Digi Python Wiki Archive

Reference Manual

Revision history—90001537

Revision	Date	Description
A	September, 2017	Converted the content from an online wiki to a reference manual.

Trademarks and copyright

Digi, Digi International, and the Digi logo are trademarks or registered trademarks in the United States and other countries worldwide. All other trademarks mentioned in this document are the property of their respective owners.

© 2017 Digi International Inc. All rights reserved.

Disclaimers

Information in this document is subject to change without notice and does not represent a commitment on the part of Digi International. Digi provides this document “as is,” without warranty of any kind, expressed or implied, including, but not limited to, the implied warranties of fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

Warranty

To view product warranty information, go to the following website:

www.digi.com/howtobuy/terms

Send comments

Documentation feedback: To provide feedback on this document, send your comments to techcomm@digi.com.

Customer support

Digi Technical Support: Digi offers multiple technical support plans and service packages to help our customers get the most out of their Digi product. For information on Technical Support plans and pricing, contact us at +1 952.912.3444 or visit us at www.digi.com/support.

Contents

Welcome to the Digi Python reference manual for developers

Start here ... 13
Reference information ... 13
What is Python? ... 13
Additional Python documenyation ... 13
Digi extras ... 14
  ConnectPort X2 ... 14
  ConnectPort X4 ... 14
  Connect WAN ... 15
  ConnectPort WAN ... 16

Digi Python programmers guide

Purpose of this guide ... 18
What is Python? ... 18
Additional Python documenyation ... 18
Getting started ... 18
Python commands in the Digi Command-Line Interface ... 19
  Python ... 19
  set Python ... 20
Loading Python programs onto a Digi device ... 20
  Using modulefinder.py to determine files to load ... 21
  Using digi_build_zip.py to automatically build a zip file ... 21
Recommended distribution of Python Interpreter ... 22
Python module reference ... 23
  Fully supported Python built-In modules ... 23
  Python standard modules with Digi-specific behavior ... 23
  File system access ... 27
Sample programs ... 27
  GPS demo ... 28
  Port Sharing Demo ... 29
Impact on payload size when using source routing ... 30

Digi provided built-in modules

Module: camera ... 34
Module: cwm ... 35
Module: digicanbus ... 36
  Description ... 36
Module: digicli ... 41
Module: digihw ... 43
Module: digij1708 ... 48
Module: digiorbcomm ... 55
Module: digipowercontrol ... 69
Module: digisms ... 74
Module: digiwdog ... 76
Module: digiweb ... 77
Module: idigidata ... 81

idigidata Python module examples 84
idigidata Python module methods 87
Module: iridium 89
Module: rci 93
Module: xbee 96

Categories

Advanced Device Discovery Protocol (ADDP) 105
Android 106
Android animation 106
Android Bluetooth test - support for Android Bluetooth test 107
Android HelloBox2D 110
Android RenderAmovingSprite 111
Android RenderAsprite 113
Android Test2D 114
Android UDP client 115
Android WDAndrolib 118
Android WatchDogDemo 119
Cellular tools 123
Simple traffic generator 124
Using Digi Realport with Python 125
Code samples 134
ADC values 135
ASM assembly code 137
Accelerometer sample 140
Advanced Device Discovery Protocol (ADDP) 141
Android animation 142
Android Bluetooth test - support for Android Bluetooth test 144
Android HelloBox2D 147
Android RenderAmovingSprite 149
Android RenderAsprite 151
Android Test2D 153
Android UDP client 154
Android WDAndrolib 158
Android WatchDogDemo 160
Battery sample 163
CAN bus sample 164
ConnectPort x 166
Detecting Cellular Status - X3 - support for detecting cellular status - X3 168
DogFighter 169
Etherios Jenova connector 173
GPS sample 179
GetConnectTankAttributes 181
IDigiMonitorSample 189
IOT Demo TradeShow 199
J1587 sample 206
J1708 sample 208
J1939 sample 209
LibDeviceCloud 211
LibFastDb 215
LibJil 218
LibJson 221
LibUtils 229
LibZkConfigProtocol 232

NET OS 9P9360 external RTC 235
NET OS Appkit Rio 237
NET OS CPU 242
NET OS Ping 243
NET OS Telnet Session 245
Reading RSSI values 248
SMTP Email 249
Scaling analog values 250
Temperature sample 253
WSBrowser 254
Wakeup sample 259
XBEE API packets 260
XBee bootloader menu 262
XBee sensor 262
XBeeComm 263
DIA 266
Auto-start Python on a Digi gateway 267
Basic web services example using DIA 270
Command line build of DIA projects 272
Core service - scheduler 275
DIA Config AIO adapter 279
DIA Device - alarm clock 280
DIA device - Runtime Totalizer 284
DIA device - sample rate reducer filter 286
DIA difference between ZigBee and DigiMesh 290
DIA Drivers and Presentations 292
DIA event uploader 293
DIA Releases 295
DIA config massa m300 serial 300
Device Cloud data streams 301
Device Cloud easy demo 302
Device Cloud RPM demo 306
Device Cloud Wiki 310
DigiMesh support in DIA 312
Enable Modbus query of DIA devices 314
Error messages 317
Example Smart Plug 319
Example XBee Serial Device 320
GE Ventostat CO2 ZigBee monitor 322
Google App Engine Device Cloud Client 326
Google Gadget LTH Sensor Example 330
Importing Modbus data from IO device 334
Microsoft PowerShell with web services 337
Modbus DIA block register map 342
Modbus DIA Client 346
Modbus DIA server 352
Motion Detection with XBee 356
Network-Time-Server-DIA-example 360
Python-based SmartPlug sensor example 362
Simple RCI by HTTP 365
Subscribing to a channel 368
Twitter DIA example 373
Understanding XBee EndPoints 377
XBee Analog I/O DIA Example 380
Data Tunneling 382

TCP to Zigbee dynamic name mapping 383
TCP to Zigbee port binding 392
UDP to XBee network 401
Understanding XBee EndPoints 405
Using Digi Realport with Python 408
Xbee transport 416
Zmatrix 420
Device Cloud 421
Archiving data files from Device Cloud 422
Auto-start Python on a Digi gateway 425
Command line build of DIA projects 427
ConnectPort X3 - GPRS, 232 interface - demo application 430
Delete a file in Device Cloud storage 436
Device Cloud Alarms 437
Device Cloud Data 439
Device Cloud data streams 441
Device Cloud web services 443
Device Cloud creation of IA configuration 443
DeviceCore 445
DeviceVendor 448
DeviceVendorSummary 449
List of Device Cloud Disconnect Reasons 450
Microsoft PowerShell with web services 451
PHP to Device Cloud 456
RCI Descriptor 457
RCI request 461
RCI do command 465
SCI 468
Sms 478
Troubleshoot Device Cloud connection 479
UI descriptor 483
XBIB display LEDs 486
XBee to Device Cloud - DataPoint Creation 492
Xbee Command to Device Cloud Cross Reference 495
Digi API Frames 498
API escape characters 499
Differences between API frame 0x10 and 0x11 501
Format API frame string 502
Raw API over ethernet to CPX2 503
Serial encap by API 504
Simple serial app quick index 507
Digi Demo Applications 511
ADC values 512
ASM assembly code 514
Advanced Device Discovery Protocol (ADDP) 516
Android animation 518
Android Bluetooth test 520
Android HelloBox2D 523
Android RenderAmovingSprite 525
Android RenderAsprite 527
Android Test2D 529
Android UDP client 530
Android WatchDogDemo 534
ConnectPort x 537
Create your own display mesh command 539

Device Cloud easy demo 540
Device Cloud RPM demo 544
DogFighter 548
Dry contact monitoring using serial signal lines 552
Fleet management demo 554
GPS Data UDP Forwarder 561
GetConnectTankAttributes 564
Google App Engine Device Cloud Client 572
How to send email via smtp 576
IOT Demo TradeShow 578
LibDeviceCloud 585
LibFastDb 589
LibJil 592
LibJson 595
LibUtils 603
LibZkConfigProtocol 606
Module: camera 609
Module: iridium 611
XBee sensor 614
Motion Detection with XBee 616
NET OS 9P9360 external RTC 620
NET OS Appkit Rio 622
NET OS CPU 627
NET OS Ping 628
NET OS Telnet Session 630
Reboot gateway at a specific time 633
Remote Power Management Demo 634
SMTP Email 642
Simple RCI by HTTP 643
Simple traffic generator 646
Smart Plug Interactive Demo 647
UDP to XBee network 650
Using Digi Realport with Python 654
Utility to set dest addr in all associated nodes 662
WSBrowser 664
XBIB display LEDs 669
XBee 868 distance/link quality demo 676
XBEE API packets 679
XBee active RFID 681
XBee Digital I/O Adapter Relay Demo 686
XBee bootloader menu 689
XBee sensor 690
XBeeComm 692
Zmatrix 693
Digi Hardware Access 695
ConnectPort serial port access 696
How to retrieve available free memory on a device 698
How to use a USB flash drive in Python 700
Locking Power Connecter 702
Module: xbee sensor 703
Power Cord Reminders 707
RS485 DB9 on Connect Products 708
Read and Write Flash USB 711
Read and write the Realtime clock 715
Transport Python programmer's guide 719

Virtual GPS NMEA Access 734
Voltage readings in XBee module 736
XBee Hardware Codes 738
XBee L-T-H sensor adapter 738
XBee Product Codes 739
XBee RS-232 adapter 744
XBEE sensors 748
Digi Products 754
Designing a sleeping XBee sensor 755
DigiMesh products 759
Digi extras 761
Python migration guide for NDS 2.8 to 2.9 767
RCI request 768
Raw API over ethernet to CPX2 773
Using terminal server to test Xbee serial adapters 774
WVA datastream to device cloud 776
Watchport Camera 779
What is your product firmware level 781
Which Digi products support Python 783
Which Python Version 786
XBee Product Codes 787
XBee RS-232 adapter 792
XBee RS-232 PH Adapter 796
XBEE sensors 797
Digi Transport Products 802
Expansion card Python options 803
FTP client 805
SFTP Client 807
SMS transport 809
Serial port transport 811
Serial data SMS on Transport 813
TCP server loopback transport 816
Telemetry 2 card digital in-to-multiple SMS example 819
Transport Python programmer's guide 823
UDP echo transport 837
Drop-in Networking Products 840
Advanced Device Discovery Protocol (ADDP) 841
ConnectPort X gateways 843
ConnectPort X2e 846
ConnectPort X5 849
DigiMesh products 850
Which Python Version 853
XBee analog I/O adapter 854
XBee Digital I/O Adapter 859
XBee display 863
XBee RS-232 adapter 864
XBee RS-485 adapter 868
XBee RS-232 PH Adapter 870
XBEE sensors 871
XBee Smart Plug 876
XBee USB adapter 878
XBee Wall Router 879
XStick 882
General Python 883
ADC values 884

Auto-start Python on a Digi gateway 887
ConnectPort FTP client capabilities 890
ConnectPort x 895
Creating run.py reading ZIP files 897
Estimating free flash file space 899
Gateway module checker 900
HTTP basic authentication 901
Handling socket error and Keepalive 902
Module finder 905
Python garbage collection 906
SMTP Email 909
Simple save and load to Flash 910
Sleep to wake on time 912
Using ZIP, GZIP or compression 915
Which Python Version 917
XBEE API packets 918
XBee sensor 919
Good Python Suggestions 921
Commandline file upload 922
Error messages 924
Handling socket error and Keepalive 926
Python garbage collection 929
Python self-testing code 932
RCI request 933
Transport Python programmer's guide 938
Use Telnet to configure 953
Using ZIP, GZIP or compression 955
Windows PythonPath 957
Java 958
Advanced Device Discovery Protocol (ADDP) 959
DogFighter 961
Etherios Jenova connector 965
GetConnectTankAttributes 971
IDigiMonitorSample 979
IOT Demo TradeShow 989
LibDeviceCloud 996
LibFastDb 1000
LibJil 1003
LibJson 1006
LibUtils 1014
LibZkConfigProtocol 1017
WSBrowser 1020
Modbus 1025
Modbus starting page 1026
Introduction to Modbus 1027
Create IA configuration by Python script 1029
Determine MTU 1033
Device Cloud creation of IA configuration 1035
Enable Modbus query of DIA devices 1037
How to create Modbus/RTU request in Python 1040
Importing Modbus data from IO device 1042
Integrating the Digi IA Modbus bridge to Python 1046
Modbus bridge on CPX4 1050
Modbus DIA block register map 1058
Modbus DIA Client 1062

Modbus DIA server 1068
Modbus Dia Code Add-On 1072
Modbus Example Ethernet Adapter 1073
Modbus Example Serial Adapter 1078
Modbus Example X4 Setup 1085
Modbus Floating Points 1088
Modbus Serial Over Mesh 1090
Modbus starting page 1093
Modbus ZB Fragmentation Support 1094
Modbus class design in Python 1096
Modbus on Digi Products 1097
Python CRC16 Modbus DF1 1100
Setting Serial Adapter Baud Rate 1103
Understanding XBee EndPoints 1108
NET+OS 1111
ASM assembly code 1112
NET OS 9P9360 external RTC 1114
NET OS Appkit Rio 1116
NET OS CPU 1121
NET OS Ping 1122
NET OS Telnet Session 1124
Programmable XBee 1128
Programmable XBee - getting started 1129
XBee bootloader menu 1129
XBeeComm 1131
Rockwell Allen-Bradley 1132
Importing Rockwell data to Dia and iDigi 1133
Python CRC16 Modbus DF1 1140
SMS Service 1143
Introduction to SMS service 1144
Module: digisms 1145
Sms enabling support 1147
SMS host sends 1150
Sms sample Digi to Digi 1153
TLS/SSL 1154
Tlslite 1155
Third Party Devices 1157
Device Cloud easy demo 1158
GE Ventostat CO2 ZigBee monitor 1162
Motion Detection with XBee 1166
Unsupported Third Party Dia Code 1170
DIA Device - alarm clock 1171
DIA device - Runtime Totalizer 1174
DIA device - sample rate reducer filter 1176
DIA event uploader 1180
Twitter DIA example 1182
Web Access 1186
Module: digiweb 1187
Python inside HTML 1190
Python interactive web console 1194
RCI request 1198
RCI do command 1203
Simple RCI by HTTP 1206
Web Auto Refresh 1209
Working with 802.15.4 1211

Channels, Zigbee 1211
Xbee Command to Device Cloud Cross Reference 1215
Xig 1218
Working with DigiMesh 1219
DIA difference between ZigBee and DigiMesh 1219
DIA Releases 1222
DigiMesh products 1226
DigiMesh support in DIA 1228
Quick guide to DigiMesh setup 1230
Remote XBee management with XCTU 1232
Sleep settings within DigiMesh 1235
Xbee Command to Device Cloud Cross Reference 1237
Working with Zigbee 1240
Binding multiple zigbee sockets 1241
Bootloader to force XBee reflash 1242
Channels, Zigbee 1244
Configuring Digi XBee modules 1249
Create your own display mesh command 1252
DIA difference between ZigBee and DigiMesh 1254
DIA Releases 1256
Determine MTU 1261
Differences between API frame 0x10 and 0x11 1263
Error messages 1264
Hardcoding a fixed XBee PAN ID 1266
How to detect radio series in Python 1267
Joining Under Xbee ZigBee 1268
Large ZigBee Networks and Source Routing 1270
Module: xbee 1272
Monitoring a ZigBee Network 1279
Remote XBee management with XCTU 1282
Sending AT commands to the gateway 1284
Sending broadcast transmissions 1285
Simple serial app quick index 1286
TCP to Zigbee dynamic name mapping 1289
Understanding XBee EndPoints 1298
Utility script to get/set AT commands on local/remote zigbee node 1301
Utility to set dest addr in all associated nodes 1306
Xbee Command to Device Cloud Cross Reference 1308
XBee extensions to the Python socket API 1311
XBee sleeping problems 1322
Xig 1324
XBee Zigbee 1325
XBee bootloader menu 1326
XBeeComm 1327

Categories

Reading from ZipFiles on the Gateway 1328
EmbeddedLinux - time sample 1328
EmbeddedLinux - UDP server-client 1329
IDigi Easy Demo Details - Support for iDigi Easy Demo 1330
The Basic Framework of the main function in your first Programmable XBee Application 1331
Step 1: sys_hw_init() 1331
Step 2: sys_xbee_init() 1332
Step 3: (Step 3): sys_app_banner() 1333

Step 4: The "for-loop" 1333
Using an additional XBIB from a different kit 1333

Welcome to the Digi Python reference manual for developers

Start here

This document provides information for developers of legacy Digi products that support Python programming. The products covered in this document are primarily NDS-based products.

These include:

- Digi ConnectPort X4
- Digi Connect SP
- Digi ConnectPort TS

Note This document is an archive of an online wiki, and is available in this archived version for users who continue to rely on this content. This document is not maintained or supported, and will not be updated. However, please browse www.digi.com/support for additional documentation and resources based on your needs and the Digi products you have purchased.

Reference information

- Digi XBee, RF Gateway and Python Development Resource page
- Digi Python Learning Recommendations
- How to Esablish TCP Connectiosn to XBee Nodes Using Python on a ConnectPort X Coordinator

What is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

Additional Python documenyation

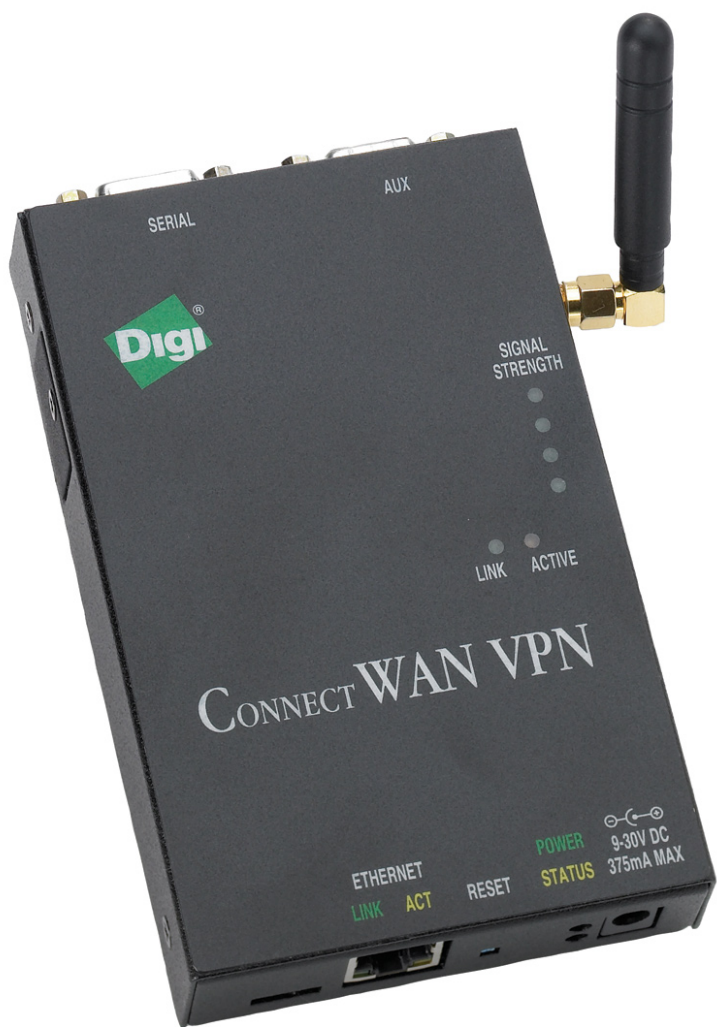
For more information on the Python Programming Language, go to www.Python.org and click the Documentation link.

Digi extras

ConnectPort X2

ConnectPort X4

Connect WAN



ConnectPort WAN

Digi Python programmers guide

Purpose of this guide 18
What is Python? 18
Additional Python documenyation 18
Getting started 18
Python commands in the Digi Command-Line Interface 19
Loading Python programs onto a Digi device 20
Recommended distribution of Python Interpreter 22
Python module reference 23
Sample programs 27
Impact on payload size when using source routing 30

Purpose of this guide

This guide introduces the Python programming language by showing how to create and run a simple Python program. It describes how to load and run Python programs onto Digi devices, either through the command-line or web user interfaces. It reviews Python modules, particularly those modules with Digi-specific behavior. Several sample Python programs are included on the Software and Documentation CD. This guide describes how to run the executable programs and describes program files.

What is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

Additional Python documenyation

For more information on the Python Programming Language, go to www.Python.org and click the Documentation link.

Getting started

This section shows how to create a very simple Python program, named hello.py, and the steps necessary to run that program on a Digi device.

First Program: “Hello, World!”

Step 1: Write the program.

In a text editor, create a file named hello.py, with the following contents:

```
# hello.py – Simple demonstration program
print “Hello Digi World!”
```

Step 2: Test the program locally.

Digi has attempted to expose all the currently available Python functionality using subsets of standard Python APIs. This means you should be able to port programs between a Digi device and other Python running systems with a minimum of modifications. The tools of a real PC provide a friendlier environment in which to check for issues in the program and debug it. While the hello.py program is simple, it is a good practice to run programs locally before attempting to move them to the Digi device.

Step 3: Move the program onto the Digi device.

In a web browser, access the web interface of the Digi device.

1. Log in to the device.
2. Using the menu, navigate to the Applications > Python page.

3. In the Upload Files section of the Python page, type in the location or browse to select the hello.py file created earlier.
4. Once selected, click the Upload button to place the file into the file system of the device.

Later, when creating more substantial programs, this same mechanism is used to load modules and ZIP files containing modules and packages on the Digi device's file system.

Step 4: Run the program.

1. Telnet or SSH to the Digi device and run this command:

```
Python hello.py
```

2. The program should output Hello Digi World and then exit.

Congratulations! You have just successfully run a Python program with the interpreter embedded on your Digi device.

Python commands in the Digi Command-Line Interface

The Digi command-line interface has two commands for configuring and executing Python programs on Digi devices:

- Python: Manually executes a Python program.
- set Python: Configure Python programs to execute when a Digi device boots.

Detailed descriptions of the commands follow.

Python

Purpose

Manually executes a Python program from a Digi device's command line.

The Python command is similar to a command executed on a PC. However, other than a program name and arguments for the program, the command takes no arguments itself.

Syntax

```
Python [TFTP server ip:]filename [program args...]
```

[TFTP server ip:]filename

The main file to be executed. This file can be either a file on the file system accessed through the web UI, or a file accessible through a TFTP server on the network. This TFTP functionality reduces the number of times that you may need to place a program on the file system while developing and refining functionality. However, the TFTP behavior only works for the main program. Modules and packages must still be present on the file system to be used.

If no filename is given, an interactive shell is spawned. To exit the interactive shell, you must import sys and call sys.exit(), because exit() is not implemented in DigiPython.

program args

The arguments for the program.

set Python

Purpose

Configures Python programs to execute on device boot.

Syntax

```
set Python [range=1-4] [state={on|off}] [command=filename,args]
```

Options

range=1-4

Range specifies the index or indices to view or modify with the command.

state={on|off}

When the state is set to on, the specified command is run when the device boots.

command=filename,args

The program filename to execute and any arguments to pass with the program, similar to the arguments for the Python command. The command option allows for programs to be run from a TFTP server; however, this usage is not recommended. If there are spaces to provide arguments, make sure to wrap the entire command in quotation marks.

Loading Python programs onto a Digi device

As demonstrated in the *hello.py* example, Python programs are loaded onto the Digi device using the Digi web interface page *Applications* > *Python*. Program files and modules can be uploaded directly to the Digi device's file system through this page. However, files can be uploaded one at a time only, and there is no support for directory manipulation. This makes adding entire libraries of modules difficult and adding packages impossible. To address this issue, Digi provides a module called *zipimport*, allowing collections of modules and packages to be included in a single upload.

To decide which modules work well on the device, or to share information on your experiences, please check out the Module Notes page.

To use *zipimport*, add all the files to be moved onto the Digi device to a file named *Python.zip*. By default, Digi's embedded copy of the Python interpreter checks for the presence of this file, and examines it when performing an import. Digi provides a *Python.zip* file which includes several useful Python standard library modules that are known to work well on our device. This file comes pre-loaded on your Python enabled device, or the latest version will always be available at Digi's product support website as part number 40002643.

Besides *Python.zip*, the *zipimport* module can be used with additional .zip files, provided the Python program knows of their presence and modifies its environment accordingly. Internally, the files accessed through the *Applications* -> *Python* web page are stored in a directory called **WEB/Python/**. To use additional .zip files, add them to the *sys.path* variable. This causes zipimport to search that file for .zip files as well. See the GPS sample application for an example.

To add mymodules.zip to the search path, you will need to following in your application.

```
import sys
sys.path.append("WEB/Python/mymodules.zip")
```

This is perfectly adequate and recommended for a production environment. However, during development it is often desirable to be able to load a modified version of a zipfile onto the device without rebooting. In order to do this, we must remove our zipfile from zipimport's directory cache.

The following code should allow you to load and reload mymodules.zip without rebooting.

```
                import sys, zipimport
zip = "WEB/Python/mymodules.zip"
if zip in zipimport._zip_directory_cache:
    del zipimport._zip_directory_cache[zip]

sys.path.append(zip)
```

Using modulefinder.py to determine files to load

For most programs, determining which files should be moved onto the Digi device should be fairly simple, most likely you will be writing most program modules and content yourself. However, when using third party modules or those provided by the standard distribution, a tree of dependencies may exist, making it difficult to determine which files must be placed on the device.

The standard Python distribution provides a tool called *modulefinder.py* that is useful in this scenario. This tool examines imports in Python programs to build a list of modules that may be used.

Using digi_build_zip.py to automatically build a zip file

digi_build_zip.py is an experimental utility to automatically build a zip file containing modules and packages required to run a Python application on the Digi platform. At its heart, *digi_build_zip.py* uses the modulefinder module to analyze a given script to build the zip file with some added intelligence. However, this analysis method is not perfect and errs on the side of inclusion. For example, many Python standard libraries perform dynamic feature support detection and import further packages if the OS feature is supported. modulefinder parses these import statements and includes packages and code that will never be used by the application script running on the Digi platform.

Using *digi_build_zip.py* is simple: execute the script with the name of the application script to be loaded on the Digi device as an argument. For example:

```
C:\My Project> Python digi_build_zip.py my_project.py
```

By default, *digi_build_zip.py* creates a zip file using the base name of the script. For example, *my_project.py* becomes *my_project.zip. digi_build_zip.py* will normally act silently until the zip file is written, unless any errors or warnings occur. Additional script options are available by specifying the --help option.

For example, here is output for the regular expression module:

```
Name File
--------
m __main__ re.py
m _sre
m array /usr/lib/Python2.4/lib
dynload/array.so
m copy_reg copy_reg.py
m sre sre.py
m sre_compile sre_compile.py
m sre_constants sre_constants.py
m sre_parse sre_parse.py
m sys
```

```
m types types.py
```

When using *modulefinder.py*, the list needs to be trimmed manually to minimize the size of the *Python.zip* file and keep dependencies to a minimum. *modulefinder.py* lists files even if they exist in an execution path that the program will never use and would therefore not need to be present. It also lists modules that are built-in or may be dynamically loaded in your environment. Choosing which programs to load on the Digi device requires care, because some files may require functionality not present on the Digi device. Further, because Python is a dynamic run-time interpreted language, this missing functionality may not cause errors in program execution until the code is interpreted.

Recommended distribution of Python Interpreter

The current version of the Python interpreter embedded in Digi devices is 2.4.3. Please use modules known to be compatible with this version of the Python language only.

To obtain Python 2.4.3 for your Windows or Linux computer, it can be downloaded directly from this page: http://www.Python.org/download/releases/2.4.3/.

Python module reference

Fully supported Python built-In modules

These modules are fully supported in the Digi implementation of Python:

- array
- binascii
- cStringIO
- cmath
- collections
- errno
- itertools
- math
- operator
- pyexpat
- select
- strop
- struct
- zipimport
- zlib

Python standard modules with Digi-specific behavior

These Python modules have some Digi-specific behavior and limitations that are important to be aware of when designing an application:

os

Use of the os module in Digi devices is currently very limited. The primary purpose in exposing it is to allow access to the serial ports, which are presented as nodes in the file system. Serial ports are available as files with the path in the form /com/0 with the zero replaced by the zero-based index of the serial port to control.

Please see also Digi Connect Port Serial Port Access for a fuller discussion of using the os module to access serial ports.

The file system currently does not allow directory traversal or manipulation, and all file specification must be performed using absolute file names only. All files accessible through the *Applications* > *Python* web page are placed in the directory prefix *WEB/Python/*.

When importing the os.py module, the module provides some functionality that will not work on Digi systems. These calls should work completely:

- open
- close
- read
- write

- lseek
- remove
- unlink
- isatty

However, not all of the above calls may be available on devices running SarOS. You can check if the above calls exist by connecting via SSH or telnet to a device and running `import os` followed by accessing the desired function as an attribute (such as running `os.open`)

These calls will not return complete information, and only the attributes *st_size, st_blocks*, and *st_mode* will contain correct information:

- stat
- fstat

Rather than deal explicitly with the limitations of the os modules for files, it is recommended to use Python's built-in *file* objects, which have full functionality directly.

All files necessary to use the *os* module are included in the *Python.zip* provided by Digi.

random

The random class or built-in calls to the module-level instance functions work using a time-based seed. However, the *SystemRandom* class cannot be used, because there is no OS support for a */dev/urandom* device.

The *random.py* file necessary to use the built-in random module is included in the *Python.zip* provided by Digi.

re

The *re* module works correctly. All files necessary to use regular expressions are included in the *Python.zip* file provided by Digi.

socket

Most of the standard Python socket API is available for use on Digi devices. However, there are some exceptions and limitations in functionality.

Although *gethostname()* is supported, *gethostname_ex()* is not.

Normally, socket implementations make the assumption that one cannot bind two sockets to the same host and port. This is expected, and Linux- and NDS-based devices will raise an error message upon the attempt to bind the second socket to a host and port with a pre-existing socket. SarOS devices follow the non-standard behavior of silently letting the second socket replace the first socket. This makes socket-related logical errors much harder to debug on SarOS devices. In addition, SarOS devices have very few sockets available, while Linux and NDS have socket implementations that are largely constrained by the number of concurrent files can be opened.

Service and protocol lookup functions and the related name resolution functionality are not available. The Digi device does not currently maintain a database of symbolic service names. This means that the *getservbyname()*, *getservbyport()* and *getprotobyname()* functions are not available.

Digi has extended the standard socket API to provide access to ZigBee mesh networks. Please see Zigbee Extensions to the Python socket API for details

For examples of socket error handling, as well as TCP Keepalive usage see: Handling_Socket_Error_and_Keepalive

termios

A limited-functionality version of the termios library is present in the Digi device. It is not expected that the library will need to be used often for configuration, because the Digi device provides a mechanism for lasting configuration of the serial port using the user interfaces provided. Any configuration of the serial port using the *termios* module in Python will possibly interfere with other ways in which the serial port can be used in the system. If using the *termios* module, make sure your Python code is the only element in the system using the port.

Please see also Digi Connect Port Serial Port Access for simple Python examples of using the *termios* module to set baud rate and other common port settings.

Supported flags

Iflags	Description
IXON	Enable recognition of software flow control characters for output flow control.
IXOFF	Enable generation of software flow control characters for input flow control.
IXANY	Un-pause output flow control on reception of any character.
INPCK	Check the parity bit on incoming data.
IGNBRK	Ignore breaks in the data stream.
IGNPAR	Ignore parity errors in the data stream.
PARMRK	Encode errors in the data stream as 0xff 0x00 <ch>.
DOSMODE	When used with PARMRK, encode the second byte of the error string as: 0x10 – A break was received. 0x08 – A framing error was received. 0x02 – An overrun occurred.
ISTRIP	Strip incoming characters to 7-bits wide.
Oflags	
ONLCR	Insert a carriage return before every outgoing newline in the data stream.
OCRNL	Insert a newline after every outgoing carriage return in the data stream.
ONOCR	Do not transmit carriage return if terminal state is already at column 0.
ONLRET	Newline characters perform carriage return on attached terminal.
TABDLY	Expand tabs in outgoing data stream to the number of spaces that will bring the terminal column to an 8 character tab stop.
Cflags	
CSIZE	Number of data bits **CS5**, **CS6**, **CS7**, **CS8**.
CSTOPB	Number of stop bits: 2 when CSTOPB is set, 1.5 if CS5 is also set.
PARENB	Enable parity generation.
PARODD	Odd parity when **PARENB** is set; even if **PARODD** is clear.
CRTSCTS	Hardware flow control.

Iflags	Description
C_cc	
VSTART	Software flow control start character.
VSTOP	Software flow control stop character.
VLNEXT	Next character is not interpreted with any special meaning.

Notably, no local modes are supported. Also, **VMIN** and **VTIME** are not supported; all *os.reads()* return immediately if any received data is available. However, there are *oflags*; the **OPOST** flag is not recognized. Setting an *oflag* is sufficient to turn on processing for that flag. If configured, the **LNEXT** character will be recognized despite the fact that canonical mode is not supported. The receiver is always enabled, so **CREAD** is not supported.

The *ispeed* and *ospeed* members of the attribute list passed to *tcsetattr()* must be identical. However, it is not required that they be specified using the existing symbolic defines. They may be specified numerically, and the *tcsetattr* call will succeed if the system can provide that baud rate within an error of 5%.

Non-standard routines

Two non-standard routines have also been added to the termios module:

- tcgetstats()
- tcsetsignals()

tcgetstats()

Purpose

Get modem signal and event status.

Syntax

```
tcgetstats(fd) -> [estat, mstat]
```

Description

The *estat* variable is a bit-mask that reports status using the following constants:

EV_OPU	Output paused unconditionally (using tcflow())
EV_OPS	Output paused by software flow control
EV_OPH	Output paused by hardware flow control
EV_IPU	Input paused unconditionally
EV_IPS	Input paused by driver logic

The *mstat* variable is a bit-mask that reports signal status using the following constants:

MS_RTS, **MS_CTS**, **MS_DTR**, **MS_DSR**, **MS_RI**, **MS_DCD**

tcsetsignals()

Purpose

Set the output signals of the serial port.

Syntax

```
tcsetsignals(fd, sigmask) -> None
```

Description

When the outgoing signals RTS and DTR are not being used for flow control, they can be controlled with this function. Sigmask is a bit-mask that should be set using the MS_RTS and MS_DTR constants.

thread and threading

The built-in **thread** and helper threading modules are available for use as needed in the system. They operate normally. However, because of the manner in which the Python interpreter is embedded in the system, error handling and thread exiting need to be done with care. Take all possible measures to ensure that threads started from a main thread of execution exit prior to the main thread exiting. Failure to do so will cause undefined behavior.

Consider whether threads are required in the application. If they are used, the main thread should include a top level *try-except* or *try-finally* construct that captures all exceptions and performs a loop that waits on a *join()* call or equivalent for all child threads. In addition, the main thread should be as simple as possible to minimize possible unforeseen termination.

time

The *time* module is believed to work well. As Digi does not yet have time-zone support, the *tzset()* function is not available. The *clock()* function has a resolution of milliseconds only, not microseconds, and tracks the uptime of the device. If the system you are using has a configured real-time clock, the *time()* routine will return the correct value; otherwise it too will return the system uptime, like the *clock()* function.

File system access

Digi devices may support several varieties of file systems. Each distinct file system in the device is prefixed in the pathname with a volume specifier. All Python-enabled Digi devices provide the *WEB volume*, which contains the *WEB/Python* directory from which programs and modules are managed. Some devices provide support for attached USB mass storage devices such as flash drives, and mini-hard drives. If USB mass storage devices are supported, when attached they will be added as volumes lettered A-Z based on their order of enumeration in the device. For more information see: How to use a USB Flash drive in Python.

File systems on Digi devices are subject to all of the limitations mentioned for the os module. Namely, directory operations are not supported and files must be specified with complete absolute path names including volume. For an example using an attached USB mass storage device, see the port sharing demo. All file system calls provided including write operations are fully blocking and will not complete until all data provided has been committed to the storage medium.

Sample programs

Several sample Python programs are included on the Software and Documentation CD for your Digi device. This section describes the sample programs and files.

GPS demo

The GPS demo in the Python/Samples/gps directory on the Software and Documentation CD Gps_demo.zip) communicates serially with a GPS receiver using the NMEA 0183 data format and presents the location information retrieved through that data stream as a web page redirection to Google Maps as well as through remote procedure calls using the XML-RPC protocol. It is not necessary to have a GPS receiver to run and view results from the application. In the absence of GPS data, it reports the location of the Greenwich Royal Observatory. New releases of Digi firmware provide support for GPS devices as part of the built-in functionality. If using a product with an embedded GPS, you should modify the script to open the */gps/0* file for the NMEA data stream.

Run the GPS demo

1. Load the demo files onto the device by selecting **Applications > Python** from the Digi device main menu, and using the file-loading function.
2. (Optional) Attach a GPS device to the serial port and configure the port for 4800 8N1 and the GPS for NMEA.
3. Run the **gps_demo.py** program.
4. Point your web browser at port 8080 of the device running the demo. It should generate an HTTP redirect with a query to maps.google.com for its location.
5. Run the following script on a PC to use the XML-RPC behavior:

```
# Get position information from gps_demo.py
import xmlrpclib

s = xmlrpclib.Server("http://<ip of device>:8080")
print s.position()
```

Files in GPS demo

The GPS demo consists of the following file:

gps_demo.py

This file includes the main application logic, which ties together the GPS communication library and the XML-RPC and HTTP modules from the standard library in a shared framework.

Notable aspects of the main application:

- The program modifies the **sys.path** program variable to add **WEB/Python/gps.zip** to the module search path. This ensures that the additional libraries will be found in the import process for the SimpleXMLRPCServer which follows.
- The program creates a built-in way to force an exit using another socket. There is no current signal support provided by the Digi environment, thus no external default means of generating an interrupt in a program. For testing, it is helpful to be able to cause a program to exit. When finished, this logic could be removed or conditionally enabled/disabled based on arguments so that a port scan or other network activity can not inadvertently cause program termination.
- Asynchronous processing of data using select. The built-in select module works for all socket and serial operations and ensures that the program runs only when it has work to perform.

- **sys.argv** support. The program allows overriding the default run-time parameters by processing the argv array.

nmea.py

This file contains a library that performs the parsing and extraction of location information from an NMEA sentence stream on the serial port. Notable aspects of this library include:

- The library uses the re module to perform protocol recognition. Each NMEA sentence is fairly simple consisting of a starting dollar sign, a two character device type identifier, a three character sentence identifier, a comma separated list of sentence specific data elements and an optional two digit checksum proceeded by an asterisk if present.
- The library does not care what the source is for the provided data. Any endpoint providing NMEA data can be processed through this library through its provided feed function.
- The core functionality is provided as a class object so that multiple instances of parsing can be generated and exist simultaneously in one program.

The library does a number of things that could be tuned for a GPS application and which require more complex manipulation and knowledge of the data stream. It does not verify that checksums are present on the sentences that are required to have them. The templated sentence lists used to extract data elements do not handle all sentence types; with more sentence types, the elements being added to the class dictionary would become cumbersome. Furthermore, there are possible performance considerations. In testing, with a single 4800 bps data stream, it was far from an issue. However, string manipulation of this form, which is heavy on slicing and splitting, requires several dynamic memory and copy operations, because strings are immutable and each operation creates and populates entirely new strings.

gps.zip

The XML-RPC and HTTP behavior is performed by taking advantage of additional files from the Python 2.4.3 standard library distribution. These files have ways of being used that can attempt to do things such as directory manipulation and hostname lookup. Because of the potential of using these files incorrectly, they have not been included in the base **Python.zip** file. However, the problematic behaviors of the modules are present only when the modules are used in particular ways; if used properly you can still take advantage of most of their power.

This list of files to include in the **gps.zip** file was generated through manual inspection of the imports performed by the only module from the fine included on the top level; **SimpleXMLRPCServer**. This process could also be performed using the **modulefinder.py** module. However, manually scanning the modules imports allows for reviewing each module for possible problems and usages that could appear when running on the Digi device.

Port Sharing Demo

The port sharing demo program in Python/Samples/sharing Sharing.zip) demonstrates an asynchronous socket server that allows multiple socket clients shared access to a single serial port. The demo program does this by using the select module and standard API calls. Much like the GPS demo, it listens on a socket to provide the main application behavior, and provides a socket to ask the application to exit as well for ease of debugging. However, if no sockets are attached, the demo application spools any data received to a file on an attached USB mass storage device.

Run the port sharing demo

1. In the web interface of the Digi device, go to Applications -> Python and load the demo files onto the Digi device.
2. Configure the serial port to match any attached serial device.
3. In the command-line interface for the Digi device, execute the Python command, specifying the file sharing.py.
4. Connect to port 8001 with up to five TCP clients and read/write data to the sockets.
5. Disconnect all TCP connections and generate data on the serial port. This data will be spooled in a file on any attached USB mass storage device.

Files in port sharing demo

sharing.py

The *sharing.py* file contains the master logic for the application. The majority of the program logic is in the *process()* function. The port-sharing logic runs in the main thread, while a child thread is created to spool any queued data to the file system. This is done to ensure that file system blocking will not interrupt processing of any additional data during the write.

Notable aspects:

- Since most of the application logic is in process, it should be easy to extend this program to create a thread per serial port to run the process routine on multiple serial ports. Because of resource limitations on the Digi device, using this routine or select would be the preferred approach to extend the program to multiple ports, rather than running multiple copies of the program.
- The program takes care to request termination and perform a join on child threads when exiting, to avoid the main thread exiting before all child threads have exited. This avoids causing undefined behavior and instability when the main thread exits.
- The program imposes a maximum limit on client connections and I/O request sizes. Once again, this is to place a boundary on resource use rather than allowing a situation where usage could grow to be unbounded.

spooler.py

The Spooler class in the *spooler.py* file is an extremely simple sub-class of the thread object from *threading.py*. It retrieves objects from a queue object, and if the objects are strings, writes them to a file specified in the constructor. As mentioned for the *sharing.py* program, this program runs as a thread to allow file system blocking to not hold off the continued execution of program logic in the main thread. The thread object was designed to allow a mechanism for the main thread to ask it to terminate as well. This provides a way to cleanly exit the entire application, with child threads being terminated before the main thread.

Impact on payload size when using source routing

In large ZigBee networks, many-to-one and source routing can be used to reduce routing overhead and improve network performance. They are supported in XBee RF modules that run ZB and SE firmware. They are enabled when the AR (Aggregate Routing Notification) parameter of an XBee RF

module, usually in the ConnectPort X gateway, is set to less than 0xFF. When source routing is enabled, the gateway will automatically capture and use the source route from each remote node. When data is received from a node, the route from the node is stored in the gateway. When transmitting data to the same node, the gateway will insert this route into the packet, causing it to be routed back to the node along the same path. An application does not need to change how it addresses transmissions to the node. When the source route is inserted into a packet, it becomes part of the packet's data payload. This reduces the maximum size of the payload that is available to applications. The source route size is 2 bytes per hop plus 1 byte. The maximum size is 21 bytes. The NP (Maximum RF Payload Bytes) parameter of the XBee RF module in the ConnectPort X gateway indicates the maximum payload size available, including any source route. To allow space for a source route, the application is recommended to use a maximum data payload size of NP-21. If the application transmits a payload size that is too large, a TX status of 0x74 is returned. The application must be receiving TX status frames to receive this status. The data packet will be dropped.

For more information, see this Wiki article on large ZigBee networks and source routing: http://www.digi.com/wiki/developer/index.php/Large_ZigBee_Networks_and_Source_Routing

Digi provided built-in modules

These modules may be available as built-in modules on your system. They are not provided as Python source code, but are extensions provided by the Digi device to expose internal functionality specific to our products.

See the Availability section of each module page for support information. Common support limitations are summarized here. "No (HW)" means the device lacks the required hardware. "No (FW)" means the underlying operating environment lacks the required support.

Module	ConnectPort X2/WiX2	ConnectPort X3/X3R	Connectport X5R/X5F	ConnectCore 3G 9P	X-Trak 3
Module: camera	No (HW)	No (HW)	No (HW)	No (HW)	No (HW)
Module: digicli	Yes	No (FW)	Yes	No (FW)	No (FW)
Module: digihw	No (HW)	Yes	Yes	Yes	Yes
Module: digipowercontrol	No (HW)	Partial	Yes	Partial	Partial
Module: digisms	No (HW)	Yes	Yes	No (HW)	Yes
Module: digiwdog	No (HW)	No (HW)	Yes	No (HW)	No (HW)
SSL/TLS/HTTPS	Yes	No (FW)	Yes	No (FW)	No (FW)

Module: camera

Introduction

Built-in Python module provided by Digi for interacting with a Watchport Camera/V2 camera.

Functions

get_image()

Purpose

Retrieve the most recent camera image as a JPEG.

Syntax

```
get_image() . (image, timestamp)
```

- *image* is buffer containing a JPEG of the most recent camera image
- *timestamp* is the timestamp the image was acquired by the device, in milliseconds since device power on.

Description

Returns the most recent image from the attached Watchport camera. The camera parameters affecting the resolution, image quality, frame rate, etc are configured via the device web interface or command line interface.

The image is a standard JPEG.

The timestamp returned with the image is the time when the image was acquired by the device. The value is the same type as the uptime of the system (milliseconds since last boot time). The timestamp can be used to find the elapsed time between two images. It can also be used to determine if two calls to get_image() are the same image. Note, the timestamp of the image will be different than the current uptime of the device. This is because camera images are acquired continuously by the device based on the Frame Delay setting. The most recent image acquired by the device is what is returned by get_image() which can be as old as the value in Frame Delay.

If an image is not available (no camera attached, camera disabled, etc) (None, None) is returned.

Example

```
##Internal module
import camera

##Destination of the image
fh = open('WEB/Python/demo_image.jpg', 'wb')

##image, and timestamp
image, timestamp = camera.get_image()

##write buffer and close
```

```
fh.write(image)
fh.close()
```

The above code saves the current camera image to a file on the device file system.

Note the file handle must be opened in binary mode. Writing the image buffer without specifying 'b' will result in a corrupted image.

Availability

Products which support this module

This feature is available only on Digi products with built-in USB host ports.

Products which DO NOT support this module

This feature is not currently supported on:

- Digi ConnectPort X2/WiX2
- Digi ConnectPort X3

Module: cwm

Provides a bridge between this device and Connectware Manager.

There are currently no public functions in module cwm.

Users are encouraged to use the cwm_data.py module and associated cwm_data.zip to interact with Connectware.

Module: digicanbus

digicanbus.py

This module was designed in conjunction with the Digi ConnectPort X5, the first Digi Python-enabled product to offer CAN bus connectivity. The module is designed to allow scripts to manipulate CAN messages at a logical level, with the embedded operating system providing support for the encoding/decoding of messages on the line.

Limited J1939 support is also provided, in order to simplify the interpretation of CAN messages as J1939 PDUs. Knowledge of individual PDU encodings is application specific, and must be supplied by the script.

Module help

Help on built-in module digicanbus:

Name

digicanbus - Provides Python abstraction for an underlying CAN controller

File

(built-in)

Description

The CANHandle class provides functions to initialize a CAN bus and transfer raw CAN messages. It also provides functions to communicate over the CAN bus via the abstraction of the SAE J1939 protocol.

The J1939_PDU class provides methods to convert raw CAN identifiers into SAE J1939 identifier fields and vice-versa. When using the SAE J1939 abstraction, this class is used directly to represent input and output messages. The class can also be used, however, to transmit and receive SAE J1939 messages using the raw CAN interface.

Classes

```
__builtin__.object
CANHandle
J1939_PDU

class CANHandle(__builtin__.object)
 |  Provides a programming interface to an underlying CAN controller
 |
 |  Instantiating a CANHandle associates a set of methods with an internal
 |  CAN bus instance.  The methods allow one to request configuration of the
 |  bus, register message filters and receive callbacks, and submit messages
 |  for transmission.
 |
 |  CANHandle objects are instantiated with a single "can_bus_id" argument,
 |  which is an integer identifier for the desired CAN bus.
```

```
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see x.__class__.__doc__ for signature
 |
 |  configure(...)
 |      configure([options]) -> None
 |
 |      The CAN bus will be stopped if running, configuration parameters
 |      will be applied, and the bus will be reinitialized.
 |
 |      With no arguments, the current configuration will be re-applied.
 |
 |      Option Arguments:
 |
 |      bitrate=n -- 'n' defaults to 250000, with a maximum of 1000000
 |
 |  register_filter(...)
 |      register_filter(width, identifier, mask, method, return_arg) -> None
 |
 |      Ask bus to accept and deliver messages matching filter via method
 |
 |      Associate a filter with a reception method function and a return
 |      argument.  Only a single copy of each unique set is stored
 |      internally.  The registration is with the underlying bus
 |      instance, not with the handle.
 |
 |      The mask parameter must be of the same form as the
 |      identifier, and indicates which bits in the identifier are
 |      significant for matching.
 |
 |      The method must be a callable object of the form:
 |
 |      method(width, identifier, remote_frame, payload, return_arg)
 |
 |      The "return_arg" argument is not examined by the bus
 |      infrastructure.
 |
 |      An exception is raised if the number of filters is exceeded.
 |
 |      NOTE: The callback method will be executed in another thread.
 |      Shared data will need to be protected from race conditions due
 |      to concurrent access.
 |
 |  send(...)
 |      send(width, identifier, remote_frame, payload) -> True | False
 |
 |      Non-blocking transmit of a message on the CAN bus.
Arguments:
 |      width -- 11 or 29, referring to the bits in the CAN identifier
 |      identifier -- 11 or 29 bit integer value
 |      remote_frame -- True or False to indicate a remote or data frame.
 |      payload -- If a remote frame, payload is an integer count of the
 |                 expected bytes.  If a data frame, payload is a string
 |                 of length 0 - 8 of the bytes to be transmitted.
 |
 |      Return Value:
 |      The function will return a true value if the message was
```

```
 |      successfully queued, and false if there is no queue space.
 |
 |  unregister_filter(...)
 |      unregister_filter(width, identifier, mask, method, return_arg) -> None
 |
 |      Remove filter/method/argument pairing from the bus filter list
 |
 |      The filter list of the bus is searched for a match of the same
 |      calling parameters.  If a match is found, that registration is
 |      removed from the list.  If no match is found, an exception is
 |      raised.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T

class J1939_PDU(__builtin__.object)
 |  Provides a means to map J1939 PDU fields to a CAN identifier
 |
 |  J1939_PDU objects are instantiated with no parameters (creating
 |  a template PDU), or with a tuple representing a CAN message:
 |    (width, identifier, remote_frame, payload)
 |  Some fields overlap, per the J1939 specification.  Manipulation of
 |  (for instance) the PGN field modifies most of the fields at once.
 |
 |  Methods defined here:
 |
 |  CANMsgTuple(...)
 |      CANMsgTuple() -> (width, identifier, remote_frame, payload)
 |
 |  PDU1Check(...)
 |      PDU1Check() -> a true value if PDU is in PDU1 format
 |
 |  PDU2Check(...)
 |      PDU2Check() -> a true value if PDU is in PDU2 format
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see x.__class__.__doc__ for signature
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  DA = <attribute 'DA' of 'digicanbus.J1939_PDU' objects>
DP = <attribute 'DP' of 'digicanbus.J1939_PDU' objects>
 |      Data Page
 |
 |  EDP = <attribute 'EDP' of 'digicanbus.J1939_PDU' objects>
 |      Extended Data Page
 |
 |  GE = <attribute 'GE' of 'digicanbus.J1939_PDU' objects>
 |      Group Extension (PDU2 format)
 |
 |  PF = <attribute 'PF' of 'digicanbus.J1939_PDU' objects>
 |      PDU Format
 |
 |  PGN = <attribute 'PGN' of 'digicanbus.J1939_PDU' objects>
 |      Parameter Group Number
```

```
 |
 |  PS = <attribute 'PS' of 'digicanbus.J1939_PDU' objects>
 |      PDU Specific
 |
 |  SA = <attribute 'SA' of 'digicanbus.J1939_PDU' objects>
 |      Source Address
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  payload = <attribute 'payload' of 'digicanbus.J1939_PDU' objects>
 |      message payload
 |
 |  priority = <attribute 'priority' of 'digicanbus.J1939_PDU' objects>
```

Examples

```
            #
# Manipulating raw CAN messages
#
from digicanbus import CANHandle
DATA_FRAME=False
REMOTE_FRAME=True
h = CANHandle( 0 )
h.configure( bitrate=125000 )
h.send( 11, 0x500, DATA_FRAME, "%c%c%c" % (0x1e, 0x10, 0x10) )
h.send( 11, 0x500, DATA_FRAME, "%c%c%c" % (0x1e, 0x10, 0x00) )
h.send( 11, 0x280, REMOTE_FRAME, 8 )

def fn( width, id, remote, payload, ts, arg ):
  print "Width %d message received to id %x" % (width, id)

h.register_filter( 11, 0x500, 0x400, fn, 'filter 1' )
h.register_filter( 11, 0x500, 0x100, fn, 'filter 2' )
```

Application Developer Notes

Due to the "callback" architecture used in the digij1708 and digicanbus drivers, application developers should be aware of the effects of the code executed in the callback thread context. In particular, if the callback thread executes a significant amount of code or blocks on external resources, this can potentially lead to "falling behind" on bus data. To minimize the possibility of falling behind on data buffers, one should think about the following when writing the callback method which will be executed in the platform callback thread context:

- Is my callback performing significant processing? The longer your callback is executing, the more likely that data from the bus will back up and, eventually, be dropped. If there is significant processing to be done on the data, the callback should focus on putting the data somewhere where another Python thread can process the data.
- Does my callback block on external resources? Beyond pure CPU processing time, blocking calls (file, socket, logging, etc.) will prevent the callback thread from grabbing and processing the next data item. As a rule, potentially blocking calls should be avoided.

- Does any code in my callback release the GIL (Global Interpreter Lock)? For optimal throughput, any call which releases the GIL (Global Interpreter Lock) should be avoided. This includes locks, files, and sockets. Efficient, thread-safe alternatives to the Queue module can be constructed by using GIL-atomic (executed in a single bytecode) operations on data structures like Python's list or deque. More information on this technique can be here: http://effbot.org/pyfaq/what-kinds-of-global-value-mutation-are-thread-safe.htm.

Availability

Products which support this module

Only Digi Python enabled products with a CAN bus hardware interface support this module, including the ConnectPort X5 family of products.

Module: digicli

WARNING! Using this module limits your ability to port your Python code to other product families. To maintain portability, use the Module: rci instead.

The digicli module exports a single function, also named digicli. This function takes a single string as an argument. That string is a command to be executed by the command line interpreter of the Digi box. The status and result of the command will be returned in a tuple. The first element is a boolean that indicates whether the command succeeded. The second element of the tuple is a list of the return text from the command separated into lines. For all Command Line commands, refer to the "*Command Reference: Digi Connect Family*" located on this page: http://www.digi.com/support/productdetl.jsp?pid=3442&osvid=0&tp=3&tp2=0.

Note CLI commands capable of producing lines with greater than 255 characters can cause problems. Newer versions of the operating system attempt to address this danger by splitting lines into multiple strings in the output list. A string with no trailing '\n' character is continued in the next string of the list.

Example

This example will demonstrate extracting the MAC address from a device with one Ethernet or 802.11 interface.

```
import digicli

status, output = digicli.digicli('show net')

if status:
    for line in output:
        if line.find('MAC Address') >= 0:
            l = line.split(':')
            print "".join(l[1:]).strip()
```

Notes

The more complex show network output of devices with more than one Ethernet or 802.11 interfaces requires a different parsing method.

Availability

Products which support this module

- Most of the Digi Connect/ConnectPort products support this module, such as WAN, X2, X4 and X5.
- The ConnectPort LTS supports this module

Products which DO NOT support this module

- The Digi ConnectPort X3 (all variations) does not support this.
- Transport does not support this module

- Passport and Cm products do not support Python
- Future gateways will NOT support this module.

Module: digihw

This module is for interacting with special hardware within Digi some products. An exception will be thrown if you attempt access hardware which does not exist within your model.

LED control

These routines allow a user to control the available LED's. Refer to the Hardware Reference Manual supplied with the kit to determine which LED's are available.

```
user_led_set(value [, led=1]) return None
```

The user controlled LED is made to match the logic state of the "value" parameter. A true value turns on the LED, and a false value turns it off. The "led" parameter indicates which user LED to blink, a value of one indicates LED 1 (the default), a value of two indicates LED 2. The led parameter is optional on platforms with only one user LED

Example: Blink the LED's

```
import digihw

# turn off user LED's after the user presses the button

# Turn on user LED's
digihw.user_led_set(1, 1)

# Add a 1-second delay delay
time.sleep(1.0)

# Turn off the LED's
digihw.user_led_set(0, 1)
```

Note The Digi ConnectPort X4 only has 1 LED which is numbered one (1). So the digihw.user_led_set(1, 1) works, and digihw.user_led_set(1, 2) causes an error.

Request exclusive access by RCI

The operating code with products - such as the X4/X5 - do not make use of the Python accessible LED after the unit has power. However, the ConnectPort X2e by default might affects the two available LED at any time. Therefore you need to request exclusive access to the LED to avoid conflict.

Issue the following RCI command to request exclusive access to the LED. If such exclusive-access is not required you will receive an error such as "setting group unknown".

```
<rci_request version="1.1">
  <set_setting>
    <led_control>
      <network_connectivity>user</network_connectivity>
    </led_control>
  </set_setting>
</rci_request>"""
```

Give up exclusive access by RCI

When your program exits, it should restore LED control to the operating system using the following RCI command. Note: if you do not restore 'OS' control, the LED will remain in the last state set by your Python until a device reset or reboot occurs.

```
<rci_request version="1.1">
  <set_setting>
    <led_control>
      <network_connectivity>os</network_connectivity>
    </led_control>
  </set_setting>
</rci_request>"""
```

Availability - Python controllable LED

In general, most Digi Python-enabled products DO NOT support this module.

Digi Python-enabled products which do support this feature include:

Product	RCI Request for COntrol	LED #1	LED #2	Remarks
ConnectCore 3G 9P 9215	No	Yes	Yes	
ConnectPort X2e	Yes	Yellow	Green	Both LED are clustered under the 'World'/Network Status icon
ConnectPort 4/4H	No	Yellow	None	LED is labeled STATUS on the faceplate, requires firmware 82001536_K or newer
ConnectPort 5	No	Yellow	None	LED is labeled STATUS on the faceplate

General I/O pins

These routines allow access to GPIO pins from Python. For a description of the available GPIO pins for your device refer to the Hardware Reference manual. Only pins which are not in use for other functions can be used.

The GPIO API uses names which correspond to the pins on the module connector. For example X2_3 refers to the third pin on connector X2, this is shown on the schematics provided with the kit on the system connectors sheet.

Refer to the GPIO appendix in the hardware reference manual which is supplied with the kit.

```
gpio_get_value(gpio_number) return value
```

Read the specified GPIO pin. An exception is raised if the gpio is invalid. Return Value: Value of the GPIO pin

```
gpio_set_value(gpio_number, value) return None
```

Set the specified GPIO pin. Value is zero or non zero. An exception is raised if the gpio is invalid.

```
gpio_set_input(gpio_number) return None
```

Set the specified GPIO pin as an input. An exception is raised if the gpio is invalid

Examples

Wake up on a GPIO button press

```
            # Routine which waits for a user to press user button 2 on the
Connect Core 9P 9215
def wait_for_button ():
    # Wait for the user to press button 2
    digihw.gpio_set_input(digihw.X2_52)
    print "Press Button 2 to continue"
    while(True):
        val = digihw.gpio_get_value(digihw.X2_52)
        if val == 0:
            break
    while(True):
        val = digihw.gpio_get_value(digihw.X2_52)
        if val == 1:
            break
```

Set GPIO pins low

```
import digihw

digihw.gpio_set_value(digihw.X2_35, 0)
digihw.gpio_set_value(digihw.X2_48, 0)
digihw.gpio_set_value(digihw.X2_52, 0)
digihw.gpio_set_value(digihw.X2_50, 0)
digihw.gpio_set_value(digihw.X2_36, 0)
digihw.gpio_set_value(digihw.X2_37, 0)
digihw.gpio_set_value(digihw.X2_38, 0)
```

Read a range of GPIO pins

```
            # Instead of the symbolic name, you can specify an index into the
GPIO
# table (zero based index)
import digihw

for pinNumber in range (low_test_range, high_test_range):
    val = digihw.gpio_get_value(pinNumber)
    if  val < 0:
```

```
        print "The last pin is [%d]" % (pinNumber)
        break
    print "Value for pin [%d] is [%d]" % (pinNumber, val)

# Set all the pins high
for pinNumber in range (low_test_range, high_test_range):
    print "Setting pin [%d] to [%d]" % (pinNumber, 1)
    digihw.gpio_set_value(pinNumber, 1)
```

Availability - GPIO pins

In general, *most Digi Python-enabled products DO NOT support this module*.

Digi Python-enabled products which do support this feature include:

- Digi ConnectCore 3G 9P 9215
- Digi ConnectPort X5

GPS location

The GPS support in the ConnectPort X5 is managed by a self-contained, dedicated module, which delivers relatively standard NMEA strings to the embedded operating system. While it is possible for an application to retrieve and parse raw NMEA strings, if necessary, (a device named "/gps/0" exists on the system, and can be opened and read), this has two major disadvantages: (1) it adds unnecessary complexity to every application that wishes to do even the most common retrieval of information from the GPS, and (2) it is unnecessarily exclusive; only one thread, for all practical purposes, can have the special device open and be reading from it at one time.

To simplify the most common operation, the AccelePort X5 embedded operating system offers the "gps_location" command in its "digihw" embedded Python module. Any thread can call "gps_location" at any time and retrieve position information (deemed the most common extraction from an NMEA stream). The position information returned by "gps_location" is associated with a timestamp. This time is relative to the system's concept of time. This choice was made intentionally, as it seemed prudent to allow these samples to be compared with the results of a call to the "time" module's "time" function... the generally available time reference for Python scripts running in the system.

```
gps_location():
    return (latitude, longitude, altitude, timestamp)
```

Returns a 4-tuple combining floating-point latitude, longitude, and altitude, as well as a timestamp taken when the sample was first parsed.

Note For ConnectPort X3/X-trak3 devices additional information is available namely -- fixtype, speed and direction as shown below

```
gps_location():
    return (latitude, longitude, altitude, timestamp, speed, direction,fixtype)
```

Returns a 7-tuple combining Fix type (2D and 3D) floating-point latitude, longitude, altitude,UTC time stamp (long integer) taken when the sample was first parsed ,speed and direction.

The timestamp is in a form consistent with that returned by the time() function of the time module, and is not drawn directly from the NMEA stream, and is thus not supplied by the GPS satellites.

Availability - GPS location

In general, **most Digi Python-enabled products DO NOT support this module**.

Digi Python-enabled products which do support this feature include:

- Digi ConnectCore 3G 9P 9215
- Digi ConnectPort X4 and X4H (with external GPS hardware added)
- Digi ConnectPort X5

Module: digij1708

This module was designed in conjunction with the Digi ConnectPort X5, the first Digi Python-enabled product to offer J1708 bus connectivity. The module is designed to allow scripts to manipulate J1708 messages at a logical level, with the embedded operating system providing support for the encoding/decoding of these messages on the line.

Limited J1587 support is also provided, in order to simplify the interpretation of J1708 messages as J1587 PIDs. Knowledge of individual PID encodings is application specific, and must be supplied by the script.

Module help

Help on built-in module digij1708:

NAME

digij1708 - Provides Python abstraction for an underlying J1708 bus

FILE

(built-in)

DESCRIPTION

The J1708Handle class provides functions to initialize a J1708 bus and transfer raw J1708 messages. It also provides functions to communicate over the J1708 bus via the abstraction of the SAE J1587 protocol.

CLASSES

```
__builtin__.dict(__builtin__.object)
    J1587_PIDdict
__builtin__.object
    J1708Handle
exceptions.ValueError(exceptions.StandardError)
    J1587_PID_ListError
    J1587_PID_PayloadError
    J1587_PID_ValueError

class J1587_PID_ListError(exceptions.ValueError)
 |  Method resolution order:
 |      J1587_PID_ListError
 |      exceptions.ValueError
 |      exceptions.StandardError
 |      exceptions.Exception
 |
 |  Methods inherited from exceptions.Exception:
 |
 |  __getitem__(...)
 |
 |  __init__(...)
```

```
|
|  __str__(...)
```

```
class J1587_PID_PayloadError(exceptions.ValueError)
|  Method resolution order:
|      J1587_PID_PayloadError
|      exceptions.ValueError
|      exceptions.StandardError
|      exceptions.Exception
|
|  Methods inherited from exceptions.Exception:
|
|  __getitem__(...)
|
|  __init__(...)
||  __str__(...)
```

```
class J1587_PID_ValueError(exceptions.ValueError)
|  Method resolution order:
|      J1587_PID_ValueError
|      exceptions.ValueError
|      exceptions.StandardError
|      exceptions.Exception
|
|  Methods inherited from exceptions.Exception:
|
|  __getitem__(...)
|
|  __init__(...)
|
|  __str__(...)
```

```
class J1587_PIDdict(__builtin__.dict)
|  Provides a means to map J1708 payloads to/from a dictionary of J1587 PIDs
|
|  J1587_PIDdict objects are instantiated either with no
|  parameters (creating an empty PID dictionary to be filled),
|  or with a J1708 payload string (in which case the PIDs are
|  automatically parsed out of the string into the PID
|  dictionary):
|
|     piddict = J1587_PIDdict([payload])
|
|  The PID dictionary is indexed by PID number.  The value
|  associated with the PID is the corresponding payload
|  substring from the 1708 payload.
|
|  All standard dictionary operations are supported.
|
|  Method resolution order:
|      J1587_PIDdict
|      __builtin__.dict
|      __builtin__.object
|
|  Methods defined here:
|
|  J1708_payload(...)
|      J1708_payload([pidlist]) -> j1708_payload_string
```

```
 |
 |      Encode PIDs from the PID dictionary into a string.
 |
 |      Argument:
 |      pidlist -- A sequence of PID numbers to encode.
 |
 |                 If omitted, an attempt will be made to include

 |                 the entire dictionary of PIDs in the output
 |                 string, in sorted order of PID.
 |
 |                 If supplied, the PIDs in the sequence, in the
 |                 order of the sequence, will be included in the
 |                 output string.
 |
 |      Return Value:
 |      The resulting string is in a form suitable for transmission
 |      in a single J1708 frame.
 |
 |      Exceptions:
 |      J1587_PID_PayloadError -- invalid payload in PID dictionary
 |      J1587_PID_ValueError -- invalid PID number
 |      J1587_PID_ListError -- invalid PID selection list
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see x.__class__.__doc__ for signature
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from __builtin__.dict:
 |
 |  __cmp__(...)
 |      x.__cmp__(y) <==> cmp(x,y)
 |
 |  __contains__(...)
 |      D.__contains__(k) -> True if D has a key k, else False
 |
 |  __delitem__(...)
 |      x.__delitem__(y) <==> del x[y]
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
 |  __ge__(...)
 |      x.__ge__(y) <==> x>=y
 |
 |  __getattribute__(...)
 |      x.__getattribute__('name') <==> x.name
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(...)
 |      x.__gt__(y) <==> x>y
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __iter__(...)
 |      x.__iter__() <==> iter(x)
```

```
|
|  __le__(...)
|      x.__le__(y) <==> x<=y
|
|  __len__(...)
|      x.__len__() <==> len(x)
|
|  __lt__(...)
|      x.__lt__(y) <==> x<y
|
|  __ne__(...)
|      x.__ne__(y) <==> x!=y
|
|  __repr__(...)
|      x.__repr__() <==> repr(x)
|
|  __setitem__(...)
|      x.__setitem__(i, y) <==> x[i]=y
|
|  clear(...)
|      D.clear() -> None.  Remove all items from D.
|
|  copy(...)
|      D.copy() -> a shallow copy of D
|
|  get(...)
|      D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
|
|  has_key(...)
|      D.has_key(k) -> True if D has a key k, else False
items(...)
|      D.items() -> list of D's (key, value) pairs, as 2-tuples
|
|  iteritems(...)
|      D.iteritems() -> an iterator over the (key, value) items of D
|
|  iterkeys(...)
|      D.iterkeys() -> an iterator over the keys of D
|
|  itervalues(...)
|      D.itervalues() -> an iterator over the values of D
|
|  keys(...)
|      D.keys() -> list of D's keys
|
|  pop(...)
|      D.pop(k[,d]) -> v, remove specified key and return the corresponding value
|      If key is not found, d is returned if given, otherwise KeyError is raised
|
|  popitem(...)
|      D.popitem() -> (k, v), remove and return some (key, value) pair as a
|      2-tuple; but raise KeyError if D is empty
|
|  setdefault(...)
|      D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D
|
|  update(...)
|      D.update(E, **F) -> None.  Update D from E and F: for k in E: D[k] = E[k]
|      (if E has keys else: for (k, v) in E: D[k] = v) then: for k in F: D[k] = F
```

```
[k]
|
|  values(...)
|      D.values() -> list of D's values|
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from __builtin__.dict:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  fromkeys = <built-in method fromkeys of type object>
|      dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v.
|      v defaults to None.
```

```
class J1708Handle(__builtin__.object)
|  Provides a programming interface to an underlying J1708 controller
|
|  Instantiating a J1708Handle associates a set of methods with an
|  internal J1708 bus instance.  The methods allow one to request
|  configuration of the bus, register message receive callbacks, and
|  submit messages for transmission.
|
|  J1708Handle objects are instantiated with a single "j1708_bus_id"
|  argument, which is an integer identifier for the desired J1708 bus.
|
|  Methods defined here:
|
|  __init__(...)
|      x.__init__(...) initializes x; see x.__class__.__doc__ for signature
|
|  configure(...)
|      configure([options]) -> None
|
|      The J1708 bus will be stopped if running, the bus will be
|      reconfigured, then restarted with the new configuration.
|
|      With no arguments, the current configuration will be re-applied.
|
|      Options Arguments:
|
|      mid=n -- SAE J1708 message identification character.  No two
|               devices on the bus should share a message identifier,
|               per specification.  'n' must be between 0 and 255.  If
|               the mid option is left unconfigured, the ability to
|               send is disabled.
|
|  register_callback(...)
|      register_callback(method, return_arg) -> None
|
|      Associate a callback with a reception method function and a return
|      argument.  Only a single copy of each unique set is stored
|      internally.  The registration is with the underlying bus
|      instance, not with the handle.
|
|      The method must be a callable object of the form:
|
|      method(mid, payload, return_arg)
```

```
 |
 |      The "return_arg" argument is not examined by the bus
 |      infrastructure.
 |
 |      An exception is raised if the number of callbacks is exceeded.
 |
 |      NOTE: The callback method will be executed in another thread.
 |      Shared data will need to be protected from race conditions due
 |      to concurrent access.
 | send(...)
 |      send(priority, payload) -> True | False
 |
 |      Non-blocking transmit of a message on the J1708 bus.  The configured
 |      message identifier will be used in the transmission.  If the message
 |      identifier is not selected, the send will fail with an exception.
 |
 |      Arguments:
 |      priority -- 1-8, as defined by the SAE J1708 specification
 |      payload -- a string, and the length should be 19 characters or less
 |
 |      Return Value:
 |      The function will return a true value if the message was
 |      successfully queued, and false if there is no queue space.
 |
 |  unregister_callback(...)
 |      unregister_callback(method, return_arg) -> None
 |
 |      The callback list of the bus is searched for a match of the same
 |      calling parameters.  If a match is found, that registration is
 |      removed from the list.  If no match is found, an exception is
 |      raised.
 |
 |      Registered callbacks will automatically be unregistered when the
 |      handle goes out of scope, or is otherwise deleted.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

Examples

```
from digicanbus import J1708Handle
h = J1708Handle( 0 )
h.configure( mid=117 )
my_priority = 8
h.send( my_priority, "%c%c%c" % (0x1e, 0x10, 0x10) )
h.send( my_priority, "%c%c%c" % (0x1e, 0x10, 0x00) )
def fn( mid, payload, arg ):
  print "Message of length %d received from module ID %d" % (len(payload), mid)

h.register_callback( fn, 'callback 1' )
```

Application Developer Notes

Due to the "callback" architecture used in the digij1708 and digicanbus drivers, application developers should be aware of the effects of the code executed in the callback thread context. In particular, if the callback thread executes a significant amount of code or blocks on external resources, this can potentially lead to "falling behind" on bus data. To minimize the possibility of falling behind on data buffers, one should think about the following when writing the callback method which will be executed in the platform callback thread context:

- **Is my callback performing significant processing?** The longer your callback is executing, the more likely that data from the bus will back up and, eventually, be dropped. If there is significant processing to be done on the data, the callback should focus on putting the data somewhere where another Python thread can process the data.
- **Does my callback block on external resources?** Beyond pure CPU processing time, blocking calls (file, socket, logging, etc.) will prevent the callback thread from grabbing and processing the next data item. As a rule, potentially blocking calls should be avoided.
- **Does any code in my callback release the GIL (Global Interpreter Lock)?** For optimal throughput, any call which releases the GIL (Global Interpreter Lock) should be avoided. This includes locks, files, and sockets. Efficient, thread-safe alternatives to the Queue module can be constructed by using GIL-atomic (executed in a single bytecode) operations on data structures like Python's list or deque. More information on this technique can be here: http://effbot.org/pyfaq/what-kinds-of-global-value-mutation-are-thread-safe.htm.

Availability

Products which support this module

Only Digi Python enabled products with a J1708 bus hardware interface support this module, including the ConnectPort X5 family of products.

Module: digiorbcomm

digiorbcomm.py

This module was designed in conjunction with the Digi ConnectPort X5 Fleet, the first Digi Python-enabled product to offer ORBCOMM satellite network connectivity. The module is designed to allow scripts to manipulate messages from the ORBCOMM serial interface API at a logical level, with the embedded operating system providing support for the encoding/decoding of messages on the line, and servicing the link layer of the ORBCOMM serial interface protocol.

Knowledge of the ORBCOMM serial interface API is required to take full advantage of this module.

Module help

Help on built-in module digiorbcomm:

NAME

digiorbcomm

FILE

(built-in)

DESCRIPTION

Provides an interface to allow Python scripts to interact with an ORBCOMM satellite modem module in the system.

CLASSES

```
__builtin__.object
    CommCmd
    ConfigCmd
    PosDeterCmd
    PosStatus
    ReceiveCallback
    SCOrigDefMsg
    SCOrigDefRep
    SCOrigGlobGram
    SCOrigMsg
    SCOrigPosRep
    SCOrigRep
    SCTermGlobGram
    SCTermMsg
    SCTermUserCmd
    SatOrbElem
    SatPlaneOrbElem
    SatStateVec
    Status
    SysAnnounce
    SysResp
```

```
class CommCmd(__builtin__.object)
 |  ORBCOMM Serial Interface Communications Command
 |
 |  Methods defined here:
 |
 |  done(...)
```

```
 |      done() -> None
```

```
 |  submit(...)
```

```
 |      submit() -> None

 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.CommCmd' objects>
 |      destination ORBCOMM Gateway
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.CommCmd' objects>
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.CommCmd' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  type_code = <attribute 'type_code' of 'digiorbcomm.CommCmd' objects>
 |      type of action requested
 |
 |  value = <attribute 'value' of 'digiorbcomm.CommCmd' objects>
 |      generic value pertinent to type of action requested
```

```
class PosStatus(__builtin__.object)
 |  ORBCOMM Serial Interface Position Status Message
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  lat_code = <attribute 'lat_code' of 'digiorbcomm.PosStatus' objects>
 |      coded geodetic latitude
 |
 |  lon_code = <attribute 'lon_code' of 'digiorbcomm.PosStatus' objects>
 |      coded geodetic longitude
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.PosStatus' object...
```

```
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.PosStatus' objects>
|      ORBCOMM Serial Interface message type
|
|  pos_calc_active = <attribute 'pos_calc_active' of 'digiorbcomm.PosStat...
|      state of position determination process
```

```
class ReceiveCallback(__builtin__.object)
 |  Register a callback for ORBCOMM message reception.
 |
 |  digiorbcomm.ReceiveCallback(cb, arg) -> handle
 |
 |  Returns a handle to be used for deregistration later.  The
 |  callback will be handled as long as the callback handle
 |  exists.  If the handle goes out of scope or is deallocated,
 |  the callback will be unregistered.
 |
 |  The "arg" parameter is arbitrary, and will be stored to be
 |  passed to the callback function with each message.
 |
 |  Callback functions should be of the form:
 |      cb(msg, arg) -> None
 |
 |  CB Param:
 |      msg: An ORBCOMM message class instance
 |      arg: Callback argument supplied during registration
 |
 |  Remember, call back functions will be executed in another
 |  thread.  Shared data will need to be protected from race
 |  conditions due to concurrent access.
 |
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

```
class SCOrigDefMsg(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Originated Default Message
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigDefMsg' o...
 |      DTE assigned, used to identify among multiple messages
 |
 |  msg_body = <attribute 'msg_body' of 'digiorbcomm.SCOrigDefMsg' objects...
```

```
|      message body
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigDefMsg' obj...
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigDefMsg' objects...
|      ORBCOMM Serial Interface message type
```

```
class SCOrigDefRep(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Originated Default Report
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigDefRep' o...
 |      DTE assigned, used to identify among multiple messages
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigDefRep' obj...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigDefRep' objects...
 |      ORBCOMM Serial Interface message type
 |
 |  user_data = <attribute 'user_data' of 'digiorbcomm.SCOrigDefRep' objec...
 |      up to six bytes of user data
```

```
class SCOrigGlobGram(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Originated Globalgram
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCOrigGlobGram' objects>
 |      destination ORBCOMM Gateway ID
 |
 |  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigGlobGram'...
```

```
|      DTE assigned, used to identify among multiple messages
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigGlobGram' o...
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigGlobGram' objec...
|      ORBCOMM Serial Interface message type
|
|  or_ind = <attribute 'or_ind' of 'digiorbcomm.SCOrigGlobGram' objects>
|      originator/recipient indicator
|
|  user_data = <attribute 'user_data' of 'digiorbcomm.SCOrigGlobGram' obj...
|      up to 229 bytes of user data
```

```
class SCOrigMsg(__builtin__.object)
|  ORBCOMM Serial Interface SC-Originated Message
|
|  Methods defined here:
|
|  done(...)
|      done() -> None
|
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  ack_level = <attribute 'ack_level' of 'digiorbcomm.SCOrigMsg' objects>
|      message acknowledgement level
|
|  cc_rcpnt_addr = <attribute 'cc_rcpnt_addr' of 'digiorbcomm.SCOrigMsg' ...
|      Sequence of strings or integers, representing copied recipients of the
message
|
|  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCOrigMsg' objects>
|      destination ORBCOMM Gateway ID
|
|  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigMsg' obje...
|      DTE assigned, used to identify among multiple messages
|
|  msg_body = <attribute 'msg_body' of 'digiorbcomm.SCOrigMsg' objects>
|      message body
|
|  msg_body_type = <attribute 'msg_body_type' of 'digiorbcomm.SCOrigMsg' ...
|      message body type
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigMsg' object...
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigMsg' objects>
|      ORBCOMM Serial Interface message type
|
|  polled = <attribute 'polled' of 'digiorbcomm.SCOrigMsg' objects>
|      polled by Gateway or initiated by SC
```

```
|
|  priority = <attribute 'priority' of 'digiorbcomm.SCOrigMsg' objects>
|      message priority level
|
|  rcpnt_addr = <attribute 'rcpnt_addr' of 'digiorbcomm.SCOrigMsg' object...
|      Sequence of strings or integers, representing direct recipients of the
message
|
|  subj = <attribute 'subj' of 'digiorbcomm.SCOrigMsg' objects>
|      message subject, or None to indicate no subject
```

```
class SCOrigPosRep(__builtin__.object)
|  ORBCOMM Serial Interface SC-Originated Position Report
|
|  Methods defined here:
|
|  done(...)
|      done() -> None
|
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  lat_code = <attribute 'lat_code' of 'digiorbcomm.SCOrigPosRep' objects...
|      coded geodetic latitude
|
|  lon_code = <attribute 'lon_code' of 'digiorbcomm.SCOrigPosRep' objects...
|      coded geodetic longitude
|
|  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigPosRep' o...
|      DTE assigned, used to identify among multiple messages
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigPosRep' obj...
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigPosRep' objects...
|      ORBCOMM Serial Interface message type
```

```
class SCOrigRep(__builtin__.object)
|  ORBCOMM Serial Interface SC-Originated Report
|
|  Methods defined here:
|
|  done(...)
|      done() -> None
|
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
```

```
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCOrigRep' objects>
 |      destination ORBCOMM Gateway ID
 |
 |  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SCOrigRep' obje...
 |      DTE assigned, used to identify among multiple messages
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCOrigRep' object...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCOrigRep' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  or_ind = <attribute 'or_ind' of 'digiorbcomm.SCOrigRep' objects>
 |      originator/recipient indicator, only values 0-3
 |
 |  polled = <attribute 'polled' of 'digiorbcomm.SCOrigRep' objects>
 |      polled by Gateway or initiated by SC
 |
 |  serv_type = <attribute 'serv_type' of 'digiorbcomm.SCOrigRep' objects>
 |      service type
 |
 |  user_data = <attribute 'user_data' of 'digiorbcomm.SCOrigRep' objects>
 |      up to six bytes of user data
```

```
class SCTermGlobGram(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Terminated Globalgram
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_dgram_ref_num = <attribute 'gwy_dgram_ref_num' of 'digiorbcomm.SCT...
 |      used to identify among multiple Globalgrams
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCTermGlobGram' objects>
 |      originating ORBCOMM Gateway ID
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCTermGlobGram' o...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCTermGlobGram' objec...
 |      ORBCOMM Serial Interface message type
 |
 |  or_ind = <attribute 'or_ind' of 'digiorbcomm.SCTermGlobGram' objects>
 |      originator/recipient indicator
 |
```

```
|  user_data = <attribute 'user_data' of 'digiorbcomm.SCTermGlobGram' obj...
|      user data
```

```
class SCTermMsg(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Terminated Message
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCTermMsg' objects>
 |      originating ORBCOMM Gateway ID
 |
 |  msg_body = <attribute 'msg_body' of 'digiorbcomm.SCTermMsg' objects>
 |      message body string
 |
 |  msg_body_type = <attribute 'msg_body_type' of 'digiorbcomm.SCTermMsg' ...
 |      message body type
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCTermMsg' object...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCTermMsg' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  or_addr = <attribute 'or_addr' of 'digiorbcomm.SCTermMsg' objects>
 |      originator indicator or address, or None
 |
 |  rcpnt_addr = <attribute 'rcpnt_addr' of 'digiorbcomm.SCTermMsg' object...
 |      list of copied recipients, or None
 |
 |  subj = <attribute 'subj' of 'digiorbcomm.SCTermMsg' objects>
 |      message subject string, or None
```

```
class SCTermUserCmd(__builtin__.object)
 |  ORBCOMM Serial Interface SC-Terminated User Command
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
```

```
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SCTermUserCmd' objects>
 |      originating ORBCOMM Gateway ID
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SCTermUserCmd' ob...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SCTermUserCmd' object...
 |      ORBCOMM Serial Interface message type
 |
 |  user_data = <attribute 'user_data' of 'digiorbcomm.SCTermUserCmd' obje...
 |      user data
```

```
class SatOrbElem(__builtin__.object)
 |  ORBCOMM Serial Interface Satellite Orbital Elements
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  mean_anom = <attribute 'mean_anom' of 'digiorbcomm.SatOrbElem' objects...
 |      Mean Anomaly for this satellite
 |
 |  mean_motion = <attribute 'mean_motion' of 'digiorbcomm.SatOrbElem' obj...
 |      ECEF position coordinates, 'y' component
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SatOrbElem' objec...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SatOrbElem' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  plane_id = <attribute 'plane_id' of 'digiorbcomm.SatOrbElem' objects>
 |      Identifies the satellite plane
 |
 |  sat_id = <attribute 'sat_id' of 'digiorbcomm.SatOrbElem' objects>
 |      ID of satellite to which this data pertains
```

```
class SatPlaneOrbElem(__builtin__.object)
 |  ORBCOMM Serial Interface Satellite Plane Orbital Elements
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
```

```
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  decay = <attribute 'decay' of 'digiorbcomm.SatPlaneOrbElem' objects>
|      decay in orbit
|
|  eccent = <attribute 'eccent' of 'digiorbcomm.SatPlaneOrbElem' objects>
|      eccentricity of orbit
|
|  epoch = <attribute 'epoch' of 'digiorbcomm.SatPlaneOrbElem' objects>
|      epoch time (milliseconds since 1/1/93)
|
|  inclin = <attribute 'inclin' of 'digiorbcomm.SatPlaneOrbElem' objects>
|      inclination
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SatPlaneOrbElem' ...
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SatPlaneOrbElem' obje...
|      ORBCOMM Serial Interface message type
|
|  orbit_num = <attribute 'orbit_num' of 'digiorbcomm.SatPlaneOrbElem' ob...
|      number of complete orbits since epoch
|
|  perigee = <attribute 'perigee' of 'digiorbcomm.SatPlaneOrbElem' object...
|      argument of perigee
|
|  plane_id = <attribute 'plane_id' of 'digiorbcomm.SatPlaneOrbElem' obje...
|      Identifies the satellite plane
|
|  raan = <attribute 'raan' of 'digiorbcomm.SatPlaneOrbElem' objects>
|      right ascension of ascending node
|
|  sat_quan = <attribute 'sat_quan' of 'digiorbcomm.SatPlaneOrbElem' obje...
|      total number of satellites in the system
```

```
class SatStateVec(__builtin__.object)
|  ORBCOMM Serial Interface Satellite State Vector
|
|  Methods defined here:
|
|  done(...)
|      done() -> None
|
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

```
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SatStateVec' obje...
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SatStateVec' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  sat_id = <attribute 'sat_id' of 'digiorbcomm.SatStateVec' objects>
 |      ID of satellite to which this data pertains
 |
 |  x = <attribute 'x' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF position coordinates, 'x' component
 |
 |  x_dot = <attribute 'x_dot' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF velocity, 'x' component
 |
 |  y = <attribute 'y' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF position coordinates, 'y' component
 |
 |  y_dot = <attribute 'y_dot' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF velocity, 'y' component
 |
 |  z = <attribute 'z' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF position coordinates, 'z' component
 |
 |  z_dot = <attribute 'z_dot' of 'digiorbcomm.SatStateVec' objects>
 |      ECEF velocity, 'z' component
```

```
class Status(__builtin__.object)
 |  ORBCOMM Serial Interface Status Message
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  active_mha_msg_ref = <attribute 'active_mha_msg_ref' of 'digiorbcomm.S...
 |      MHA # of message being transferred (0xFF = no msg)
 |
 |  check_errs = <attribute 'check_errs' of 'digiorbcomm.Status' objects>
 |      number of downlink packets with bad checksum since last status packet
 |
 |  gwy_list = <attribute 'gwy_list' of 'digiorbcomm.Status' objects>
 |      list of tuples of ORBCOMM Gateway's and their minimum acceptable message
priority
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.Status' objects>
 |      ORBCOMM message status
 |
```

```
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.Status' objects>
|      ORBCOMM Serial Interface message type
|
|  queued_ib_msgs = <attribute 'queued_ib_msgs' of 'digiorbcomm.Status' o...
|      number of SC-Originated messages in SC memory
|
|  queued_ob_msgs = <attribute 'queued_ob_msgs' of 'digiorbcomm.Status' o...
|      number of SC-Terminated messages in SC memory
|
|  sat_in_view = <attribute 'sat_in_view' of 'digiorbcomm.Status' objects...
|      current satellite number, 0 if no satellite in view
|
|  sc_diag_code = <attribute 'sc_diag_code' of 'digiorbcomm.Status' objec...
|      diagnostic code from self test
|
|  sc_state = <attribute 'sc_state' of 'digiorbcomm.Status' objects>
|      state of SC, message transport processes
|
|  stored_sats = <attribute 'stored_sats' of 'digiorbcomm.Status' objects...
|      number of stored satellite orbital elements
|
|  time = <attribute 'time' of 'digiorbcomm.Status' objects>
|      24 bit integer representing number of seconds since 00:00:00 UTC, resets
midnight Sunday
|
|  total_sats = <attribute 'total_sats' of 'digiorbcomm.Status' objects>
|      total number of satellites in system
|
|  week = <attribute 'week' of 'digiorbcomm.Status' objects>
|      UTC time week, starting January 6, 1980
```

```
class SysAnnounce(__builtin__.object)
|  ORBCOMM Serial Interface System Announcement
|
|  Methods defined here:
|
|  done(...)
|      done() -> None
|
|  submit(...)
|      submit() -> None
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
|
|  announce_code = <attribute 'announce_code' of 'digiorbcomm.SysAnnounce...
|      system provided information
|
|  dest_or_ind = <attribute 'dest_or_ind' of 'digiorbcomm.SysAnnounce' ob...
|      destination OR indicator
|
|  gwy_id = <attribute 'gwy_id' of 'digiorbcomm.SysAnnounce' objects>
|      originating ORBCOMM Gateway
|
|  msg_status = <attribute 'msg_status' of 'digiorbcomm.SysAnnounce' obje...
```

```
|      ORBCOMM message status
|
|  msg_type = <attribute 'msg_type' of 'digiorbcomm.SysAnnounce' objects>
|      ORBCOMM Serial Interface message type
```

```
class SysResp(__builtin__.object)
 |  ORBCOMM Serial Interface System Response
 |
 |  Methods defined here:
 |
 |  done(...)
 |      done() -> None
 |
 |  submit(...)
 |      submit() -> None
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
 |
 |  ack_mask = <attribute 'ack_mask' of 'digiorbcomm.SysResp' objects>
 |      which recipients apply to this ACK
 |
 |  diag_code = <attribute 'diag_code' of 'digiorbcomm.SysResp' objects>
 |      diagnostic code
 |
 |  gwy_ref_num = <attribute 'gwy_ref_num' of 'digiorbcomm.SysResp' object...
 |      ORBCOMM Gateway assigned, used for delivery confirmation
 |
 |  mha_ref_num = <attribute 'mha_ref_num' of 'digiorbcomm.SysResp' object...
 |      DTE assigned, used to uniquely identify messages
 |
 |  msg_status = <attribute 'msg_status' of 'digiorbcomm.SysResp' objects>
 |      ORBCOMM message status
 |
 |  msg_type = <attribute 'msg_type' of 'digiorbcomm.SysResp' objects>
 |      ORBCOMM Serial Interface message type
 |
 |  origin = <attribute 'origin' of 'digiorbcomm.SysResp' objects>
 |      type of originator of this acknowledgment
 |
 |  origin_id = <attribute 'origin_id' of 'digiorbcomm.SysResp' objects>
 |      ID of originator, either sat_id or gwy_id
 |
 |  status = <attribute 'status' of 'digiorbcomm.SysResp' objects>
 |      status of message transfer or message enquiry
```

Functions

```
param_request(...)
    param_request(param [, data]) -> result
```

Execute a request for a parameter "get", or a parameter "set" if the optional "data" parameter is supplied.

The struct module is recommended for composing the "data" parameter and parsing the "result" string.

Arguments

param -- integer parameter number, corresponding to those found in the ORBCOMM Serial Interface Specification.

data -- a string to be supplied as the value for a parameter set operation.

Return value

The result is always the responding value string from the ORBCOMM modem for the requested parameter, or an exception is thrown.

Availability

Products which support this module

ConnectPort X5 Fleet with optional ORBCOMM satellite modem

Module: digipowercontrol

Low power mode functionality is provided by the digipowercontrol which is supported on Digi products with the special power control hardware. An exception will be thrown if you attempt to use these routines on a device which does not support power control.

The power control device is not responsible for determining when a device or its peripherals should be powered off. This intelligence is left to the applications running in the device. The following operations are supplied via the embedded Python "digipowercontrol" module so that applications can:

- Turn off the system power
- Turn off the system power until a future time (scheduled wakeup)
- Independently control the power of the cellular modem (default state is on when the device is first connected to power)
- Independently control the power of an Orbcomm satellite modem, if present (default state is off whenever the processor is powered on)
- Determine the current state of the power sources managed by the power control module
- Get a hint, when possible, as to the reason the system was powered on

Descriptions of supported methods and types follow.

Methods

cellular_power_get()

Purpose

Get the state of power to the cellular module.

Syntax

```
cellular_power_get() . state
```

Description

Get the state of power to the cellular module.

Return

Return a true value if enabled, a false value if disabled.

cellular_power_set()

Purpose

Control the state of power to the cellular module.

Syntax

```
cellular_power_set(state) . None
```

Description

Control the state of power to the cellular module. A state argument with a true value enables power, and a false value disables it (low power mode).

Return

None

orbcomm_power_get()

Purpose

Get the state of power to the orbcomm satellite module.

Syntax

```
orbcomm_power_get() . state
```

Description

Get the state of power to the orbcomm satellite module.

Return

Return a true value if enabled, a false value if disabled.

orbcomm_power_set()

Purpose

Control the state of power to the orbcomm satellite module.

Syntax

```
orbcomm_power_set(state) . None
```

Description

Control the state of power to the orbcomm satellite module. A state argument with a true value enables power, and a false value disables it (low power mode).

Return

None

system_power_get()

Purpose

Get the state of system power.

Syntax

```
system_power_get() . state
```

Description

Get the state of system power. This function exists for future flexibility and completeness, since when system power is off, the value False can never be returned.

Return

Return a true value if enabled, a false value if disabled.

system_power_set()

Purpose

Control the state of system power.

Syntax

```
system_power_set(state[, duration]) . None
```

Description

Control the state of system power. A state argument with a true value enables power, and a false value disables it (low power mode). If the power is being changed to the disabled state, the optional duration argument can be supplied to indicate how many seconds (at most) the device should remain powered off. NOTE: This function may return control to the Python interpreter for a period of time before the main processor fully shuts down.

Return

None

wake_on_external_irq_set()

Purpose

Enable the power control module to wake up and reset on an external IRQ.

Syntax

wake_on_external_irq_set(gpio_pin, trigger_type) . None

Description

Enable the power control module to wake up and reset on an external IRQ, call system_power_set() to put the device to sleep after calling this function.

The first argument indicates which GPIO pin is used.

The second argument indicates the trigger type, which can be:

WAKE_ON_IRQ_RISING_EDGE -- trigger on rising edge

WAKE_ON_IRQ_FALLING_EDGE -- trigger on falling edge

WAKE_ON_IRQ_LOGIC_ONE -- trigger on logic one

WAKE_ON_IRQ_LOGIC_ZERO -- trigger on logic zero

Return

None.

wake_reason()

Purpose

Enable the power control module to wake up and reset on an external IRQ.

Syntax

```
wake_reason() . reasons | None
```

Description

Returns a Python "set" object populated with one or more values indicating why the power controller enabled processor power.

A return of None indicates that no reason is available. This might occur on a device initiated reboot, since the power controller did not cycle the power to initiate processor operation.

Return

- None if device is booting due to normal power up.
- WAKE_REASON_ACCEL = 2
- WAKE_REASON_IGNITION = 0
- WAKE_REASON_POWERLOSS = 7
- WAKE_REASON_RTC = 1

Note that not all products support all reasons.

Power control examples

Wake up and reset on an external IRQ trigger

```
            #
# Cause the device to enter low power sleep mode and
# reset when the wakeup button (GPIO 4) is pressed
# on the CC9P3G 9215 developement board
#

import digipowercontrol
import sys

# The GPIO described in the Hardware Reference Manual for your board
# In this case GPIO4 is connected to the wake up button on the board.
wakeup_button_gpio = 0x04

# Wake up when the button is pressed, for the this board
# the GPIO4 is pulled low when the button is pressed, so we
# wakeup on the falling edge.
digipowercontrol.wake_on_external_irq_set(wakeup_button_gpio ,
                                         digipowercontrol.WAKE_ON_IRQ_FALLING_
EDGE)

# Go to sleep, reset when a trigger occurs, this routine does not return
digipowercontrol.system_power_set(0)
```

Wake up and reset after a given duration

```
            #
# Cause the device to enter low power sleep mode and
# wake up after the number of seconds entered
```

```
#

import digipowercontrol
import sys
duration = 0.0
print("Enter the duration in seconds:")
duration = float(raw_input('-->'))

# Go to sleep, wake-up and reset after the number of seconds entered
# This function will not return
digipowercontrol.system_power_set(0, duration)
```

Availability

Products which support this module

Digi Python-enabled products which do support this feature include:

Function	ConnectPort X3	ConnectPort X5	ConnectCore 3G 9P
cellular_power_get()	No	Yes	No
cellular_power_set()	No	Yes	No
orbcomm_power_get()	No	Optional HW	No
orbcomm_power_set()	No	Optional HW	No
system_power_get()	Yes	Yes	Yes
wake_on_external_irq_set()	No	No	Yes
wake_reason()	Yes	Yes	No

Products which DO NOT support this module

In general, **most Digi Python-enabled products DO NOT support this module because very special hardware design is required**.

Module: digisms

Provides an interface to allow the Digi Cellular SMS handling functionality to pass SMS messages to Python applications for processing. By default SMS is disabled, so you might use the web interface (or other methods) to enable support for SMS. Your cellular data plan must support SMS - by default most data plans DO NOT.

Receiving SMS Messages

The module contains a single type named Callback that is initialized with a callback function as follows:

```
digisms.Callback(cb) -> handle
```

Returns a handle to be used for deregistration later. The callback will be handled as long as the callback handle exists. If the handle goes out of scope or is deallocated, the callback will be unregistered.

Callback functions should be of the form:

```
cb(SMSMessage)
```

An 'SMSMessage' object contains these members:

message

Contents of the message

source_addr

Source address that sent the message

timestamp

Service Center assigned timestamp

The function's return value is ignored.

Transmitting SMS messages

The 'digisms' module provides a 'digisms.send' function to transmit an SMS message over the cellular network.

The send routine is of the form: > send(destination, message)

destination

A string that contains the phone number to direct the message to.

message

The message to send. Exceeding the maximum character length of a single SMS message will cause the gateway to transmit multiple messages.

Limitations & Suggestions

Remember, call back functions will be executed in another thread. Shared data will need to be protected from race conditions due to concurrent access.

Availability

Products which support this module

This feature is available as of firmware revision 2.9.0 for cellular gateways with GSM modules. (See What is your product firmware level.)

Products which DO NOT support this module

This feature is not currently supported on:

- Digi ConnectPort X3

Module: digiwdog

The systems hardware watchdog is exposed in the *digiwdog* built-in module. This module contains a single class named Watchdog, each instance of which represents a task or event in the system that promises to stroke the watchdog at least as often as a given rate, failure of which to do so will result in the system rebooting.

The *Watchdog* class constructor takes two arguments. The first argument is the promised stroke rate. This is given in seconds and must be no less than ten seconds. A second optional argument allows a descriptive name to be attached to the object for identification of watchdog objects in the system and possible future use in reporting watchdog activity.

The resulting *Watchdog* object from a call to the constructor contains a single method named *stroke()*. *stroke()* takes no arguments and must be called at least as often as the promised rate in order to keep the system up.

When a Watchdog object goes out of scope, it will automatically unregister itself, so that allowing an object to go out of scope and not be stroked is not a failure that will cause the system to reboot. objects can be explicitly removed with the del keyword in Python to explicitly lose the references and unregister.

Example for X4 (NDS platforms)

The following example strokes the watchdog for awhile, but then forces it to reset when it decides to stop.

```
import digiwdog
import time

wd = digiwdog.Watchdog(10, "ForceReset")

for i in range(20):
    print i
    time.sleep(1)
    if i % 10 == 0:
        wd.stroke()

while True:
    print "woof"
    time.sleep(1)
```

Example for X2e (linux platforms)

Unfortunately, the watchdog function on the linux platforms is similar, but different. Also, any space in the name (like "Force Reset" instead of "forcerest") breaks the WatchDog functionality.

```
import watchdog
import time

# notice the 2 parameters are swapped from the X4/NDS example
wd = watchdog.Watchdog("ForceReset", 10)

for i in range(20):
    print i
    time.sleep(1)
```

```
    if i % 10 == 0:
        # notice the 'stroke' function is named 'heartbeat'
        wd.heartbeat()

while True:
    print "woof"
    time.sleep(1)
```

Example for all platforms

```
import sys
import time

if sys.platform == 'linux2':
    import watchdog
else:
    import digiwdog

if sys.platform == 'linux2':
    wd = watchdog.Watchdog("ForceReset", 10)
else:
    wd = digiwdog.Watchdog(10, "ForceReset")

for i in range(20):
    print i
    time.sleep(1)
    if i % 10 == 0:
        if sys.platform == 'linux2':
            wd.heartbeat()
        else:
            wd.stroke()

while True:
    print "woof"
    time.sleep(1)
```

Availability

Products which support this module

This feature is available as of firmware revision 2.6.2 on Digi products with hardware RealTime clocks.

Products which DO NOT support this module

This feature is not currently supported on:

- Digi ConnectPort X2
- Digi ConnectPort WiX2
- Digi ConnectPort X3

Module: digiweb

Provides an interface to allow the Digi web server to call into Python and allow Python code to handle web page requests.

The module contains a single type named Callback that is initialized with a callback function as follows:

Summary

digiweb.Callback(cb) -> handle

Returns a handle to be used for deregistration later. The callback will be handled as long as the callback handle exists. If the handle goes out of scope or is deallocated, the callback will be unregistered.

Callback functions should be of the form: > cb(type, path, headers, args) -> None | (type, data)

parameters

type

HTTP request type. Can be 'HEAD', 'GET' or 'POST'.

path

URL path component.

headers

A dictionary containing header information from the client. Contains host, agent and referer strings

args

Request Type specific

> **GET**
>
> URL-decoded form arguments in a dictonary
>
> **POST**
>
> Body of POST request in a single string

For either request type, If there is nothing provided in the request, None will be specified instead.

return values

- **None** if this path or request type cannot be handled by the callback routine.
- 2-tuple
  - type: For a successful request, determines the MIME type the server will report. Choose from TextHtml, TextPlain or TextXml.
  - data: Body of response content.

Limitations & suggestions

- Remember, call back functions will be executed in another thread. Shared data will need to be protected from race conditions due to concurrent access.
- Do not try to use common page names like "index.htm", for while they may NOT exist within the Digi product, the Digi web server (as do all web servers) treat certain default pages as magic.

- If you are creating a page name for users to remember, you may not want to require any extension at all, plus make them case insensitive. Telling users to use "192.168.1.1/status" is far easier than remembering if it is "status.htm" or "status.html" or "Status.HTML".
- To add an image to your page, use this example HTML tag . The images must be uploaded to your Python directory by the same web ui page as for the Python code. Be very careful of the case, as 'Image.jpg' is not the same as 'image.jpg' or 'Image.JPG'. Also, your flash space is highly limited so keep image small and highly compressed. JPG at lower than normal compression seem the best. BMP are too large. PNG may be 'functionally too rich', so larger than required. GIF also tend to be larger than JPG unless you only have a few colors (line drawings etc).

Examples

The following code will serve any unrecognized path in the system with information on the request for one minute then exit.

```
import digiweb

template = """
<html><head><title>Request info</title></head>
<body>
%s request for path '%s'
<hr>
Headers: %s
<hr>
Args: %s
</body>
</html>
"""

def pageinfo(type, path, headers, args):
    return (digiweb.TextHtml, template % (type, path, headers, args))

if __name__ == "__main__":
    import time
    hnd = digiweb.Callback(pageinfo)
    time.sleep(60)
```

This example, like the one above will display request information, but it will make args be a dict object instead of a str during a POST request so it can be treated the same way as a GET request.

```
import digiweb
import cgi

template = """
<html><head><title>Request info</title></head>
<body>
%s request for path '%s'
<hr>
Headers: %s
<hr>
Args: %s
</body>
```

```
</html>
"""

def pageinfo(type, path, headers, args):
    tmp = {}
    if type=="POST":
      tmp = cgi.parse_qs(args)
      args=tmp
      for key in args:
        args[key] = args[key][0]
      return (digiweb.TextHtml, template % (type, path, headers, args))

if __name__ == "__main__":
    import time
    hnd = digiweb.Callback(pageinfo)
    time.sleep(60)
```

Availability

This feature is available as of firmware revision 2.8.0.

Module: idigidata

Methods

send_to_idigi()

Purpose

To pass data from the device to the cloud

Syntax

```
send_to_idigi (data, filename [, collection, content_type, archive=False,
append=False, timeout=30]) -> (success, error, errmsg)
```

Description

- Send data to the iDigi data service and wait for the response.
- Parameter *data* is the string or binary data to send.
- Parameter *filename* is the server filename to store data.
- Optional parameter *collection* is the subcollection where the file should be stored.
- Optional parameter *content_type* is the MIME type of the data.
- Optional parameter *archive* is True to archive the data.
- Optional parameter *append* is True to append the data to an existing resource.
- Optional parameter *timeout* is maximum time in seconds to wait for a response. (timeout=30 default is suitable only if the 'complexity' of your upload job is low. For example, if you are uploading in a DIA channel format, and have 200 samples of say 5 different channels, then 30 seconds is fine & the real upload may takes seconds (or sub-seconds). However, if you upload 200 samples and they represent 200 different channels, then often 30 seconds is not enough as the cloud needs to lookup, validate, and locate each channel one by one.)

Return

Returns (success, error, errmsg) if successful. raises an Exception on error. success is True if the data was stored sucessfully. error is the status code of the transfer. errmsg is the status message of the transfer.

register_callback()

Purpose

Register a function to be called when a device request message is received. Device requests are of the form:

```
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="00000000-00000000-000000FF-FF000000"/>
    </targets>
    <requests>
      <device_request target_name="data_callback_example">
        my payload string
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

Callback functions should be of the form: function (target, data) -> reply

Parameter target is the request handler identifier string. Parameter data is the received message string or binary data.

Returns the reply data to send.

Syntax

```
register_callback (target, function) -> handle
```

Description

Register a function to be called when a device request message is received.

Parameter target is the request handler identifier string. Parameter function is the function to call.

Return

Returns a handle to the registered callback; raises an Exception on error.

unregister_callback()

Purpose

Unregister a function to be called when a message is received for a service.

Syntax

```
unregister_callback (handle)
```

Description

Unregister the callback function

Parameter handle is the callback handle returned by register_callback().

Return

None

Examples

Example #1

This sample toggles the state of the relay and sends the results up to Device Cloud every 2 minutes as file "data.txt". change DESTINATION below to match your device

"""This sample was run on an X2e with a Smart Plug.
"""

```
import time

import idigidata
import xbee


# configure known destination:
DESTINATION="[00:13:a2:00:40:6b:65:d6]!"
filename = "data.txt"
collection = "readings"

while(1):
  xbee.ddo_set_param(DESTINATION, 'D4', 4) ## Turning the power relay off
  data = xbee.ddo_get_param(DESTINATION, 'D4') ## Getting a reading of D4

  #send file to cloud
  idigidata.send_to_idigi('data is %02X'%ord(data), filename, collection)
  time.sleep(120)

  xbee.ddo_set_param(DESTINATION, 'D4', 5) ## Turning the power relay on
  data = xbee.ddo_get_param(DESTINATION, 'D4') ## Getting a reading of D4

  #send file to cloud
  idigidata.send_to_idigi('data is %02X'%ord(data), filename, collection)
  time.sleep(120)
```

Example #2

To access the callback function from Device Cloud you need to send an SCI request as follows:

```
<sci_request version="1.0">
  <data_service>
    <targets>
      <device id="00000000-00000000-000000FF-FF000000"/>
    </targets>
    <requests>
      <device_request target_name="data_callback_example">
        D4
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

The callback function will return the value of D4, If the following is sent:

```
<sci_request version="1.0">
  <data_service>
```

```
    <targets>
       <device id="00000000-00000000-000000FF-FF000000"/>
    </targets>
    <requests>
      <device_request target_name="data_callback_example">
        STOP
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

The callback function will be unregistered and "CALLBACK STOPPED" will be returned.

"""This sample was run on an X2e with a Smart Plug change DESTINATION below to match your device
"""

```
import time

import idigidata
import xbee

# configure known destination:
DESTINATION="[00:13:a2:00:40:6b:65:d6]!"
filename = "data.txt"
collection = "readings"

def data_callback(data_callback_example,xml):
    if xml.find("D4")>-1:
        data = xbee.ddo_get_param(DESTINATION, 'D4') ## Getting a reading of D4
        print "Successful callback"
        return "D4 reading is %02X"%ord(data)
    elif xml.find("STOP")>-1:
        idigidata.unregister_callback(handle)
        print "CALLBACK STOPPED"
        return "CALLBACK STOPPED"
    else:
        return "Please use D4 to get reading"


handle = idigidata.register_callback("data_callback_example", data_callback)

while(True):

    xbee.ddo_set_param(DESTINATION, 'D4', 4) ## Turning the power relay off

    time.sleep(120)

    xbee.ddo_set_param(DESTINATION, 'D4', 5) ## Turning the power relay on

    time.sleep(120)
```

idigidata Python module examples

send_to_idigi

Uset the `send_to_idigi` method to:

- Upload data directly to data streams
- Place files in the device's Data Service

You can find more information in the Device Cloud Programming Guide. (Direct device uploads, file data)

Data stream upload using CSV:

```
import idigidata

# Timestamp corresponds to Sep 1 2015, 12:00:00 UTC.
# This example uploads only one data point, but devices can upload more than
# one data point per request.
# This point will appear under the data stream <device-id>/temperature
doc = """\
#TIMESTAMP,DATA,DATATYPE,STREAMID
1441108800000,75,INTEGER,temperature
"""

# The leading 'DataPoint/' in the upload URL is enough to tell Device Cloud
# that these are data points. The file name would be used as the stream ID,
# but we specify the STREAMID field in the upload, so that is used instead.
success, code, msg = idigidata.send_to_idigi(doc, "DataPoint/upload.csv")

if not success:
    print "Got error code %d (%s) uploading data" % (code, msg)
```

Data stream upload using XML:

You can reuse the code for the CSV upload example above if you change the filename to `"DataPoint/upload.xml"` and replace the `doc` value like this:

```
doc = """
<DataPoint>
    <data>75</data>
    <streamId>temperature</streamId>
    <timestamp>1441108800000</timestamp>
    <dataType>integer</dataType>
</DataPoint>
"""
```

You can also upload a list of data points. Just wrap the `DataPoint` elements in a `list` element, like this:

```
<list>
    <DataPoint>
        ...
    </DataPoint>
    <DataPoint>
        ...
    </DataPoint>
    ...
</list>
```

Data stream upload of binary content:

The binary upload format lets you upload arbitrary data into a data stream, but limits you to one data point per upload. The filename determines the data stream ID.

- DataPoint/temperature.bin uploads to the stream <device-id>/temperature
- DataPoint/hello/world.bin uploads to the stream <device-id/hello/world

General file data:

You can use `send_to_idigi` to upload a string or a blob of binary data into a file. This example will create a file named `file.txt` in your device's file data in Device Cloud.

```
import idigidata

doc = "Hello, world!"

idigidata.send_to_idigi(doc, "file.txt")
```

register_callback

This example handles a target name of "reply", and adds "to you too" to the request content to make the response.

Code to execute on the gateway:

```
import idigidata
import time


def example_callback(target, data):
    # Remove leading and trailing whitespace
    data = data.strip()
    # Print out what we got
    print data

    # Response will be:
    # <device_request target_name="reply" status="0">... to you too</device_
request>
    return data + " to you too"

# Register the callback function for the target name "example".
handle = idigidata.register_callback("reply", example_callback)

while True:
    # Loop doing nothing to keep the program running.
    # example_callback will be called when the SCI request is received.
    time.sleep(1)
```

SCI request to send in Device Cloud:

```
<sci_request version="1.0">
    <data_service>
        <targets>
            <!-- Replace this id value with your gateway's Device ID -->
            <device id="00000000-00000000-00409DFF-FF000000" />
        </targets>
        <requests>
            <device_request target_name="reply">
                Hello
            </device_request>
        </requests>
```

```
    </data_service>
</sci_request>
```

unregister_callback

After registering a data callback function, you can remove that callback with `unregister_callback`.

```
handle = idigidata.register_callback("example", example_callback)
# ...

idigidata.unregister_callback(handle)
```

idigidata Python module methods

```
send_to_idigi(data, filename[, collection, content_type, archive=False,
append=False, timeout=30]) -> (success, error, errmsg)
```

Send data to the Device Cloud data service and wait for the response.

Arguments:

- `data`: the string or binary data to send.
- `filename`: the server filename where the data will be stored.
- `collection` (optional): the subcollection where the file should be stored.
- `content_type` (optional): the MIME type of the data.
- `archive` (optional): set to True to archive the data.
- `append` (optional): set to True to append the data to an existing resource.
- `timeout` (optional): the maximum time in seconds to wait for a response.

Returns:

A tuple containing: a success boolean, the status code of the transfer, and the status message of the transfer.

```
register_callback(target, function) -> handle
```

Register a callback function of the form `(target, data) -> response` to be called when the gatewaya device request with the specified target name is received.

Arguments:

- `target`: the device request target name to register for.
- `function`: the callback function to register.

Returns:

A handle to the registered callback

```
unregister_callback(handle)
```

Unregister a device request callback.

Arguments:

- handle: a handle to a registered callback. (See `register_callback`.)

Returns:

None

Module: iridium

The Iridium Module

Introduction

Built-in Python module provided by Digi for utilizing the Iridium satellite interface.

Functions

Power Control

```
iridium_power_get()
```

- In digipowercontrol module
- Returns a truth value

```
iridium_power_set(value)
```

- Accepts a truth value
- Errors result in exception
- No return value

Status

```
digi_iridium.state()
```

- No parameters
- Returns a dictionary
- Values including:

power (a truth value) serial_number (a Python string)

- --network_availability (a truth value)
- --signal_strength (0-5)
- If power is off, only power state returned

Send

```
digi_iridium.send(msg)
```

- No return value
- Error results in exception

- Blocks until message sent or rejected
- Parameter is payload of message
- Expected to be a Python string
- Always transferred as a “binary” blob
- No destination address!

Receive

```
digi_iridium.Callback(fn)
```

- Returns a callback handle
- Callback remains registered while handle exists “fn” is a one parameter function
- Parameter is message as a Python string
- Use a lambda wrapper to pass context if needed (see example)
- Incoming messages delivered to all registered callbacks
- Receive “Gotchas”
- --Controlling Latency
- --Source Address Management
- --“Hidden” Behaviors

Python Example

```
            # Roughly once per second, print updated
# statistics.  Whenever we receive a message,
# attempt to retransmit with location added

import digipowercontrol as dpc
import digi_iridium as di
import digihw as dhw
import Queue

dpc.iridium_power_set(True) # Power on the Iridium modem

data_queue= Queue.Queue() # Prepare receive path

def handle_rx_msg(input_queue, payload):
    input_queue.put(payload)

cbhandle = di.Callback(lambda msg: \
    handle_rx_msg(data_queue, msg))

txcnt = 0
rxcnt = 0
while True:
    print "RxCnt: %-10d  TxCnt: %-10d" % \
        (rxcnt, txcnt)

    try:
```

```
        msg = data_queue.get(True, 1.0)
        rxcnt = rxcnt + 1
        try:
            msg += ' ' + str(dhw.gps_location())
        except:
            msg += ' (location unknown)'
        try:
            print “Trying to transmit a message"
            di.send(msg)
            txcnt = txcnt + 1
        except:
            print "Currently unable to transmit”
        except Queue.Empty: pass
```

Code Analysis

```
            # Roughly once per second, print updated
# statistics.  Whenever we receive a message,
# attempt to retransmit with location added
import digipowercontrol as dpc
import digi_iridium as di
import digihw as dhw
import Queue
```

- Not representative of a “finished” Python app
- Libraries required for the application
- Python trick for space savings (not necessary)

```
# Power on the Iridium modem
dpc.iridium_power_set(True)
```

- Ensure that the Iridium is powered on
- Default power state of modem is “off”

```
            # Prepare receive path
data_queue= Queue.Queue()
def handle_rx_msg(input_queue, payload):
    input_queue.put(payload)
cbhandle = di.Callback(lambda msg: handle_rx_msg(data_queue, msg))
```

- Queue to pass data from callback to main code
- Lambda wrapper example, demonstrating how to pass two parameters to a callback

```
while True:
    print "RxCnt: %-10d  TxCnt: %-10d" % rxcnt, txcnt)
    try:
        msg = data_queue.get(True, 1.0)
        rxcnt = rxcnt + 1
```

```
        :
    except Queue.Empty: pass
```

- Simple sample main loop with Queue “polling”

```
try:
    msg += ' ' + str(dhw.gps_location())
except:
    msg += ' (location unknown)‘
```

- Try/except block in case the GPS location function throws an exception
- Compose response based on the received message

```
try:
    print “Trying to transmit a message“
    di.send(msg)
    txcnt = txcnt + 1
except:
    print "Currently unable to transmit”
```

- Simple transmission sample
- Transmission will fail if the satellite network is not available at transmission time
- Successful transfer to the Iridium gateway servers will “almost always” result in a successful end delivery
- No feedback to device if gateway end fails

Availability

Products which support this module

This feature is available only on the Digi Connect X5/

Module: rci

This module allows you to handle RCI Requests.

Note There are limitations on the XML data which may be passed through an RCI callback on many Digi devices (e.g. the ConnectPort X2, X4, and X8). These limits are:

- Max attribute length (e.g. <tag attribute="this field length" />): 50 characters
- Max number of attributes (per element): 10 attributes
- Max element tag length: 50 characters

RCI as Client

To pass a direct RCI request to the local device, use the rci.process_request() request.

```
process_request(request) return response
```

This example shows how you would make use of the process_request method to retrieve the uptime of the Digi device. This does NOT require a callback to be installed. The response is generated by the core services within the Digi product.

```
# import the rci module to get access to the process_request method
import rci

# This is the rci request that will return device statistics
request = '<rci_request version="1.1"><query_state><device_stats/></query_
state></rci_request>'

# Response will store the response from the rci request
response = rci.process_request(request)
```

The contents of response will be similar to:

```
<rci_reply version="1.1">
        <query_state>
                <device_stats>
                        <cpu>9</cpu>
                        <uptime>341961</uptime>
                        <datetime>Mon May 11 10:24:47 2009</datetime>
                        <totalmem>16777216</totalmem>
                        <usedmem>13243060</usedmem>
                        <freemem>3534156</freemem>
                </device_stats>
       </query_state>
</rci_reply>
```

RCI as Server

Enabling the Digi device to respond to RCI request received by serial, TCP, HTTP or Device Cloud/SCI.

```
add_rci_callback(name, callback) return None
```

Function callback is called when name is sent as a rci do_command target.

Callback will be called with a string representing the xml contained within the do_command. callback returns a string which will be returned to the caller as the result of the request. The returned string must be valid xml. Returning invalid xml will result in invalid xml being returned to the requester. Note, plain text is valid xml. Binary data must be base64 encoded. If no reply is expected, an empty string ("") must be returned

This is a blocking call. callback will be called from the thread that calls this function.

There are no limitations on what the callback function does, However, device processing waits for a response so it is recommended to keep processing to a minimum. Best practice is to respond immediately and then perform processing in another thread. If no response is returned from callback within 45 seconds, the device will stop waiting and return a warning to the RCI requester.

```
stop_rci_callback(name) return None
```

Stops the previously called add_rci_callback with the specified name.

```
process_request(request) return response
```

Processes an RCI request and returns the response.

Example #1

This example will print out any data sent to it and then send the data back as the reply

```
import rci

def rci_callback(xml):
        print xml
        return "received: %s" % (xml)

rci.add_rci_callback("rci_callback_example", rci_callback)
```

Running this, send a RCI request as follows to the device

```
<rci_request version="1.1">
  <do_command target="rci_callback_example">
ping
  </do_command>
</rci_request>
```

In reply the device will send:

```
<rci_reply version="1.1"><do_command target="rci_callback_example">
received:ping
</do_command></rci_reply>
```

Example #2

This example will use the producer/consumer design pattern to have asynchronous RCI calls. The work in the consumer thread in this case will just write the messages to standard out. This design pattern is good to use if you are using the RCI mechanism to do some action that can take awhile to process instead of using it as a mechanism to retrieve prepared data from the device.

```
import rci
import thread
from Queue import Queue

# queue to hold messages between producer and consumer threads
q = Queue(10)

def rci_callback(xml):
        # Stop listening if the message contains the string "STOP"
        if xml.find("STOP")>-1:
                rci.stop_rci_callback("rci_callback_example")
                return "Stopped"

        # Because we have limited time and are blocking the response in the
        #  callback thread we will act as a producer, adding incomming messages
        #  to a Queue to be processed by a consumer thread
        q.put(xml)
        return "Ok"

def producer_thread():
        print "listening for RCI callbacks"
        #Start listening for RCI messages, block until unregistered or an
exception occurs
        rci.add_rci_callback("rci_callback_example", rci_callback)
        print "STOP received, no longer listening"

# start a thread to handle RCI requests
thread.start_new_thread(producer_thread, ())

# Main thread will now become a consumer of the queue
while True:
        data = q.get()
        print data
```

Module: xbee

Additional ZigBee functionality is provided by the zigbee module included on the Software and Documentation CD media included with your Python-enabled Digi product. Descriptions of supported methods and types follow.

Methods

ddo_get_param()

Purpose

Get a Digi Device Objects parameter value.

Syntax

```
ddo_get_param(addr_extended, id[, timeout, order=False]) . string
```

Description

- Get a DDO parameter id by using the 64-bit address given by *addr_extended*.
- Parameter addr_extended is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.
- Parameter *id* is a 2-byte string such as 'NI'. To make DDO parameter requests of remote radios, all radio module firmware versions must support this capability. For a description of valid *id* values, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).
- Optional parameter **timeout** is maximum time in seconds to wait for a response.
- Optional parameter **order** is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Return

To properly interpret the binary string returned from this function, please see the API manual for the radio module. It may be useful to use the i module to construct the type into a more useful data type.

An exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a try:/except: statement.

This call may block for many seconds if the remote node is sleeping or offline - therefore avoid application designs which repeatedly query nodes which might be offline. For example do not blindly attempt to read ten DDO parameters from the same node in a row; if the first ddo_get_param() call timeouts, then so should the remaining nine. Instead, when the first DDO call fails, record the node as offline and return at a future time to try reading the ten DDO parameters again.

ddo_set_param()

Purpose

Set a Digi Device Objects parameter value.

Syntax

```
ddo_set_param(addr_extended, id[, value, timeout, order=False, apply=True]) .
boolean
```

Description

Set a DDO parameter id by using the 64-bit address given by *addr_extended* and the given value value.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.

Parameter *id* is a 2-byte string such as 'NI'. To make DDO parameter requests of remote radios, all radio module firmware versions must support this capability. For a description of valid values for *id*, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).

Parameter *value* must be either a string or an integer. Do not submit any value when the parameter does NOT require a value - as for example as with the 'FR' command to reboot the Xbee device.

Optional parameter *timeout* is maximum time in seconds to wait for a response.

Optional parameter *order* is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Optional parameter *apply* is **True** to apply changes to node settings immediately. If *apply* is **False**, changes are queued in the node until a command with *apply* set to **True** or the AC command is sent to the node.

Note If *addr_extended* is a broadcast address (such as "00:00:00:00:00:00:FF:FF!") which will have no response, then you must set *timeout*=0 or ddo_set_param() will throw an exception and fail.

Return

A boolean True or False is returned if the remote node accepts or rejects the SET command. Otherwise an exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except** statement. See **ddo_get_param()** for more usage hints.

ddo_command()

Purpose

Execute a Digi Device Objects command.

Syntax

```
ddo_command(addr_extended, id[, param, timeout, order=False, apply=True]) .
string or None
```

Description

Execute a DDO command given by id by using the 64-bit address given by *addr_extended* and the optional parameter *param*.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.

Parameter *id* is a 2-byte string such as 'NI'. To send DDO commands to remote radios, all radio module firmware versions must support this capability. For a description of valid values for *id*, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).

Parameter *param* must be either a string or an integer. Do not submit any value when the command does NOT require a value - as for example as with the 'FR' command to reboot the Xbee device.

Optional parameter *timeout* is maximum time in seconds to wait for a response.

Optional parameter *order* is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Optional parameter *apply* is **True** to apply changes to node settings immediately. If *apply* is **False**, changes are queued in the node until a command with apply set to True or the AC command is sent to the node.

Return

A string is returned if the command produces a result. To properly interpret the binary string returned from this function, please see the API manual for the radio module. It may be useful to use the *struct* module to construct the type into a more useful data type.

An exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except statement**. See **ddo_get_param()** for more usage hints.

get_node_list()

Purpose

Perform a node discovery.

Syntax

```
get_node_list([refresh=True, clear=refresh, discover_digi=False, discover_
zigbee=False]) . (node, node, …, node)
```

Description

Perform a node discovery and return a tuple of nodes.

If the *refresh* parameter is set to **True**, this function will block and a fresh node discovery is performed. If no discovery methods are selected, a method appropriate for the network type will be used.

If *refresh* is set to **False**, this function returns a cached copy of the node discovery list. This cached version may include devices that were unable to be discovered within the discovery timeout imposed during a blocking call. If discovery methods are selected, newly discovered nodes will be added to the cached list.

If the *clear* parameter is set to **True**, this function will clear the cached list and perform a network discovery. If the *clear* parameter is set to **False**, this function will add newly discovered nodes to the

existing cached list. If the clear parameter not specified, this function will clear the cached list if a network discovery is being performed.

If the *discover_digi* parameter is set to **True** this function will block and network discovery of Digi nodes will be performed. This obtains extended information supported by Digi nodes.

If the *discover_zigbee* parameter is set to **True** this function will block and network discovery of ZigBee nodes will be performed. This obtains information from standard ZigBee nodes.

register_joining_device()

Purpose

Register a new node into the local trust center key table.

Syntax

```
register_joining_device(addr_extended, key)
```

Description

This method is available on a gateway running a Smart Energy profile trust center.

Register a new node with the 64-bit address given by *addr_extended*, and set its initial trust center link key to *key*.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!".

Parameter *key* is a binary string of up to 16 bytes. If key is less than 16 bytes, the upper bytes are padded with 0.

Return

An exception is thrown if a registration error occurs, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except** statement.

unregister_joining_device()

Purpose

Unregister a node from the local trust center key table.

Syntax

```
unregister_joining_device(addr_extended)
```

Description

This method is available on a gateway running a Smart Energy profile trust center.

Remove the node with the 64-bit address given by *addr_extended* and its key from the local trust center key table.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!".

Return

An exception is thrown if the given node is not registered, an error occurs, or if the parameter is malformed. Therefore you must wrap any calls with a **try-except** statement.

Classes

node

Name

node – a Python object returned from a node discovery

Attributes

Name	Type	Description	Example
type	string	node role/type in mesh	is in ['coordinator', 'router', 'end']
addr_ extended	string	64-bit extended hardware address	"[00:13:a2:00:40:0a:07:8d]!"
addr_short	string	16-bit network assigned address	"[49c3]!"
addr_parent	string	16-bit network parent address	"[fffe]!"
source_route	tuple	tuple of 16-bit network addresses in the source route from the node.First element is a neighbor of the node. Last element is a neighbor of the gateway node.Empty if the node is one hop away, or no source route has been received from the node.Present in gateway firmware version 2.15 and later.	("[1234]!", "[5678]!")
profile_id	int	node profile ID	0xC105 or 49413
manufacturer_ id	int	node manufacturer ID	0x101E or 4126
label	string	node’s string label (Setting 'NI')	"TK103U"
device_type	int	node’s device type (Setting 'DD').Upper 16 bits contain the module type. Lower 16 bits contain the product type.Will be 0 if the node does not support 'DD'	0x00030001

Methods

to_socket_addr()

Purpose

Transform a node into a socket address tuple

Syntax

```
to_socket_addr([endpoint,] [profile_id,] [cluster_id,] [use_short]) .
    (“address!”, endpoint, profile_id, cluster_id)
```

Desc ription

Transform this node into a socket address tuple, suitable for use with functions from the socket modules. If use_short is True, the short node address representation is used instead of the 64-bit extended address, which is used by default.

ZigBee Module Examples

Perform a Network Node Discovery

```
            #
# Perform a node discovery and print out# the list of discovered nodes to stdio.
#

# import the zigbee module into its own namespace:
import zigbee

# Perform a node discovery:
node_list = zigbee.getnodelist()

# Print the table:
print "%12s %12s %8s %24s" % \
    ("Label", "Type", "Short", "Extended")
print "%12s %12s %8s %24s" % \
    ("-" * 12, "-" * 12, "-" * 8, "-" * 24)

for node in node_list:
    print "%12s %12s %8s %12s" % \
        (node.label, node.type, \
            node.addr_short, node.addr_extended)
```

Use DDO to Read Temperature from XBee Sensor

```
            #
# Collect a sample from a known XBee Sensor adapter
# and parse it into a temperature.
#


# import zigbee and xbee_sensor modules:
import zigbee
import xbee_sensor


# configure known destination:
DESTINATION="[00:13:a2:00:40:0a:07:8d]!"


# Note: for clarity, the try: except: statements required to handle timeout is
not shown
```

```
# ensure sensor is powered from adapter:
zigbee.ddo_set_param(DESTINATION, 'D2', 5)
zigbee.ddo_set_param(DESTINATION, 'AC', '')


# get and parse sample:
sample = zigbee.ddo_get_param(DESTINATION, '1S')
xbee_temp = xbee_sensor.XBeeWatchportT()
xbee_temp.parse_sample(sample)
print "Temperature is: %f degrees Celsius" % (xbee_temp.temperature)
```

Create your own display-mesh command

This page includes a fully functional application using ddo_get_param() and getnodelist(): Create Your Own Custom Node List on a Digi ConnectPort X2/X4/X8 gateway

Categories

Advanced Device Discovery Protocol (ADDP) 105
Android 106

Advanced Device Discovery Protocol (ADDP)

What is ADDP?

ADDP (Advanced Device Discovery Protocol) is a proprietary protocol developed by Digi International that allows devices on a local network to be found regardless of their network configuration.

How does it work?

ADDP uses a client/server model. The client is the application that is searching for devices. The server is the device that is being search for.

In the simplest terms, the client application sends out a specially formatted UDP broadcast packet on the network. ADDP servers listening for the packet, will receive it, and send an ADDP response back to the client. Once this process is complete, the client can then send configuration requests to the device. These can include things like network settings, and reboot requests.

Java library

A subset of the protocol has been implemented in Java. You can find the jar file here: ADDP Library

The associated javadoc documentation can be found here: ADDP Java doc.

This library allows you to search synchronously, and asynchronously for devices on the network. You can then use it to reconfigure the device's network settings, or reboot the device.

Java sample application

A simple discovery sample application can be found here: AddpSample.zip.

Basic usage

First, instantiate the AddpClient object.

```
AddpClient addpClient = new AddpClient();
```

Next, call SearchForDevices() and check the return value. Then get the devices, and walk the hashtable.

```
if (addpClient.SearchForDevices()) {
    AddpDeviceList deviceList = addpClient.getDevices();

    Enumaration<AddpDevice> e = deviceList.elements();
    while(e.hasMoreElements()) {
        AddpDevice device = e.nextElement();

    // do something with the device here
    System.out.println(device.toString());

    // if device is not configured for DHCP, then turn it on and reboot.
    if (device.getDHCP() == 0) {
      addpClient.setDHCP(device, true, "dbps");
      addpClient.rebootDevice(device, "dbps");
    }
  }
}
```

Android

Android animation

Android sample animation test

(Android modules, i.MX51 and i.MX53) Android program, when this application runs on the android device, it will change the background images for every 1/2 second, we can start and stop the animation using the button displayed on the application.

Test files

This sample program contains several files, the /src folder contains the source files.

Animation test sample application

The Android Animation Test sample application can be found here: Animation2.zip

Basic usage

Compile, load and run program using Android environment.

Sample of Animation2Activity.java file:

```
package animation2.test;

//import canvas.paint.CanvasActivity.DemoView;
import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.widget.*;

public class Animation2Activity extends Activity {
    /** Called when the activity is first created. */

    Button b;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.setupButton();

    }


    private void setupButton() {
                b = (Button) this.findViewById(R.id.startFAButtonId);
                b.setOnClickListener(new Button.OnClickListener() {
                        public void onClick(View v) {
                                parentButtonClicked(v);
                        }
```

```
                });
        }


private void parentButtonClicked(View v) {
            animate();
    }

private void animate() {
            ImageView imgView = (ImageView) findViewById(R.id.animationImage);
            // imgView.setVisibility(ImageView.VISIBLE);
            imgView.setBackgroundResource(R.drawable.ani);

            AnimationDrawable frameAnimation = (AnimationDrawable)
imgView.getBackground();


            if (frameAnimation.isRunning()) {
                    frameAnimation.stop();
                    b.setText("Start");
            } else {
                    frameAnimation.start();
                    b.setText("Stop");
            }
    }

public boolean onTouchEvent(MotionEvent event) {

        int eventaction = event.getAction();

        switch (eventaction)
        {
          case MotionEvent.ACTION_DOWN:              // finger touches the screen
                 this.setupButton();

                 break;
          case MotionEvent.ACTION_MOVE:             // finger moves on the screen
                  //this.setupButton();
                  break;
          case MotionEvent.ACTION_UP:                   // finger leaves the screen

              break;
       }

       return true;

    }

}
```

Android Bluetooth test - support for Android Bluetooth test

Android sample Bluetooth test

'***(Android supported modules)***' Android program, when this application runs on the android device it invokes the bluetooth interface.

Test files

This sample program contains several files, the /src folder contains the source files.

Bluetooth test sample application

The Android Animation Test sample application can be found here: BlueToothTest.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of MainActivity.java file:

```
package com.digi.bluetoothtest;

import java.io.IOException;
import java.io.OutputStream;
import java.lang.reflect.Method;
import java.util.Set;
import java.util.UUID;

import android.os.Bundle;
import android.os.Message;
import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothClass;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
        private static final int REQUEST_ENABLE_BT = 1;
        private Button _scanBlueToothButton;
        private EditText _logEditText;

        private BluetoothAdapter _bluetoothAdapter = null;


        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                connectUi();

                _bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

                if (_bluetoothAdapter == null) {
```

```
                        Toast.makeText(this, "No BT adapter", Toast.LENGTH_
LONG).show();
                        return;
                }

                if (!_bluetoothAdapter.isEnabled()) {
                        Intent enableBt = new Intent(BluetoothAdapter.ACTION_
REQUEST_ENABLE);
                        startActivityForResult(enableBt, REQUEST_ENABLE_BT);
                }
                else {}
        }


        private void scanBlueToothButton_OnClick() {
                ProgressDialog dialog = ProgressDialog.show(this, "",
                        "Loading. Please wait...", true);

                _bluetoothAdapter.disable();
                _bluetoothAdapter.enable();

                _bluetoothAdapter.startDiscovery();

                dialog.dismiss();

                Set<BluetoothDevice> devices = _bluetoothAdapter.getBondedDevices
();
                StringBuilder sb = new StringBuilder();

                if (devices.size() > 0) {
                        for (BluetoothDevice device : devices) {
                                sb.append(device.getName());
                                sb.append("\n");
                                // 1e0ca4ea-299d-4335-93eb-27fcfe7fa848

                                try {
                                        Method m = device.getClass().getMethod(
                                                "createRfcommSocket", new Class[]
{ int.class });

                                        BluetoothSocket sock = (BluetoothSocket)
m.invoke(device, 1);

                                        sock.connect();

                                        OutputStream stream =
sock.getOutputStream();
                                        stream.write("Hello bt world!".getBytes
());
                                        stream.close();
                                } catch (Exception e) {
                                        e.printStackTrace();

                                }
                        }
                }
                _logEditText.setText(sb.toString());
        }
```

```
	@Override
	protected void onActivityResult(int requestCode, int resultCode, Intent 
data) {
		super.onActivityResult(requestCode, resultCode, data);
		
		switch (requestCode) {
			case REQUEST_ENABLE_BT:
				break;

		}

	}


	@Override
	public boolean onCreateOptionsMenu(Menu menu) {
		getMenuInflater().inflate(R.menu.activity_main, menu);
		return true;
	}


	private void connectUi() {
		_scanBlueToothButton = (Button) findViewById
(R.id.scanBluetoothButton);
		_scanBlueToothButton.setOnClickListener(new View.OnClickListener
() {
			
			public void onClick(View v) {
				scanBlueToothButton_OnClick();
			}
		});
		
		_logEditText = (EditText) findViewById(R.id.logEditText);
	}
}
```

Android HelloBox2D

Android sample HelloBox2D test

(Android supported modules) Android program, This application is a porting of the Box2D into android environment.

Test files

This sample program contains several files, the /src folder contains the source files.

HelloBox2D test sample application

The Android HelloBox2D Test sample application can be found here: HelloBox2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of HelloBox2DActivity.java file:

```
package com.digi.box2d;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;

public class HelloBox2DActivity extends Activity {
         private PhysicsWorld mWorld;
        private Handler mHandler;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mWorld = new PhysicsWorld();
        mWorld.create();

        mHandler = new Handler();
        mHandler.post(update);

    }

    @Override
    protected void onPause() {
        super.onPause();

        mHandler.removeCallbacks(update);
    }

    private Runnable update = new Runnable() {
        public void run() {
            mWorld.update();
            if (mWorld.FallingBox.isAwake()){
                 mHandler.postDelayed(update, (long) (mWorld.timeStep*1000));
            } else {
                mHandler.removeCallbacks(update);
            }
        }
    };
}
```

Android RenderAmovingSprite

Android sample render moving sprite tests

(Android supported modules) Android program, It renders a sprite on the screen and moves it.

Test files

This sample program contains several files, the /src folder contains the source files.

Render Moving Sprite Test Sample Application

The Android Render a Moving Sprite Test sample application can be found here: RenderAMovingSprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderAMovingSpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.view.Window;

public class RenderAMovingSpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                requestWindowFeature(Window.FEATURE_NO_TITLE);

                mGLSurfaceView = new GLSurfaceView(this);

                MyRenderer spriteRenderer = new MyRenderer(this);

                GLSprite sprite = new GLSprite(R.drawable.digi);

                DisplayMetrics dm = new DisplayMetrics();

                getWindowManager().getDefaultDisplay().getMetrics(dm);

                Grid spriteGrid = null;

                // Setup a quad for the sprite to use. All sprites will use the
                // same sprite grid instance.
                spriteGrid = new Grid(2, 2, false);
                spriteGrid.set(0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, null);
                spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
                spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
                spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);

                sprite.x = 100;
                sprite.y = 150;
                sprite.width = 64;
                sprite.height = 64;

                sprite.setGrid(spriteGrid);

                Runtime r = Runtime.getRuntime();
                r.gc();

                spriteRenderer.sprite = sprite;
                spriteRenderer.setVertMode(true, true);
                mGLSurfaceView.setRenderer(spriteRenderer);

                setContentView(mGLSurfaceView);

                Thread gameThread = new Thread(new Game(sprite, this));
```

```
                gameThread.start();
        }
}
```

Android RenderAsprite

Android sample Render a sprite test

(Android supported modules) Android program, Renders a stationary sprite on the screen.

Test files

This sample program contains several files, the /src folder contains the source files.

Render a sprite test sample application

The Android Render a Sprite Test sample application can be found here: RenderASprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderASpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;

public class RenderASpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mGLSurfaceView = new GLSurfaceView(this);

        MyRenderer spriteRenderer = new MyRenderer(this);

        GLSprite sprite = new GLSprite(R.drawable.digi);

        DisplayMetrics dm = new DisplayMetrics();

        getWindowManager().getDefaultDisplay().getMetrics(dm);

        Grid spriteGrid = null;

        // Setup a quad for the sprite to use.  All sprites will use the
        // same sprite grid instance.
        spriteGrid = new Grid(2, 2, false);
        spriteGrid.set(0, 0,  0.0f, 0.0f, 0.0f, 0.0f , 1.0f, null);
        spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
        spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
```

```
        spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);

        sprite.x = 100;
        sprite.y = 100;
        sprite.width = 64;
        sprite.height = 64;

        sprite.setGrid(spriteGrid);

        Runtime r = Runtime.getRuntime();
        r.gc();

        spriteRenderer.sprite = sprite;
        spriteRenderer.setVertMode(true,true);

        mGLSurfaceView.setRenderer(spriteRenderer);

        setContentView(mGLSurfaceView);


    }
}
```

Android Test2D

Android sample Test2D test

(Android supported modules) Android program, a 2D rendering test using canvas.draw.

Test files

This sample program contains several files, the /src folder contains the source files.

Test2D test sample application

The Android Render a Moving Sprite Test sample application can be found here: Test2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of Test2DActivity.java file:

```
package com.digi.test2d;

import android.app.Activity;
import android.os.Bundle;
import android.view.Display;
import android.view.Window;
import android.view.WindowManager;

public class Test2DActivity extends Activity{
        private drawView view;

    /** Called when the activity is first created. */
    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                        WindowManager.LayoutParams.FLAG_FULLSCREEN);

        Art.loadBitmaps(getResources());

        Display display = getWindowManager().getDefaultDisplay();

                view = new drawView(this, display.getWidth(), display.getHeight
());

        setContentView(view);
    }
}
```

Android UDP client

Android sample for UDP client test

(Android modules i.MX51 and i.MX53) Android program, when this application runs on the android device, it will show "temp" and "humi" buttons on the android UI, and as we click on those buttons it will communicate with the UDPserver.

Test files

This sample program contains several files and the /src folder contains the source files.

UDP Claient Test Sample Application

The Android UDP Client Test sample application can be found here: AndroidUDPClient.zip.

Basic usage

Sample of ChatServerActivity.java file:

```
package test.chat.serv;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Iterator;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class ChatServerActivity extends Activity {
    private static final String host = null;
        private int port;
        String str=null;
        /** Called when the activity is first created. */
        TextView txt5,txt1;
        byte[] send_data = new byte[1024];
        byte[] receiveData = new byte[1024];
        String modifiedSentence;
        Button bt1,bt2,bt3,bt4;


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txt1   = (TextView)findViewById(R.id.textView1);
        txt5   = (TextView)findViewById(R.id.textView5);

        bt1 = (Button) findViewById(R.id.button1);
        bt2 = (Button) findViewById(R.id.button2);
        bt3 = (Button) findViewById(R.id.button3);
        bt4 = (Button) findViewById(R.id.button4);
        //textIn.setText("oncreate");




        bt1.setOnClickListener(new View.OnClickListener(){
                public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);
                str="temp";
                 try {
                                client();
                                //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                }
        }



    });

bt2.setOnClickListener(new View.OnClickListener(){
```

```
    public void onClick(View v) {
            // Perform action on click
            //textIn.setText("test");
            //txt2.setText("text2");
            //task.execute(null);

            str="test";
           try {
                             client();
                             //txt1.setText(modifiedSentence);
                    } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                    }

             }
});

bt3.setOnClickListener(new View.OnClickListener(){
    public void onClick(View v) {
            // Perform action on click
            //textIn.setText("test");
            //txt2.setText("text2");
            //task.execute(null);

            str="humi";
            try {
                            client();
                            //txt1.setText(modifiedSentence);
                    } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                    }

    }
});

    bt4.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {
                    // Perform action on click
                    //textIn.setText("test");
                    //txt2.setText("text2");
                    //task.execute(null);
            txt1.setText("null");
            txt5.setText("null");

            }

    });





}


public void client() throws IOException{
```

```
        DatagramSocket client_socket = new DatagramSocket(2362);
        InetAddress IPAddress =  InetAddress.getByName("10.80.1.95");

        //while (true)
        //
{
                send_data = str.getBytes();
                //System.out.println("Type Something (q or Q to quit): ");

            DatagramPacket send_packet = new DatagramPacket(send_data,str.length
(), IPAddress, 2362);
            client_socket.send(send_packet);
          //chandra
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
                client_socket.receive(receivePacket);
                modifiedSentence = new String(receivePacket.getData());
                //System.out.println("FROM SERVER:" + modifiedSentence);
                if(modifiedSentence.charAt(2)=='%')
                 txt5.setText(modifiedSentence.substring(0, 3));
                else
                        txt1.setText(modifiedSentence);
                modifiedSentence=null;
                client_socket.close();

          // }

        }

 }
```

Android WDAndrolib

Android sample WDAndrolib test

'***(Android supported modules)***' Android program, This is a library project which demonstrates how to access C/C++ code from java using JNI. Here WD driver written in C is made available to java apis. This library is used with WatchDogDemo application in this folder.

Test files

This sample program contains several files, the /src folder contains the source files.

WDAndrolib test sample application

The Android Render a Moving Sprite Test sample application can be found here: WDAndroLib.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of WDLib.java file:

```
package com.digi.wdandrolib;

public class WDLib {


  static {
    System.loadLibrary("WDAndro");
  }
  /**
    * Open() will initialize WatchDog Timer on module.
    * If not tick'ed module will restart in default timeout value.
    * @return On success a Handler for WatchDog Timer.
    *          On failure < 0
    */
    public native int open();

  /**
    * setTimeOut() will set a timeout value for WatchDog Timer.
    * @param fd Handler returned from open().
    *
    * @param wdTimeOut TimeOut Value in seconds which WatchDog Timer will expire.
    *
    * @return On success Zero. On failure -1 is returned.
    */
  public native int setTimeOut( int fd, int wdTimeOut);

  /**
    * keepAliveFor() is used to reset WatchDog Timer to zero.
    * In other words for "keepalive" seconds module will not reset.
    * @param fd Handler returned from open().
    * @param keepalive Time in seconds where module is kept alive is not
rebooted.
    * @return On success zero.
    *         On failure -1.
    */
  public native int keepAliveFor(int fd, int keepalive);

}
```

Android WatchDogDemo

Android sample WatchDogDemo test

(Android module CCWMX53) Android program, Demonstrates how to access Watch Dog in Android on Digi modules.

Test files

This sample program contains several files, the /src folder contains the source files.

WatchDogDemo test sample application

The Android Watch Dog DemoTest sample application can be found here: WatchDogDemoHome.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of WatchDogDemoHome.java file:

```
package com.digi.wddemo;

import com.digi.wdandrolib.WDLib;

import android.app.Activity;
import android.os.Bundle;
import android.os.CountDownTimer;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.TextView;

public class WatchDogDemoHome extends Activity {

        private Button btnWDtest;
        private Spinner timeOutSpinner;
        private Spinner keepAliveSpinner;
        private int timeOut;
        private int keepAliveValue;
        private int wdHandler;
        private TextView counterTextField;
        private TextView welcomeTextField;
        private int initialized = 0;

        WDLib wdObject = new WDLib();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.home);

        btnWDtest = (Button) findViewById(R.id.startWDbutton);
        timeOutSpinner = (Spinner) findViewById(R.id.timeOutSpinner);
        keepAliveSpinner = (Spinner) findViewById(R.id.KeepalivesSpinner);
        counterTextField = (TextView) findViewById(R.id.timeRemaining);
        welcomeTextField = (TextView) findViewById(R.id.wText);

        configureGUI();
        configureWD();
    }
    void configureGUI() {
        timeOutSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {
            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                timeOut = Integer.parseInt(timeOutSpinner.getSelectedItem
().toString());
                Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > timeout is
:" + timeOut);
            }
            public void onNothingSelected(AdapterView<?> parent) {
                timeOut = 15;
            }
        });
```

```
        keepAliveSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {

            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                keepAliveValue = Integer.parseInt
(keepAliveSpinner.getSelectedItem().toString());
               Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > keepalive is
:" + keepAliveValue);
        }
        public void onNothingSelected(AdapterView<?> parent) {
            keepAliveValue = 15;
        }
    });
}

public void configureWD() {
    btnWDtest.setOnClickListener(
      new OnClickListener() {
              public void onClick(View view) {

                      if (initialized == 0){

                             /*
                              * Opening watchdog
                              * */
                              wdHandler = wdObject.open();
                              initialized = 1;
                              Log.d("WDAndro", ".WatchDogDemoHome.configureWD() >
fd is :" + wdHandler);
                              if(wdHandler > 0){
                                      /*
                                       * Setting timeout for watchdog
                                       * */
                                       if(wdObject.setTimeOut(wdHandler, timeOut)
== 0)
                                               Log.d("WDAndro",
".WatchDogDemoHome.configureWD() > timeout is :" + timeOut  + " seconds");
                                       else
                                               Log.d("WDAndro", "Failed to set
Timeout");
                             }

                             /*
                              * A countdown timer to show remaining time for
watchdog to reboot
                              * */
                              new CountDownTimer(keepAliveValue * 1000, 1000) {
                                public void onTick(long millisUntilFinished) {
                                       counterTextField.setText("Keepalive for: "
+ millisUntilFinished / 1000 +" seconds");
                                       Log.d("WDAndro",
".WatchDogDemoHome.configureWD() > keepalive for :" + millisUntilFinished/1000  +
" seconds");
                              }

                                public void onFinish() {
                                       counterTextField.setText("Rebooting in "+
```

```
timeOut +" seconds");
                              }
                            }.start();

                            /*
                             * Spawning a seperate timer for ticking watchdog
so that this process wont hang UI
                             * */
                             new CountDownTimer(keepAliveValue * 1000, 1000) {

                                public void onTick(long millisUntilFinished) {
                                        /*
                                         * Ticking watchdog for a second
                                         * */
                                      wdObject.keepAliveFor(wdHandler, 1);
                                }

                                public void onFinish() {
                                }
                             }.start();

                        }

                }
        });

    }


}
```

Cellular tools

Simple traffic generator

Send a single UDP packet per time period

This simple script sends a UDP packet to a remote IP at fixed time-periods, and can be used to hold open a VPN tunnel which auto-closes when idle.

```
            # Simple UDP client to push some nonsense data

from socket import *
import time

# put the IP address to UDP to here: safer NOT to use DNS
HOST = '192.168.196.6'

# put a valid port number here - 7 is echo server, usually disabled these days
# but we won't be expecting an answer anyway
PORT = 7

# put something to send here
DATA = "Hi"

# put time to delay here - is in seconds, so 5 * 60 = once per 5 minutes
# the expression will have Python calc the seconds for us
SLEEP_TIME = 5 * 60

while 1:
# we recreate, close and free up socket every time
# this is more robust if the time delay is large
udpSock = socket(AF_INET, SOCK_DGRAM)
print "Sending data <%s>" % DATA
udpSock.sendto(DATA,(HOST,PORT))
udpSock.close()

time.sleep(SLEEP_TIME)
```

Using Digi Realport with Python

Digi Realport is a set of operating system drivers which make remote IP-based serial ports appear as local physical ports. Traditionally Digi Realport uses TCP/IP only and talks to a very special low-level driver in Digi products. Unfortunately, at present these low-level drivers in products such as the X4 and X8 gateways literally expects to talk to the hardware serial ports. Thus there is no way to connect standard Digi Realport to a Python script.

However the latest versions of Digi Realport for Windows has added a UDP mode which ONLY moves serial data, emulating a 3-wire RS-232 cable. The data it sends is well packed into a single UDP packet and works very well with protocols such as Modbus or Rockwell DF1. Since none of the Digi Realport protocol is included, any Python script can wait on the UDP socket and interact with a remote Windows-based host application.

Supported products

Digi Realport for Windows (such as P/N 40002549_xx.zip) Older versions or versions for a different OS may not have UDP support.

Digi products which support Python Zmatrix.

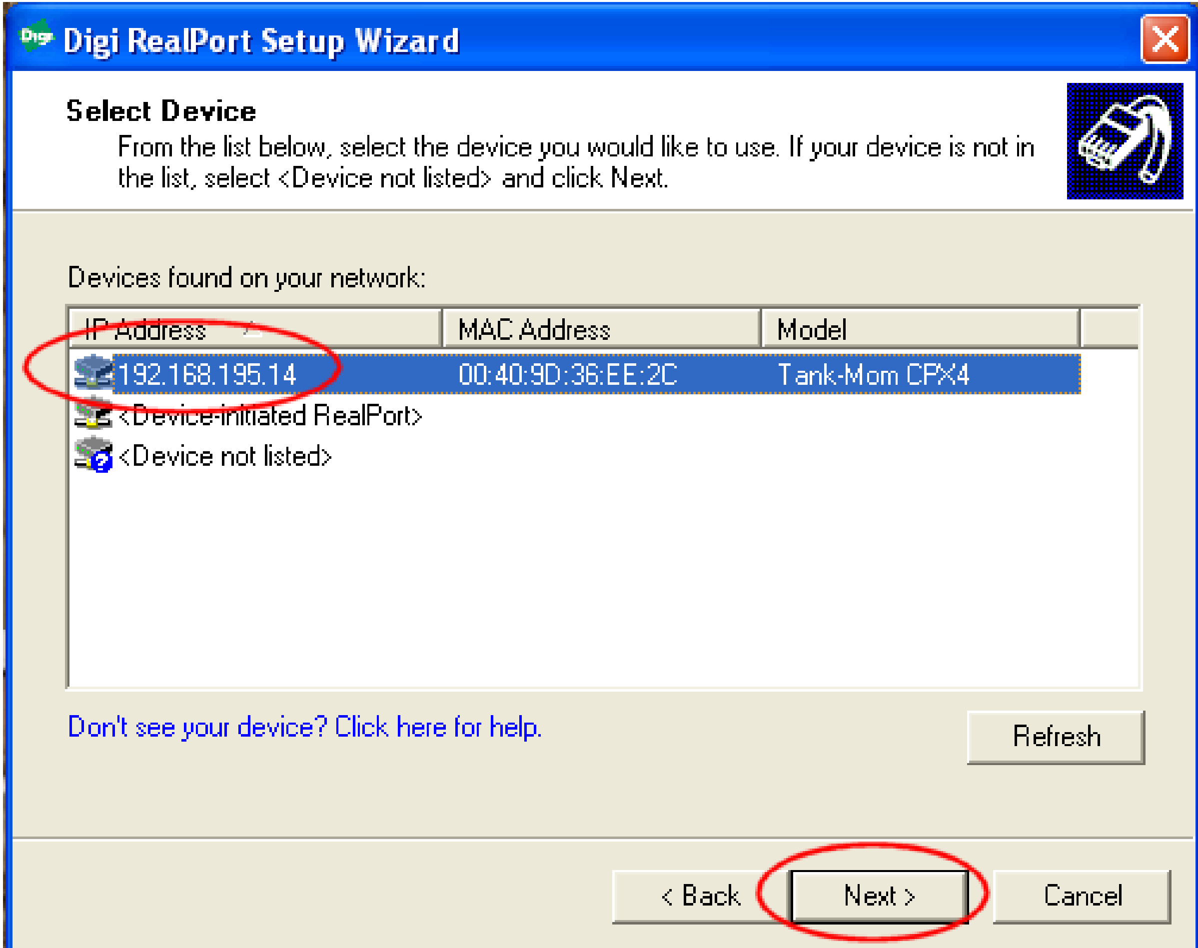
Setting up Digi Realport under Windows

Download the latest version of Digi Realport. This March_2010_ver.4.4.365.0_Realportdriver.zip is for Microsoft Windows XP/2003/Vista/2008 (both 32/64 bit arch).

CLICK HERE to find Realport for another Operating System or verify you have the latest Microsoft Windows version.

Installation

Unpack (unzip) the files in a suitable directory. Run the SETUP.EXE and you should see this this display:

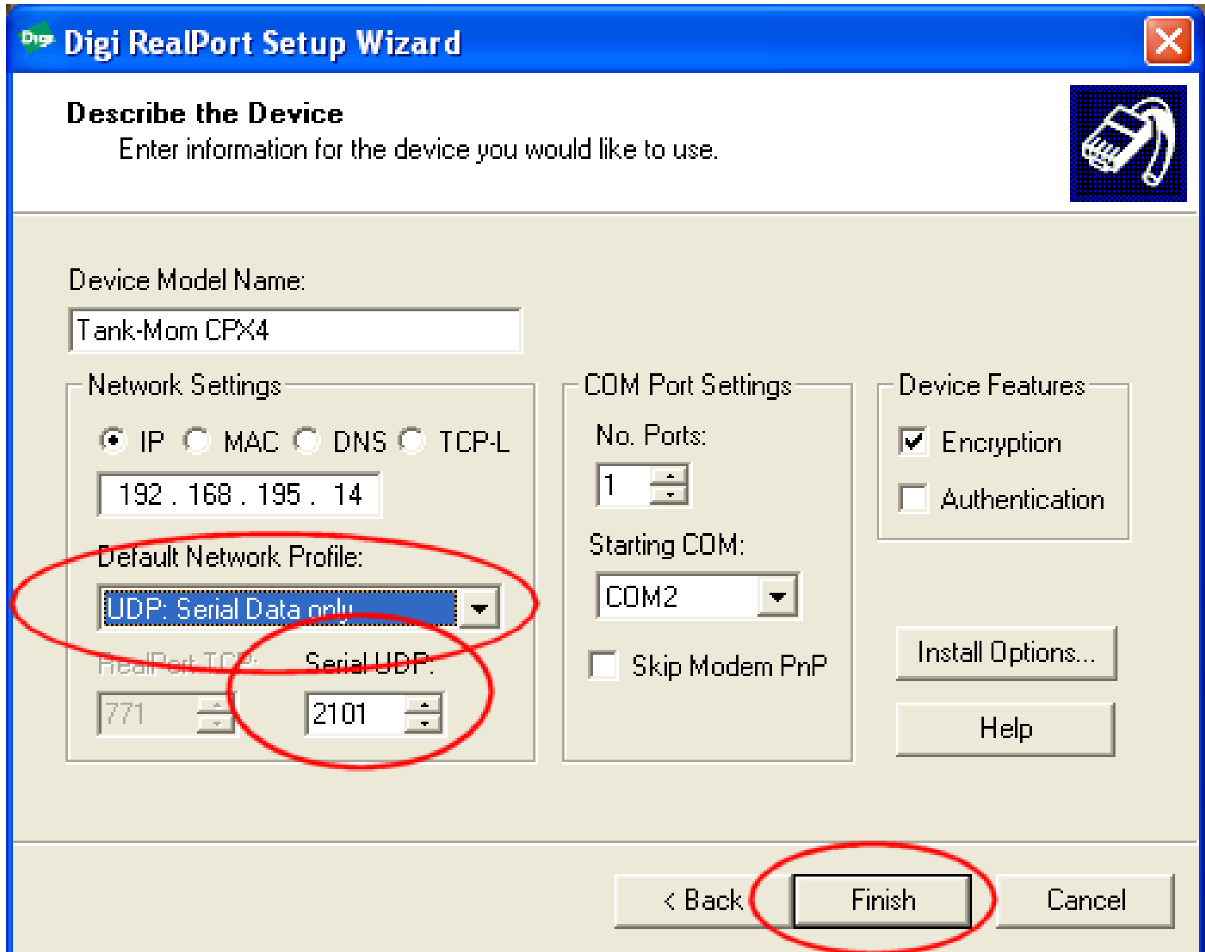


It is easiest to install Digi Realport with your device (or one of the same model) sitting next to you on your local Ethernet. This way the setup program should find it by browsing, plus learn all of the correct capabilities automatically - once installed, it is easy to change the IP address if your device is remotely located over wide-area-network such as cellular. In the example above, we'll be enabling Realport to a Digi ConnectPort X4 gateway. If you don't see your product listed, it might not have a proper IP address configured or your Windows firewall is blocking the UDP multi-cast being used. Always temporarily disable your Windows firewall when looking for LOST Digi devices, since they will reply to the Wizard with unreachable or even NULL (0.0.0.0) IP addresses and firewalls will always discard such 'mal-formed' UDP packets.

When you see this display, then:

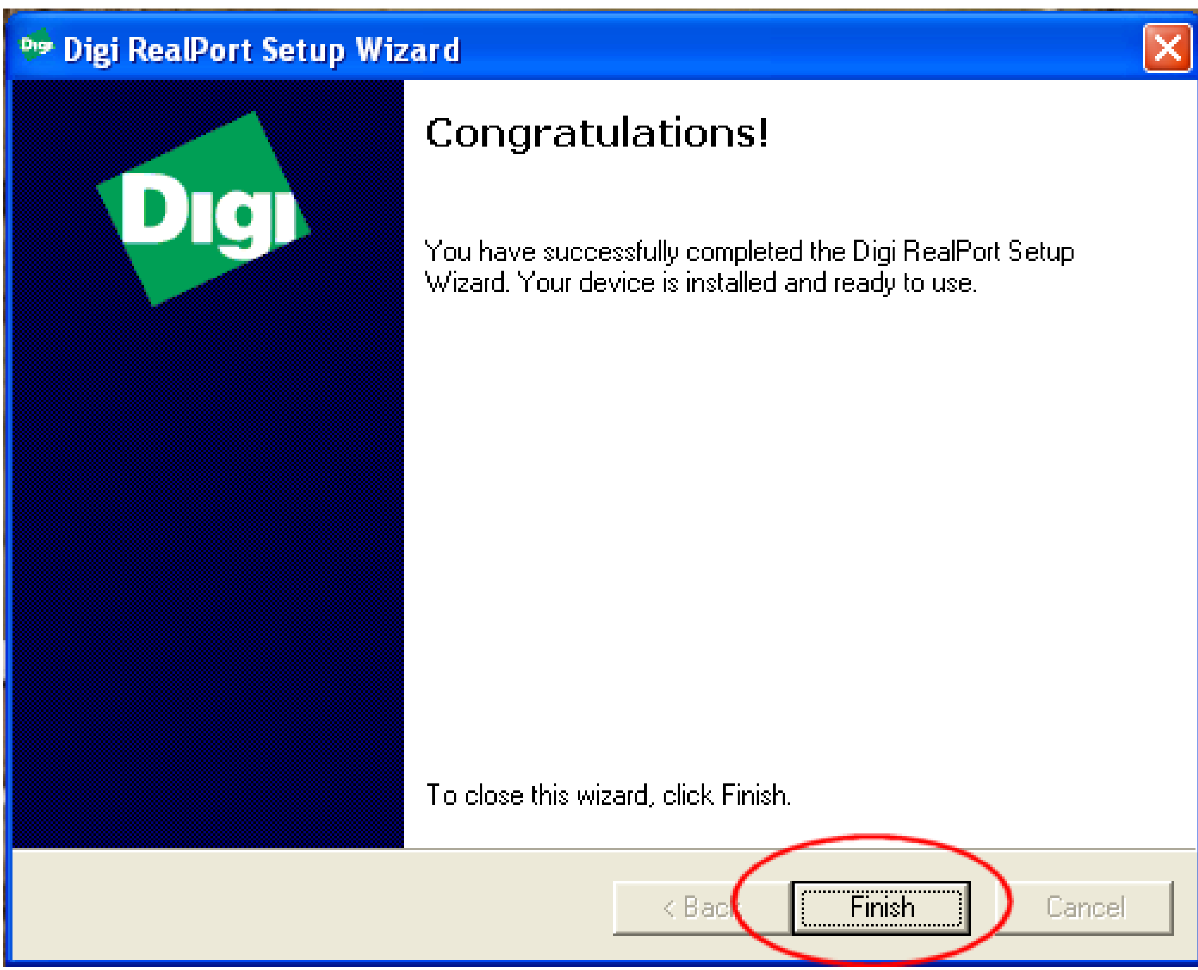
1. Select the device to configure.
2. Click **Next**.

   You should the see this display, where you can configure the basic features to use:



3. Change the Default Network Profile to **UDP: Serial Data only**. This changes Digi Realport to use UDP/IP instead of TCP/IP, plus the Serial Data only warning means all control signals, the ability to change baud rate (etc) is lost in this mode. Digi Realport in UDP mode literally mimics a 3-wire RS-232 line with only Txd, Rxd and signal ground lines.
4. Set the appropriate UDP port number to target as destination - the default of 2101 is likely okay.
5. Tweak other settings as desired. In this display we are configuring COM2 - you could move this to COM6 or other values. Also, with Realport in UDP mode things like Encryption and Authentication are NOT usable, even if the device supports it.
6. Click **Finish**.

   You should see the setup now do some work, run through a few progress displays and finally show this display:

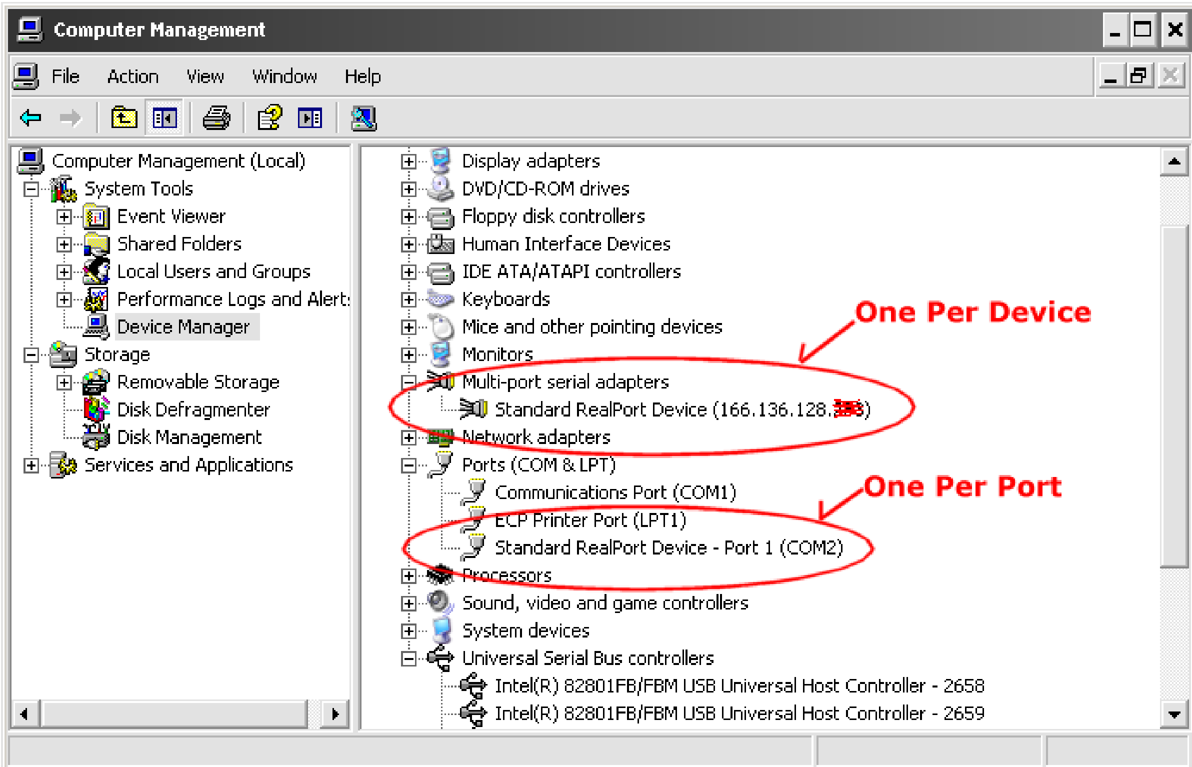


At this point, the computer will send any serial data written by a Windows application to COM2 to the IP address 192.168.195.14 in UDP packets to port 2101.

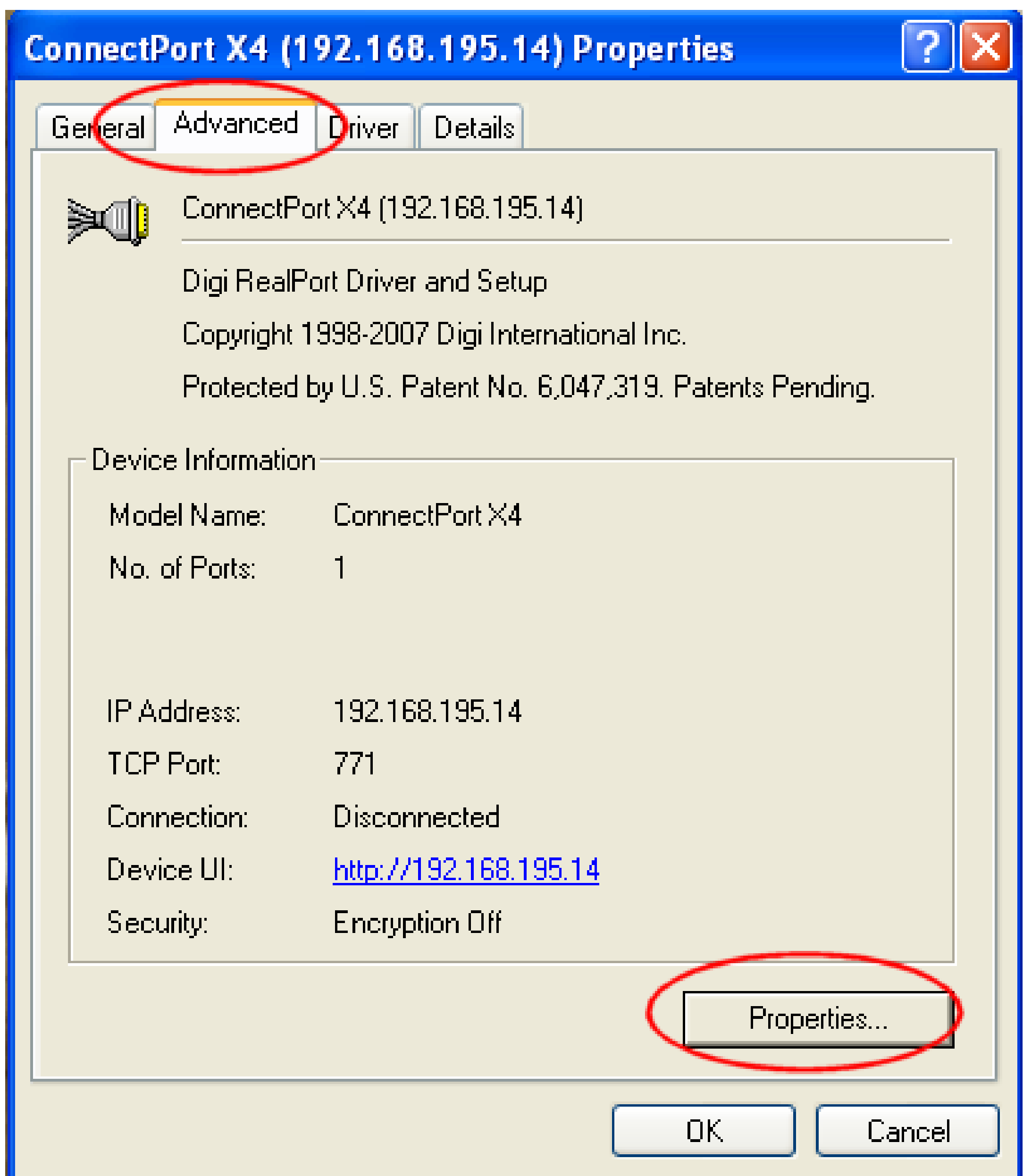
Changing settings within Realport in UDP mode

After you've installed Digi Realport, you can change setting through the Device Manager. You can open it several ways:

1. Right click My Computer, click Manage, Click Device Manager
2. Open System Properties, click the Hardware Tab, click the Device Manager



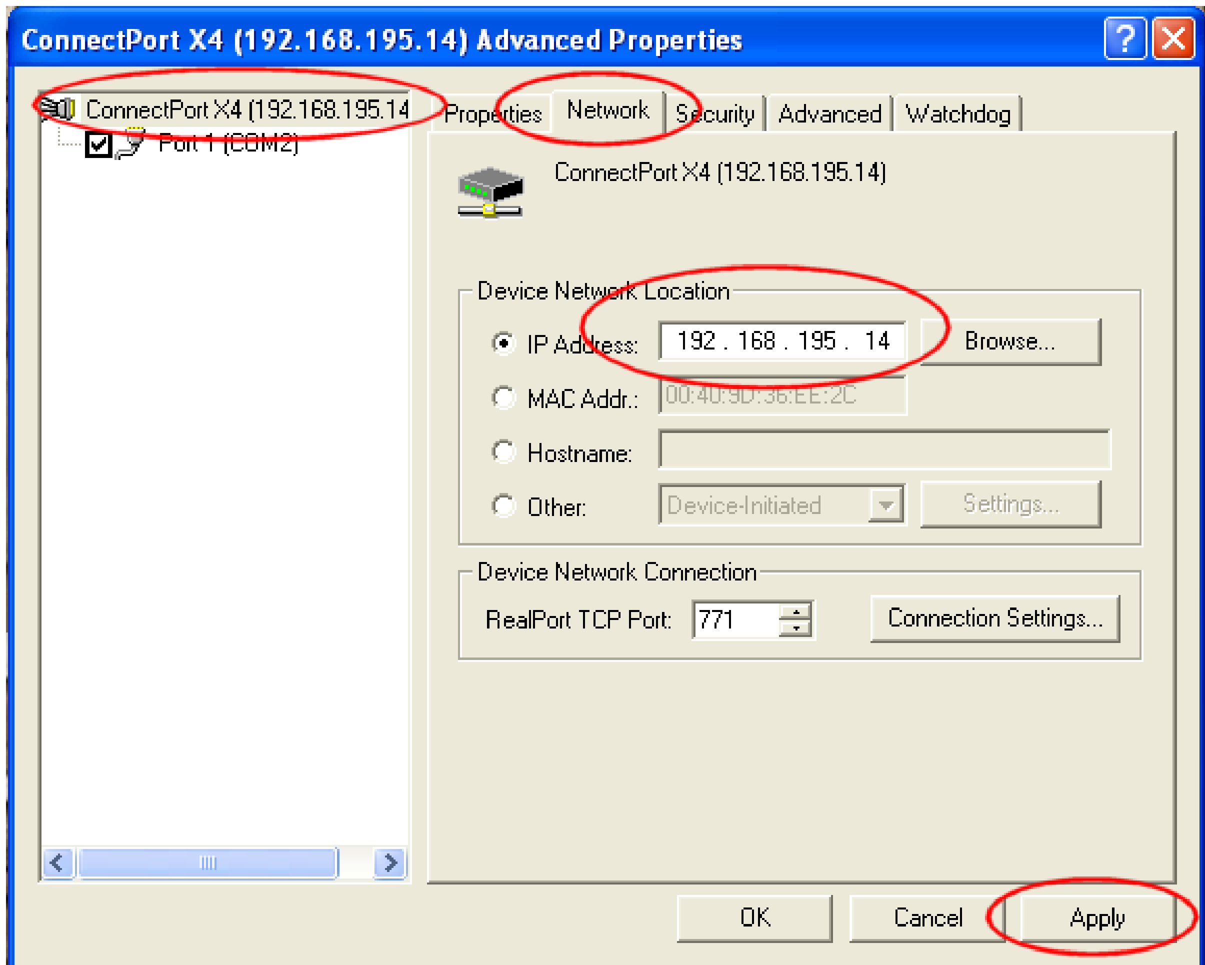
The Digi Realport device installs itself as a Multi-Port Adapter, right click it and select Properties. You will see this display:



Select the Advanced Tab, and select the Properties button. You will notice the Connection status is Disconnected - this is true since this is UDP/IP, not TCP/IP.

To Change the IP address

Select your Device (in this case ConnectPort X4) in the left panel and not the Port, then select the Network tab. Here you can select to use and change an IP address, or you can change to use a DNS Hostname. So while in this case the device was installed locally with a non-routing IP of 192.168.195.14, you could change it here to be the actual public IP such as 166.x.x.x. Remember to hit Apply when done.

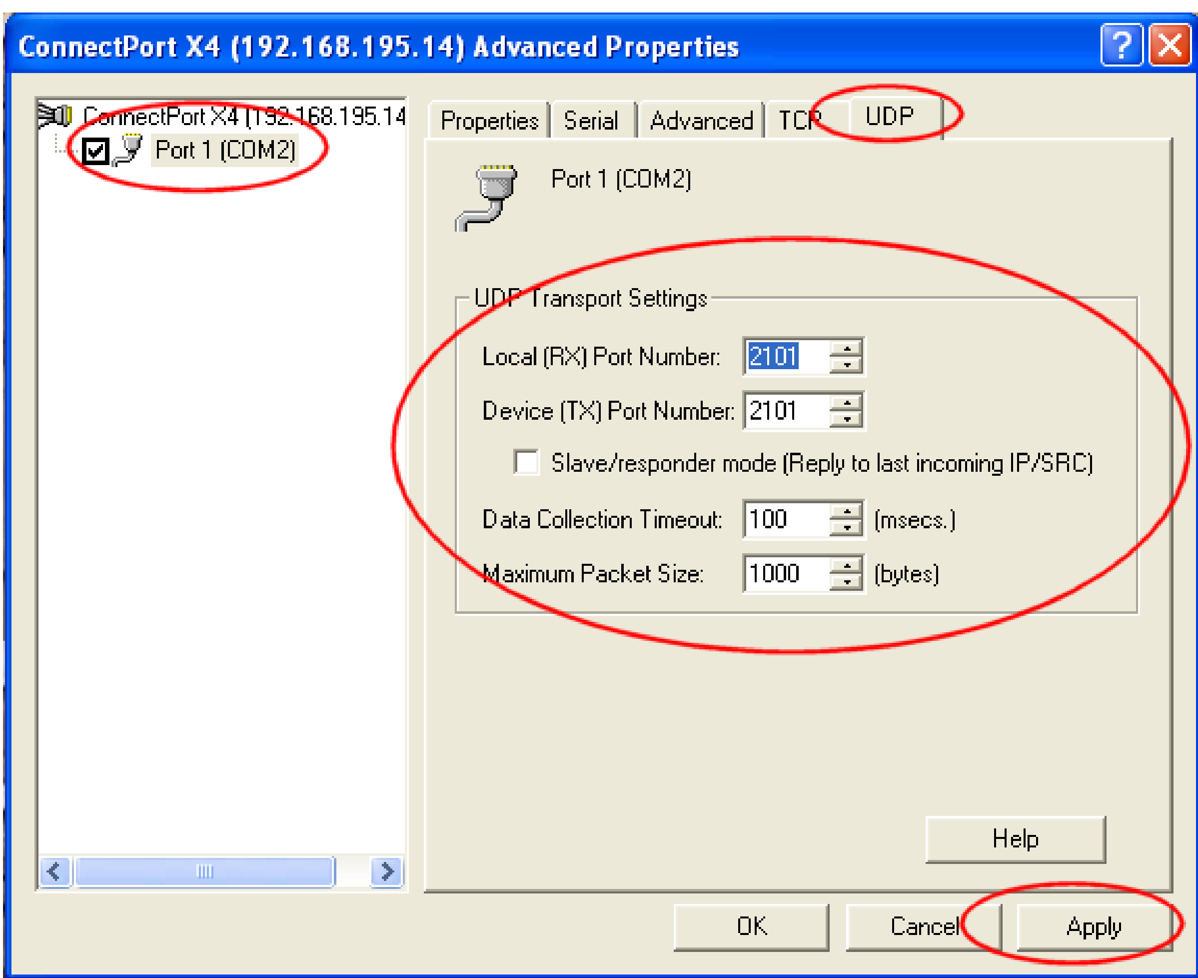


To Change the UDP settings

1. Select the **Port 1**, then the tab. The COM2 shown here will match the port you installed Realport on. Pressing the HELP button will - surprise - show you some fairly complete help information.

   The Local (RX) Port is the UDP source port within UDP packets, while the Device (TX) Port is the UDP destination port. In this example, the two numbers of 2101 match, but they can be any valid port numbers. The default is ideal for Windows-based clients polling the remote Digi product configured in UDP Sockets serial port profile. It sends data from the application to the Digi, and it must return it to the Local (RX) Port number.

   If the Digi product is a WANIA, CPX4 or DOIAP with the IA Engine active sending requests to the Windows host, then check the box labeled as Slave/responder mode, which disables the Device (TX) Port. In this mode, Digi realport remembers the UDP source/destination information to send the responses back to the Digi product.



2. Click **Apply** when you are done.

Code samples

This category is a set of pages that contain small demonstration samples that focus on a particular interface on a Digi device.

Subcategories

There are two subcategories to this cataegory.

- EmbeddedLinux - time sample
- EmbeddedLinux - UDP server-client

ADC values

Program to receive ADC values from ZigBee

(ZigBee routers and sensors) Receive and parse the ADC values coming from the ZigBee routers and sensors.

How does it work?

This application receives data packets from routers and end devices and prints the parsed output to the console. These data packets include Light, Temperature and Humidity values.

Test files

This sample program contains two files, Get_Router_Sensor_reading.py and RouterSensor_reading_application_notes.doc. The program file is Get_Router_Sensor_reading.py.

ADC value test sample application

The ADC Value sample application can be found here: Get_Router_Sensor_reading.zip.

Basic usage

See Application Note here: RouterSensor_reading_application_notes.zip.

Sample of Get_Router_Sensor_reading.py file:

##

```
# Copyright (c)2012, Digi International (Digi). All Rights Reserved.      #
#                                                                          #
# Permission to use, copy, modify, and distribute this software and its   #
# documentation, without fee and without a signed licensing agreement, is #
# hereby granted, provided that the software is used on Digi products only #
# and that the software contain this copyright notice,  and the following  #
# two paragraphs appear in all copies, modifications, and distributions as #
# well. Contact Product Management, Digi International, Inc., 11001 Bren   #
# Road East, Minnetonka, MN, +1 952-912-3444, for commercial licensing     #
# opportunities for non-Digi products.                                     #
#                                                                          #
# DIGI SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED   #
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A          #
# PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, #
# PROVIDED HEREUNDER IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND. #
# DIGI HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,         #
# ENHANCEMENTS, OR MODIFICATIONS.                                          #
#                                                                          #
# IN NO EVENT SHALL DIGI BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,      #
# SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,   #
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF   #
# DIGI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.                #
############################################################################
```

```
import os
import sys
import socket
import struct
```

```
import time
import zigbee
import datetime
import traceback
from socket import *

humidity = 0
light = 0
temp = 0

def parse_packet(payload):
    print "parser function"

def calc_light(value):
    mv = float(value * 1200) / 1023
    return mv

def calc_temp(value):
    mv = float(value * 1200) / 1023
    degc = ((mv - 500.0) / 10.0) - 4.0
    #degf = (degc * 1.8) + 32.0
    return degc

def calc_humidity(value):
    mv = (value / 1023.0) * 1200
    humidity = (((mv * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
    return humidity

try:

    sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)
    ''' Bind to ("", 0xe8, 0xc105, 0x92)to receive all incoming 'IS' responses
    and decode them per the XBee documentation
    '''
    sd.bind(("", 0xe8, 0xc105, 0x92))
    print "socket is bound"

    while 1:
        try:
            payload, src_addr = sd.recvfrom(255)
            len_payload = len(payload)

            print datetime.datetime.now()
            src = src_addr[0][1:24]
            print "Source Address: %s" %src

            #print len_payload
            if len_payload == 8:
                fixed_byte, digital_bits, analog_set, lig, tmp = struct.unpack
(">bhbhh", payload[:8])
                light = calc_light(lig)
                temp = calc_temp(tmp)
                print "light - %s" %light
                print "temperature - %s" %temp

            elif len_payload == 10:
                b1,b2,b3,b4,b5,b6,lig,tmp = struct.unpack(">bbbbbbhh", payload
[:10])
```

```
                print b1
                print b2
                print b3
                print b4
                print b5
                print b6
#                print b7
#                print b8
#                print b9
#                print b10
#                print "fixed_byte is %d" %fixed_byte
#                print "digital_bits is %d" %digital_bits
#                print "analog_set is %d" %analog_set
                print "lig is %d" %lig
               print "temp is %d" %tmp
               light = calc_light(lig)
               temp = calc_temp(tmp)
               print "light - %s" %light
               print "temperature - %s" %temp


           elif len_payload == 12:
               print "sensor"
               print len_payload
               fixed_byte, digital_bits, analog_set, dont_know_bit1, dont_know_
bit2, lig, tmp, humidity = struct.unpack(">bhbbbhhh", payload[:12])
               light = calc_light(lig)
               temp = calc_temp(tmp)
               humidity = calc_humidity(humidity)
               print "light is %d" %light
               print "temp is %d" %temp
               print "humidity is %d" %humidity
           else:
               print "invalid payload"
               print len_payload

           print " "

        except Exception, e:
            print "Exception %s" %e
            traceback.print_exc()

except socket.error, s:
    print "socket exception: %s" %s

sd.close()
print "socket is closed"
```

ASM assembly code

Program to execute assembly code in NET+OS

ASM TEST (For NET+OS 7.4.2 - 7.5.2 modules) To showcase how to execute Assembly code in NETOS.

How does it work?

Test files

This sample program contains two files, root.c and appconf.h. The main function is in root.c.

ASM test sample application

The ASM Test sample application can be found here: ASMSample.zip.

Basic usage

Compile, load and run program using NET+OS environment.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

void assembly_delay();

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;

}
#define LOOP_MAX (25)
void applicationStart (void)
{
           int loopIdx = 0;
            printf ("Hello World!\n");

           // continually call a routine that will execute one instruction 4
times in one asm call
           for (loopIdx = 0; loopIdx < LOOP_MAX; loopIdx++)
           {
               assembly_delay();
           }

           // now execute the same assembly directly
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");

          // now do it in one call
          asm volatile(
          "mov         r0, r0\n\t"
          "mov         r0, r0\n\t"
          "mov         r0, r0\n\t"
          "mov         r0, r0\n\t"
          );


          printf("Test done\n");


          tx_thread_suspend(tx_thread_identify());
```

```
    }

        void assembly_delay()
              {

                 asm volatile("mov   r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t");
              }
  }
```

Accelerometer sample

This page contains a simple demonstration of the accelerometer device on Python enabled Digi devices. It is meant to be instructive on how to use the functionality.

This sample assumes a basic understanding of accelerometer principles and vector math. For information on accelerometers and how they work, go here: accelerometer primer.

```
            ## From the digihw embedded library, import the accelerometer
from digihw import accelerometer

## Import the Queue library to act as a thread safe object between callback and
## the main thread.
from Queue import Queue

## Create a empty Queue object
sample_queue = Queue()

## Get a handle from the accelerometer
accel = accelerometer()

## print a sample.
print accel.sample()

## Define a callback function when the threshold is crossed.
## Have it append the sample to the queue

def callback(sample, sample_queue):
    print "Callback function was called"
    print "Sample: ", sample
    print "Argument: ", arg

    sample_queue.put(sample)

threshold_in_g = 3.0
print "Registering %.2f g threshold for accelerometer" %threshold_in_g

accel.register_threshold(threshold_in_g, callback, sample_queue)

print "Please lightly shake the accelerometer"
sample = sample_queue.get()
## The sum of the Gs should be equal to one, however sometimes the total value of
Gs will have a +/-1% error
print "The sample that triggered the accelerometer was:\n", sample
```

Product compatibility

Known to work on:

- Digi ConnectPort X5 variants
- Digi X-Trak 3

Known to NOT work on:

Most Digi gateways lack the required hardware to support this code.

Advanced Device Discovery Protocol (ADDP)

What is ADDP?

ADDP (Advanced Device Discovery Protocol) is a proprietary protocol developed by Digi International that allows devices on a local network to be found regardless of their network configuration.

How does it work?

ADDP uses a client/server model. The client is the application that is searching for devices. The server is the device that is being search for.

In the simplest terms, the client application sends out a specially formatted UDP broadcast packet on the network. ADDP servers listening for the packet, will receive it, and send an ADDP response back to the client. Once this process is complete, the client can then send configuration requests to the device. These can include things like network settings, and reboot requests.

Java library

A subset of the protocol has been implemented in Java. You can find the jar file here: ADDP Library

The associated javadoc documentation can be found here: ADDP Java doc.

This library allows you to search synchronously, and asynchronously for devices on the network. You can then use it to reconfigure the device's network settings, or reboot the device.

Java sample application

A simple discovery sample application can be found here: AddpSample.zip.

Basic usage

First, instantiate the AddpClient object.

```
AddpClient addpClient = new AddpClient();
```

Next, call SearchForDevices() and check the return value. Then get the devices, and walk the hashtable.

```
if (addpClient.SearchForDevices()) {
    AddpDeviceList deviceList = addpClient.getDevices();

    Enumaration<AddpDevice> e = deviceList.elements();
    while(e.hasMoreElements()) {
        AddpDevice device = e.nextElement();

    // do something with the device here
    System.out.println(device.toString());

    // if device is not configured for DHCP, then turn it on and reboot.
    if (device.getDHCP() == 0) {
      addpClient.setDHCP(device, true, "dbps");
      addpClient.rebootDevice(device, "dbps");
    }
  }
}
```

Android animation

Android sample animation test

(Android modules, i.MX51 and i.MX53) Android program, when this application runs on the android device, it will change the background images for every 1/2 second, we can start and stop the animation using the button displayed on the application.

Test files

This sample program contains several files, the /src folder contains the source files.

Animation test sample application

The Android Animation Test sample application can be found here: Animation2.zip

Basic usage

Compile, load and run program using Android environment.

Sample of Animation2Activity.java file:

```
package animation2.test;

//import canvas.paint.CanvasActivity.DemoView;
import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.widget.*;

public class Animation2Activity extends Activity {
    /** Called when the activity is first created. */

    Button b;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.setupButton();

    }


    private void setupButton() {
                b = (Button) this.findViewById(R.id.startFAButtonId);
                b.setOnClickListener(new Button.OnClickListener() {
                        public void onClick(View v) {
                                parentButtonClicked(v);
                        }
                });
        }
```

```
private void parentButtonClicked(View v) {
            animate();
    }

private void animate() {
            ImageView imgView = (ImageView) findViewById(R.id.animationImage);
            // imgView.setVisibility(ImageView.VISIBLE);
            imgView.setBackgroundResource(R.drawable.ani);

            AnimationDrawable frameAnimation = (AnimationDrawable)
imgView.getBackground();


            if (frameAnimation.isRunning()) {
                    frameAnimation.stop();
                    b.setText("Start");
            } else {
                    frameAnimation.start();
                    b.setText("Stop");
            }
    }

public boolean onTouchEvent(MotionEvent event) {

        int eventaction = event.getAction();

        switch (eventaction)
        {
          case MotionEvent.ACTION_DOWN:              // finger touches the screen
                 this.setupButton();

                 break;
          case MotionEvent.ACTION_MOVE:             // finger moves on the screen
                  //this.setupButton();
                  break;
          case MotionEvent.ACTION_UP:                   // finger leaves the screen

              break;
       }

       return true;

    }

}
```

Android Bluetooth test - support for Android Bluetooth test

Android sample Bluetooth test

'***(Android supported modules)***' Android program, when this application runs on the android device it invokes the bluetooth interface.

Test files

This sample program contains several files, the /src folder contains the source files.

Bluetooth test sample application

The Android Animation Test sample application can be found here: BlueToothTest.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of MainActivity.java file:

```
package com.digi.bluetoothtest;

import java.io.IOException;
import java.io.OutputStream;
import java.lang.reflect.Method;
import java.util.Set;
import java.util.UUID;

import android.os.Bundle;
import android.os.Message;
import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothClass;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
        private static final int REQUEST_ENABLE_BT = 1;
        private Button _scanBlueToothButton;
        private EditText _logEditText;

        private BluetoothAdapter _bluetoothAdapter = null;
```

```
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                connectUi();

                _bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

                if (_bluetoothAdapter == null) {
                        Toast.makeText(this, "No BT adapter", Toast.LENGTH_
LONG).show();
                        return;
                }

                if (!_bluetoothAdapter.isEnabled()) {
                        Intent enableBt = new Intent(BluetoothAdapter.ACTION_
REQUEST_ENABLE);
                        startActivityForResult(enableBt, REQUEST_ENABLE_BT);
                }
                else {}
        }


        private void scanBlueToothButton_OnClick() {
                ProgressDialog dialog = ProgressDialog.show(this, "",
                        "Loading. Please wait...", true);

                _bluetoothAdapter.disable();
                _bluetoothAdapter.enable();

                _bluetoothAdapter.startDiscovery();

                dialog.dismiss();

                Set<BluetoothDevice> devices = _bluetoothAdapter.getBondedDevices
();
                StringBuilder sb = new StringBuilder();

                if (devices.size() > 0) {
                        for (BluetoothDevice device : devices) {
                                sb.append(device.getName());
                                sb.append("\n");
                                // 1e0ca4ea-299d-4335-93eb-27fcfe7fa848

                                try {
                                        Method m = device.getClass().getMethod(
                                                "createRfcommSocket", new Class[]
{ int.class });

                                        BluetoothSocket sock = (BluetoothSocket)
m.invoke(device, 1);

                                        sock.connect();

                                        OutputStream stream =
sock.getOutputStream();
                                        stream.write("Hello bt world!".getBytes
());
                                        stream.close();
```

```
                                } catch (Exception e) {
                                        e.printStackTrace();

                                }
                        }
                }
                _logEditText.setText(sb.toString());
        }


        @Override
        protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
                super.onActivityResult(requestCode, resultCode, data);

                switch (requestCode) {
                        case REQUEST_ENABLE_BT:
                                break;

                }

        }


        @Override
        public boolean onCreateOptionsMenu(Menu menu) {
                getMenuInflater().inflate(R.menu.activity_main, menu);
                return true;
        }


        private void connectUi() {
                _scanBlueToothButton = (Button) findViewById
(R.id.scanBluetoothButton);
                _scanBlueToothButton.setOnClickListener(new View.OnClickListener
() {
                        public void onClick(View v) {
                                scanBlueToothButton_OnClick();
                        }
                });

                _logEditText = (EditText) findViewById(R.id.logEditText);
        }
}
```

Android HelloBox2D

Android sample HelloBox2D test

(Android supported modules) Android program, This application is a porting of the Box2D into android environment.

Test files

This sample program contains several files, the /src folder contains the source files.

HelloBox2D test sample application

The Android HelloBox2D Test sample application can be found here: HelloBox2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of HelloBox2DActivity.java file:

```
package com.digi.box2d;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;

public class HelloBox2DActivity extends Activity {
         private PhysicsWorld mWorld;
        private Handler mHandler;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mWorld = new PhysicsWorld();
        mWorld.create();

        mHandler = new Handler();
        mHandler.post(update);

    }

    @Override
    protected void onPause() {
        super.onPause();

        mHandler.removeCallbacks(update);
    }

    private Runnable update = new Runnable() {
        public void run() {
            mWorld.update();
            if (mWorld.FallingBox.isAwake()){
                 mHandler.postDelayed(update, (long) (mWorld.timeStep*1000));
            } else {
                mHandler.removeCallbacks(update);
```

```
                }
            }
        };
}
```

Android RenderAmovingSprite

Android sample render moving sprite tests

(Android supported modules) Android program, It renders a sprite on the screen and moves it.

Test files

This sample program contains several files, the /src folder contains the source files.

Render Moving Sprite Test Sample Application

The Android Render a Moving Sprite Test sample application can be found here: RenderAMovingSprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderAMovingSpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.view.Window;

public class RenderAMovingSpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                requestWindowFeature(Window.FEATURE_NO_TITLE);

                mGLSurfaceView = new GLSurfaceView(this);

                MyRenderer spriteRenderer = new MyRenderer(this);

                GLSprite sprite = new GLSprite(R.drawable.digi);

                DisplayMetrics dm = new DisplayMetrics();

                getWindowManager().getDefaultDisplay().getMetrics(dm);

                Grid spriteGrid = null;

                // Setup a quad for the sprite to use. All sprites will use the
                // same sprite grid instance.
                spriteGrid = new Grid(2, 2, false);
                spriteGrid.set(0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, null);
                spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
                spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
                spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);
```

```
                sprite.x = 100;
                sprite.y = 150;
                sprite.width = 64;
                sprite.height = 64;

                sprite.setGrid(spriteGrid);

                Runtime r = Runtime.getRuntime();
                r.gc();

                spriteRenderer.sprite = sprite;
                spriteRenderer.setVertMode(true, true);
                mGLSurfaceView.setRenderer(spriteRenderer);

                setContentView(mGLSurfaceView);

                Thread gameThread = new Thread(new Game(sprite, this));

                gameThread.start();
        }
}
```

Android RenderAsprite

Android sample Render a sprite test

(Android supported modules) Android program, Renders a stationary sprite on the screen.

Test files

This sample program contains several files, the /src folder contains the source files.

Render a sprite test sample application

The Android Render a Sprite Test sample application can be found here: RenderASprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderASpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;

public class RenderASpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mGLSurfaceView = new GLSurfaceView(this);

        MyRenderer spriteRenderer = new MyRenderer(this);

        GLSprite sprite = new GLSprite(R.drawable.digi);

        DisplayMetrics dm = new DisplayMetrics();

        getWindowManager().getDefaultDisplay().getMetrics(dm);

        Grid spriteGrid = null;

        // Setup a quad for the sprite to use.  All sprites will use the
        // same sprite grid instance.
        spriteGrid = new Grid(2, 2, false);
        spriteGrid.set(0, 0,  0.0f, 0.0f, 0.0f, 0.0f , 1.0f, null);
        spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
        spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
        spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);

        sprite.x = 100;
        sprite.y = 100;
        sprite.width = 64;
        sprite.height = 64;
```

```
        sprite.setGrid(spriteGrid);

        Runtime r = Runtime.getRuntime();
        r.gc();

        spriteRenderer.sprite = sprite;
        spriteRenderer.setVertMode(true,true);

        mGLSurfaceView.setRenderer(spriteRenderer);

        setContentView(mGLSurfaceView);


    }
}
```

Android Test2D

Android sample Test2D test

(Android supported modules) Android program, a 2D rendering test using canvas.draw.

Test files

This sample program contains several files, the /src folder contains the source files.

Test2D test sample application

The Android Render a Moving Sprite Test sample application can be found here: Test2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of Test2DActivity.java file:

```
package com.digi.test2d;

import android.app.Activity;
import android.os.Bundle;
import android.view.Display;
import android.view.Window;
import android.view.WindowManager;

public class Test2DActivity extends Activity{
        private drawView view;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                        WindowManager.LayoutParams.FLAG_FULLSCREEN);

        Art.loadBitmaps(getResources());

        Display display = getWindowManager().getDefaultDisplay();

                view = new drawView(this, display.getWidth(), display.getHeight
());

        setContentView(view);
    }
}
```

Android UDP client

Android sample for UDP client test

(Android modules i.MX51 and i.MX53) Android program, when this application runs on the android device, it will show "temp" and "humi" buttons on the android UI, and as we click on those buttons it will communicate with the UDPserver.

Test files

This sample program contains several files and the /src folder contains the source files.

UDP Claient Test Sample Application

The Android UDP Client Test sample application can be found here: AndroidUDPClient.zip.

Basic usage

Sample of ChatServerActivity.java file:

```
package test.chat.serv;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Iterator;





import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class ChatServerActivity extends Activity {
    private static final String host = null;
        private int port;
        String str=null;
        /** Called when the activity is first created. */
        TextView txt5,txt1;
        byte[] send_data = new byte[1024];
        byte[] receiveData = new byte[1024];
```

```
        String modifiedSentence;
        Button bt1,bt2,bt3,bt4;


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txt1   = (TextView)findViewById(R.id.textView1);
        txt5   = (TextView)findViewById(R.id.textView5);

        bt1 = (Button) findViewById(R.id.button1);
        bt2 = (Button) findViewById(R.id.button2);
        bt3 = (Button) findViewById(R.id.button3);
        bt4 = (Button) findViewById(R.id.button4);
        //textIn.setText("oncreate");




        bt1.setOnClickListener(new View.OnClickListener(){
                public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);
                str="temp";
                 try {
                                client();
                                //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                }
        }



    });

bt2.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);

                str="test";
               try {
                                 client();
                                 //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                 }
    });
```

```
    bt3.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);

                str="humi";
                try {
                                client();
                                //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

        }
    });

        bt4.setOnClickListener(new View.OnClickListener(){
                public void onClick(View v) {
                        // Perform action on click
                        //textIn.setText("test");
                        //txt2.setText("text2");
                        //task.execute(null);
                txt1.setText("null");
                txt5.setText("null");

                }

        });





    }


    public void client() throws IOException{


        DatagramSocket client_socket = new DatagramSocket(2362);
        InetAddress IPAddress =  InetAddress.getByName("10.80.1.95");

        //while (true)
        //
{
                send_data = str.getBytes();
                //System.out.println("Type Something (q or Q to quit): ");

            DatagramPacket send_packet = new DatagramPacket(send_data,str.length
(), IPAddress, 2362);
            client_socket.send(send_packet);
          //chandra
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
                client_socket.receive(receivePacket);
```

```
                modifiedSentence = new String(receivePacket.getData());
                //System.out.println("FROM SERVER:" + modifiedSentence);
                if(modifiedSentence.charAt(2)=='%')
                 txt5.setText(modifiedSentence.substring(0, 3));
                else
                        txt1.setText(modifiedSentence);
                modifiedSentence=null;
                client_socket.close();

          // }

        }

 }
```

Android WDAndrolib

Android sample WDAndrolib test

'***(Android supported modules)***' Android program, This is a library project which demonstrates how to access C/C++ code from java using JNI. Here WD driver written in C is made available to java apis. This library is used with WatchDogDemo application in this folder.

Test files

This sample program contains several files, the /src folder contains the source files.

WDAndrolib test sample application

The Android Render a Moving Sprite Test sample application can be found here: WDAndroLib.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of WDLib.java file:

```
package com.digi.wdandrolib;

public class WDLib {


  static {
    System.loadLibrary("WDAndro");
  }
  /**
    * Open() will initialize WatchDog Timer on module.
    * If not tick'ed module will restart in default timeout value.
    * @return On success a Handler for WatchDog Timer.
    *           On failure < 0
    */
    public native int open();

  /**
    * setTimeOut() will set a timeout value for WatchDog Timer.
    * @param fd Handler returned from open().
    *
    * @param wdTimeOut TimeOut Value in seconds which WatchDog Timer will expire.
    *
    * @return On success Zero. On failure -1 is returned.
    */
  public native int setTimeOut( int fd, int wdTimeOut);

  /**
    * keepAliveFor() is used to reset WatchDog Timer to zero.
    * In other words for "keepalive" seconds module will not reset.
    * @param fd Handler returned from open().
    * @param keepalive Time in seconds where module is kept alive is not
rebooted.
    * @return On success zero.
    *         On failure -1.
```

```
    */
  public native int keepAliveFor(int fd, int keepalive);

}
```

Android WatchDogDemo

Android sample WatchDogDemo test

(Android module CCWMX53) Android program, Demonstrates how to access Watch Dog in Android on Digi modules.

Test files

This sample program contains several files, the /src folder contains the source files.

WatchDogDemo test sample application

The Android Watch Dog DemoTest sample application can be found here: WatchDogDemoHome.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of WatchDogDemoHome.java file:

```
package com.digi.wddemo;

import com.digi.wdandrolib.WDLib;

import android.app.Activity;
import android.os.Bundle;
import android.os.CountDownTimer;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.TextView;

public class WatchDogDemoHome extends Activity {

        private Button btnWDtest;
        private Spinner timeOutSpinner;
        private Spinner keepAliveSpinner;
        private int timeOut;
        private int keepAliveValue;
        private int wdHandler;
        private TextView counterTextField;
        private TextView welcomeTextField;
        private int initialized = 0;

        WDLib wdObject = new WDLib();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.home);

        btnWDtest = (Button) findViewById(R.id.startWDbutton);
        timeOutSpinner = (Spinner) findViewById(R.id.timeOutSpinner);
```

```
        keepAliveSpinner = (Spinner) findViewById(R.id.KeepalivesSpinner);
        counterTextField = (TextView) findViewById(R.id.timeRemaining);
        welcomeTextField = (TextView) findViewById(R.id.wText);

        configureGUI();
        configureWD();
    }
    void configureGUI() {
        timeOutSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {
            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                timeOut = Integer.parseInt(timeOutSpinner.getSelectedItem
().toString());
                Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > timeout is
:" + timeOut);
            }
            public void onNothingSelected(AdapterView<?> parent) {
                timeOut = 15;
            }
        });

        keepAliveSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {

            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                keepAliveValue = Integer.parseInt
(keepAliveSpinner.getSelectedItem().toString());
               Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > keepalive is
:" + keepAliveValue);
        }
        public void onNothingSelected(AdapterView<?> parent) {
            keepAliveValue = 15;
        }
    });
}

public void configureWD() {
    btnWDtest.setOnClickListener(
      new OnClickListener() {
              public void onClick(View view) {

                      if (initialized == 0){

                             /*
                              * Opening watchdog
                              * */
                              wdHandler = wdObject.open();
                              initialized = 1;
                              Log.d("WDAndro", ".WatchDogDemoHome.configureWD() >
fd is :" + wdHandler);
                              if(wdHandler > 0){
                                      /*
                                       * Setting timeout for watchdog
                                       * */
                                       if(wdObject.setTimeOut(wdHandler, timeOut)
== 0)
                                               Log.d("WDAndro",
```

```
".WatchDogDemoHome.configureWD() > timeout is :" + timeOut  + " seconds");
                                        else
                                                Log.d("WDAndro", "Failed to set
Timeout");
                             }

                             /*
                              * A countdown timer to show remaining time for
watchdog to reboot
                              * */
                              new CountDownTimer(keepAliveValue * 1000, 1000) {
                                public void onTick(long millisUntilFinished) {
                                       counterTextField.setText("Keepalive for: "
+ millisUntilFinished / 1000 +" seconds");
                                       Log.d("WDAndro",
".WatchDogDemoHome.configureWD() > keepalive for :" + millisUntilFinished/1000  +
" seconds");
                              }

                                public void onFinish() {
                                       counterTextField.setText("Rebooting in "+
timeOut +" seconds");
                                }
                              }.start();

                              /*
                               * Spawning a seperate timer for ticking watchdog
so that this process wont hang UI
                               * */
                               new CountDownTimer(keepAliveValue * 1000, 1000) {

                                  public void onTick(long millisUntilFinished) {
                                          /*
                                           * Ticking watchdog for a second
                                           * */
                                        wdObject.keepAliveFor(wdHandler, 1);
                                  }

                                  public void onFinish() {
                                  }
                               }.start();

                        }

                }
        });

    }


}
```

Battery sample

This page contains a sample application that demonstrates the use of the battery_voltage Python API on Digi devices.

```
            ## From the digihw embedded library, import the battery_voltage
object
from digihw import battery_voltage

## Import the Queue library to act as a thread safe object between callback and
## the main thread.
from Queue import Queue

## Create a empty Queue object
sample_queue = Queue()

## Get a handle to the battery_voltage object
battery = battery_voltage()

## Get a sample
sample =  battery.sample()
print "Current battery sample: ", sample

## Define a callback when the threshold is reached.  The second argument is
## expected to be a queue object.

def callback(sample, sample_queue):
  print "Callback function was called"
  print "Sample: ", sample
  print "Placing sample in queue: ", str(sample_queue)
  sample_queue.put(sample)

## The threshold for the battery voltage is reached when the measured voltage
## drops below the set threshold.  For example, if the threshold is set to
## 10 volts, the measured voltage is 9 volts, the threshold is reached.

## Here we set the threshold to whatever the sample is + 1 volt.
## This will likely cause the threshold event to trigger.

threshold_in_v = sample + 1.0
print "Setting the voltage threshold to %.2f for the battery" %threshold_in_v

battery.register_threshold(threshold_in_v, callback, sample_queue)

print "Waiting for threshold to be triggered"
sample = sample_queue.get()

print "Sample that triggered the threshold: ", sample
```

CAN bus sample

This page contains a sample that demonstrates the usage of the CAN bus using the Python API on the Digi devices.

```
            ## Import the CAN module
from digicanbus import *
import struct, time, sys

speed = 125000
if len(sys.argv) >= 2:
    speed = int(sys.argv[1])

## The digicanbus module has CANHandle().  A function that returns a handle to
## the current CAN bus.

## Specify the CAN bus number.
print "Getting handle to CAN bus 0"
handle = CANHandle(0)

## First we configure the bus to the speed specified
print "Configuring for %d bps" %speed
handle.configure(speed)

## We create a simple callback function to use with the CAN filters.
## The callback must have the following parameters defined:
## width: specifies either 11 or 29 bit message
## identifier: The identifier that was matched
## remote_frame: Boolean indicating a remote frame (RTR)
## payload: 0-8 bytes of payload for the message
## return_arg:  Argument specified when creating the filter

## We will use the return_arg parameter to determine which filter triggered the
## callback.

def callback_1(width, identifier, remote_frame, payload, return_arg):
    print "\ncallback_1 function was called"
    print '11 or 29 bit: ', width, ' bit'
    print 'Identifier matched: ', identifier
    print 'Is remote frame: ', remote_frame
    print 'Payload: ', struct.unpack('%dB'%len(payload), payload)
    print 'Return arg: ', return_arg

## We create a tuple, which contains all the information needed for the filter
## Width:  11 or 29 bit message
## Identifier: Which CAN identifier will be selected
## Mask:  Which bits of the identifier matter when matching
## callback function:  The function will be called when something is matched
## return_arg:  Argument passed to the call back when

## We create a tuple, which contains all the information needed for the filter
## Width:  11 or 29 bit message
## Identifier: Which CAN identifier will be selected
## Mask:  Which bits of the identifier matter when matching
## callback function:  The function will be called when something is matched
## return_arg:  Argument passed to the call back when

## Note: Multiple filters can use the same callback function
```

```
## Below we are exploring different scenarios with the filters.

print "Defining filters:"
## Filter 1 will only trigger on messages with the 0x700 identifier.
## This is done by saying that the bits 0x700 must be on, and all bits will be
## measured in the mask (0x7FF).

filter_1 = (11, 0x700, 0x7FF, callback_1, 'filter_1')
print "Filter 1: ", filter_1

## Filter 2 will trigger on messages between 0x700 and 0x7FF.
## This is done by saying that the bits 0x700 must be on, but only the 0x700
## bits will be measured in the mask.

filter_2 = (11, 0x700, 0x700, callback_1, 'filter_2')
print "Filter 2: ", filter_2

## Filter 3 will trigger on all 29 bit messages.
## This is done by setting the width to 29, and using values 0x0 for identifier
## and mask.

filter_3 = (29, 0x0, 0x0, callback_1, 'filter_3')
print "Filter 3: ", filter_3

## Register the filters on the CAN bus
print "Registering filters..."
handle.register_filter(*filter_1)
handle.register_filter(*filter_2)
handle.register_filter(*filter_3)

counter = 0
print "Hit enter to send a CAN message, type 'quit' to exit"
while raw_input().lower() != 'quit':
    counter += 1
    msg = (11, counter % 0x500, False, str(counter % 99999))
    handle.send(*msg)

## Unregister the filters created using the stored tuples
print "Unregistering filters"
handle.unregister_filter(*filter_1)
handle.unregister_filter(*filter_2)
handle.unregister_filter(*filter_3)
```

ConnectPort x

Python program for ConnectPort X products

ConnectPort X Test (Python program) This example program sets hex value to KY parameter.

Test files

This sample program contains two files. File name "ReadMe.doc" and "Set_ddo_param_KY_with_hexvalues.py".

ConnectPort hex value test sample application

The Set_ddo_param_KY_with_hexvalues.py Python Test sample application can be found here: Set_ddo_param_KY_with_hexvalues.zip.

Basic Usage

Provide input in values below;

1. DESTINATION = “Provide the Extended address or OUI of the node to which KY(encryption) needs to be set”
2. Value = Hex value for the KY parameter

```
Sample code;
# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'
```

Sample of Set_ddo_param_KY_with_hexvalues.py file:

```
''' This program accomplish setting hex value to KY(encryption) parameter
    without any error.
        KY(Link Key) - Set the 128-bit AES link key.
        KY parameter take either int or string as values,
        but we cannot provide hex values, this application helps in providing
        hex values using a user defined function to the KY parameter.
'''

import sys
import os
import zigbee
import xbee
from _zigbee import *
import time
import traceback

# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'

# Converts a character string of hex digits into a byte string
def hex_str_to_bin_str(hex_string):
```

```
    start_index = 0
    if hex_string[0:2] == "0x":
        start_index = 2
    result = ""
    for index in range(start_index, len(hex_string), 2):
        byte = int(hex_string[index:index+2], 16)
        print byte
        result += chr(byte)
        print result
return result

try:
        # ddo_set_param - set a Digi Device Objects parameter value.
        # KY(Link Key) - Set the 128-bit AES link key.
        zigbee.ddo_set_param(DESTINATION, "KY", hex_str_to_bin_str(value))
        # EE(Encryption Enable)-this parameter set the encryption enable setting.
      zigbee.ddo_set_param(DESTINATION, "EE", 1)
      # WR - Write parameter values to non-volatile memory so that parameter
modifications
      # persist through subsequent resets.
          zigbee.ddo_set_param(None, "WR", 1)
      print "Writing parameters, waiting five seconds..."
      time.sleep(5)

except Exception, e:
    print "Exception %s" %e
    straceback.print_exc()

print "end of the program"
```

Detecting Cellular Status - X3 - support for detecting cellular status - X3

How to detect when the ConnectPort X3's cellular link is up?

Since the basic ConnectPort X3 has only a single IP address (and NO Ethernet port) detecting the status of the cellular connection is as easy as confirming that the IP address changes from 0.0.0.0 to something else. The cell connection is generally quite fast - becoming true within 10 to 20 seconds after Python starts executing. However, it has been observed to take up to 2 minutes in rare situations. This is largely controlled by the local cell tower situation.

In theory your code could wait 'forever' for the IP to go valid, but that is generally not a safe program solution - especially if your X3 is battery powered. It is better to call this routine with a 30 second wait time, then externally decide to optionally retry based on context.

Note that detecting cellular link is not as easy on the ConnectPort X4 and other 'routing' gateways. The code presented here will merely return the current IP Address assigned to the Ethernet port! The cellular link uses one of several 'PPP' objects.

Sample code

The following function returns either an empty string or the assigned IP address.

```
import time
import rci

def wait_for_cell_connection(tout = 0.0, check = 2.0):
    """Wait for a cell connection - for X3's only IP to become valid

    Wait 'tout' seconds, return '' if timeout; 0.0 = wait forever
    Recheck RCI call every 'check' seconds; 0.5 secs or longer
    """

    request = '<rci_request version="1.1"><query_state><boot/></query_
state></rci_request>'
    start = time.clock()
    if check < 0.5:
        # give the subsystems some CPU time
        check = 0.5

    while True:

        if tout > 0.0:
            if ((time.clock() - start) > tout):
                # print 'waited too long for IP, aborting'
                return ''

        response = rci.process_request(request)
        offs = response.find('ip_address' )
        if offs < 0:
            print 'Bad RCI Response Seen'
            # print response[:120]
        else:
            response = response[offs:]
            # result will be either:
```

```
        # ip_address>0.0.0.0</ip_address or ip_address>166.130.2.99</ip_ad
        # ... is XML, but avoid the overhead of XML parsing
        pre = response.find('>' )
        pst = response.find('<' )
        response = response[pre+1:pst]

        if(response == '0.0.0.0'):
           # print 'No IP yet - is 0.0.0.0'
            pass
        else:
            # print 'Have IP: ' + response
             return response

    # work-around for an early X3 fw issue where the next RCI call
    # may return corrupted data if this is not freed.
    del response

    # delay - give other parts of the X3 system CPU time
    time.sleep(check)
```

Product compatibility

Known to work on:

Digi ConnectPort X3, X3R

Known to NOT work on:

This code does NOT would with most Digi gateways because the it returns the Ethernet IP, not the cellular IP.

DogFighter

DogFighter sample test

(**For java supported modules**) Java program; An image rendering sample.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java DogFighter test sample application

The DogFighter Test sample application can be found here: DogFighter.zip.

Basic usage

Compile, load and run program using java environment.

Sample of DogFighter.java file:

```
package com.digi.DogFighter;

import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Cursor;
```

```
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.image.BufferStrategy;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferInt;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class DogFighter extends Canvas implements Runnable {
        private static final long serialVersionUID = 1L;

        private static final int WIDTH = 320;
        private static final int HEIGHT = 240;
        private static final int SCALE = 2;

        private boolean running;
        private Thread thread;

        private Game game;
        private Screen screen;
        private BufferedImage img;
        private int[] pixels;
//      private InputHandler inputHandler;
//      private Cursor emptyCursor, defaultCursor;
//      private boolean hadFocus = false;

        // handles the FPS counter
        private long elapsed;
        private long start_time;

        Random random = new Random();

        public DogFighter() {
                Dimension size = new Dimension(WIDTH * SCALE, HEIGHT * SCALE);
                setSize(size);
                setPreferredSize(size);
                setMinimumSize(size);
                setMaximumSize(size);

                game = new Game();
                screen = new Screen(WIDTH, HEIGHT);
                img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_
RGB);
                pixels = ((DataBufferInt) img.getRaster().getDataBuffer
()).getData();
                screen.pixels = pixels;

                // inputHandler = new InputHandler();

               // addKeyListener(inputHandler);
               // addFocusListener(inputHandler);
               // addMouseListener(inputHandler);
               // addMouseMotionListener(inputHandler);
//             emptyCursor = Toolkit.getDefaultToolkit().createCustomCursor(
//                             new BufferedImage(16, 16, BufferedImage.TYPE_INT_ARGB),
```

```
//                              new Point(0, 0), "empty");
//              defaultCursor = getCursor();
        }

        public synchronized void start() {
               if (running)
                      return;
               running = true;
               thread = new Thread(this);
               thread.start();
       }

        public synchronized void stop() {
                if (!running)
                        return;
                running = false;
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        @Override
        public void run() {
                int count = 0;
                elapsed = 0;
                start_time = System.nanoTime();
                game.init();

                //runs the game for one tick at a time
                while (running) {
                        // used to estimate FPS
                        if (count >= 200) {
                                elapsed = (System.nanoTime() - start_time) / 200;
                                start_time = System.nanoTime();
                                count = 0;
                        }

                       // run the game
                       game.doTick();

                       // render it
                       render();

                       count++;
               }

        }

        private void render() {
                // get the next frame buffer (we use triple buffering?)
                BufferStrategy bs = getBufferStrategy();
                if (bs == null) {
                        createBufferStrategy(3);
                        return;
                }

               // have the screen object render the game
```

```
                screen.render(game);

                // draw in the FPS counter
                screen.draw("FPS:" + (int) (1 / (elapsed / 1000000000.0)), 0, 0);

                // draw the screen onto the BufferedImage
                //for (int i = 0; i < WIDTH * HEIGHT; i++)
{
                //pixels[i] = screen.pixels[i];
                //}

                // get the next frame buffer
                Graphics g = bs.getDrawGraphics();
                // draw the screen to it.
                g.fillRect(0, 0, getWidth(), getHeight());
                g.drawImage(img, 0, 0, WIDTH * SCALE, HEIGHT * SCALE, null);
                g.dispose();
                // flip the buffer to the front
                bs.show();
        }


        // creates and configures the window
        public static void main(String[] args) {
                DogFighter df = new DogFighter();

                JFrame frame = new JFrame("DogFighter Sample");

                JPanel panel = new JPanel(new BorderLayout());
                panel.add(df, BorderLayout.CENTER);

                frame.setContentPane(panel);
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setResizable(false);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);

                df.start();
         }
}
```

Etherios Jenova connector

Etherios Jenova connector sample test

(**For java supported modules**) Standalone java application via which you can send linux commands to a linux host via DC.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java Ethrios Jenova connector test sample application

The Etherios Jenova Connector Test sample application can be found here: EtheriosJenovaConnector.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Base64.java file:

```
package com.etherios.jenovaconnector;

public final class Base64<b1> {
        static private final int BASELENGTH = 128;
        static private final int LOOKUPLENGTH = 64;
        static private final int TWENTYFOURBITGROUP = 24;
        static private final int EIGHTBIT = 8;
        static private final int SIXTEENBIT = 16;
        static private final int FOURBYTE = 4;
        static private final int SIGN = -128;
        static private final char PAD = '=';
        static private final boolean fDebug = false;
        static final private byte[] base64Alphabet = new byte[BASELENGTH];
        static final private char[] lookUpBase64Alphabet = new char
[LOOKUPLENGTH];
        static {
                for (int i = 0; i < BASELENGTH; ++i) {
                        base64Alphabet[i] = -1;
                }

                for (int i = 'Z'; i >= 'A'; i--) {
                        base64Alphabet[i] = (byte) (i - 'A');
                }
                for (int i = 'z'; i >= 'a'; i--) {
                        base64Alphabet[i] = (byte) (i - 'a' + 26);
                }

                for (int i = '9'; i >= '0'; i--) {
                        base64Alphabet[i] = (byte) (i - '0' + 52);
                }

                base64Alphabet['+'] = 62;
                base64Alphabet['/'] = 63;
```

```
                for (int i = 0; i <= 25; i++)
                        lookUpBase64Alphabet[i] = (char) ('A' + i);

                for (int i = 26, j = 0; i <= 51; i++, j++)
                        lookUpBase64Alphabet[i] = (char) ('a' + j);

                for (int i = 52, j = 0; i <= 61; i++, j++)
                        lookUpBase64Alphabet[i] = (char) ('0' + j);

                lookUpBase64Alphabet[62] = (char) '+';
                lookUpBase64Alphabet[63] = (char) '/';
        }

        protected static boolean isWhiteSpace(char octect) {
                return (octect == 0x20 || octect == 0xd || octect == 0xa ||
octect == 0x9);
        }


        protected static boolean isPad(char octect) {
                return (octect == PAD);
        }

        protected static boolean isData(char octect) {
                return (octect < BASELENGTH && base64Alphabet[octect] != -1);
        }

        protected static boolean isBase64(char octect) {
                return (isWhiteSpace(octect) || isPad(octect) || isData(octect));
        }

        /**
        * Encodes hex octects into Base64
        *
        * @param binaryData
        *            Array containing binaryData
        * @return Encoded Base64 array
        */
        public static String encode(byte[] binaryData) {
                if (binaryData == null)
                        return null;

                int lengthDataBits = binaryData.length * EIGHTBIT;
                if (lengthDataBits == 0) {
                        return "";
        }

        int fewerThan24bits = lengthDataBits % TWENTYFOURBITGROUP;
        int numberTriplets = lengthDataBits / TWENTYFOURBITGROUP;
        int numberQuartet = fewerThan24bits != 0 ? numberTriplets + 1
                        : numberTriplets;
        char encodedData[] = null;

        encodedData = new char[numberQuartet * 4];

        byte k = 0, l = 0, b1 = 0, b2 = 0, b3 = 0;
        int encodedIndex = 0;
        int dataIndex = 0;
```

```
        if (fDebug) {
                System.out.println("number of triplets = " + numberTriplets);
        }

        for (int i = 0; i < numberTriplets; i++) {
                b1 = binaryData[dataIndex++];
                b2 = binaryData[dataIndex++];
                b3 = binaryData[dataIndex++];

                if (fDebug) {
                        System.out.println("b1= " + b1 + ", b2= " + b2 + ", b3= "
+ b3);
                }

                l = (byte) (b2 & 0x0f);
                k = (byte) (b1 & 0x03);

                byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                                : (byte) ((b1) >> 2 ^ 0xc0);

                byte val2 = ((b2 & SIGN) == 0) ? (byte) (b2 >> 4)
                                : (byte) ((b2) >> 4 ^ 0xf0);

                byte val3 = ((b3 & SIGN) == 0) ? (byte) (b3 >> 6)
                                : (byte) ((b3) >> 6 ^ 0xfc);

                if (fDebug) {
                        System.out.println("val2 = " + val2);
                        System.out.println("k4   = " + (k << 4));
                        System.out.println("vak  = " + (val2 | (k << 4)));
                }

                encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[val2 | (k <<
4)];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[(l << 2) |
val3];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[b3 & 0x3f];
        }

        // form integral number of 6-bit groups
        if (fewerThan24bits == EIGHTBIT) {
                b1 = binaryData[dataIndex];
                k = (byte) (b1 & 0x03);

                if (fDebug) {
                        System.out.println("b1=" + b1);
                        System.out.println("b1<<2 = " + (b1 >> 2));
                }

                byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                                : (byte) ((b1) >> 2 ^ 0xc0);

                encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[k << 4];
                encodedData[encodedIndex++] = PAD;
                encodedData[encodedIndex++] = PAD;
        } else if (fewerThan24bits == SIXTEENBIT) {
                b1 = binaryData[dataIndex];
```

```
                b2 = binaryData[dataIndex + 1];
                l = (byte) (b2 & 0x0f);
                k = (byte) (b1 & 0x03);

               byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                               : (byte) ((b1) >> 2 ^ 0xc0);

               byte val2 = ((b2 & SIGN) == 0) ? (byte) (b2 >> 4)
                               : (byte) ((b2) >> 4 ^ 0xf0);

               encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
               encodedData[encodedIndex++] = lookUpBase64Alphabet[val2 | (k <<
4)];
               encodedData[encodedIndex++] = lookUpBase64Alphabet[l << 2];
               encodedData[encodedIndex++] = PAD;
        }

        return new String(encodedData);
     }

    /**
    * Decodes Base64 data into octects
    *
    * @param encoded
    *            string containing Base64 data
    * @return Array containind decoded data.
    */
    public static byte[] decode(String encoded) {
            if (encoded == null)
                    return null;

            char[] base64Data = encoded.toCharArray();

            // remove white spaces
            int len = removeWhiteSpace(base64Data);

            if (len % FOURBYTE != 0) {
                    return null;
                    // should be divisible by four
            }


            int numberQuadruple = (len / FOURBYTE);

            if (numberQuadruple == 0)
                    return new byte[0];

            byte decodedData[] = null;
            byte b1 = 0, b2 = 0, b3 = 0, b4 = 0;
            char d1 = 0, d2 = 0, d3 = 0, d4 = 0;
            int i = 0;
            int encodedIndex = 0;
            int dataIndex = 0;

            decodedData = new byte[(numberQuadruple) * 3];

            for (; i < numberQuadruple - 1; i++) {
                    if (!isData((d1 = base64Data[dataIndex++]))
                                    || !isData((d2 = base64Data[dataIndex++]))
```

```
                                    || !isData((d3 = base64Data[dataIndex++]))
                                    || !isData((d4 = base64Data[dataIndex++])))
                            return null;
                     // if found "no data" just return null
                      b1 = base64Alphabet[d1];
                      b2 = base64Alphabet[d2];
                      b3 = base64Alphabet[d3];
                      b4 = base64Alphabet[d4];

                     decodedData[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                     decodedData[encodedIndex++] = (byte) (((b2 & 0xf) << 4) | ((b3
>> 2) & 0xf));
                     decodedData[encodedIndex++] = (byte) (b3 << 6 | b4);
              }

              if (!isData((d1 = base64Data[dataIndex++]))
                            || !isData((d2 = base64Data[dataIndex++]))) {
                      return null;
                      // if found "no data" just return null
              }

              b1 = base64Alphabet[d1];
              b2 = base64Alphabet[d2];
              d3 = base64Data[dataIndex++];
              d4 = base64Data[dataIndex++];

              if (!isData((d3)) || !isData((d4))) {
                     // Check if they are PAD characters
                     if (isPad(d3) && isPad(d4)) {
                             // Two PAD e.g. 3c[Pad][Pad]
                              if ((b2 & 0xf) != 0)// last 4 bits should be zero
                                      return null;

                              byte[] tmp = new byte[i * 3 + 1];

                              System.arraycopy(decodedData, 0, tmp, 0, i * 3);
                              tmp[encodedIndex] = (byte) (b1 << 2 | b2 >> 4);

                              return tmp;
                    } else if (!isPad(d3) && isPad(d4)) {
                              // One PAD e.g. 3cQ[Pad]
                              b3 = base64Alphabet[d3];

                              if ((b3 & 0x3) != 0)// last 2 bits should be zero
                              return null;

                              byte[] tmp = new byte[i * 3 + 2];
                              System.arraycopy(decodedData, 0, tmp, 0, i * 3);
                              tmp[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                              tmp[encodedIndex] = (byte) (((b2 & 0xf) << 4) | ((b3
>> 2) & 0xf));

                              return tmp;
                      } else {
                              return null;
                              // an error like "3c[Pad]r", "3cdX", "3cXd", "3cXX"
where X is
                              // non data
                      }
```

```
            } else {
                    // No PAD e.g 3cQl
                    b3 = base64Alphabet[d3];
                    b4 = base64Alphabet[d4];

                    decodedData[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                    decodedData[encodedIndex++] = (byte) (((b2 & 0xf) << 4) |
((b3 >> 2) & 0xf));
                    decodedData[encodedIndex++] = (byte) (b3 << 6 | b4);
            }
            return decodedData;
        }

        /**
        * remove WhiteSpace from MIME containing encoded Base64 data.
        *
        * @param data
        *           the byte array of base64 data (with WS)
        * @return the new length
        */

         protected static int removeWhiteSpace(char[] data) {
                 if (data == null)
                         return 0;


                 // count characters that's not whitespace
                int newSize = 0;
                int len = data.length;

                for (int i = 0; i < len; i++) {
                        if (!isWhiteSpace(data[i]))
                                data[newSize++] = data[i];
                }

                return newSize;
       }


}
```

GPS sample

This page contains a sample of how to use the GPS API on a Digi device.

```
## The returned values from the gps_location call are NMEA parsed into a
## tuple: (latitude, longitude, altitude, timestamp)

## ConnectPort X3 /X-Trak 3 devices the returned values from the gps_location
call are NMEA parsed into
## a tuple: (latitude, longitude, altitude, timestamp, speed, direction,fix type)

## The sample returned is the latest received from the GPS device.  The
## timestamp is assigned to it when it was successfully parsed.

## In cases where there are no sample available, an exception is raised.

"""\
    GPS Location API Sample

    This example reads and displays all the values from the device's
    integrated GPS each 5 seconds, using the GPS Location API.

    Displays Latitude, Longitude, Altitude, Timestamp in GMT/GPS time
    FixType  1D 2D 3D
    Speed  in meter per second
    Direction in Degrees
"""

# imports
import sys
import os
import time
import digihw   ## From the digi embedded library, import the parsed gps_location

# variables
is_reading = True


# Ensure to set TRUE for Variants of X-Trak 3, X3 & X3R devices
# Ensure to set FALSE for X4 & X5 devices
Extended_GpsInfo = True


def get_formmated_value(value):
    """
        Returns the given value formatted in degrees, minutes and seconds.
    """
    working_value = value
    if working_value < 0:
      working_value = working_value * -1
    deg = working_value;
    gpsdeg = int(deg)
    remainder = deg - (gpsdeg * 1.0)
    gpsmin = remainder * 60.0
    remainder2 = gpsmin - int(gpsmin)*1.0
    gpsseg = int(remainder2*60.0)

    final_value = "%sdeg %smin %ssec" %(gpsdeg, int(gpsmin), gpsseg)
```

```
    return final_value

def get_latitude_hemisphere(latitude):
    """
        Returns the hemisphere depending on the latitude (S or N)
    """
    if latitude < 0:
        return "S"
    return "N"

def get_longitude_hemisphere(longitude):
    """
        Returns the hemisphere depending on the longitude (W or E)
    """
    if longitude < 0:
        return "W"
    return "E"

# Read GPS information every 5 seconds
while is_reading:
    try:
        print "Reading GPS data...\r\n"
        gps_data = digihw.gps_location()

        if Extended_GpsInfo == True:
            latitude, longitude, altitude, timestamp, Speed, Direction, FixType =
gps_data
        else:
            latitude, longitude, altitude, timestamp = gps_data


        print "Latitude  : %s %s" %(get_formmated_value(latitude),
                                     get_latitude_hemisphere(latitude))
        print "Longitude : %s %s" %(get_formmated_value(longitude),
                                     get_longitude_hemisphere(longitude))
        print "Altitude  : %d meters" %altitude
        print "Date      : % s" %time.ctime(timestamp)

        if Extended_GpsInfo == True:
            print "Speed     : %f meter/s" %Speed
            print "Direction : %d degrees" %Direction
            print "FixType   : %dD" %FixType
         # Wait 1 seconds
        print "Please, wait 1 seconds...\r\n"
        time.sleep(1)
    except:
        print "Couldn't read GPS data. You need to get a better GPS \
               signal.\nPlease, ensure the GPS antenna is correctly connected and
\
               has an open view of the sky.\r\n"

        # Wait 20 seconds
        print "Please, wait 20 seconds...\r\n\r\n"
        time.sleep(20)
```

GetConnectTankAttributes

GetConnectTankAttributes sample test

(**For java supported modules**) This application gets the connect tank attributes from the device cloud and displays it to the user.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java GetConnectTankAttributes test sample application

The GetConnectTankAttributes Test sample application can be found here: GetConnectTankAttributes.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.DCT;
//imports
import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JT
xtArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

/* Brief description of the terms used in the program.
- JPanel - JPanel is a generic lightweight container
- JLabel - A JLabel object can display either text, an image, or both
- JTextField -  JTextField is a lightweight component that allows the editing of
a single line of text
- JButton - An implementation of a "push" button when it is clicked an event
should takes place
- JTable - The JTable is used to display and edit regular two-dimensional tables
of cells
- Font - Setting Font for the Components
*/

public class Main extends JFrame {


       /**
        *
        */
        private static final long serialVersionUID = 1L;

        private JTextField txtUsername;
        private JTextField txtPassword;
        private JTable tblDevices;
        private JTable tblAttributeList;
        private DeviceListTableModel devicesTableModel;
        private AttributeTableModel AttributeListTableModel;

        private String username;
        private String password;
        private String highlightedDevice;
        public String indicator = null;
        public String login_indicator = null;
        final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
        final JLabel credentialsInforLabel = new JLabel("Enter Device Cloud
credentials, click Connect and please wait for few seconds!!");
        private static final Color cl_black = new Color(21, 45, 60);
        private static final Color cl_dkgray = new Color(110, 110, 110);
        private static final Color cl_grn = new Color(12, 130, 68);
        public Font label = new Font("Times New Roman", Font.BOLD, 15);
        final JLabel wrongDeviceInfo;
        JPanel devicesScrollPanePanel = new JPanel();
        JPanel attributeScrollPanePanel = new JPanel();
        JPanel motherPanel = new JPanel();

        JPanel basePanel = new JPanel();
        JPanel extraPanel = new JPanel();
```

```
        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public Font fontDevice = new Font("Times New Roman", Font.PLAIN, 13);
        public Font fontAttribute = new Font("Times New Roman", Font.PLAIN, 15);
        public final JLabel waitInfo = new JLabel("Please select Connect Tank
device and wait!!");


        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }
        //constructor of the Main class which is responsible for the GUI
        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 950);

                setTitle("Digi Connect Tank");

                //adding main panel to the main window
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));
                credentialsInforLabel.setSize(2, 2);
                credentialsInforLabel.setFont(font_bold);
                credentialsInforLabel.setForeground(cl_grn);

                // panel which holds the user info
                JPanel credentialsPanel = new JPanel();
                credentialsPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
5));
                credentialsPanel.add(credentialsInforLabel);
                // adding Panels to the mother panel
                motherPanel.add(credentialsPanel);
                // adding basePanel which holds other panels to main panel i.e.,
mother panel
                motherPanel.add(basePanel);
```

```
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                //userPanel which holds username and username textfield
                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
                // declaring a Label
                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);
                userLabel.setFont(label);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);

                //passPanel which holds password label and password textfield
                JPanel passPanel = new JPanel();
                //adding passPanel to basePanel
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
                passLabel.setFont(label);

                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);

                //Panel which holds URL combo box
                JPanel dropListConnectPanel = new JPanel();
                //adding panel to basic panel
                basePanel.add(dropListConnectPanel);
                final JLabel correctDeviceInfo = new JLabel("Latest values sent
by the Connect Tank to Device Cloud!");
                correctDeviceInfo.setSize(5, 5);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(cl_grn);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select Connect Tank and wait for few seconds!");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(cl_grn);

                waitInfo.setVisible(false);
                waitInfo.setSize(5, 5);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(cl_grn);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(cl_grn);
                wrongCredentialsInfo.setFont(font_bold)        ;
```

```
                //combo box which has list of URL's
                JComboBox dropDownURLList = new JComboBox();
                dropDownURLList.addItem(listURLItems[0]);
                dropDownURLList.addItem(listURLItems[1]);
                //adding URL's list to panel
                dropListConnectPanel.add(dropDownURLList);
                dropDownURLList.setSize(50,25);
                dropDownURLList.setFont(label);

                // declaring a button
                    JButton btnConnect = new JButton("Connect");
                    //adding button to the panel
                    dropListConnectPanel.add(btnConnect);
                    btnConnect.setOpaque(true);
                    btnConnect.setSize(new Dimension(50, 25));
                    //set font to button
                    btnConnect.setFont(label);

                    //adding scroll panel to main motherPanel
                    motherPanel.add(devicesScrollPanePanel
                    devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));
                    //adding user info to panel
                    devicesScrollPanePanel.add(wrongCredentialsInfo);
                    devicesScrollPanePanel.add(waitInfo);
                    //declaring a scroll pane
                    JScrollPane devicesScrollPane = new JScrollPane();
                    //adding scroll pane to panel
                    devicesScrollPanePanel.add(devicesScrollPane);
                    //table which displays all the devices in teh user account
                    tblDevices = new JTable();
                    tblDevices.setFillsViewportHeight(true);
                    tblDevices.setPreferredScrollableViewportSize(new Dimension
(800, 150));
                    devicesScrollPane.setViewportView(tblDevices);
                    tblDevices.setColumnSelectionAllowed(true);
                    // object of deviceLidtTableModel
                    devicesTableModel = new DeviceListTableModel();
                    tblDevices.setModel(devicesTableModel);
                    tblDevices.setRowHeight(22);
                    //set font for the table
                    tblDevices.setFont(fontDevice);
                    // Setting column width
                    tblDevices.getColumnModel().getColumn(0).setPreferredWidth
(140);
                    tblDevices.getColumnModel().getColumn(1).setPreferredWidth
(300);
                    tblDevices.getColumnModel().getColumn(2).setPreferredWidth
(125);
                    tblDevices.getColumnModel().getColumn(3).setPreferredWidth
(124);
                    tblDevices.getColumnModel().getColumn(4).setPreferredWidth
(124);
                    tblDevices.getColumnModel().getColumn(5).setPreferredWidth
(124);
                    tblDevices.getColumnModel().getColumn(6).setPreferredWidth
(124);
```

```
                    //adding terminalOuter to main motherPanel
                    motherPanel.add(terminalOuter);

                    terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                    terminalOuter.add(cmdInputPanel);
                    cmdInputPanel.add(correctDeviceInfo);
                    cmdInputPanel.add(wrongDeviceInfo);

                    terminalOuter.add(attributeScrollPanePanel);

                    JScrollPane attributeListScrollPane = new JScrollPane();

                    attributeScrollPanePanel.add(attributeListScrollPane);

                   //table which displays attributes of a selected device
                   tblAttributeList = new JTable();
                   tblAttributeList.setFillsViewportHeight(true);
                   tblAttributeList.setPreferredScrollableViewportSize(new
Dimension(600, 300));
                   attributeListScrollPane.setViewportView(tblAttributeList);
                   tblAttributeList.setColumnSelectionAllowed(true);
                   AttributeListTableModel = new AttributeTableModel();
                   tblAttributeList.setModel(AttributeListTableModel);
                   tblAttributeList.getColumnModel().getColumn
(0).setPreferredWidth(0);
                   tblAttributeList.setRowHeight(40);
                   tblAttributeList.setFont(fontAttribute);
                   tblAttributeList.getColumnModel().getColumn
(1).setPreferredWidth(130);
                   tblAttributeList.getColumnModel().getColumn
(2).setPreferredWidth(60);


                   terminalOuter.setVisible(true);

                   //listener for Connect button
                   btnConnect.addActionListener(new ActionListener() {
                           public void actionPerformed(ActionEvent arg0) {
                                   //method to call after clicking Connect button
                                   btnConnect_onClick();
                           }
                   });

                   //listener for dropdownlist combo box
                dropDownURLList.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                            chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                    }
                  });

                //listener for devices table
                    tblDevices.addMouseListener(new java.awt.event.MouseAdapter()
{
                            public void mouseClicked(java.awt.event.MouseEvent e)
{

                                    int row=tblDevices.rowAtPoint(e.getPoint());
```

```
                                    correctDeviceInfo.setVisible(false);
                                    wrongDeviceInfo.setVisible(false);

                                    highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                    try {
                                            // calls connect_cloud()
                                            indicator =
AttributeListTableModel.connect_cloud(username, password,chosenServer,
highlightedDevice);
                                    if(indicator.compareTo("correct device") !=
0){

                                            waitInfo.setVisible(false);
                                            correctDeviceInfo.setVisible(false);
                                            wrongDeviceInfo.setVisible(true);
                                            tblDevices.removeAll();
                                            tblAttributeList.removeAll();

                                            }
                                    else
                                    {
                                            correctDeviceInfo.setVisible(true);
                                    }
                                    }
                                    catch(Exception e1){
                                            System.out.println(e1);
                                    }
                                    System.out.println(indicator);

                            }
                    });

                    this.pack();

                    this.setVisible(true);

        }
        // this method is called when you click Connect button
        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                wrongCredentialsInfo.setVisible(false);
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        waitInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }
```

```
}
```

IDigiMonitorSample

iDigiMonitorSample sample test

(**For java supported modules**) This application is a simple UI to manage device cloud monitors.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java iDigiMonitorSample test sample application

The iDigiMonitorSample Test sample application can be found here: IDigiMonitorSample.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.monitor;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputSt
ea
;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.zip.InflaterInputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import com.digi.utils.Base64;

public class Main extends UIMain implements MonEvent {
        private static final long serialVersionUID = 1L;

        private ProcessingThread processor;
        private String monitor_ids[];
        private Monitor monitor;

        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
```

```
	}

	public Main() {
		super();

		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				if (processor != null)
					processor.shutdownThread();

				if (monitor != null)
					monitor.kill();

				// wait for threads to shut down.
				try {
					Thread.sleep(1000);
				} catch (InterruptedException e1) {
				}

				System.exit(0);
			}
		});

		btnConnect.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent arg0) {
				btnConnect_OnClick();
			}
		});

		btnCreate.addActionListener(new ActionListener() {

			@Override
			public void actionPerformed(ActionEvent arg0) {
				btnCreate_OnClick();
			}
		});

		btnDelete.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnDelete_OnClick();
			}
		});

		btnStartStop.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnStartStop_OnClick();
			}
		});

		btnUpdate.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnUpdate_OnClick();
			}
		});
```

```
                // hide left panel until we have some login info.
                pnlControls.setVisible(false);

                // set up worker threads/objects
                processor = new ProcessingThread();
                processor.start();
                monitor = new Monitor();
                monitor.addListener(this);

        }

        // some basic logging functions
        private void log(String msg) {
                txtLog.append(msg);
                txtLog.setCaretPosition(txtLog.getText().length());
        }

        private void logln(String msg) {
                log(msg + "\r\n");
        }

        // checks to see if we have the required information to do some work
        private boolean hasUserInfo() {
                String user = txtUsername.getText();
                String pass = new String(txtPassword.getPassword());
                if (user.isEmpty() == true || pass.isEmpty() == true)
                        return false;

                return true;
        }

        // connects to idigi and updates the list
        private void btnConnect_OnClick() {
                if (hasUserInfo()) {
                        processor.doRefreshMonitorList();

                        pnlControls.setVisible(true);
                        btnConnect.setVisible(false);
                }
        }

        // tells the processor to create a new monitor
        private void btnCreate_OnClick() {
                if (hasUserInfo())
                        processor.doCreateMonitor();
        }

        // tells the processor to delete the current monitor
        private void btnDelete_OnClick() {
                if (hasUserInfo())
                        processor.doDeleteMonitor();
        }

        // toggles the state of the monitor class
        private void btnStartStop_OnClick() {
                if (hasUserInfo()) {
                        if (monitor.isRunning()) {
                                log("\r\nStopping monitor...");
                                btnStartStop.setText("Start Watching");
```

```
                                monitor.stop();
                                logln("done!");
                        } else {
                                logln("\r\nStarting monitor...");
                                btnStartStop.setText("Stop Watching");
                                monitor.start(cboServer.getSelectedItem
().toString(),
                                                txtUsername.getText(),
                                                new String
(txtPassword.getPassword()),
                                                Integer.parseInt((String)
cboMonitor.getSelectedItem()));


                        }

                }

        }

        // refreshes the monitor list.
        private void btnUpdate_OnClick() {
                if (hasUserInfo())
                        processor.doRefreshMonitorList();
        }

       // Called by Monitor when data is written to the socket.
        @Override
        public void onDataWrite(byte[] raw, int length) {
                logln("\r\nData Written:");
                for (int i = 0; i < length; i++) {
                        log(String.format("%02X", raw[i]) + " ");
                }
                logln("");

        }

        // Called by Monitor when data is read from the socket
        @Override
        public void onDataRead(byte[] raw, int length) {
                logln("\r\nData Read:");

                // displays the raw data in hex.
                for (int i = 0; i < length; i++) {
                        log(String.format("%02X", raw[i]) + " ");
                }
                logln("");

                // 0x03 is the packet type 'PublishMessage'
                // Everything after byte 16 is XML, so pull that data out
                // convert to String, and display.
                StringBuilder xml = new StringBuilder();

                if (raw[1] == 0x3 && raw[10] == 0x01) // if we're using gzip

        // compression, decompress and

        // show to screen
                {
```

```
                        try {
                                InputStream inputstream = new InflaterInputStream
(
                                                new ByteArrayInputStream(raw, 16,
raw.length));

                                byte[] decompressed_buffer = new byte[2000];
                                int len;
                                while (inputstream.available() == 1) {
                                        len = inputstream.read(decompressed_
buffer);

                                        for (int i = 0; i < len; i++)
                                                xml.append(String.format("%c",
decompressed_buffer[i]));
                                }
                                inputstream.close();

                        } catch (Exception ex) {
                                System.out.println(ex);
                        }

                }

                else if (raw[1] == 0x3) {
                        for (int i = 16; i < length; i++) {
                                xml.append(String.format("%c", raw[i]));
                        }
                }

                logln(utils.parseXML(xml.toString()));
        }

        // performs the monitor deletion (called ONLY from the context of the
        // processor thread)
        private void deleteMonitor() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                logln("\r\nDeleting monitor " + (String)
cboMonitor.getSelectedItem()
                               + "...");
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor/" + (String)
cboMonitor.getSelectedItem();
                URL url;
                try {
                        logln("Sending HTTP DELETE to " + path);

                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("DELETE");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
```

```
+ ":" + Password)
                                                                    .getBytes
()));

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }

                        logln("Response:");
                        logln(xml.toString());
                        refreshMonitorList();

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }

        // performs the monitor update (called ONLY from the context of the
        // processor thread)
        private void refreshMonitorList() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                logln("\r\nRefreshing monitor list...");
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor";
                URL url;
                try {
                        logln("Sending HTTP GET to " + path);
                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("GET");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
+ ":" + Password)
```

```
                                                            .getBytes
()));

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {
                                                Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }
                        logln("Response:");
                        logln(xml.toString());
                        parseMonitorList(xml.toString());

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }

        // (called ONLY from the context of the processor thread)
        private void createMonitor() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor";
                String gzip = new String("none");

                logln("\r\nCreating monitor...");
                URL url;
                try {
                        CreateMonitor cm = new CreateMonitor();
                        cm.setVisible(true);
                        String topics = cm.getTopics();

                        // strip out gzip
                        if (topics.contains("gzip")) {
                                if (topics.contains(",gzip"))
                                        topics = topics.replace(",gzip", "");
                                else
                                        topics = topics.replace("gzip", "");
                                gzip = "gzip";
                        }

                        if (topics.isEmpty()) {
```

```
                                logln("No topics selected... stopped.");
                                return;
                        }
                        logln("Sending HTTP POST to " + path);

                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("POST");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
+ ":" + Password)
                                                                        .getBytes
()));

                        String payload = "\r\n<Monitor>"
                                        + "<monTopic>"
                                        + topics
                                        + "</monTopic>"
                                        +
"<monTransportType>tcp</monTransportType>"
                                        + "<monFormatType>xml</monFormatType>"
                                        + "<monBatchSize>1</monBatchSize>"
                                        + "<monCompression>"
                                        + gzip
                                        + "</monCompression>"
                                        +
"<monBatchDuration>0</monBatchDuration><monStatus>ACTIVE</monStatus>"
                                        + "</Monitor>";

                        logln("HTTP POST Payload:");
                        logln(utils.parseXML(payload));
                        logln("Sending....");

                        long packetsize = payload.length();

                        conn.setRequestProperty("Content-Length", packetsize +
"");

                        OutputStream out = conn.getOutputStream();

                        out.write(payload.getBytes());
                        out.close();

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
```

```
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {
                                                Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }

                        logln("Response:");
                        logln(xml.toString());

                        refreshMonitorList();

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        // parses the XML to pull out the monitor ID's
        private void parseMonitorList(String XML) {
                try {
                        DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
                        DocumentBuilder db = dbf.newDocumentBuilder();

                        // create the XML document.
                        Document dom = db.parse(new ByteArrayInputStream
(XML.getBytes()));
                        Element elem = dom.getDocumentElement();

                        NodeList nl = elem.getElementsByTagName("Monitor");

                        cboMonitor.removeAllItems();

                        if (nl != null) {
                                monitor_ids = new String[nl.getLength()];

                                for (int i = 0; i < monitor_ids.length; i++) {
                                        cboMonitor.addItem
(utils.getTextFromElement(
                                                        ((Element) nl.item(i)),
"monId"));
                                }

                                }


                } catch (Exception e) {
                }
        }

        // This is the thread that does all the data processing.
        // It is done this way so that the UI does not hang while
        // waiting for HTTP transactions to complete, and parsing to execute.
```

```
        private class ProcessingThread extends Thread {
                private int _command;
                private boolean _run;

                public synchronized void shutdownThread() {
                        _command = 1;
                }

                public synchronized void doRefreshMonitorList() {
                        _command = 2;
                }

                public synchronized void doCreateMonitor() {
                        _command = 3;
                }

                public synchronized void doDeleteMonitor() {
                        _command = 4;
                }

                @Override
                public void run() {
                        super.run();

                        _run = true;
                        _command = 0;
                        while (_run) {
                                try {
                                        switch (_command) {
                                        case 0:
                                                Thread.sleep(100);
                                                break;
                                        case 1:
                                                _run = false;
                                                break;
                                        case 2:
                                                refreshMonitorList();
                                                _command = 0;
                                                break;
                                        case 3:
                                                createMonitor();
                                                _command = 0;

                                                break;
                                        case 4:
                                                deleteMonitor();
                                                _command = 0;
                                                break;
                                        }
                                } catch (Exception ex) {
                                }
                        }
                }
        }

}
```

IOT Demo TradeShow

IOT_Demo_TradeShow sample test

(**For java supported modules**) This application lists attributes and provides a way to turn the FAN ON and OFF.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java IOT_Demo_TradeShow Test Sample Application

The IOT_Demo_TradeShow Test sample application can be found here: IOT_Demo_TradeShow.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.etherios;

import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import com.digi.etherios.DataTableModel;
import com.digi.etherios.DeviceListTableModel;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

public class Main extends JFrame {

       /**
        *
        */
        private static final long serialVersionUID = 1L;

        private JTextField txtUsername;
        private JTextField txtPassword;
        private JTextField txtRefreshInterval;
        private JTable tblDevices;
        private JTable tblAttributeList;

        private DeviceListTableModel devicesTableModel;
        private DataTableModel dataTableModel;

        private String username;
        private String password;
        private String interval;
        private String highlightedDevice;
        public String indicator = null;
        public String login_indicator = null;
        final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
        private static final Color cl_black = new Color(21, 45, 60);
        private static final Color cl_btn_grn = new Color(10, 148, 54);
        private static final Color cl_dkgray = new Color(110, 110, 110);
        private static final Color cl_ltgray = new Color(153, 153, 153);
        private static final Color cl_white = new Color(255, 255, 255);
        final JLabel wrongDeviceInfo;
        JPanel devicesScrollPanePanel = new JPanel();
        JPanel cmdOutputScrollPanePanel = new JPanel();
        JPanel motherPanel = new JPanel();

        JPanel basePanel = new JPanel();
        JPanel extraPanel = new JPanel();

        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        private static String[][] data = null;
        private static int rowCount = 0;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public final JLabel waitInfo = new JLabel("Please select device on which
IOT_demo program is running and wait for few seconds!!");
        boolean clickedonce = false;
```

```
        private static String[] ValueList = null;

        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }

        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 750);

                setTitle("Digi Connect Tank");
                //getContentPane().setLayout(new GridLayout(0, 1, 0, 0));
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));

                motherPanel.add(basePanel);
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);


                JPanel passPanel = new JPanel();
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
```

```
                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);
               JPanel dropListConnectPanel = new JPanel();
                basePanel.add(dropListConnectPanel);
                JPanel intervalPanel = new JPanel();
                basePanel.add(intervalPanel);

                final JLabel correctDeviceInfo = new JLabel("Latest values of
selected device in Device Cloud and will be refreshed based on interval!!");
                correctDeviceInfo.setSize(4, 4);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(Color.BLACK);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select device on which IOT_demo program is running");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(Color.BLACK);


                waitInfo.setVisible(false);
                waitInfo.setSize(7,7);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(Color.BLACK);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(Color.black);

                JComboBox dropDownURLList = new JComboBox();
                dropDownURLList.addItem(listURLItems[0]);
            dropDownURLList.addItem(listURLItems[1]);
            dropListConnectPanel.add(dropDownURLList);
            dropDownURLList.setBackground(cl_dkgray);
            dropDownURLList.setForeground(cl_black);
            dropDownURLList.setSize(50,25);

            JLabel intervalLabel = new JLabel("  Interval in min");
                intervalLabel.setForeground(cl_black);
                intervalLabel.setBackground(cl_dkgray);
                intervalLabel.setSize(60,25);
                dropListConnectPanel.add(intervalLabel);

                txtRefreshInterval = new JTextField();
                dropListConnectPanel.add(txtRefreshInterval);
                txtRefreshInterval.setColumns(5);

                JButton btnConnect = new JButton("Connect");
                dropListConnectPanel.add(btnConnect);

                Font f = new Font("Arial", Font.BOLD, 13);
                btnConnect.setOpaque(true);
                btnConnect.setBackground(cl_dkgray);
                btnConnect.setForeground(cl_black);
                btnConnect.setSize(new Dimension(50, 25));
```

```
                btnConnect.setFont(f);

                motherPanel.add(devicesScrollPanePanel);
                devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));

                devicesScrollPanePanel.add(wrongCredentialsInfo);
                devicesScrollPanePanel.add(waitInfo);
                JScrollPane devicesScrollPane = new JScrollPane();
                devicesScrollPanePanel.add(devicesScrollPane);


                tblDevices = new JTable();
                tblDevices.setFillsViewportHeight(true);

                tblDevices.setPreferredScrollableViewportSize(new Dimension(800,
150));

                devicesScrollPane.setViewportView(tblDevices);
                tblDevices.setColumnSelectionAllowed(true);
                devicesTableModel = new DeviceListTableModel();
                tblDevices.setModel(devicesTableModel);
                tblDevices.setRowHeight(22);
                tblDevices.getColumnModel().getColumn(0).setPreferredWidth(140);
                tblDevices.getColumnModel().getColumn(1).setPreferredWidth(300);
                tblDevices.getColumnModel().getColumn(2).setPreferredWidth(125);
                tblDevices.getColumnModel().getColumn(3).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(4).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(5).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(6).setPreferredWidth(124);
                motherPanel.add(terminalOuter);

                terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                terminalOuter.add(cmdInputPanel);
                cmdInputPanel.add(correctDeviceInfo);
                cmdInputPanel.add(wrongDeviceInfo);
                terminalOuter.add(cmdOutputScrollPanePanel);

                JScrollPane cmdOutputsListScrollPane = new JScrollPane();

                cmdOutputScrollPanePanel.add(cmdOutputsListScrollPane);

                tblAttributeList = new JTable();
                tblAttributeList.setFillsViewportHeight(true);

                tblAttributeList.setPreferredScrollableViewportSize(new Dimension
(600, 350));


                cmdOutputsListScrollPane.setViewportView(tblAttributeList);
                tblAttributeList.setColumnSelectionAllowed(true);


                dataTableModel = new DataTableModel();
                tblAttributeList.setModel(dataTableModel);
                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(0);
                tblAttributeList.setRowHeight(40);
```

```
                tblAttributeList.setFont(font);

                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(60);
                tblAttributeList.getColumnModel().getColumn(1).setPreferredWidth
(130);
                tblAttributeList.getColumnModel().getColumn(2).setPreferredWidth
(60);


                terminalOuter.setVisible(true);

                btnConnect.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                btnConnect_onClick();
                        }
                });


            dropDownURLList.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                }
            });

                tblDevices.addMouseListener(new java.awt.event.MouseAdapter(){
                        public void mouseClicked(java.awt.event.MouseEvent e){


                                if (clickedonce  == false){
                                        clickedonce = true;

                                        int row=tblDevices.rowAtPoint(e.getPoint
());
                                        correctDeviceInfo.setVisible(false);
                                        wrongDeviceInfo.setVisible(false);


                                        highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                        try {
                                                indicator =
dataTableModel.connect_cloud(username, password, chosenServer,
highlightedDevice);
                                                BasicThread refreshThread = new
BasicThread(username, password, chosenServer, highlightedDevice,
interval,dataTableModel);
                                                refreshThread.start();

                                        if(indicator.compareTo("correct device")
!= 0){
                                                System.out.println("indicator :
"+indicator);
                                                correctDeviceInfo.setVisible
(false);
                                                wrongDeviceInfo.setVisible(true);
                                                tblDevices.removeAll();
                                                tblAttributeList.removeAll();
```

```
                                                }
                                        else
                                        {
                                                correctDeviceInfo.setVisible
(true);

                                        }
                                }
                                        catch(Exception e1){
                                                System.out.println(e1);
                                        }
                                        System.out.println(indicator);

                                }
                                else {
                                        System.out.println("Clicked already");
                                }
                        }


                });
                this.pack();

                this.setVisible(true);
        }

        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                interval = txtRefreshInterval.getText();
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }

}
```

J1587 sample

This page contains a sample of the J1587 Python API on Digi devices.

```
            ## The J1587 support native to the Digi product is limited to the
## interpretation of the raw J1708 messages into a J1587 PID that contains the
## discrete samples from the J1708 message.

## No J1587 communication is explicitly provided.  All messages are converted
## back into J1708 messages of the same meaning then sent across the J1708
## communication bus.

## Many of the steps required to make use of the J1587 PID require the same
## setup as the J1708 bus, as it is built upon that.

## Import the J1708 Python API
import digij1708
import struct

##Create a handle to the J1708 bus through the class J1708Handle()
##The class takes 1 parameter, the bus number to get a handle to.
print "Getting a handle to the J1708 Bus"
handle = digij1708.J1708Handle(0)

##Configure the bus to a particular mid. in this case 0x48
handle.configure(0x48)

##Create a callback function.  This function requires three parameters:
##  mid, payload, and arg

## mid:  The message ID
## payload: The payload of the message
## arg:  An arbitrary parameter defined when setting the callback.

def callback_1(mid, payload, arg):
  print "\nMid: ", mid
  print "Payload: ", payload
  print "Arg: ", arg

  print "Converting to J1587 PID message"

  ## The payload is used to instantiate the J1587 PID.   The PID implements
  ## Python dictionary functions, so iteration over the keys or direct
  ## key look up is possible.

  pid = digij1708.J1587_PIDdict(payload)
  for key in pid:
    print "Pid: %s = %s" %(key, pid[key])

## Register the callback on the J1708 bus
handle.register_callback(callback_1, 'foo')
print "Type 'quit' to exit, or type a J1587 payload message then hit enter:\n"
  while 1:
  input = raw_input()
  if input.lower() in ['q', 'qu', 'qui', 'quit']:
    break

  input = input.strip()
```

```
  ## create a fake message J1587 compatible
  if len(input) == 0:
    msg = struct.pack('=2B', 45, 65)

  ## Use it to create a J1587 PID.
  pid = digij1708.J1587_PIDdict(msg)

  ## Print the discrete samples within the PID
  print "Sending pid with values:"
  for key in pid:
    print "Pid: %s = %s" %(key, ord(pid[key]))

  ## Convert the message into a J1708 message, and send it out the J1708 handle
  msg = pid.J1708_payload()
  try:
    handle.send(1, msg)
  except Exception, e:
    print e

## Unregister the call back, must use exact same input as the register call
print "Unregistering callback"
handle.unregister_callback(callback_1, 'foo')
```

J1708 sample

This page contains a sample of how to use the J1708 Python API on a Digi device.

```
            ## Import the J1708 Python API
import digij1708
import struct

##Create a handle to the J1708 bus through the class J1708Handle()
##The class takes 1 parameter, the bus number to get a handle to.
print "Getting a handle to the J1708 Bus"
handle = digij1708.J1708Handle(0)

##Configure the bus to a particular mid. in this case 0x48
handle.configure(0x48)

##Create a callback function.  This function requires three parameters:
##  mid, payload, and arg

## mid:  The message ID
## payload: The payload of the message
## arg:  An arbitrary parameter defined when setting the callback.

def callback_1(mid, payload, arg):
  print "\nMid: ", mid
  print "Payload: ", payload
  print "Arg: ", arg

## Register the callback on the J1708 bus
handle.register_callback(callback_1, 'foo')
print "Type 'quit' to exit, or type a message in to send then hit enter:\n"
while 1:
  input = raw_input()
  if input.lower() in ['q', 'qu', 'qui', 'quit']:
    break

  input = input.strip()

  ## If no input, create a fake message
  if len(input) == 0:
    msg = struct.pack('=2B', 45, 12)

  ## If input, take up to 19 characters of it
  else:
    if len(input) > 19:
      msg = input[:19]
  else:
    msg = input

  ## Send the message at priority 1
  try:
    handle.send(1, msg)
  except Exception, e:
    print e

## Unregister the call back, must use exact same input as the register call
print "Unregistering callback"
handle.unregister_callback(callback_1, 'foo')
```

J1939 sample

This page contains a sample on how to use the J1939 Python API on a Digi device.

```
            ## Import the CAN module
from digicanbus import *
import struct, time, sys

  baud = 125000
if len(sys.argv) >= 2:
baud = int(sys.argv[1])

## The digicanbus module has CANHandle().  A function that returns a handle to
## the current CAN bus.

## Specify the CAN bus number.
print "Getting handle to CAN bus 0"
handle = CANHandle(0)

## configure(bitrate)
## Configures the CAN bus to a specific bps and starts it.  This must be called
## at least once.

print "Configuring the bus to %d bps" %baud
handle.configure(baud)

## We create a function to be called when a J1939 message that is matched will
## be passed to.

def callback_1(width, identifier, remote_frame, payload, return_arg):
  PDU = J1939_PDU(width, identifier, remote_frame, payload)
  print "====PDU received===="
  for opt in ['DA', 'DP', 'EDP', 'GE', 'PF', 'PGN',
              'PS', 'SA', 'priority', 'payload']:

    print opt + " = " + str(PDU.__getattribute__(opt))

  print "Converting to a raw can message, sending it over the CAN bus"
  raw_msg = PDU.CANMsgTuple()
  print "%s %s %s %s" %(raw_msg[0], hex(raw_msg[1]), raw_msg[2], raw_msg[3])
  try:
    handle.send(*raw_msg)
  except Exception, e:
    print e
  else:
    print "Message succesfully sent"

## We create a filter that will trigger on 29 bit messages
filter_1 = (29, 0x0, 0x0, callback_1, 'filter_1')
print "Filter 1: ", filter_1

## Register the filters on the CAN bus
print "Registering filters..."
handle.register_filter(*filter_1)

counter = 0
print "Hit enter to send J1939_PDU, type 'quit' to exit"
while raw_input().lower() != 'quit':
```

```
  counter += 1
  P = J1939_PDU()
  P.DA = 0x76
  P.PGN = 0xF001
  P.payload = 'msg' + str(counter)
  can_msg = P.CANMsgTuple()
  handle.send(*can_msg)

## Unregister the filters created using the stored tuples
print "Unregistering filters"
handle.unregister_filter(*filter_1)
```

LibDeviceCloud

LibDeviceCloud sample test

(**For java supported modules**) This program is a library for communicating to the device cloud in java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibDeviceCloud Test Sample Application

The LibDeviceCloud Test sample application can be found here: LibDeviceCloud.zip.

Basic usage

Compile, load and run program using java environment.

Sample of MonitorTest.java file:

```
package com.digi.devicecloud.test;

import java.io.IOException;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

import com.digi.devicecloud.DeviceCloud;
import com.digi.devicecloud.MonitorConfig;
import com.digi.devicecloud.Result;
import com.digi.devicecloud.WsMonitor;
import com.digi.devicecloud.monitor.MonConnectionRequestPacket;
import com.digi.devicecloud.monitor.MonConnectionResponsePacket;
import com.digi.devicecloud.monitor.MonPacket;
import com.digi.devicecloud.monitor.MonPublishMessagePacket;
import com.digi.devicecloud.monitor.TcpMonitor;
import com.digi.devicecloud.monitor.TcpMonitorListener;
import com.digi.json.JsonArray;
import com.digi.json.JsonObject;

public class MonitorTest implements TcpMonitorListener {
        private static String username = "";
        private static String password = "";
        private static String hostname = "login.etherios.com";
        private static final int MONITOR_ID = 109537;

        private DeviceCloud cloud = new DeviceCloud(hostname, username,
password);


        public static void main(String[] args) {
                MonitorTest test = new MonitorTest();

                try {
                        test.listenMonitor();
```

```
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

        while (true) {
                try {
                        Thread.sleep(10000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

}


private void start() {
        try {

                WsMonitor monitorWs = new WsMonitor(cloud);

                List<Result> results = monitorWs.list();

                for (int i = 0; i < results.size(); i++) {
                        Result result = results.get(i);

                        JsonArray items = result.getItems();

                        for (int j = 0; j < items.size(); j++) {
                                JsonObject obj = items.getJsonObject(j);

                                System.out.println(obj.display());
                        }
                }

                // create a monitor
                // createMonitor();
                // listen to monitor
                // listenMonitor();

                // while (true) {
                // Thread.sleep(1000);
                // }

        } catch (Exception ex) {
                ex.printStackTrace();
        }
}


private void createMonitor() throws IOException, ParseException {
        WsMonitor monitorWs = new WsMonitor(cloud);

        MonitorConfig config = new MonitorConfig();

        // using compresion
        config.setCompression(MonitorConfig.COMPRESSION_ZLIB);
        // using JSON
        config.setFormatType(MonitorConfig.FORMAT_JSON);
```

```
                // listening to Device Core changes
                config.setTopic(MonitorConfig.TOPIC_DEVICE_CORE);
                // using TCP/IP
                config.setTransportType(MonitorConfig.TRANSPORT_TYPE_TCP);

                // create the monitor
                Result result = monitorWs.create(config);

                // print the monitor ID
                System.out.println(result.getDataAsString());
        }


        private void listenMonitor() throws InterruptedException {
                // create a new monitor with a high timeout
                TcpMonitor monitor = new TcpMonitor(900000);

                // add myself as a listener
                monitor.addListener(this);

                // start the monitor
                monitor.start("login.etherios.com", 3200, false);

                // create the request packet
                MonConnectionRequestPacket request = new
MonConnectionRequestPacket(
                username, password, MONITOR_ID);

                // send the packet
                monitor.sendPacket(request);

        }


        @Override
        public void tcpMonitorIncommingPacket(TcpMonitor tcpMonitor, MonPacket
packet) {
                System.out.println("Recieved: " + packet.getType());

                if (packet.getType() == MonPacket.TYPE_PUBLISH_MESSAGE) {
                        MonPublishMessagePacket publish =
(MonPublishMessagePacket) packet;

                        try {
                                System.out.println(new JsonObject(new String(
                                                publish.getPayload())).display
());
                        } catch (ParseException e) {
                                e.printStackTrace();
                        }
                }

                if (packet.getType() == MonPacket.TYPE_CONNECTION_RESPONSE) {
                        MonConnectionResponsePacket response =
(MonConnectionResponsePacket) packet;

                        System.out.println("Connection Response: " +
response.getStatus());
                }
```

```
        }


        @Override
        public void tcpMonitorConnected(TcpMonitor tcpMonitor) {
                // TODO Auto-generated method stub

        }
}
```

LibFastDb

LibFastDb sample test

(**For java supported modules**) This program is an interface for easily accessing MySQL from java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibFastDb Test Sample Application

The LibFastDb Test sample application can be found here: LibFastDb.zip.

Basic usage

```
Compile, load and run program using java environment.
```

Sample of FastDb.java file:

```
package com.digi.fastdb;

import java.lang.reflect.Field;
import java.sql.PreparedStatement;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.digi.fastdb.utils.utils;

public class FastDb {
        private static FastDb _instance = null;

        private Set<String> _tables = new HashSet<String>();


        public static FastDb getInstance() {
                if (_instance == null) {
                        _instance = new FastDb();
                }
                return _instance;
        }


        private FastDb() {

        }


        public boolean objectExists(Object object) throws Exception {
                checkTables(object.getClass());
                return utils.objectExists(object);
        }


        public void writeNewObject(Object object) throws Exception {
```

```
                checkTables(object.getClass());
                utils.writeNewObject(object);
        }


       /**
        * Attempts to push an update to the store based on the object's primary
key
        *
        * @param object
        * @throws Exception
        */
        public void pushObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.pushObject(object);
        }

       /**
        * If the object exists.. then the object is updated. Otherwise a new one
is
        * created.
        *
        * @param object
        * @throws Exception
        */
        public void saveObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.saveObject(object);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement) throws Exception {
                checkTables(klass);
                return getObjects(klass, statement, true);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement, boolean includeChildren) throws Exception {
                checkTables(klass);
                return utils.getObjects(klass, statement, includeChildren);
        }


        public void deleteObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.deleteObject(object);
        }


        public void removeLink(Object parent, String fieldName, Object
fieldValue) throws Exception {
                checkTables(parent.getClass());

                Field field = parent.getClass().getDeclaredField(fieldName);

                utils.deleteLink(parent, field, fieldValue);
        }
```

```
        private void checkTables(Class<?> klass) throws Exception {
                String tableName = utils.getTableName(klass);

                if (_tables.contains(tableName)) {
                        return;
                }

                if (utils.tableExists(tableName)) {
                        _tables.add(tableName);
                        return;
                }

                List<String> schemas = utils.generateTableSchema(klass);
                for (int i = 0; i < schemas.size(); i++) {
                        utils.createTable(schemas.get(i));
                }


        }
}
```

LibJil

LibJil sample test

(**For java supported modules**) This program is an interpreter for the JIL language.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJil Test Sample Application

The LibJil Test sample application can be found here: LibJil.zip.

Basic usage

Compile, load and run program using java environment.

Sample of JilProgram.java file:

```
package com.digi.jil;

import java.util.Hashtable;

import com.digi.jil.lang.JilStatement;
import com.digi.json.JsonObject;

public class JilProgram {
        private JilStatement _rootNode;
        private Hashtable<String, JilStatement> _labels;


        public JilProgram(JsonObject source) throws Exception {
                _labels = new Hashtable<String, JilStatement>();
                _rootNode = JilStatementFactory.newStatement(this, source);
                _rootNode.setOid("r");
        }


        public void step(JilContext jilContext) throws Exception {
                String oid = jilContext.getStatementOid();

                if (oid == null) {
                        jilContext.setStatementOid("r");
                        oid = "r";
                }

                JilStatement statement = find(oid);

                JilResult result = statement.execute(jilContext);

                switch (result.state) {
                // there was a horrible error
                        case JilResult.RESULT_ERROR:
                                handleError(result, jilContext);
                                break;
                        case JilResult.RESULT_GOTO:
```

```
                                handleGoto(result, jilContext);
                                break;
                        case JilResult.RESULT_NEXT_SIBLING:
                                handleNextSibling(result, jilContext);
                                break;
                        case JilResult.RESULT_COMPLETE:
                                handleComplete(result, jilContext);
                                break;
                }
        }

        public void registerLabel(JilStatement label) {
                _labels.put(label.getLabelName(), label);
        }


        public JilStatement lookupLabel(String labelName) {
                if (_labels.containsKey(labelName)) {
                        return _labels.get(labelName);
                }

                return null;
        }


        private void handleComplete(JilResult result, JilContext jilContext)
throws Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_COMPLETE);
        }


        private void handleError(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_FAILED);
        }


        private void handleGoto(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setStatementOid(result.nextOid);
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
                // TODO, set other stuff !?
        }


        private void handleNextSibling(JilResult result, JilContext jilContext)
throws Exception {
                // find next command
                String oid = result.nextOid;

                if (oid == null || "r".equals(oid)) {
                        // we are done!
                        jilContext.setStatementOid(oid);
                        jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_
COMPLETE);
                        return;
                }

                oid = JilUtils.nextSibling(oid);
```

```
		JilStatement statement = find(oid);
		while (statement == null) {
			oid = JilUtils.parseParent(oid);

			if (oid == null || "r".equals(oid)) {
				// we are done!
				jilContext.setStatementOid(oid);
				jilContext.setExecutionStatus(JilStatement.EXEC_
STATUS_COMPLETE);
				return;
			}

			oid = JilUtils.nextSibling(oid);

			statement = find(oid);
		}

		jilContext.setStatementOid(oid);
		jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
	}


	public JilStatement find(String oid) {

		return _rootNode.find(oid);
	}

}
```

LibJson

LibJson sample test

(**For java supported modules**) This program is a JSON parser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJson Test Sample Application

The libJson Test sample application can be found here: LibJson.zip

Basic usage

Compile, load and run program using java environment.

Sample of JsonArray.java file:

```
package com.digi.json;

import java.text.ParseException;
import java.util.LinkedList;
import java.util.List;

public class JsonArray {
        private List<Object> _objects = new LinkedList<Object>();


        public JsonArray() {

        }


        public JsonArray(String string) throws ParseException {
                JsonTokenizer tokenizer = new JsonTokenizer(string);
                fromTokenizer(tokenizer);

        }


        public JsonArray(JsonTokenizer tokenizer) throws ParseException {
                fromTokenizer(tokenizer);
        }


        private void fromTokenizer(JsonTokenizer tokenizer) throws ParseException
{
                if (!tokenizer.isNextTokenStartArray()) {
                        throw new ParseException("Not an array!", -1);
                }
                // pop the open brace
                tokenizer.nextToken();

                // check for empty array
                if (tokenizer.isNextTokenFinishArray()) {
```

```
                        tokenizer.nextToken();
                        return;
                }

                while (true) {
                        _objects.add(tokenizer.parseValue());

                        // if not comma, then it should've been a ']'
                        if (!tokenizer.nextToken().equals(",")) {
                                break;
                        }
                }

        }


        public int size() {
                return _objects.size();
        }


        public Object get(int index) {
                return _objects.get(index);
        }


        public Object get(String path) {
                List<String> dir = JsonTokenizer.parsePath(path);

                return get(dir);
        }


        protected Object get(List<String> directions) {
                if (directions.size() == 0)
                        return this;

                String value = directions.remove(0);

                value = value.replace("[", "");
                value = value.replace("]", "");

                int item = Integer.parseInt(value);

                Object obj = _objects.get(item);
                if (obj instanceof JsonObject) {
                        return ((JsonObject) obj).get(directions);
                }
                else if (obj instanceof JsonArray) {
                        return ((JsonArray) obj).get(directions);
                }

                return obj;
        }

        public String getString(int index) {
                return (String) get(index);
        }
```

```
        public JsonObject getJsonObject(int index) {
                return (JsonObject) get(index);
        }


        public JsonArray getJsonArray(int index) {
                return (JsonArray) get(index);
        }


        public int getInt(int index) {
                return Integer.parseInt(getString(index));
        }


        public long getLong(int index) {
                return Long.parseLong(getString(index));
        }


        public float getFloat(int index) {
                return Float.parseFloat(getString(index));
        }


        public double getDouble(int index) {
                return Double.parseDouble(getString(index));
        }


        public Object set(int index, Object value) {
                while (index >= _objects.size())
                        _objects.add(null);

                return _objects.set(index, value);
        }


        protected void set(List<String> directions, Object value) throws
ParseException {
                if (directions.size() == 0)
                        throw new ParseException("Invalid path", 0);

                String item = directions.remove(0);

                item = item.replaceAll("\\[", "");
                item = item.replaceAll("\\]", "");

                int index = Integer.parseInt(item);

                if (directions.size() == 0) {
                        set(index, value);
                }
                else {
                        // already exists
                        if (_objects.size() > index) {
                                Object obj = _objects.get(index);
```

```
				if (obj instanceof JsonObject) {
					JsonObject jo = (JsonObject) obj;

					jo.put(directions, value);
					return;
				}
				else if (obj instanceof JsonArray) {
					JsonArray ja = (JsonArray) obj;
					String child = directions.get(0);
					if (child.equals("[]"))
						ja.add(directions, value);
					else
						ja.set(directions, value);
					return;
				}
			}
			else {

				String child = directions.get(0);
				if (child.startsWith("[") && child.endsWith("]"))
{
					JsonArray ja = new JsonArray();

					set(index, ja);

					if (child.equals("[]"))
						ja.add(directions, value);
					else
						ja.set(directions, value);

					return;
				}
				else {
					JsonObject jo = new JsonObject();

					set(index, jo);

					jo.put(directions, value);
					return;
				}
			}

		}

	}


	public boolean add(Object value) {
		return _objects.add(value);
	}


	public void add(int index, Object value) {
		_objects.add(index, value);
	}


	protected void add(List<String> directions, Object value) throws
ParseException {
```

```
		if (directions.size() == 0)
			throw new ParseException("Invalid path", 0);

		String item = directions.remove(0);

		if (!item.equals("[]"))
			throw new ParseException("This should never happen!", 0);

		if (directions.size() == 0) {
			add(value);
		}
		else {
			String child = directions.get(0);
			if (child.startsWith("[") && child.endsWith("]")) {
				JsonArray ja = new JsonArray();

				_objects.add(ja);

				if (child.equals("[]"))
					ja.add(directions, value);
				else
					ja.set(directions, value);

				return;
			}
			else {
				JsonObject jo = new JsonObject();

				_objects.add(jo);

				jo.put(directions, value);
				return;

			}

		}

	}


	public Object remove(int index) {
		return _objects.remove(index);
	}


	protected Object remove(List<String> directions) throws ParseException {
		if (directions.size() == 0)
			throw new ParseException("Invalid path!", -1);
		String item = directions.remove(0);

		item = item.replaceAll("\\[", "");
		item = item.replaceAll("\\]", "");

		int index = Integer.parseInt(item);

		if (directions.size() == 0) {
			return remove(index);
		}
		else {
```

```
                        Object obj = get(index);

                        if (obj instanceof JsonObject) {
                                JsonObject jo = (JsonObject) obj;

                                return jo.remove(directions);
                        }
                        else if (obj instanceof JsonArray) {
                                JsonArray jo = (JsonArray) obj;

                                return jo.remove(directions);
                        }

                }
                return null;

        }


        public void clear() {
                _objects.clear();
        }


        public String display() {
                return display(0).toString();
        }


        protected StringBuilder display(int depth) {
                StringBuilder sb = new StringBuilder();

                sb.append("[\n");
                for (int i = 0; i < _objects.size(); i++) {
                        Object obj = _objects.get(i);

                        sb.append(JsonTokenizer.repeat("  ", depth + 1));
                        if (obj == null) {
                                sb.append("null");
                        }
                        else if (obj instanceof JsonObject) {
                                sb.append(((JsonObject) obj).display(depth + 1));
                        }
                        else if (obj instanceof JsonArray) {
                                sb.append(((JsonArray) obj).display(depth + 1));
                        }
                        else if (obj instanceof String) {
                                sb.append(JsonTokenizer.escapeString((String)
obj));
                        }
                        else {
                                sb.append(obj + "");
                        }

                        if (i < _objects.size() - 1) {
                                sb.append(",");
                        }

                        sb.append("\n");
```

```
		}
		sb.append(JsonTokenizer.repeat("  ", depth));
		sb.append("]");

		return sb;
	}


	public StringBuilder toStringBuilder() {
		StringBuilder sb = new StringBuilder();

		sb.append("[");
		for (int i = 0; i < _objects.size(); i++) {
			Object obj = _objects.get(i);

			if (obj == null) {
				sb.append("null");
			}
			else if (obj instanceof JsonObject) {
				sb.append(((JsonObject) obj).toStringBuilder());
			}
			else if (obj instanceof JsonArray) {
				sb.append(((JsonArray) obj).toStringBuilder());
			}
			else if (obj instanceof String) {
				sb.append(JsonTokenizer.escapeString((String)
obj));
			}
			else {
				sb.append(obj + "");
			}
			if (i < _objects.size() - 1) {
				sb.append(",");
			}
		}
		sb.append("]");

		return sb;
	}

	public boolean has(int index) {
		if (index < _objects.size())
			return true;

		return false;
	}


	protected boolean has(List<String> directions) throws ParseException {
		if (directions.size() == 0)
			throw new ParseException("Invalid path!", -1);

		String item = directions.remove(0);

		item = item.replaceAll("\\[", "");
		item = item.replaceAll("\\]", "");

		int index = Integer.parseInt(item);
```

```
                if (!has(index))
                        return false;
                else if (directions.size() == 0)
                        return true;

                Object obj = get(index);

                if (obj instanceof JsonObject) {
                        JsonObject jo = (JsonObject) obj;

                        return jo.has(directions);
                }
                else if (obj instanceof JsonArray) {
                        JsonArray ja = (JsonArray) obj;

                        return ja.has(directions);
                }

                return false;
        }


        @Override
        public String toString() {
                return toStringBuilder().toString();

        }

}
```

LibUtils

LibUtils sample test

(**For java supported modules**) This program is a collection of utility classes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibUtils Test Sample Application

The libUtils Test sample application can be found here: LibUtils.zip.

Basic usage

Compile, load and run program using java environment.

Sample of ConfigFile.java file:

```
package com.digi.utils;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;

/**
* Loads a simple config file into memory. Config files are of the following
* form
*
* ### Comments start with a #
* key = value
*
* @author mcarver
*
*/
public class ConfigFile {
        private static Hashtable<String, ConfigFile> _configFiles = new
Hashtable<String, ConfigFile>();

        private Hashtable<String, String> _parameters;
        private String _filename;


       /**
        * Loads the config file at filename and caches the data. If the config
file
        * is already loaded, then the cache is returned.
        *
        * @param filename
        * @return
        * @throws IOException
        */
        public static ConfigFile getInstance(String filename) throws IOException
```

```
{
                if (_configFiles.containsKey(filename)) {
                        return _configFiles.get(filename);
                }

                ConfigFile config = new ConfigFile(filename);

                _configFiles.put(filename, config);

                return config;
        }


        private ConfigFile(String file) throws IOException {
                _parameters = new Hashtable<String, String>();
                _filename = file;

                loadFile();
        }


        private void loadFile() throws IOException {

                File file = new File(_filename);

                BufferedReader stream = new BufferedReader(new InputStreamReader(
                                new FileInputStream(file)));

                String line = null;
                while ((line = stream.readLine()) != null) {

                        line = line.trim();

                        if (line.startsWith("#")) {
                                continue;
                        }

                        if (!line.contains("=")) {
                                continue;
                        }

                        String name = line.substring(0, line.indexOf("=")).trim
().toLowerCase();
                        String value = line.substring(line.indexOf("=") + 1).trim
();

                        _parameters.put(name, value);
                }

                stream.close();

        }


        /**
         * Returns the value of the parameter named name. Names are case
         * insensitive.
         *
         * @param name
```

```
         * @return
         */
        public String getString(String name) {
                if (!_parameters.containsKey(name.toLowerCase())) {
                        return null;
                }

                return _parameters.get(name.toLowerCase());
        }
}
```

LibZkConfigProtocol

LibZkConfigProtocol sample test

(**For java supported modules**) This program is a protocol library ZooKeeper (zk.digi.com) which uses to communicate with its slave nodes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibZkConfigProtocol Test Sample Application

The libZkConfigProtocol Test sample application can be found here: LibZkConfigProtocol.zip.

Basic usage

Compile, load and run program using java environment.

Sample of FieldField.java file:

```
package com.digi.configurepackets;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class FileField extends Field {
        public StringField fileName;
        public File file;


        FileField(DataInputStream dataIn) throws IOException {
                fileName = new StringField(dataIn);
                int fileLength = dataIn.readInt();
                file = File.createTempFile("FileField_", ".tmp");

                OutputStream out = new BufferedOutputStream(new FileOutputStream
(file));
                try {
                        int pos = 0;
                        int read = 0;
                        int packet_size = 2048;
                        if (packet_size > fileLength) {
                                packet_size = fileLength;
                        }

                        byte[] data = new byte[packet_size];
                        while (pos < fileLength) {
                                read = dataIn.read(data);
```

```
                                if (read > 0) {
                                        out.write(data, 0, read);
                                        pos += read;
                                        if (fileLength - pos < packet_size) {
                                                packet_size = fileLength - pos;
                                                data = new byte[packet_size];
                                        }
                                }
                                else if (read == 0) {
                                        try {
                                                Thread.sleep(100);
                                        } catch (InterruptedException e) {}
                                }
                                else if (read < 0) {
                                        throw new IOException(
                                                        "Could not read entire file from
stream!");
                                        }
                                }

                        if (pos != fileLength) {
                                throw new IOException(
                                                "There was an error reading the
file from the data stream!");
                        }
                } catch (IOException ex) {
                        out.close();
                        file.delete();
                        out = null;
                } finally {
                        if (out != null) {
                                out.close();
                        }
                }

        }


        public FileField(File file) {
                fileName = new StringField(file.getName());
                this.file = file;
        }


        public FileField(String name, File file) {
                fileName = new StringField(name);
                this.file = file;
        }


        public void delete() {
                file.delete();
        }


        @Override
        public int length() {
                return fileName.length() + 4 + (int) file.length();
        }
```

```
		@Override
	public void toDataStream(DataOutputStream dataOut) throws IOException {
		fileName.toDataStream(dataOut);
		dataOut.writeInt((int) file.length());

		InputStream in = new BufferedInputStream(new FileInputStream
(file));

		try {
			int read = 0;
			byte[] packet = new byte[2048];
			while ((read = in.read(packet)) != -1) {
				if (read > 0) {
					dataOut.write(packet, 0, read);
				}
				else if (read == 0) {
					try {
						Thread.sleep(100);
					} catch (InterruptedException e) {}
				}
			}
		} finally {
			in.close();
		}
	}


}
```

NET OS 9P9360 external RTC

Program for 9P9360 external RTC (DS1337) in NET+OS

NET+OS 9P9360 External RTC (For NET+OS 7.4.2 - 7.5.2 modules) This example adds support for the onboard DS1337 RTC on the 9P9360 in NET+OS.

Test files

This sample program contains six files. The main function is in root.c.

9P9360 External RTC (DS1337) Test Sample Application

The 9P9360 External RTC (DS1337) Test sample application can be found here: 9P9360_External_RTC_(DS1337).zip.

Basic usage

```
Copy rtc_drv.custom over
\netos\src\bsp\devices\ns9xxx\ns9360\rtc\rtc_drv.c
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include "rtc.h"
#include <time.h>

void applicationTcpDown (void)
{
        static int ticksPassed = 0;

        ticksPassed++;
}

static NaRtcTime_t rtc_time;

static char* const WEEKDAYS[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday","Friday", "Saturday"};

static char* const MONTHS[12] = {"January", "Febuary", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};

void applicationStart (void)
{
        NaStatus rc;
        char* data;
        time_t t;

        while(1)
        {
                t = time(NULL);
                data = ctime(&t);
```

```
                printf("time(): %s", data);
                rc = naRtcGetTime(0, &rtc_time);
                switch (rc)
                {
                        case NASTATUS_SUCCESS:
                                // Wednesday, December 16, 2009 at 12:00:00
                                // day of the week/month index starts at 1, our
arrays start at 0
                                printf("Date/Time: ");
                                printf("%s, ",WEEKDAYS[rtc_time.dayOfWeek-1]);
                                printf("%s %i, %i at", MONTHS[rtc_time.month-1],
rtc_time.date, rtc_time.year);
                                printf(" %i:%i:%i\n", rtc_time.hours, rtc_
time.minutes, rtc_time.seconds);
                                break;
                        case NASTATUS_RTC_NOT_INITIALIZED:
                                printf("The real time clock is not
initialized.\n");
                                break;
                        case NASTATUS_RTC_FAIL:
                                printf("The real time clock was unsuccesffully
updated with the new time. It's given invalid rtc_time.\n");
                                break;
                        case NASTATUS_RTC_NOT_AVAILABLE:
                                printf("The real time clock doesn't support this
action.\n");
                                break;
                }
                tx_thread_sleep(500);
        }

        printf ("Hello World!\n");
        tx_thread_suspend(tx_thread_identify());
}
```

NET OS Appkit Rio

Program for controlling the Rabbit RIO chip in NET+OS

Appkit RIO Test (For NET+OS 7.4.2 - 7.5.2 modules) A driver for controlling the Rabbit RIO chip.

Test files

This sample program contains eight files. The main function is in root.c.

CPU test sample application

The Appkit Rio Driver Test sample application can be found here: RIO_Appkit_Driver.zip.

Basic usage

```
Test Hardware setup on RIO APPKIT board:
--With an LED (or Scope). Connect ground (neg) to P4.9(GND). Connect VCC(pos)
      to P4.8(RC0P0). This LED is where the majority of all the tests will be
      performed
--With a STDP(single throw, double poll) switch. Connect the common pin to
      P4.4(RC1P0). Connect one poll to VCC and the other poll to GND. This pin
      will be used for all the input tests.
BSP SETUP

Digi Connect ME 9210
Change the following in gpio.h:
#define BSP_GPIO_MUX_SERIAL_A              BSP_GPIO_MUX_SERIAL_2_WIRE_UART
#define BSP_GPIO_MUX_IRQ_1                 BSP_GPIO_MUX_USE_PRIMARY_PATH
#define BSP_GPIO_MUX_IRQ_1_CONFIG                      BSP_GPIO_MUX_IRQ_FALLING_EDGE

Right Click on your project. Go to properties. Select Net+OS from the list.
Under bsp_sys.h change 'Dialog Port' and 'STDIO Port' to 'Serial Port C'

ConnectCore 9P 9215
Change the following in gpio.h:
#define BSP_GPIO_MUX_IRQ_3                 BSP_GPIO_MUX_USE_3RD_ALTERNATE_PATH
#define BSP_GPIO_MUX_IRQ_3_CONFIG                      BSP_GPIO_MUX_IRQ_FALLING_EDGE
```

Known issues

The Input/Output pins on the RIO APPKIT board are all floating. This means that any pin not pulled up or down and set for an input will report random state changes due to cross talk and other interference.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

#include "rio_appkit.h"


void applicationTcpDown (void)
```

```
{
        static int ticksPassed = 0;
        ticksPassed++;
}

void TestGPIO();
void TestPWM();
void TestGPIOInput();
void TestHardwareReset();
void TestSoftReset();
void TestPPM();
void TestInterrupt();

void applicationStart (void)
{
#if (PROCESSOR == ns9215)
       /*
        * ConnectCore 9P9215 Connections
        * GPIO67 = X21.C9 - Reset Line
        * GPIO101(IRQ3) = X21.D16 - Interrupt Line (IRQ3)
        */
        naRIOInit(5,67,EXTERNAL3_INTERRUPT);
#elif (PROCESSOR == ns9210)
       /*
        * Connect ME 9210 Setup
        * GPIO13 = P3.20 - Reset Line
        * GPIO2 = P3.12 - Interrupt Line (IRQ1)
        */
        naRIOInit(5,13,EXTERNAL1_INTERRUPT);
#endif

        printf("Starting RIO Test\n");

        TestHardwareReset();
        TestSoftReset();
        TestGPIO();
        TestPWM();
        TestGPIOInput();
        TestPPM();
        TestInterrupt();

    tx_thread_suspend(tx_thread_identify());
}

void TestGPIO()
{
        int port,pin;
        int x;
        int retval;

        printf ("Starting GPIO Test...\n");
        for (x=0; x < 20; x++)
        {
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, TRUE);

                printf("All are high\n");
                tx_thread_sleep(10);
```

```
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, FALSE);

                printf("All are low\n");
                tx_thread_sleep(10);
        }
        printf ("GPIO Test complete...\n");
}

void TestPWM()
{
        int retval;
        int x,i;
       /*
        * 16666666 = 60hz
        *  20000000 = 50hz
        *  10000000 = 100hz
        */
        long period = 10000000;

        printf ("Starting PWM Test.\n");
        retval = naRIOReset(TRUE);
        retval = naRIOSetPrescaler(period);

        retval = naRIOSetPWM(0,0,period,0);
        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 100; x++)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
                for (x = 100; x >=1 ; x--)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
        }
        printf("PWM Test Complete...\n");
}

void TestHardwareReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Hardware Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(TRUE);
        printf("RIO has been reset!\n");
```

```
        printf("Hardware Reset Test complete...\n");
        tx_thread_sleep(500);
}

void TestSoftReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Software Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(FALSE);
        printf("RIO has been reset!\n");
        printf("Software Reset Test Compelte...\n");
        tx_thread_sleep(500);
}

void TestGPIOInput()
{
        BOOL val = FALSE;
        BOOL last = FALSE;
        int retval;
        int x;

        retval = naRIOReset(TRUE);

        printf("Starting GPIO Input Test Port 1, Pin 0...\n");

        retval = naRIOSetInput(1,0);
        for (x=0; x<20; x++)
        {
                // wait for a state change
                while ( retval == last)
                        val = naRIOReadInput(1,0);

                printf("Pin is now %d\n", retval);
                last = retval;
        }

        printf("GPIO Input Test Complete...\n");
}

void TestPPM()
{
        int retval;
        long period = 10000000;
        int x,i;

        naRIOReset(TRUE);

        printf("Starting PPM Test. Port 0, Pin 0...\n");
        retval = naRIOSetPrescaler(period);
```

```
        /* set pins 2,3 on port0 to a PWM */
        retval = naRIOSetPWM(0,2,period,period/3);
        retval = naRIOSetPWM(0,3,period,period/3);

        /* set pin0 to a PPM with a phase shift */
        retval = naRIOSetPPM(0,0,period,0,period/3);

       /*
        * set pin1 (the dead pin) to a GPIO.
        * pin1's match register is being used,
        * so it can only be used for GPIO
        */
        retval = naRIOSetOutput(0,1,FALSE);

        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 360; x++)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
                for (x = 360; x >=1 ; x--)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
        }

        printf("PPM Test Complete...\n");
}

void InterruptCallBack(int port, int pin)
{
        int retval;
        printf("Callback called on port: %d, pin: %d\n", port, pin);
        retval = naRIOResetGPIOInterrupt(1,0);
}

void TestInterrupt()
{
        int retval;
        printf("Starting Interrupt Test. Port 1, Pin 0...\n");
        retval = naRIOReset(TRUE);
        retval = naRIOGPIOInterrupt(1,0, InterruptCallBack);
}
```

NET OS CPU

Program to read/write CPU registers in NET+OS

CPU TEST (For NET+OS 7.4.2 - 7.5.2 modules) Shows how to read/write directly from CPU registers in NET+OS.

Test files

This sample program contains two files, root.c and readme.txt. The main function is in root.c.

CPU test sample application

The CPU Test sample application can be found here: Read-Write_CPU_Registers.zip.

Basic usage

Compile, load and run program using NET+OS.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

volatile unsigned long *reg_ptr = (unsigned long *) 0xA0902008;

void applicationStart (void)
{
        printf ("Register Value: 0x%X\n", *reg_ptr);
        *reg_ptr = (*reg_ptr) | 0x6C;
        printf ("Register Value: 0x%X\n", *reg_ptr);

    tx_thread_suspend(tx_thread_identify());
}

void applicationTcpDown (void){    static int ticksPassed = 0;    ticksPassed++;}
```

NET OS Ping

Program to generate a ping from a NET+OS module

PING TEST (For NET+OS 7.4.2 - 7.5.2 modules) Developed to generate a ping from the NET+OS development module.

How does it work?

PING TEST is a client based application set up to run on a NET+OS development module. It sends ping requests to a specific IP address entered in the root.c file.

Test files

This sample program contains three files, ping.c, ping.h and root.c. The main function in ping.c pings a target IP address. In the file ping.h you can specify the amount of system ticks per second.

Ping test sample application

A simple Ping Test sample application can be found here: Ping_test.zip.

Basic usage

First, open the root.c file and enter the ipaddress you would like to ping and save file

```
retval = ping("10.4.110.1");
```

Second, Build application, load the application into the embedded module and run it.

Sample of root.c file:

```
{
        int retval;

        while(1)

        {
                retval = ping("10.4.110.1");

                if(retval == -1)
                {
                        printf("Error\n");
                }
                else if(retval == 0)
                {
                        printf("No response\n");
                }
                else
                {
                        printf("Response heard\n");
                }
        }
```

```
        return;
}
```

NET OS Telnet Session

Program for Telnet_Customized_SessionID in NET+OS

NET+OS Telnet Session ID (For NET+OS 7.4.2 - 7.5.2 modules) Showcase how to customize telnet session ID using NETOS APIs. Sample application showing working of a Telnet Server.

Test files

This sample program contains six files. The main function is in root.c.

Telnet Session ID Test Sample Application

The Telnet Session ID Test sample application can be found here: Telnet_Customized_SessionID.zip.

Basic usage

Before Building and debugging please check the following in bsp_cli.h

```
1. #define BSP_CLI_TELNET_ENABLE   FALSE
2. #define BSP_CLI_ENABLE  FALSE

Debug/Run :

Open command prompt and
$telnet (ipaddress) 5000
login :
password :


Enter any characters and hit enter.
You can start more than one telnet session.
|Open another command prompt and :
$telnet (ipaddress) 5000

This application will display the below details to serial port for each session:

Session ID
Bytes Received for the particular Session
Username of the session.

Charlie & Bob :) :)
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include <sysAccess.h>


#include "telnet.h"
#include <bsp_api.h>


/*
```

```
* Set this to 1 to run the manufacturing burn in tests.
*/
int APP_BURN_IN_TEST = 0;


void applicationStart (void)

{
        int ret_telnet;
        unsigned long port_count = 0;
        iconfig_ptr.max_entries = TELNET_MAX_SERVERS;

    /*  Add Username digi password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "digi", "sysadm", NASYSACC_LEVEL_RW, NULL);

    /*  Add Username sysadm password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "sysadm", "sysadm", NASYSACC_LEVEL_R, NULL);

    /*  Add Username debug password debug to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "debug", "debug", NASYSACC_LEVEL_R, NULL);

    /*
         *Initializing Telnet Server .
        */
    if( (ret_telnet = TSInitServer(&iconfig_ptr)) != SUCCESS){
    printf("Return value : %d\n",ret_telnet);
            if(ret_telnet == TS_NO_MEMORY)
                    printf("Unable to allocate memory.\n");
            else if(ret_telnet == TS_INVALID_PARAMETER)
                    printf("The value in parameter is invalid.\n");
            else if(ret_telnet == TS_SYSTEM_ERROR)
                    printf("An internal error occurred.\n");
            else
                    printf("Can't open without calling
TSInitServer.\nExiting........\n");
            }
    /*
        *Configuring Telnet Server
        *Here only one server is configured.
        * If you want you can configure one more.
        * passing port_count+1
        */

        if(setup_telnet_server(port_count) != 0){
                printf("Telnet Server Setup Failed\n");
                return;
        }

        TS_t *list;
        while(1)
        {
                tx_thread_sleep(5*NABspTicksPerSecond);

                if(tx_semaphore_get(&semaphore_ptr,TX_WAIT_FOREVER ) != TX_
SUCCESS){
                        printf("Semaphore Get Failed\n");
                        return;
                }
                list = head;
```

```
                while(list != NULL)
                {
                        printf("\nSession ID : %d\n",list->ts_id);
                        printf("Bytes Received in this Session : %d\n",list-
>byte_recv);
                        printf("Username : %s\n",list->username);
                        list = list->ts_nxt;
                }
                if(tx_semaphore_put(&semaphore_ptr) != TX_SUCCESS){
                        printf("Semaphore Put Failed\n");
                        return;}
                }

        }

}

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;
/*
* Code to handle error condition if the stack doesn't come up goes here.
*/
}
```

Reading RSSI values

This page contains a simple demonstration of capturing RSSI values using the digicli module on Python enabled Digi devices. It is meant to be instructive on how to use the functionality.

```
import sys, re, digicli
# this line is a regex for the Network Address when using "display xbee"
p = re.compile ( '(\[[0-9a-f]{4,4}\]\!)' )
# this line is a regex for the RSSI value visible when using "display xbee addr=
[0123]!" (for each discovered module)
q = re.compile ( 'rssi \(DB\):(.*)\r' )
# this line is a regex for the MAC address of the module using "display xbee
addr=[0123]!" (for each discovered module)
r = re.compile ( 'Status of device:( [0-9a-f:\!]{24,24})' )

print "Node\t\tRSSI\t\tMAC Address"
status, output = digicli.digicli('display xbee')

# first we find all the Network node numbers:
for line in output:
node = p.findall(line)

    # then we query each node number for details
    if node != []:
        # here is an example of using digicli with a variable -- note that node
[0] converts node from a list to a str
        status, output2 = digicli.digicli(' display xbee addr=' + node[0] )
        for line2 in output2:
            rsval = q.findall(line2)

            #and then we pick up the MAC address
            if rsval != []:
                for line3 in output2:
                    maddr = r.findall(line3)
                    if maddr != []:
                        print node, "\t", rsval, "\t", maddr

# exit the script and go back to command line
sys.exit()
```

SMTP Email

Python program for SMTP Email

SMTP Email Notification (Python program) This example application sends an email notification to the specified email-id's.

Test files

This sample program contains one file. File name "SMTP_email_notofication.py".

SMTP Email Notification Test Sample Application

The SMTP_email_notofication.py Python Test sample application can be found here: SMTP_email_notofication.zip.

Basic usage

Provide inputs where neccesary.

Sample of SMTP_email_notofication.py file:

```
#!/usr/bin/Python

import os
import sys
import smtplib
import traceback

##########################################################################
##########################################################################
'''###PROVIDE INPUTS HERE###'''
EMAIL_HOST = "mail.<your_domian>.com"
sender = 'M2M'  #ANY NAME
# Receivers email id
receivers = ['first_name.last_name@digi.com', 'second_name.last_name@digi.com']
message = """ From: Digi Sample <M2M@digi.com>
To: <first_name.lastname@digi.com>,<second_name.last_name@digi.com>
CC: <first_name.lastname@digi.com>
Subject: SMTP e-mail test

This is a test e-mail
"""
##########################################################################
##########################################################################
try:
      smtpObj = smtplib.SMTP('EMAIL_HOST', 25)
      smtpObj.sendmail(sender, receivers, message)
      print "Successfully sent email"
except Exception, e:
      print "Exception %s" %e
      traceback.print_exc()

print "End of the program"
```

Scaling analog values

Most products reading analog values return a 16-bit integer in a limited range defined by the hardware. For example, most Digi products return a value from 0 to 1023 which represents a reading within 0 to 10.25v or 0 to 23.5mA. Yet knowing that your room temperature is 675 binary or 4.56 volts or 17.2mA is not very useful.

You will want to SCALE this value to something more useful like 76.3 degrees Fahrenheit or 297.6 degrees Kelvin.

While this conversion is not difficult, it is complex to envision since you need to fit two distinct line equations to obtain a result. Below is a simple example which does this in two steps. First it converts the raw binary input into a percentage range 0-100% represented as 0.0 to 1.0 appropriate for the sensor (this might go below 0% or above 100%). Second, it converts the range percentage into the actual published sensor value.

An example

The sensor

A Minco model TT321PD1M precision RTD temperature probe with 4-20mA interface defines 4mA as -58.0 DegF and 20mA as 122.0 DegF. So the Digi product is placed into Current Loop mode and returns a value like 17.3mA - how is this converted to DegF?

The first step is to stop thinking of the input as 17.3mA, but to instead think of it as a percentage of the vendor-defined range. In this example the range is from a 0% signal representing -58.0 DegF, while a 100% signal represents 122.0 DegF. The table below shows the theoretical relationship of 4-20mA on a Digi product which converts 0-23.5mA to binary 0-1023.

mA	percent of 0-23.5mA	percent of 4-20mA	binary in 0 to 1023	Example DegF
0 mA	0%	-25%	0	-103 DegF
4 mA	17%	0%	174	-58 DegF
8 mA	34%	25%	348	-13 DegF
12 mA	51%	50%	522	+32 DegF
16 mA	68%	75%	697	+77 DegF
20 mA	85%	100%	871	+122 DegF
24 mA	102%	125%	1045	+167 DegF

Note While this allows defining temperatures 'out-of-range' such as -103 DegF at 0mA, users should never make active use of sensor data when outside of their published range, which in this example is 4-20mA and -58.0 to +122 DegF.

Example 'Terms' for the scaling

It is convenient to think of the conversion as using a five-element tuple of the form (**raw_zero, raw_span, user_zero, user_span, units_str**), where:

- raw_zero is the binary input which is declared 0.00 percent of range
- raw_full is the binary input which is declared 100.00 percent of range (is not in the tuple)

- raw_span is the binary delta (the 'span') between raw_zero and raw_full, so (raw_full - raw_zero)
- user_zero is the user-desired unit-of-measure to return at 0.00 percent of range
- user_full is the user-desired unit-of-measure to return at 100.00 percent of range (is not in the tuple)
- user_span is the delta (the 'span') between user_zero and user_full, so (user_full - user_zero)
- units_str is the string for the user-desired unit-of-measure

Some example term tuples used in the code sample below:

(0, (1023-0), 0, (10.25-0), 'V')

Binary 0 is called 0% while binary 1023 is called 100%. 0-100% is returned as 0.0V to 10.25V

(0, (1023-0), 0, (23.5-0), 'mA')

Binary 0 is called 0% while binary 1023 is called 100%. 0-100% is returned as 0.0mA to 23.5mA

(174, (871-174), 0, (100-0), 'prc')

Binary 174 is called 0% while binary 871 is called 100%. 0-100% is returned as 0prc to 100prc

(174, (871-174), -58.0, (122.0-(-58.0)), 'DegF')

Binary 174 is called 0% while binary 871 is called 100%. 0-100% is returned as -58.0DegF to 122.0DegF

Implementation Note: *It may be easier to allow customers to configure the 'zero' and 'full' values, then calculate the 'span' during start-up. However, recalculating the 'span' as '(full-zero)' every processing cycle is a waste of CPU resources, so retain the calculated span for reuse. The Python code sample included here assumes the 2nd and 4th terms are the span, not the full.*

The Python scale code

```
def sample():

    # Note: the 'get_raw_analog(x)' call is imaginary
    #       Substitute in the call required to obtain a raw binary number

    # Report Analog One as 0-10.25v
    trm_1 = (0, 1023, 0, 10.25, 'V')
    ain_1 = scale_value(get_raw_analog(0), trm_1)
    print 'Analog 1 was %0.2f %s' % (ain_1, trm1[4])

    # Report Analog Two as 0-23.5mA
    trm_2 = (0, 1023, 0, 23.5, 'mA')
    ain_2 = scale_value(get_raw_analog(1), trm_2)
    print 'Analog 2 was %0.2f %s' % (ain_2, trm2[4])

    # Report Analog Three as 0-100% for 4-20mA (might be < 0% or > 100%)
    # Note that we shift the expected raw range away from 0-1023 to 174-871
    trm_3 = (174, (871-174), 0, 100, 'prc')
    ain_3 = scale_value(get_raw_analog(2), trm_3)
    print 'Analog 3 was %0.2f %s' % (ain_3, trm3[4])

    # Put it all togather
    # Report Analog Four as DegF for the example RTD 4-20mA sensor
    trm_4 = (174, (871-174), -58.0, (122.0-(-58.0)), 'DegF')
    ain_4 = scale_value(get_raw_analog(3), trm_4)
    print 'Analog 4 was %0.2f %s' % (ain_4, trm4[4])
```

```
    return


def scale_value(raw_in, terms):
    """Scale the raw_in value (assumed raw int) to user-units float
    terms = (raw_zero, raw_span, user_zero, user_span, units_str)"""

    # msg = ' >> terms = %s' % terms
    scaled = None

    if len(terms) > 2 and terms[1] != 0:
        # first we calculate 0-100% as 0.0 to 1.0
        # so y = (raw_in - raw_zero) / raw_span
        scaled = float(raw_in - terms[0]) / terms[1]
        # msg += ' prc=%0.4f' % scaled

        # second we optionally change to user units
        if len(terms) > 4 and terms[2] is not None:
            scaled =  terms[2] + (scaled * terms[3])
            # msg += ' usr=%0.4f%s' % (scaled,terms[4])

    # print msg
    return scaled
```

Temperature sample

This page contains a sample on how to use the Temperature API on Digi devices.

```
## From the digihw library, import the temperature handle
from digihw import temperature

## Get a sample reading in Celsius
sample = temperature()
print "Celsius: ", sample

## Convert it to Fahrenheit
f_sample = (9.0/5.0) * sample + 32
print "Fahrenheit: ", f_sample
```

WSBrowser

WSBrowser sample test

(**For java supported modules**) This program is a webservice browser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java WSBrowser Test Sample Application

The WSBrowser Test sample application can be found here: WSBrowser.zip.

Basic usage

Compile, load and run program using java environment.

Sample of WSB.java file:

```
package com.digi.wsb;

import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.text.ParseException;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import com.digi.json.JsonObject;
import com.digi.utils.Base64;
import com.digi.utils.misc;

public class WSB extends WsbUi {
        private static final long serialVersionUID = 1L;


       /**
        * @param args
        */
        public static void main(String[] args) {
                new WSB().setVisible(true);
        }


        public WSB() {
                super();

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
```

```
                                System.exit(0);
                        }
                });
        }


        @Override
        public void goButton_onClick(ActionEvent e) {
                try {
                        String method = ((String) methodComboBox.getSelectedItem
()).toUpperCase();
                        String url = urlTextField.getText();
                        String username = usernameTextField.getText();
                        String password = new String
(passwordTextField.getPassword());
                        String mime = (String) mimeContentType.getSelectedItem();
                        String payload = postDataTextArea.getText();

                        if (username.equals("")) {
                                username = null;
                        }

                        if ("GET".equals(method) || "DELETE".equals(method)) {
                                getData(method, url, username, password);
                        }
                        else if ("POST".equals(method) || "PUT".equals(method)) {
                                pushData(method, url, payload, mime, username,
password);
                        }
                } catch (Exception ex) {
                        log(ex);
                }

        }


        public void log(String logging) {
                try {
                        JsonObject jobj = new JsonObject(logging);
                        logTextArea.append(jobj.display() + "\n");
                } catch (Exception e) {
                        logTextArea.append(logging + "\n");
                }

                logTextArea.setCaretPosition(logTextArea.getText().length());
        }


        public void log(Exception e) {
                log(misc.getStackTrace(e));
        }


        public WebResult getData(String method, String path, String username,
String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
```

```
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");

                        // configure the authorization
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);

                        // get the steam from the server
                        InputStream str = conn.getInputStream();
                        int size = conn.getContentLength();
                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public WebResult pushData(String method, String path, String payload,
String contentType, String username, String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", contentType);
                        conn.setRequestProperty("Content-Length", payload.length
() + "");
                        conn.setDoOutput(true);

                        // configure the authorization
```

```
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);
                        log(payload);

                        // write the data
                        OutputStream out = conn.getOutputStream();
                        out.write(payload.getBytes());
                        out.close();

                        // get the steam from the server
                        InputStream str = conn.getInputStream();

                        int size = conn.getContentLength();

                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public void dumpConnectionRequest(HttpURLConnection conn) {
                Map<String, List<String>> map = conn.getRequestProperties();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
```

```
                        }

                        log(header.trim());
                }
        }


        public void dumpConnectionResponse(HttpURLConnection conn) throws
IOException {
                Map<String, List<String>> map = conn.getHeaderFields();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
                        }

                        log(header.trim());
                }
        }
}
```

Wakeup sample

This page contains a sample on how to use the wake reason Python API part of the digipowercontrol API on Digi devices.

```
## from the internal digipowercontrol module import wake_reson and constants
from digipowercontrol import *

## wake_reason provides a set object that contains the reasons why the device
## is now powered on.
wr_set = wake_reason()

## We perform a CONSTANT in wake_reason() to check if that was the event that
## caused us to start.  Multiple events can occur between the off state and
## powered on state of the device.

if WAKE_REASON_ACCEL in wr_set:
  ## The accelerometer has additional functionality that it stores a set of
  ## accelerometer readings that caused it to trigger the event.  A user can
  ## inspect them using the accelerometer.event_samples() function, which
  ## contains a before and after list of samples.

  print "Wake reason could have been accelerometer"
  from digihw import accelerometer

  handle = accelerometer()

  event_samples = handle.event_samples()

  print "These are the samples that caused the accelerometer to trigger"

  print "\nSamples from before the event"
  for sample in event_samples[0]:
    print sample

  print "\nSamples from after the event"
  for sample in event_samples[1]:
    print sample

if WAKE_REASON_IGNITION in wr_set:
  print "Wake reason could have been ignition"
if WAKE_REASON_POWERLOSS in wr_set:
  print "Wake reason could have been powerloss"
if WAKE_REASON_RTC in wr_set:
  print "Wake reason could have been RTC"
```

XBEE API packets

Program to design API packets

(**For ZigBee modules**) Program "Tx_Req 0x10" API packets generator for Zigbee modules(Xbee RF Modules (S1 {only DigiMesh}, S2, S3 and S8).

Test files

This sample program contains one file, "Tx_Req 0x10 packet generator.py".

XBee API Packet generator Test Sample Application

The XBee API Packet sample application can be found here: Tx_Req_0x10_packet_generator.zip.

Basic usage

Provide input where necessary.

Sample of Tx_Req 0x10 packet generator.py file:

```
########################################################################
#                    IMPORTS                                           #
########################################################################

import serial
import sys
import binascii


## Known Issue:
## 1. ONLY 1 byte length packet not transmitting



######  User Data #######
com_port = 'COM1'
dest_64bit = '0013A20040762859'
dest_16bit = 'FFFE'
rf_data = """!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`"""
#########################

def b_u(st):
    ## function to convert big-endian binary string into bytes[0-1] as int
    if len(st) == 1:
        return ord(st)
    else:
        length = len(st)
        sft = 8*(length-1)
                return (b_u(st[0])<<sft) + b_u(st[1:])



## create serial socket connection to talk with module
ser = serial.Serial(com_port, 9600, timeout=0.1, rtscts=True)

## convert RF data to hex
rf_hex = binascii.hexlify(rf_data)
```

```
##print "rf_hex=", rf_hex


                        ## calculate packet length
hex_len = hex(14 + (len(rf_hex)/2))
hex_len = hex_len.replace('x','0')
##print "hex_len=", hex_len

## calculate checksum
## 0x17 is the sum of all parameters minus 64bit & 16bit dest addr & payload
    checksum = 17


for i in range(0,len(dest_64bit),2):
    checksum = checksum + int(dest_64bit[i:i+2],16)

for i in range(0,len(dest_16bit),2):
    checksum = checksum + int(dest_16bit[i:i+2],16)

for i in range(0,len(rf_hex),2):
    checksum = checksum + int(rf_hex[i:i+2],16)

## checksum = 0xFF - 8-bit sum of bytes between the length and checksum
checksum = checksum%256
checksum = 255 - checksum
checksum = hex(checksum)
checksum = checksum[-2:]

## designing packet
tx_req = ("7E" + hex_len + "10" + "01" + dest_64bit
          + dest_16bit + "00" + "00" + rf_hex + checksum)
print "Tx packet = ", tx_req

## convert packet from hex to binary
data = binascii.unhexlify(tx_req)

## send data on serial line to module
ser.write(data)

## listin COM port for response
resp = ser.readline()

## convert response from binary to int
resp = b_u(resp)

## convert response from int to hex
resp = '%x' % resp
hex_data = resp.upper()
print "Response (in hex) = ", hex_data

## close connection
ser.close()
```

XBee bootloader menu

Program to bypass bootloader menu

XBee bypass bootloader menu (Xbee program) This sample application skips and bypasses freescale MCU & directly access radio.

Test files

This sample program contains several files. Program is in file "main.c".

Bypass bootloader menu Test Sample Application

The bootloader menu Test sample application can be found here: Bypass_bootloader_menu.zip.

Basic usage

```
APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250.
```

Sample of bootloader bypass menu:

```
#include <xbee_config.h>
/* #include <types.h>
#include <xbee/platform.h>*/


void main(void)
{
/*       sys_hw_init();
        sys_xbee_init();
        sys_app_banner(); */
        //APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250
        sys_reset(APP_CAUSE_BYPASS_MODE);
}
```

XBee sensor

Python program for XBee sensor

XBee sensor (Python program) This program gives a user information regaridng battery status of sensor.

Test files

This sample program contains one file. File name "check_battery_life_sensor.py".

XBee battery sensor test sample application

The check_battery_life_sensor.py Python Test sample application can be found here: Check_battery_life_sensor.zip.

Basic usage

Make sure that the sensors are in the network coordinator. Provide the inputs where neccesary.

Sample of check_battery_life_sensor.py file:

```
import sys
import os
import struct
import zigbee
import xbee
from _zigbee import *
import time
import traceback
from struct import *

''' Provide the extended address(OUI) of the xbee sensor'''
###################################################
DESTINATION="00:13:a2:00:40:86:cd:11!"
###################################################

def calculate_battery(bat_vol):
    mv = float(bat_vol)
    mv = ((mv * 1200)/1024)
    v = mv/1000
    v = round(v, 2)
    return v

try:
    print "reading parameters, waiting five seconds..."
    # %V- Supply Voltage. Reads the voltage on the Vcc pin. Scale by 1200/1024 to
    # convert to mV units.
    param_value = zigbee.ddo_get_param(DESTINATION, "%V")
    param_value = struct.unpack("h", param_value)
    str_param_value = str(param_value)        # converting into a string
    s_index = str_param_value.find("(")
    e_index = str_param_value.find(",)")
    bat_voltage = str_param_value[s_index+1:e_index]
    battery_mv = calculate_battery(bat_voltage)
    if (battery_mv) >= 2.8 and (battery_mv) <= 3.4:
        print "battery is in the range of 2.8 and 3.4 and the battery value" + \
        + "is %s" %str(battery_mv)
    else:
        print "low battery"

except Exception, e:
    print "Exception %s" %e
    traceback.print_exc()

print "end of the program"
```

XBeeComm

Program to communicate XBeeComm through Com port with Windows 7

XBeeComm Sample (For Windows 7 PC) This application can communicate with the Xbee module from the PC through the COM port. Using the different buttons on the form we can get the different Xbee parameters like pan.

How does it work?

Using this C sharp application , we can communicate with the Xbee module from the PC. Using the different buttons on the form we can get the different Xbee parameters like pan id, version of the software.

Test files

This sample program contains nine files.

XBeeComm Test Sample Application

The XBeeComm Test sample application can be found here: XbeeComm.zip.

Basic usage

Compile, load and run program using Windows Tools.

Sample Program.cs file:

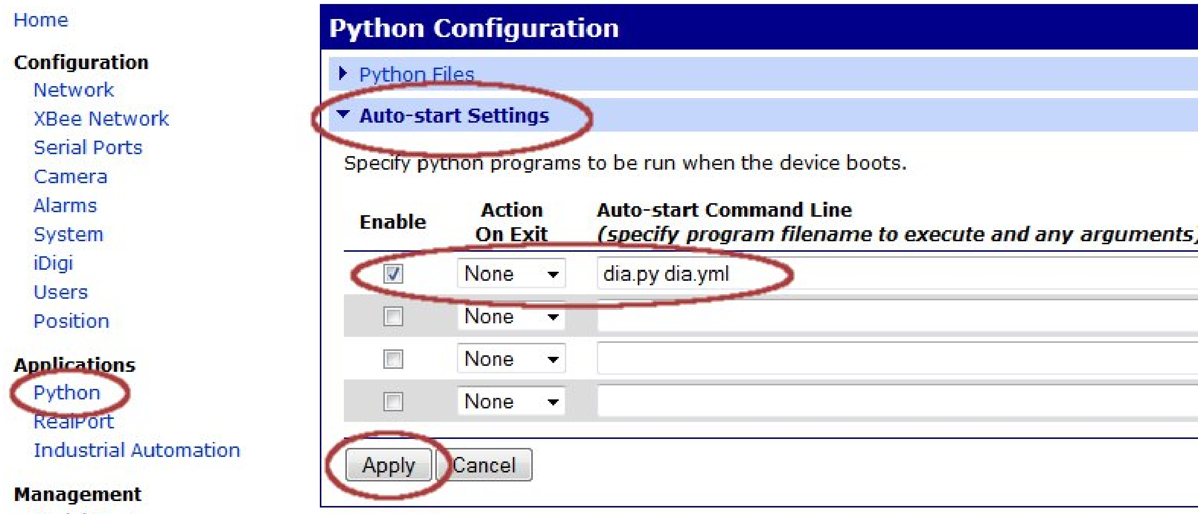
```
#using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace app2
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

DIA

Auto-start Python on a Digi gateway

By web interface

The most direct way to enable auto-start is via the device's web interface. Select the Python link on the left, then the Auto-Start Settings. You will be shown the four scripts you can auto-start. Enter the program name, including the ".py" extension. *Do not include the command "Python"!*



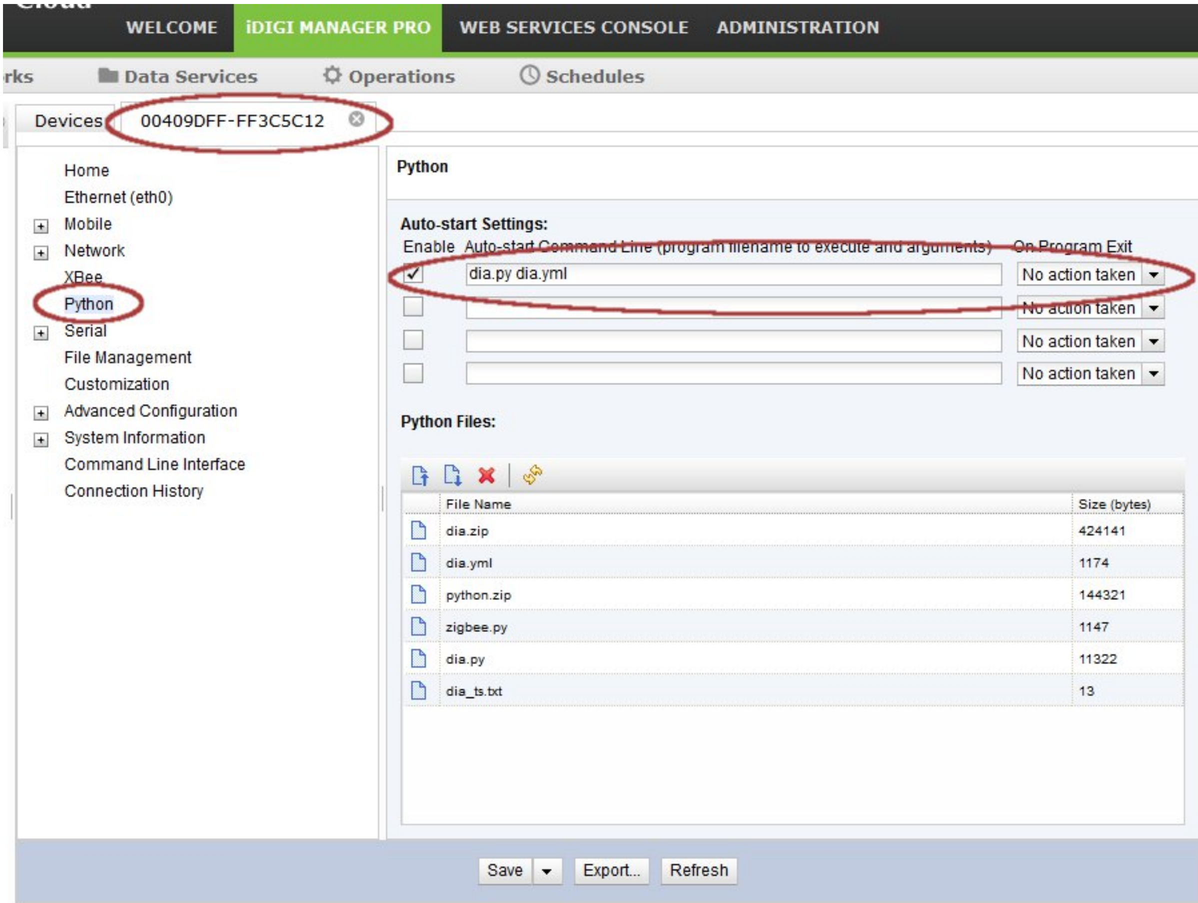
The On-Exit actions default to None, but can be set to restart the script, or reboot the gateway. **However a warning - never enable a restart/reboot option while debugging your code**, for having a simple typo in your main routine can cause the gateway to reboot in a tight loop which makes recover difficult (impossible by remote Device Cloud access). Users of the DIA framework should review this helpful change to dia.py: Rapid Reboot Detection.

Notes on ConnectPort X2e ZB

- Selecting the Python link on the left directly shows you the Auto-Start settings. Unlike on the ConnectPort X2/X4, Python files are NOT shown on this page. The Python files are accessed under the File Management link.
- The ConnectPort X2e ZB has built in Rapid Reboot Detection. The faster a reboot occurs, the longer the X2e/Linux kernel delays before auto-starting Python scripts.

By device manager

Setting auto-start via the Device Manager web console is much like setting it via the web interface. Note the above warning about the restart/reboot setting! if the device is rebooting every few seconds due to a Python error or simple typo, you will not be able to recover by Device Cloud access.



By RCI/SCI

The Python Auto-Start settings are held within the 'Python' settings group. See Digi's RCI and SCI documentation to understand how to read and write settings.

Request

```
<rci_request version="1.1">
 <query_setting>
  <Python />
 </query_setting>
</rci_request>
```

Response

```
<rci_reply version="1.1">
 <query_setting source="current" compare_to="none" encrypt="none">
  <Python><state>on</state><command>dia.py
dia.yml</command><onexit>none</onexit></Python>
  <Python
index="2"><state>off</state><command></command><onexit>nosne</onexit></Python>
  <Python
index="3"><state>off</state><command></command><onexit>none</onexit></Python>
  <Python
```

```
index="4"><state>off</state><command></command><onexit>none</onexit></Python>
 </query_setting>
</rci_reply>
```

Digi DIA notes

The DIA framework requires at least 2 special files to run - **dia.py** and **dia.zip**. You run the dia.py script to start the framework, and this file handles the diverse platform differences, mounting the dia.zip and running the framework from within. By default the dia.yml configuration is embedded in the dia.zip file. Some users prefer to manually keep a copy of dia.yml outside of the dia.zip to simplify remote browsing and changes. If this is done, you need to auto-start "dia.py dia.yml".

As of DIA version 2.0, creating a simple text file in the Python area named "nospin.txt" will enable dia.py to watch for a rapid reboot situation, which means the gateway has been rebooted more than 10 times in 20 minutes. This causes dia.py to delay launching the dia.zip files. See Also: Rapid Reboot Detection.

Digi ESP notes

The Digi ESP (Eclipse for Python) IDE use a default start file name of **dpdsrv.py**. If you use ESP to upload your Python code, you can safely put dpdsrv.py into your auto-start setting instead of dia.py.

Generally, dpdsrv.py does nothing more than run your script, which means you can often safely bypass dpdsrv.py and run your own main script. However, opening the dpdsrv.py script in an editor allows you to see what it is doing. Upon some platform (like a Windows PC), the dpdsrv.py *DOES* create a corrected search path so cannot be bypassed.

Basic web services example using DIA

Overview

This simple web Services example uses the built-in "Hello World" DIA software. DIA runs on your Digi communications gateway (ConnectPort X) and is an application (written in Python) that connects devices (both physical and virtual) to presentations (external resources like this example).

The example code is 100% AJAX, a combination of HTML and Javascript that includes a well-known call (XmlHttpRequests) to interact with web services.

Note This example only works when the HTML file is loaded from the local filesystem (not a web-server) as XmlHttpRequests are not allowed to accessed any other server than the server which served the HTML file. Traditionally when writing a web application which accesses a device attached to Device Cloud, one must have the web application server access Device Cloud on the web-browser's behalf.

Include the "Hello World" configuration information for DIA to utilized on your Digi gateway

```
## Hello World Digi Device Acquisition Framework Configuration File
devices:
  # The template device driver; creates a demonstration device
  # containing a counter and a set of channels implementing a simple
  # adder.
   - name: hello
     driver: devices.hello_world_device:HelloWorldDevice
     settings:
         update_rate: 1.0

presentations:
  # Create a new console instance on TCP port 4146.  It can be connected
  # two by using any telnet client.
  - name: console0
    driver: presentations.console.console:Console
    settings:
```

```
        type: tcp
        port: 4146

  # Create a web presentation instance.  When running on a PC this will
  # start a new web-server on a default port
  # When running on a Digi device this presentation will "extend" the
  # web-server built in to the Digi device with a new page named
  # "digi_daq.html".
  - name: web0
    driver: presentations.web.web:Web
    settings:
         page: idigi_dia.html
         port: 8081

  # The RCIHandler allows for channel dumps, querying a channel, and setting
  # a channel.  The RCIHandler parses an incoming XML formatted message
  # to determine what you wish to do.  The messages are shown below.  If
  # an error is encountered, the request you sent will have a child error
  # element specifying the error information.
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
         target_name: idigi_dia
```

This includes the virtual connector / driver (hello_world_device), and three presentations for interacting with the virtual driver: one via telnet (to port 4146), one via a simple web page (IP_address_of_gateway/idigi_dia.html) and one via a Web Services call to what is called an RCI (Remote Command Interface), using a target of idigi_dia.

Ensure that you have a Device Cloud account and that your ConnectPort-X gateway is actively connected. For additional information, visit www.etherios.com/devicecloud and visit the "resources" page.

Restart your gateway and ensure that DIA is running. You can automate DIA startup via the "Pythons->Autostart" screen in the Digi ConnectPort X webUI administration utility.

Test your DIA configuration locally by pointing your laptop to the ConnectPort-X's local IP address. This means that you need to be locally attached for this particular test. type "IP_Address_of_ConnectPort-X/idigi_dia.html" into your web browser. You should receive a web-UI that will display the three samples available to the virtual connector/driver: prefix_string, suffix_string, xtended_string (a concatenation of the first two samples).

Copy the following code into a text file and save on your laptop. Open via Firefox, Safari or Chrome. IE works, but you'll need to add a javascript function to allow it to recognize a few AJAX methods. Upon running, you'll need to include:

- Your Digi ConnectPort-X gateway address (just the last two octets - e.g. 00409DFF-FF382CF3)
- URL of the Device Cloud Connectivity Server (the URL where your Digi ConnectPort-X product is connected - found from Device Cloud when you are logged in)

- username: Your Device Cloud username
- password: Your Device Cloud password

Download the AJAX code for your PC-browser here ->IDigi-WS-Demo.zip.

Below you find an extended version of the above AJAX code. The changes are:

```
- I added the option to get a specific channel ("channel get" beside "channel_
dump")
- I added a debug option to display the RCI answer
- I used it with my CPX and added for info my ana.yml file for information
- I read all values from a complete device by "get all", or "get one" to read
only one channel value
here are the files: ->IDigi-Demo.zip.
```

Command line build of DIA projects

We highly recommend building DIA projects in Digi ESP (our Eclipse-based development environment), but there are times where that is not optimal:

- When you have a DIA driver that does not have the supporting files to integrate into ESP
- When you would like to automate the process in a script
- You just don't roll that way ;)

Doing a build at the command line is simple and efficient, and very flexible.

Python

You will need Python installed on your system to do the make process.

- For NDS gateways, such as the ConnectPort X4, X5, X2D, etc. you will need Python 2.4.3.
- For Linux based X2e Zigbee Gateways, you will need Python 2.7.1.

NOTES:

- Python 2.4.3 has been deprecated, so can be difficult to find and install for many PC platforms. The best way to get it from the ESP installation, you can even choose during install to only install Python, not the full Eclipse environment.
- Python 2.7.1 is readily available for most platforms, including Windows, Mac and Linux.

Windows or Mac

The easiest and most reliable way to get the correct Python environment on your build PC is to install ESP. It will install a compatible version of Python 2.4, Python 2.6 and Python 2.7.

ESP can be downloaded from the this page on our support website ESP download is under Development Tools.

Tip At least in the Windows installer, you will have the option to not install ESP, just the Python binaries. If you are short on space, this is a good idea. I do recommend doing the full installation, if nothing else to get the DIA documentation (part of the Help system in ESP). Who knows, you might also find ESP a good way to do your projects.

Linux or Unix

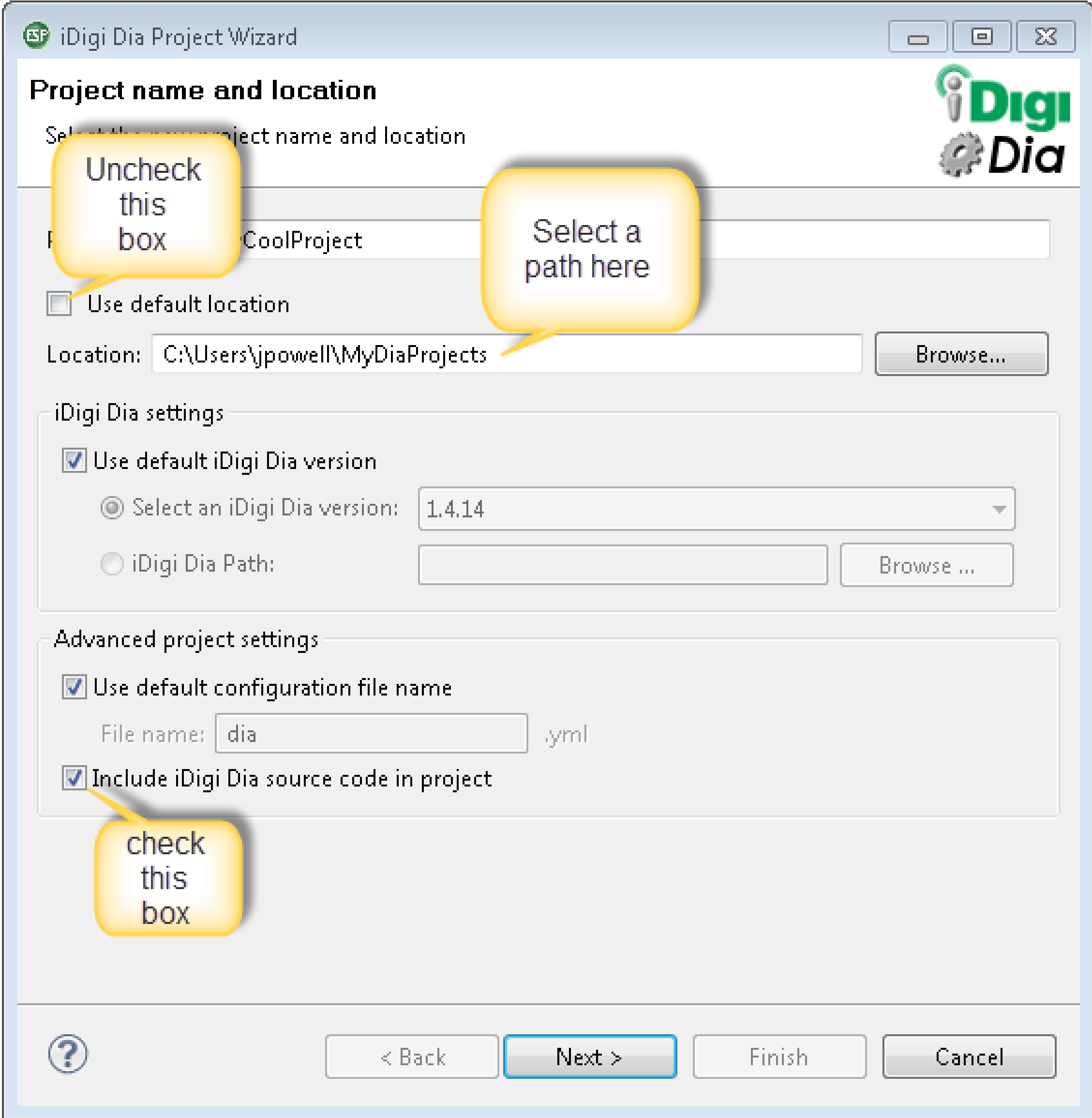
This is a bit trickier, as most distributions no longer have Python 2.4 in their repositories. Use your favorite search engine and search for instructions ("Ubuntu Python 2.4" for example). I was able to

easily find instructions for Ubuntu, and there are several resources on the web as this is a common request.

If you are only developing for the X2e platform, you should have no problem getting Python 2.7.1.

How to get the build tree

The easiest way is to create a new "DIA Project" in ESP. In the first window, give your project a name, uncheck default location, enter a path that is easy for your to access, and check the "include DIA source code in project" checkbox.



Doing the build

Now, the easy part!

From the root directory of the tree (you will see subdirectories such as src\, lib\, doc\, etc.) type:

```
C:\Python24\Python.exe .\make.py cfg\MyCoolProject.yml
```

For X2*e builds, use c:\Python27 instead*.

The path to Python.exe may need to modified to match your PC, but this is normally the correct path for Windows.

Tip Though it is normal convention to name the file "dia.yml" it is not necessary when building this way. You can have a whole bunch of .yml files for different projects, point to them during the build process, and it will work just fine. The yml (no matter what the source yml filename is) is serialized and named "dia.pyr" in dia.zip during the make process (see the 6th line of output when running the make).

You will hopefully see something like this:

```
Project make started.
Analyzing files...
Compiling files...
Zipping files...
Finished writing archive 'bin\dia.zip'
Transforming settings file 'cfg\MyCoolProject.yml' to 'pyr' format...
Adding transformed settings as 'dia.pyr' to 'bin\dia.zip'...
Project make completed successfully.
```

Your dia.zip (the built project) will be in the bin\ directory. Transfer that to the /WEB/Python/ directory on your ConnectPort/ConnectCore gateway. If you have not already done an ESP build from this version of DIA on your gateway, you will also need to copy dia.py (resides in the root directory of the build tree) to the same directory.

Core service - scheduler

Scheduling events

Most DIA devices will do things only from time to time. The DIA framework includes a simple callback event scheduler which enables callbacks to your device object in future times. This enables your device to do some action every X seconds, or react to other events failing to occur within Y seconds.

Example #1 - periodic polling

- You need to poll a remote device once per 5 minutes.
- At startup, schedule an initial poll_callback in perhaps 30 seconds or 1 minute (delayed to reduce startup congestion)
- During the poll_callback
  - Reschedule a new poll_callback in 5 minutes
  - Send your poll - you CANNOT wait for a response or you affect scheduling jitter (see Timing and Jitter below)
  - Schedule a response_callback in 10 second (note: some means of polling may cause an automation callback upon data receipt. Use of a separate response_callback is situation dependent.
- During the response callback
  - Handle any response data, or a timeout/no response
  - Handle any retry?

Example #2 - detecting when data not being received

- Your remote XBee adapter should send a new sample every 1 minute, you want to catch if no data has arrived in over 2 minutes.
- At startup, schedule a no_data_callback 125 seconds (over 2 minutes). Assuming the XBee device is already configured, it should return its first data sample within the next 60 seconds.
- During your data receive callback:
  - Cancel the old pending no_data_callback
  - Turn off any no_data alarm or signal that data is arriving
  - Reschedule a new no_data_callback for 125 seconds in the future
- During the no_data_callback:
  - Signal your data alarm as desired
  - Reschedule a new no_data_callback for the future - you may want it to be 30 times the expected data rate, so perhaps in 30 minutes or even several hours. The first good data sample received will clear the alarm condition and restore the no_data_callback to 125 seconds

Timing and Jitter

The DIA is not a hard real-time system. Scheduling a callback in 5 or even 5000 seconds does not mean you'll get a callback at the exact time requested. A single thread is doing these callbacks, plus your

callback runs in the context of that thread. That is why in the Example #1 above we are not supposed to wait for the poll. If it takes 6 seconds for a response, then any other DIA code expecting callback during those 6 seconds will NOT be called until after your response.

Your code must be a good-citizen. Never do anything in your callback which doesn't complete immediately.

Concurrency issues

If you have multiple callbacks due to multiple sources, they may be running in different thread contexts. For example, if you send a poll based on the 'scheduler' callback, but receive the data callback based on the XBee_Serial driver or other mechanism, then you may need to use semaphores (like threading.lock object) or the Python queue object. Without this you might find that a new poll callback is called WHILE an older data callback is processing.

Source

The simple DIA device file below shows examples of both uses of the scheduler.

The first 'kid' callback functions like the poll example above. It is rescheduled every 5 seconds for the first 25 seconds.

The second 'mom' callback functions like the no_data example above. It is also rescheduled every 5 seconds, but is to trigger at 10 seconds. This allows the 'kid' callback to cancel, so suppress the triggering of the 'mom' callback until the 'kid' gets tired of hiding from 'mom.

Download a zip

Blog_sched.zip

Output example

```
C:\py\dia\work>Python dia.py bin\dia.yml
Determining platform type... PC host environment assumed.

iDigi Device Integration Application Version 1.3.8
Using settings file: dia.yml
Core: initial garbage collection of 0 objects.
Core: post-settings garbage collection of 28 objects.
Starting Scheduler...
Starting Channel Manager...
Starting Device Driver Manager...
family: Starting at Wed Aug 18 08:59:14 2010
Starting Presentation Manager...
Starting Services Manager...
Core services started.

Kid says: (0, 'call 1') at Wed Aug 18 08:59:19 2010
          Mom can't find me!

Kid says: (1, 'call N') at Wed Aug 18 08:59:24 2010
          Mom can't find me!

Kid says: (2, 'call N') at Wed Aug 18 08:59:29 2010
          Mom can't find me!

Kid says: (3, 'call N') at Wed Aug 18 08:59:34 2010
          Mom can't find me!
```

```
Kid says: (4, 'call N') at Wed Aug 18 08:59:39 2010
          Where's Mom?

Mom says: that's it - time for bed!
family: Stopping at Wed Aug 18 08:59:44 2010
```

YML example

This YML assumes your device code is in the src\devices\experimental directory.

```
devices:
  - name: family
    driver: devices.experimental.blog_sched:BlogSchedDevice
```

Full Python code

```
# blog_sched.py
# - simple DIA device to show use the Core Services scheduler
#
# by Lynn August Linse, 2010-Aug-18

# imports
from devices.device_base import DeviceBase
from settings.settings_base import SettingsBase, Setting

import traceback
import time

class BlogSchedDevice(DeviceBase):

    def __init__(self, name, core_services):
        self.__name = name
        self.__core = core_services

        # we don't have any settings or properties

        ## Initialze the Devicebase interface:
        DeviceBase.__init__(self, self.__name, self.__core,
                                [], [])
        return

    def apply_settings(self):
        """Called when new configuration settings are available."""

        SettingsBase.merge_settings(self)
        accepted, rejected, not_found = SettingsBase.verify_settings(self)
        SettingsBase.commit_settings(self, accepted)
        return (accepted, rejected, not_found)

    def start(self):
        """Start the device driver.  Returns bool."""
        print '%s: Starting at %s' % (self.__name, time.ctime())

        try:
            # Fetch the Scheduler Service
            self.sched = self.__core.get_service('scheduler')

            # schedule first callback in 5 seconds - args just for show
```

```
            self.kid = self.sched.schedule_after( 5, self.__kid_call, (0,'call
1'))

            # schedule second callback in 10 seconds
            # - course, naughty kid cancels poor mom 5 time before letting her
run
            self.mom = self.sched.schedule_after( 10, self.__mom_call, ('time for
bed'))
            self.count = 0

        except:
            traceback.print_exc()
            print 'Scheduler Failed!'
            return False

        return True

    def stop(self):
        """Stop the device driver. Does nothing but returns True."""
        print '%s: Stopping at %s' % (self.__name, time.ctime())
        return True

    ## Locally defined functions:
    def __kid_call( self, args):
        """first callback event"""
        print '\nKid says: %s at %s' % (args, time.ctime())

        self.count += 1
        if self.count < 5:
            print "          Mom can't find me!"

            # cancel the pending second event callback
            self.sched.cancel( self.mom)

            # recreate our two new events in future
            self.kid = self.sched.schedule_after( 5, self.__kid_call,
(self.count,'call N'))
            self.mom = self.sched.schedule_after( 10, self.__mom_call, ('time for
bed'))

        else:
            # do NOT cancel the pending second/Mom event, which will trigger in a
few seconds
            # do NOT reschedule the first/Kid event - it stops forever at this
point!
            print "          Where's Mom?"
        return

    def __mom_call( self, args):
        """second callback event"""
        print "\nMom says: that's it - %s!" % args
        self.stop()
        return
```

DIA Config AIO adapter

Configuring the Digi AIO adapter in DIA

TODO - this page needs to be cleaned up.

Documentation notes

The general XBee Adapter manual 90000981.PDF lists the Analog Mode settings as 0 or 1, which are the actual hardware pin level, not the values to set via AT command. The AT settings to enable a digital output are either 4 (to output 0) or 5 (to output 1). Thus a more useful form of that table would be:

Terminal 1 Mode	Terminal 2 Mode	D8	D4	D6	Terminal 3 Mode	Terminal 4 Mode	P0	D7	P2
Current Loop	Current Loop	4 (low)	4 (low)	4 (low)	Current Loop	Current Loop	4 (low)	4 (low)	4 (low)
Current Loop	Ten V	4 (low)	4 (low)	5 (high)	Current Loop	Ten V	4 (low)	4 (low)	5 (high)
Ten V	Current Loop	4 (low)	5 (high)	4 (low)	Ten V	Current Loop	4 (low)	5 (high)	4 (low)
Ten V	Ten V	4 (low)	5 (high)	5 (high)	Ten V	Ten V	4 (low)	5 (high)	5 (high)
Differential	Differential	5 (high)	4 (low)	4 (low)	Differential	Differential	5 (high)	4 (low)	4 (low)

YML example

Here is an YML example from the DIA demo application called the "Tank Demo".

```
- name: aio_left
  driver: devices.xbee.xbee_devices.xbee_aio:XBeeAIO
  settings:
      xbee_device_manager: xbee_device_manager
      extended_address: "00:13:a2:00:40:0a:49:7a!"
      sample_rate_ms: 1000
      power: "On"
      channel1_mode: "TenV"
      channel2_mode: "TenV"
      channel3_mode: "TenV"
      channel4_mode: "TenV"
```

DocString from Python code

Below is the docstring copied from the driver file in ./src/devices/xbee/xbee_devices/xbee_aio.py. It is here since many users are not comfortable trying to browse or read Python files.

```
"""\
An Example DIA Driver for the XBee Analog IO Adapter
Settings:
```

```
    xbee_device_manager: must be set to the name of an XBeeDeviceManager
                         instance.

    extended_address: the extended address of the XBee Sensor device you
                      would like to monitor.
    sleep: True/False setting which determines if we should put the
           device to sleep between samples.
    sample_rate_ms: the sample rate of the XBee adapter.
    power: True/False setting to enable/disable the power output
           on terminal 6 of the adapter.
    channel1_mode: Operating input mode for pin 1 of the adapter.
                    Must be a string value comprised of one of the following:
                    "TenV" - 0-10v input available on any channel.
                    "CurrentLoop" - 0-20 mA current loop available on
                                    any channel.
                    "Differential" - +/- 2.4a differential current mode
                                     enabled on channel1 & channel2 or
                                     channel3 & channel4.
    channel2_mode: Operating input mode for pin 2 of the adapter.
                   See channel1_mode for valid setting information.
    channel3_mode: Operating input mode for pin 3 of the adapter.
                   See channel1_mode for valid setting information.
    channel4_mode: Operating input mode for pin 4 of the adapter.
                   See channel1_mode for valid setting information.
    awake_time_ms: How many milliseconds should the device remain
                   awake after waking from sleep.
    sample_predelay: How long, in milliseconds, to wait after waking
                     up from sleep before taking a sample from the
                     inputs.
"""
```

DIA Device - alarm clock

Alarm clock device

The Alarm Clock device is a low-speed general resource which can help other devices accomplish simple timed actions. It is designed to work with minutes or hours. **Users who needed timed behavior faster than once per minute should use their own thread and timer logic**.

At present it only offers:

- the ability to trigger a transform (or publish a set) every:
  - 'minute' = once per minute, when seconds = 0
  - 'hour' = once per hour, when minutes & seconds = 0
  - 'six_hour' = once per 6 hours, so at 00:05:00, 06:05:00, 12:05:00 and 18:05:00
  - 'day' = once per day, so at 00:00:00 / midnight
- It can print the line "{dev_name}: time is now 2009-05-31 10:47:00" at any of the above time intervals.

TODO

- Add a cron-like ability for other devices to request publish/sets at times such at 3:27am each Tuesday.

- Consider adding a second 'helper' thread to run complex/long jobs. For example STMP email service might take longer than 1 minute to return control to the alarm_clock thread.

Connections - inputs

There are currently no block inputs.

Connections - outputs

These are the block outputs.

Name	Type	Description
minute	tuple	Published once per minute (will always be True)..
hour	tuple	Published once per hour on the hour (will always be True).
six_hour	tuple	Published once per 6 hours, at 5 min after the hour (will always be True).
day	tuple	Published once per day, at 10 minutes after midnight (will always be True).

Notes

- The tuple sent is of form (time(), tuple(localtime(time())).
- It is converted to a tuple since the same object reference is passed to all subscribers.
- An actual example is: (1244544360.0, (2009, 6, 9, 10, 46, 0, 1, 160, -1))
- Your device might MISS sets/publishes - for example another task might cause the alarm clock to be busy LONGER than 1 minute. So you device should examine the TIME value in the set/publish to determine how many seconds have really occurred since the last event.

Settings

These are the block settings which affect operation; some should be saved to NVRAM while others are dynamic.

Name	Type	Default	Lifespan	Description
tick_rate	int	60	NVRAM	seconds between ticks - change only for debug purposes!
printf	str	'minute'	NVRAM	if and when alarm_clock prints a 'time is now' line in trace, in set ['none','minute','hour','six_hour','day']

Notes

Changing tick_rate does NOT affect the minutes etc, as those are based the real time clock; this only affects how often the thread wakes. Setting it to 15 means the thread wakes up 4 times for even 'minute' event. Setting it to 300 means 4 of 5 minute events are missed.

The Python code

I place this in my src\devices directory. However, it has few dependencies and could be placed anywhere (it affects the YML form).

YML example

This example shows the alarm clock device being named 'tick_tock', plus a second device which subscribes to tick_tock.minute, which allows that device to run once per minute without requiring a thread of it's own. *The HourMeterBlock is NOT part of DIA - it is a custom user-defined device.*

```
- name: tick_tock
  driver: devices.alarm_clock_device:AlarmClockDevice
  settings:
      printf: minute
- name: motor_01_hours
  driver: devices.blocks.counter_device:HourMeterBlock
  settings:
      active_true: True
      input_source: motor_01.channel1_input
      tick_source: tick_tock.minute
```

Sample output

This shows the once-per-minute printf with AIO adapters sleeping for 2 minutes at a time.

```
tick_tock: time is now 2009-06-10 09:23:00

XBeeAIO(smart01): AD0=0000 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank03): AD0=0501 AD1=0000 AD2=0000 AD3=0000 bat=okaytick_tock: time is
now 2009-06-10 09:24:00
XBeeAIO(itank01): AD0=0153 AD1=0000 AD2=0000 AD3=0000 bat=okay

XBeeAIO(ssi01): AD0=0087 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank02): AD0=0502 AD1=0000 AD2=0000 AD3=0000 bat=okay
tick_tock: time is now 2009-06-10 09:25:00
XBeeAIO(smart01): AD0=0000 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank03): AD0=0501 AD1=0000 AD2=0000 AD3=0000 bat=okay
```

Python Code

2009jun09_alarm_clock_device.zip

Dec-2009 Enhancements - Events

In the Dec 2009 version I have added the ability to create time-based boolean event channels. So for example, you could create a channel named 'xmas' which is True from 6PM until 10PM, and False the rest of each day. Needless to say, you could subscribe to such a channel from a Digi SmartPlug, and use it to turn On/Off your outdoor Christmas lights.

New Details

This new (Dec 2009) version allows an optional 'action_list' section, following which you can add a number of events. Each event creates a NEW output channel on the AlarmClockDevice - so in the example below the new channel named **tick_tock.xmas** becomes available. In this example, the channel tick_tock.xmas will be True during the period from about 18:00 (6pm) until 23:00 (11pm).

The 'per' setting can also be set to 'hourly', in which case the 'on'/'off' should be minutes of the hour, so an output can be true at some arbitrary minutes-after-the-hour. Note that since one of my goals is to NOT load the CPU for unnecessary accuracy, it is possible the event triggers up to a minute late. So the outdoor lights might turn on at 6:01pm instead of 6:00pm.

My main TODO item is add the notion of which days of the week - if you look into the code you see that the 'do-daily' code is 0x007F, so 7 true bits. The design goal would be to enable causing the event to fire only once per week (say on Wednesday at 3am), or only have it occur on weekdays or weekends.

Example YML for Christmas lights

```
devices:

  - name: tick_tock
    driver: devices.experimental.alarm_clock_device:AlarmClockDevice
    settings:
        tick_rate: 60
        printf: minute
        action_list:
            - event: { 'nam':'xmas', 'per':'daily', 'on':'18:00', 'off':'23:00' }
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
  - name: my_lights
    driver: devices.xbee.xbee_devices.xbee_rpm:XBeeRPM
    settings:
        xbee_device_manager: "xbee_device_manager"
        extended_address: "00:13:a2:00:40:48:5a:65!"
        sample_rate_ms: 5000
        default_state: Off
        power_on_source: tick_tock.xmas
```

Python code

2009jun09_alarm_clock_device.zip

DIA device - Runtime Totalizer

Runtime Totalizer device

Many systems benefit from tracking how long things run (how long they are on or active or True). This example DIA device logs the time period during which boolean (True/False) samples are True. The totals are saved to an ASCII text file which can be read via the web interface, plus is reloaded when the gateway reboots.

This device is also a nice example of "trade-off". It only totalizes the samples at fixed periods (for example once per minute), and it assumes the input has been in its current state for the entire past minute. It also only saves the NVRAM state at a slow rate, for example once per 15 minutes.

Could you totalize in milliseconds? Yes, of course you could. Is it useful? Probably not. If your goal is to understand how many kilowatt-hours a room light was on per month, then tracking in minutes verse milliseconds is not likely to be statistically interesting. Could you save the totals to FLASH every second? Yes, of course you could, but then your Digi gateway's FLASH might fail within the year. Assuming the gateway only reboots due to power outages once every few months, then a few dozen minutes a season are not statistically interesting.

Fortunately, the Runetime Totalizer device allows you to control the rates. The Python code included does NOT use it's own thread. Instead, it subscribes to an external DIA sample which produces period refreshes of data. This could be any IO sensor which produces data periodically, or the DIA Device - alarm clock.

Configuration and settings

The Setting for the **RunTimeDevice** consist of a paired list of **channel** and **name**. Channel is the input sample to monitor with type Boolean, and Name is the totalized value output with type float. The channel/name list must be balanced, as otherwise the DIA does not treat the YML data correctly.

The **RunTimeDevice** is passive - it has no thread. It requires an external Sample refresh via the DIA's powerful publish-subscribe subsystem to "Push" it into action. Thus these two SPECIAL channels exist (see the YML example below).

- **time_source** is any periodic sample which causes the RunTimeDevice to evaluate and totalize the other channels. In the example below it is the Minute sample from the Alarm Clock Device. The type of data sample is not important; it is the act of update/refresh which causes the RunTimeDevice to process one cycle, and it uses the change in real-time since the last cycle to update the totals.
- **publish_source** is any periodic sample which causes the RunTimeDevice to refresh/update it's own collection of Samples out, plus saves the data to the NVRAM (flash) of the Digi gateway.

These two special-settings are handled this way because the author could not get DIA to handle the YML list of undefined size correctly if the settings did not consist COMPLETELY of these pairs. This might change in the future

Example YML File for a DIO Adapter with 4 inputs

This example defines six (6) channels, two of which are special cases:

- **time_source** causes the totals to be calculated once per minute, but they are not published nor saved to NVRAM/flash at this time.

- **publish_source** causes the totals to be published and saved to NVRAM/flash once per 15 minutes
- **lights, pc_stuff, chargers, light_active** are the four totalizers, which track the on (power-up) time of four external power relays.

```
- name: my_cube
  driver: devices.experimental.runtime_device:RunTimeDevice
  settings:
    - channel: tick_tock.minute
      name: time_source
    - channel: tick_tock.15_min
      name: publish_source
    - channel: relays.relay1_status
      name: lights
    - channel: relays.relay2_status
      name: pc_stuff
    - channel: relays.relay3_status
      name: chargers
    - channel: relays.relay4_status
      name: light_active
```

Example ASCII Text file saved by RunTimeDevice

The text file named **RunTotals.txt** is saved in the WEB/Python/ directory. You can access it from the web interface in the same place in which you see Python files. Literally this is a string representation of a Python dictionary, so order is NOT predictable. Technically you can edit this file carefully and then reboot the DIA system to "reload" changed values.

```
{'lights': 54784, 'pc_stuff': 250684, 'light_active': 101462, '_total_time':
1440243, 'chargers': 0}
```

Download info

Files in the runtime_device.zip

- **readme_runtime_device.txt** is a summary similar to this Wiki page.
- **src\devices\experimental\ia_trace.py** is a simple trace facility which allows increasing or decreasing the chattiness of the print output from DIA - the class RunTimeDevice() inherits from it.
- **src\devices\experimental\string_file.py** is routines to save and reload a single Python object saved as "string", and reload with "eval(string)" - it is a simplistic pickler.
- **src\devices\experimental\runtime_device.py** is the DIA device driver to be referenced in your YML file.

Python code

2009oct12_runtime_device.zip

DIA device - sample rate reducer filter

Sample rate reduction/filter device

Consider a DIA system with 10 tanks and sleeping level sensors which wake once per hour to upload new data to X4 and DIA. Do you really need to push 240 data samples per day over your cost-sensitive cellular link into the Device Cloud? What if on the average day, only 1 or 2 of those tank levels even change?

The DIA 'Filter Device' allows you to add very simple intelligence around the forwarding of new data. In this tank example, it would allow the DIA to apply the logic "**update Device Cloud only when the tank level changes by at least 1%, plus never update more often than once per hour, but always update at least once per day**." This could reduce the total data samples moved over cellular to only 15 or 20 samples per day - while still allowing the central host to see tank level changes within an hour of the change.

The filter_device acts as a simple, time-aware filter (or transform) which applies process knowledge to reduce the frequency of sample updates. In the above example, the DIA configuration would start with ten (10) sensor device objects which blindly produce new data samples every hour - so 240 changes per day. These might have names like ['TNK01in', 'TNK02in' ...] and so on, which means 'TNK01.channel1_value' might be the level in the first tank.

Ten (10) new filter_device objects would be created to selectively copy or mirror the output of those sensor device objects, having names like ['tank01', 'tank02' ...] and so on. They might apply a logic with the Python dictionary: { 'delta':1.0, 'atleast':'24 hr', 'atmost':'59 min' } They might also rename their 'input' to become output as 'level'. So they create new channels such as 'tank01.level' which change at most 24 times per day, but at least 1 time per day. If idigi_db uploads the channels from the filter_ device instead of from the sensors, then you will have less cellular traffic.

Filter conditions / configuration

By default each filter_device supports at most 4 channels - this is hard-coded within the filter_ device.py since it creates database items EVEN for channels which are NOT configured. The syntax of the filter_device is closer to Python to reduce the system overhead in created such 'mirror channels' on a large DIA system.

The filter_device is configured with:

- A tag name ['tag1', 'tag2', ... 'tagN'] with a source channel name such as template.counter or m3_01.temperature. By default, the outgoing filter_device channel has the same name as the source. Duplicate names causes exceptions to be thrown.
- An optional rename ['rename1', 'rename2', ... 'renameN'] which can over-ride the source channel name. For example, instead of a name like 'distance', you might want 'level'.
- A filter condition dictionary ['filter1', 'filter2', ... 'filterN'] which defines the list of conditions to suppress or cause the output channel to be updated.

Filter conditions

- All channels will produce an initial sample even if the conditions are not met.
- An empty dictionary such as "filter1: {}" causes the data to always propagate, otherwise by default the data is NOT propagated if conditions exist.
- The "**atleast**" clause over-rides all others and produces a new data sample AT LEAST as often as configured time in SECONDS. Examples are { 'atleast':'24 hr'} or { 'atleast':86400}, where the string '24 hr' or '1 day' would be internally converted to 86,400 seconds.
- The "**atmost**" clause also over-rides others and suppresses new data samples to happen no more that the configured time in SECONDS. Using filter conditions to track 'events' might result in data loss if 'atmost' suppresses samples. Examples are { 'atmost':'15 min'} or { 'atmost':900}
- The "**delta**" clause produces a new data sample if the value changes up or down by at least the value configured. The value depends on the units, so for a level measured in inches the delta-value would be inches. Examples are { 'delta':0.5} or { 'delta':25}
- The "**below**" clause produces a new data sample if the value is less-than (below) the value configured. There is no hysteresis, so an 'atmost' clause may be required to slow down new data samples.
- The "**above**" clause produces a new data sample if the value is greater-than (above) the value configured. There is no hysteresis, so an 'atmost' clause may be required to slow down new data samples.
- The "**equal**" clause produces a new data sample if the value is the same as the value configured. Be warned that sensor inputs treated as floats will rare 'equal' a constant such as 2.5 or 100. (It is a TODO item to offer a solution for this)
- The "**round**" clause is processed ONLY if the sample is being produced, and it forces the sample to become a float. It is primarily of use when data uploads are handles as ASCII text. For example, it makes no sense to upload a temperature as "21.094400000000007" if the accuracy is +/- 1 degree. Adding the condition { 'round':1 } would round the float to 21.1000000, which normally will be ASCII encoded as only "21.1"

Example YML file for TemplateDevice

This example produces two new channels with the following characteristics:

- **m3_01.counter** only produces a new sample (SETs a new output) if the data sample is greater than 6. However it produces atleast 1 sample every 5 minutes (300 seconds) and never produces more than 1 new sample every 1 minute and 10 seconds (70 seconds). This channel inherited its name from the source channel template.counter
- **m3_01.other** produces a new sample (SETs a new output) after the source channel has changed at least by a value of 3 (example: 1, 4, 7 etc). It has no time restrictions.

```
devices:

  - name: template
    driver: devices.template_device:TemplateDevice
    settings:
        count_init: 0
        update_rate: 1.0

  - name: m3_01a
    driver: devices.filter_device:FilterBlockDevice
    settings:
        tag1: template.counter
        filter1: { 'above':6, 'atleast':'5 min', 'atmost':'1 min 10 sec' }
        tag2: template.counter
        rename2: 'other'
        filter2: { 'delta':3 }
```

Example YML file for Massa M3 ultrasonic sensor

The Massa M3 battery powered device wakes on schedule - for example once per hour - and pushes new data into the Device Cloud/DIA channel database. However, this does NOT mean that a new tank level has been seen. Also the Massa M3 driver produces about 12 channels, all which are 'refreshed' with a new sample at this rate. So just uploading all changed channels wastes a lot of cellular bandwidth.

This example produces four new channels with the following characteristics:

- **m3_01.distance** produces a new sample if the level changed by at least 0.5 inch, but produces atleast 1 sample every 3 hours and never more than 1 new sample per hour.
- **m3_01.temperature** produces a new sample if the temperature changed by at least 2 degrees, but produces atleast 1 sample every day and never more than 1 per 2 hours.
- **m3_01.target_strength** only produces a new sample if the ultra-sonic signal quality is less than 50%, never more than 1 per hour.
- **m3_01.battery** produces a 1 new sample every day.

```
devices:

  - name: m3_01
    driver: devices.vendors.massa.massa_m3:MassaM3
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:4b:ae:b0!"
        poll_rate_sec: 3600
        sample_rate_sec: 3600

  - name: m3_01a
    driver: devices.filter_device:FilterBlockDevice
    settings:
        tag1: m3_01.distance
        filter1: { 'delta':0.5, 'atleast':'3 hr', 'atmost':'1 hr' }
        tag2: m3_01.temperature
        filter2: { 'delta':2.0, 'atleast':'24 hr', 'atmost':'2 hr' }
        tag3: m3_01.target_strength
        filter3: { 'below':50, 'atmost':'1 hr' }
        tag4: m3_01.battery
        filter4: { 'atleast':'24 hr' }
```

Download info

Files in the filter_device.zip

- **readme_filter_device.txt** is a summary similar to this Wiki page.
- **src\common\helpers\parse_duration.py** is a Python routine to convert strings like '24 hr' into msec or sec. It supports the tags ['ms','sec','min','hr','day']
- **src\common\helpers_test_parse_duration.py** is a Python regression test routine.
- **src\devices\filter_device.py** is the DIA device driver to be referenced in your YML file.

Python code

2009oct09_filter_device.zip

TODO

- The tag interface is cumbersome and wasteful - having tags named tag1/filter1 and tag2/filter2 is not ideal. A method should be added to accept them all as tag/filter and be handled properly.
- Since 'equal' doesn't work with floats, adding a 'nearly' tag to match when a float is within some tolerance of a target would be good.
- Enable inverting boolean conditions, so a sample of 'True' can be forwarded as 'False'.

DIA difference between ZigBee and DigiMesh

Mesh setup

Although not related to DIA, people familiar with ZigBee setup will find DigiMesh a new experience.

Unlike ZigBee where new nodes can sometimes merely be powered up and joined to the correct network, DigiMesh works more like the older ZNet design where you will require either:

- Use XCTU and an XBIB board to manually configure the required Network setting
- Use a 'commissioning node' in an XBIB board with default settings to locate the new factory-fresh DigiMesh device, then using the 'Remote Configuration' setting, push the required Network settings into the new device.

See this Wiki page for a summary of the required parameters: Quick guide to DigiMesh setup.

Understanding sleeping

There is a basic difference between ZigBee and DigiMesh.

- ZigBee has powered, non-sleeping devices called routers (or parents). The coordinator is just a router with an extra role.
  - Routers 'mesh', forming the resilient mesh-topology hyped by ZigBee advocates.
  - Routers discover routes (paths) and adjust routes as old peers go offline and new peers come online.
  - Routers repeat broadcasts
  - In general routers don't say much - unless asked to move data.
- ZigBee also has sleeping end-devices (or children).
  - End-Devices do not mesh. They locate ONE specific router/parent to belong to.
  - When End-Devices wake up, they start polling their parent 10-times per second, sending data or looking for data the parent has buffered.
  - End-Devices do NOT repeat broadcasts
  - However, end-devices are rather chatty as long as they are awake!

DigiMesh

- All devices in DigiMesh are routers, forming the mesh
- DigiMesh devices can sleep, or be fully awake.
- All nodes need to understand the wake/sleep cycle, and only talk during wake cycles.

DigiMesh in DIA

What this means to DIA (and YML files) is that any ZigBee node can have any desired data sample rate. You can have 20 temperature sensors, all with random sample rates between 1 second and 1 day. The fully awake routers can send data at any time, and the sleeping end-devices can wake anytime, trusting that their parent is awake, ready and waiting to handle their data.

In contrast, under DigiMesh all data samples need to be an limited integral multiple of the entire mesh's sleep/wake cycle.

DIA/DigiMesh sleep coordinator

Within DIA, the Digi gateway acts as the designated sleep coordinator. This means DIA sets the sleep/wake times within the XBee within the gateway, and the gateway imposes the sleep/wake cycle over the entire mesh.

This is a DIA limitation, not a DigiMesh one. As of this release, DIA does not support nominated sleep coordinators, which is more appropriate to a peer-to-peer system. Most DIA systems are gateway-centric, assuming the gateway is critical and the devices have little or no function without the gateway, and therefore their sleep-coordinator.

DIA/DigiMesh sleep mode

To disable all sleeping, set the DigiMeshDeviceManager's **sleep_time** setting to zero. The SM value in all nodes is forced to 0x00.

To enable sleeping, set the DigiMeshDeviceManager's **sleep_time** and **wake_time** settings to a value higher than 10msec - for example 20000 and 10000 respectively, which means the mesh sleeps for 20 seconds, then is awake for 10 seconds. The SM value in all nodes is forced to either 0x07 or 0x08, which means 'always awake but sleep aware' and 'sleep'.

This 20/10 design means for 20 seconds none of the nodes transmit data - even the fully awake nodes (SM=0x07). Every device waits until the mesh wakes, then they all try to send messages as required, including forwarding (and 'meshing') as required. Of course the wake_time needs to be long enough to allow all the required messages to move, including passing across multiple hops.

This is a DIA limitation, not a DigiMesh one. DigiMesh includes some richer asynchronous sleep-modes, which allow nodes to wake up during the sleep-period and talk, however this implies the sleeping node is within 'ear shot' of a fully awake node since no other sleeping node will be awake to help route the message.

DIA/DigiMesh data sample rates

At this point, all nodes can use external logic to decide when (during any particular wake-time) that they wish to send data. If the node has its own processor, it can format a message and send it during the beginning portion of the wake cycle, hopefully having time to receive any responses. The DIA framework on the gateway can also send out polls. The DigiMeshDeviceManager will queue up any pending outgoing messages from DIA drivers, then sending them out when the mesh wakes.

Support for the Digi XBee's IC/IR commands is also supported. By default, a node will send data each time it awakes. This requires IR=0xFFFF, IF=0x01 and at least 1 of the XBee IO pins be configured as an input.

The DigiMeshDeviceManager's **set_if** setting can be set to True or False. If set_if=False, then the DIA does NOT interfere with the IC/IR/IF settings within the remote nodes. DIA assumes that the user has configured the remote devices as required.

If set_if=True, then the DIA uses the various sample_rate setting of drivers to calculate the correct IR/IF values. For example, if the device wishes a once per minute data sample and the mesh is waking every 30 seconds (for example, our 20/10 sleep_time/wake_time settings), the setting IR=0xFFFF and IF=0x02 means the XBee will send a data sample every second wake cycle. The DIA will round down, so if the driver sample_rate is once per 75 seconds with a wake-cycle of 30 seconds (20/10 sleep/wake), then DIA is forced to select IF=0x02, which will give you data every 60 seconds.

DIA Drivers and Presentations

Drivers

Each driver is in a zip archive. Drivers include source files as well as a DIA configuration example.

Presentations

Each driver is in a zip archive. Drivers include source files as well as DIA configuration example.

DIA event uploader

Uploading events or discrete data samples to DIA

Comparing iDigi_DB to iDigi_Upload

The stock iDigi upload presentation (named iDigi_DB) assumes you only want a sampling (or subset or snap-shot) of data samples handled by DIA. For example, if 20 tanks send new level reading to Dia once per minute, this amounts to 1200 samples per hour. Yet the Device Cloud host may not desire any more than 20 samples per hour - just one per tank per hour. Therefore the stock iDigi_DB presentation is configured with an **interval** setting (for example: 3600 seconds or 1 hour), then all configured channels are sampled and uploaded to Device Cloud once per hour.

This design does not support uploading real time events. For example, if a door sensor is monitored, then Device Cloud will only see the door status on the hour (the sample interval). Device Cloud will not record that the door was opened three times during the hour.

iDigi_Upload is a custom adaptation of the original idigi_db.py file, modified to save a copy of every new channel sample for upload. These are saved in a cache and only cleared after successful upload.

Description	iDigi_DB	iDigi_Upload
Are discrete events uploaded to Device Cloud?	No - only the I/O status at the sample interval	Yes - every new sample is uploaded
If upload fails, are samples saved and retried?	No - new samples will be taken at the next sample interval	Yes - copies are made of every sample and only deleted after successful upload (unless cache grows beyond configured limit)
If a channel doesn't change, is it uploaded every interval?	Yes - all channels are sampled at each sample interval	No - copies are made only when sample change
Is the upload size limited per interval?	Yes - at most one sample of every channels is sent	No - every changed sample is uploaded
Is memory usage limited?	Yes - existing samples are used to create upload	No - a duplicate copy of every changed sample is held until upload

New settings

iDigi_Upload supports the original iDigi_DB settings, plus adds two new settings:

cache_size

Is the maximum number of bytes of flash to use buffering for upload. It defaults as 0, which disables this drastic changes of behavior.

- When **cache_size** = 0, idigi_upload holds all data in memory only. Restarting the gateway will discard your data.

- When **cache_size** is set > 0, then idigi_upload queues a deep-copy of every new sample first in memory, then saves this to flash approximately once a minute. You will see files such as 'data_cache.txt' appear and disappear from your Python directory. These are cleared only after upload or when the caches fills, then in FIFO design the oldest data is lost. NOTE that the design keeps a backup when deleting the cache, so the actual size of flash required will be twice the configured amount.

clean_minute_interval

Is an alternative to the base **interval** setting. It accepts only minutes in the set (0, 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60)! It causes upload ON THE HOUR, then minute intervals in a predictable pattern. So setting 10 cause uploads at 00:00:00, 00:10:00, 00:20:00 and so on.

- Setting **clean_minute_interval** to zero (0) causes the base interval setting to be used.
- Setting **clean_minute_interval** to non-zero over-rides the interval setting.

Download Info

Python Code

This version requires Dia 1.4.14 or newer. The files includes are:

- src\common\helpers\sleep_aids.py
- src\samples\annotated_sample.py
- src\presentations\idigi_upload__init__.py
- src\presentations\idigi_upload\idigi_upload.py
- src\presentations\idigi_upload\string_file.py
- 2012feb24_idigi_upload.zip

DIA Releases

DIA Release 2.0.x (May 2012)

Note: this version is not released yet.

DigiMesh support

One of the largest changes is the introduction of two different XBee manager devices, so your YML file should now use either of these:

```
  driver: devices.xbee.xbee_device_manager.zigbee_device_
manager:ZigBeeDeviceManager
```

Or

```
  driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
```

Although ZigBee users should migrate to call the ZigBeeDeviceManager explicitly, YML files including the old XBeeDeviceManager will run the ZigBeeDeviceManager instead.

DigiMesh sleep design

The DIA DigiMeshDeviceManager assumes 1 of 2 designs:

- Non-sleeping, so all nodes run as SM=0
- Sleeping, with the gateway acting as the preferred Sleep Coordinator. Nodes will be set to:
  - Gateway/Sleep-Coordinator: SM=7, SO=0x05
  - Non-Sleeping Node: SM=7, SO=0x02
  - Sleeping Node: SM=8, SO=0x02

Support for sleep-coordinator 'Nomination' and the asynchronous/pin-sleep modes of DigiMesh are not yet supported.

DigiMesh YML changes

Here is an YML usage for the DigiMeshDeviceManager:

```
- name: xbman
  driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
  settings:
      dh_dl_force: True
      dh_dl_refresh_min: 300
      sleep_time: 9000
      wake_time: 1000
      set_if: True
```

The **dh_dl_force** and **dh_dl_refresh_min** settings apply to both ZigBee and DigiMesh systems and will be explained in another section of this document.

The three DigiMesh specific settings are:

- **sleep_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes wake up every 2 seconds. Set this to 0 to disable sleeping.

- **wake_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes to stay awake for 2 seconds. If sleep_time = 0, this setting is ignored.
- **set_if** is a boolean which defines if DIA will adjust the IR/IF settings in remote nodes. It defaults to False, which assumes the user manually handles data messages. If set to True, then DIA makes a best-effort to map individual driver 'sample_rate' settings to XBee IF intervals.

For example, if a XBee DM AIO adapter has a sample_rate_ms setting of 60000 (once a minute). The default DigiMeshDeviceManager settings of sleep_time = 2 seconds and wake_time = 2 seconds means a sleep-cycle of 4 seconds total. So DIA will set the XBee DM AIO adapter with IR=0xFFFF and IF=0x0F, which means the AIO adapter will send in an IO data sample when it wakes within the 15th sleep-cycle (ie: 60-sec / 4-sec). DIA rounds down if the rates do not factor cleanly.

Internal Driver Changes to Support DigiMesh

Users wishing their custom driver code to run under DigiMesh will need to make some changes to their drivers.

First, drivers must not directly set SM, SO, or IR - they must submit the required settings using the XBee Manager. For example:

```
        xbee_sleep_cfg = self._xbee_manager.get_sleep_block(
                                    self._extended_address,
                                    sleep=False,
                                    sleep_rate_ms=sample_rate,
                                    awake_time_ms=0)
```

The XBee Manager (ZigBee or DigiMesh) uses the 'sleep' parameter to select the correct SM/SO settings. The 'sleep_rate_ms' and 'awake_time_ms' parameters are used to set the setting SN/SP/ST for ZigBee and IR/IF for DigiMesh.

For clearer examples, look at the DIA drivers for the Digi Adapters.

Second, your driver should no longer set the DH or DL settings directly. The support for these have been moved to the XBee Manager since the values required may be different for different XBee technology.

Destination Address (DH/DL) Support

In this release of DIA, drivers should no longer directly set the DH/DL of XBee nodes. This function has moved to the XBeeDeviceManager classes to support different behavior required by different Xbee technologies.

Two new settings have been added which can be used with both ZigBee and DigiMesh systems:

- **dh_dl_force** is a string and can be:
    - None or False, which means do NOT change DH/DL in any node
    - True, which means do what is most appropriate for the XBee technology. For ZigBee, this forces the default aggregator address of '00:00:00:00:00:00:00:00!' into all DH/DL. For DigiMesh, this forces the gateway's SH/SL address into all DH/DL.
    - Coordinator, which means use the gateway or coordinator SH/SL address for all nodes DH/DL
    - The string of an exact address such as '00:13:a2:00:40:32:d9:51!' which is to be used.

- **dh_dl_refresh_min** is a string which defines when the DH/DL setting is affected.
  - None, which means never change DH/DL at all
  - Once, which means DH/DL will be set when nodes are configured, and also broadcast once.
  - Config, which means DH/DL will only be set when nodes are configured, and not broadcast.
  - A time with tag such as '5 min' or '1 day' which defines a repeated broadcast interval. if no tag is applied, then a number is assumed to be in minutes.

Rapid reboot detection

Many Digi gateways include an option to reboot if an auto-started Python script exits. This is a wise precaution against an expected error occurring months after reboot, but can make the unit unreachable by Device Cloud if the error occurs repeatedly at the start of the script.

For example, if the main Python script has a simple typo or a ZIP file was accidentally truncated during download, then the auto-started Python script may exit instantly, forcing a reboot within seconds of the last reboot and startup. A gateway locked in such a reboot-cycle will never be connected to Device Cloud long enough to allow fixing the problem.

This release of DIA addresses this within the main **dia.py** file. By default this feature is disabled, so users wishing this protection must manually create (or seed) a text file named nospin.txt in the Python area of the gateway.

The file name is case-sensitive, so prevent allowing Windows to rename your file Nospin.txt!

The file can be empty or contain a line of text. Once active, the file will be rewritten by dia.py upon every reboot with a timestamp, and if dia.py detects that the last 10 reboots have occurred in less than 20 minutes, then dia.py will sleep for 10 minutes before trying to start.

If you enable this feature on a gateway without time service (so it always boots as 1-Jan-1970), then after 10 reboots the gateway will always delay starting for 10 minutes. This may be undesirable, but not as undesirable as becoming unreachable from Device Cloud.

Users writing their own auto-start code should consider copying this same function to their own applications.

Other internal driver changes

Older DIA drivers tended to make very poor use of Python or object oriented paradigms. For example, much of the code within most DIA Xbee drivers is duplicated in all of the peers - literally cut-and-pasted. This is a clear violation of Object Programming concepts which assume that if all derived classes will need the same code fragment, then it should be handled by the base class, not repeated within all derived classes.

Tracer Module

Besides the previous tracer-levels of ('debug', 'info', 'warning', 'error', 'critical'), two new lower levels are defined BELOW debug.

tracer.calls

These should be used for simple debug statements showing a routine was called. This was added because some programmers pepper too many 'debug' statements such as:

```
def calculate_average(self, a, b):
    # this should now be self._tracer.calls() instead
    self._tracer.debug("calculate_average")
```

These tend to pollute the debug trace with considerable low-grade information. Defining a new sub-debug level named 'calls' allows the user to enable/disable these simple, low-grade debug lines upon demand.

tracer.xbee

For example, these are used by the DigiMesh manager to announce all of the mesh-wake and mesh-sleep messages received from the gateway XBee. This information is generally used only when debugging the xbee manager, or sleep/performance problems with drivers.

tracer lines now return a boolean value, True if active

This should be used in situations where the tracer parameters are costly to format. In the example below, the parameter is a time-expensive operation - a STRING conversion of a list of hundreds of integers. Python must evaluate all parameters whether the tracer level is true or not, which can cost a huge time penalty even when tracing is not enabled. Using an if-then statement greatly reduces the performance hit.

```
if self._tracer.debug():
    # my_list is converted to a string ONLY when debug is True
    elf._tracer.debug("list data:%s", str(my_list))
```

class DeviceBase

The DeviceBase now creates three variables which can be used by derived classes:

- **self._name** = name as passed in by def __init__(self, name, core_services, settings, properties)
- **self._core** = core_services as passed in by def __init__(self, name, core_services, settings, properties)
- **self._tracer** = get_tracer(name)

In past DIA versions, most derived class duplicated the effort to create and manage these as self.__ name, self.__core, and self.__tracer. These cause no harm, but waste resources and prevent derived classes from being created from derived classes. When porting drivers to the new DIA, you are encouraged to use DeviceBase's copies of these variables.

The DeviceBase also now has a setting named 'trace', which defaults to , which means use global trace level in self._tracer. It can hold any valid tracer-level for the Tracer module, such as 'debug', 'info', and so on.

This is used when a user has for example 20 devices, but wants only 1 of them to output tracer.debug lines, and the other 19 to output tracer.info lines.

class XBeeBase

The XBeeBase now creates and manages two variables which can be used by derived classes:

- self._xbee_manager, which is set by the code:

```
    # Fetch the XBee Manager name from the Settings Manager:
    dm = self._core.get_service("device_driver_manager")
    self._xbee_manager = dm.instance_get(
            SettingsBase.get_setting(self, "xbee_device_manager"))
```

self._extended_address, which is set by the code:

```
    # Get the extended address of the device:
    self._extended_address = SettingsBase.get_setting(self, "extended_address")
```

This also means the XBeeBase class manages the settings named "xbee_device_manager" and "extended_address". Derived classes should attempt to create or manage these settings. When porting drivers to the new DIA, you are encouraged to use XBeeBase's copies of these variables and settings.

The XBeeBase class registers a callback named self.running_indication() which derived classes can overload as desired. The default base class routine looks like this:

```
def running_indication(self):
"""
Indicate that we have completed config and are running
"""
self._tracer.info("Configuration is Complete. Running indication.")
return
```

DIA config massa m300 serial

Configuring the Massa M300 ultrasonic sensor on an RS-485 serial adapter in DIA

TODO - this page needs to be cleaned up.

YML example

YML examples should be added, and the settings below better tagged as optional, required and any default information.

```
  - name: tank07
    driver: devices.vendors.massa.massa_m300:MassaM300
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:A2:00:40:4A:21:DB!"
        poll_rate_sec: 15
```

DocString from Python code

Below is the docstring copied from the driver file in ./src/devices/vendors/massa/massa_m300.py. It is here since many users are not comfortable trying to browse or read Python files.

```
"""\
Massa M-300 Driver Connected to XBee 485 Adapter Driver

Supports a Massa M-300 device connected to a XBee 485 driver.  The following
Massa M-300 devices are supported:

    M-300/95 (min 12 in. / max. 180 in.)
    M-300/150 (min 7 in. / max. 96 in.)
    M-300/210 (min 4 in. / max. 50 in.)

Wiring information:

    M-300 Brown --> XBee 485 Pin 1 (485 Port B {+})
    M-300 Green --> XBee 485 Pin 2 (485 Port A {-})
    M-300 Black --> XBee 485 Pin 5 (GND)
    M-300 Red   --> XBee 485 Pin 6 (+12 DC)
    M-300 White --> Not Connected

XBee RS-485 DIP Switch Configuration:
    Pins 2, 3, 4 = ON (RS-485 Mode)
    Pins 5, 6    = OFF (Disable Bias, Termination)

NOTE: this driver at present does not support any sleep modes.
"""
```

Device Cloud data streams

Format for uploading data

Attribute explanation

name (required)

The traditional DIA framework use a **{instance}.{channel}** format, so for example a gateway with 2 LTH sensors might upload samples named "indoor.temperature" and "outdoor.temperature". As a Data Stream, these will have a form such as "dia/channel/00000000-00000000-00409DFF-FF6A72F6/indoor/temperature".

value (required)

The value can be anything within quotes or as a valid XML text value. Device Cloud attempts to auto-detect the type, which can lead to problems when a string of digits is misinterpreted as a number.

type (optional)

To explicitly force a type, add the type attribute:

- bool (becomes Data Stream **Integer**)
- str (becomes Data Stream **String**, as UTF-8)
- int (becomes Data Stream **Integer**, big-endian 32-bit two's compliment)
- long (becomes Data Stream **Long**, big-endian 64-bit two's compliment)
- float (becomes Data Stream **Float**, big-endian 32-bit IEEE754 floating point)
- double (becomes Data Stream **Double**, big-endian 64-bit IEEE754 floating point)

unit (optional)

Attaches a unit of measure to the value sample.

timestamp (optional)

The timestamp should be of the simple, non-duration forms in ISO8601. The simplest is the "2011-10-03T00:05:20Z", where the trailing Z means UTC time. If the timestamp is omitted or of an invalid format, then the Device Cloud will use the server system time as of processing.

Compact format example

Compact formatting treats all data as attributes, making the XML size smaller. White-space is optional and ignored.

```
  <idigi_data compact="True">
    <sample name="home.M_AC_Power_2" value="0.2" unit="W" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_AC_Power_1" value="0.2" unit="W" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_Imported_1" value="0.04690" unit="KWh" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.offline" value="False" unit="" timestamp="2011-10-03
```

```
00:04:54Z"/>
    <sample name="home.M_Imported_2" value="0.01738" unit="KWh" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.M_AC_Voltage" value="125.2092" unit="VAC" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.M_AC_Current_1" value="0.06" unit="A" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_AC_Current_2" value="0.0" unit="A" timestamp="2011-10-
03T00:05:20Z"/>
  </idigi_data>
```

Full format example

Full formatting creates a larger XML size. White-space is optional and ignored.

```
  <idigi_data>
    <sample>
      <name>home.M_AC_Power_2</name>
      <value>0.2</value>
      <unit>W</unit>
      <timestamp>2011-10-02T23:58:01Z</timestamp>
    </sample>
    <sample>
      <name>home.M_AC_Power_1</name>
      <value>1.2</value>
      <unit>W</unit>
      <timestamp>2011-10-02T23:58:01Z</timestamp>
    </sample>
  </idigi_data>
```

Device Cloud easy demo

Purpose

This page is supposed to introduce you into a sample Device Cloud application. These are the special points that are demonstrated in this demo:

- The demos shows the advantages of Device Cloud and our Drop in Networking products:

```
  - Build local intelligence with the Python programming engine that works even
without Cellular Connectivity
  - Connect to the Devices easily via Device Cloud without knowing the IP address
```

- The local intelligence toggles power control adapter either based on sensor info or I/O toggle (Push button)
- Remote connection to the demo is done via a hosted website that includes the access via webservices (XML Messages)

```
  This website shows as well the Energy information provided by the smartplug
(load, work, Current, Voltage..)
```

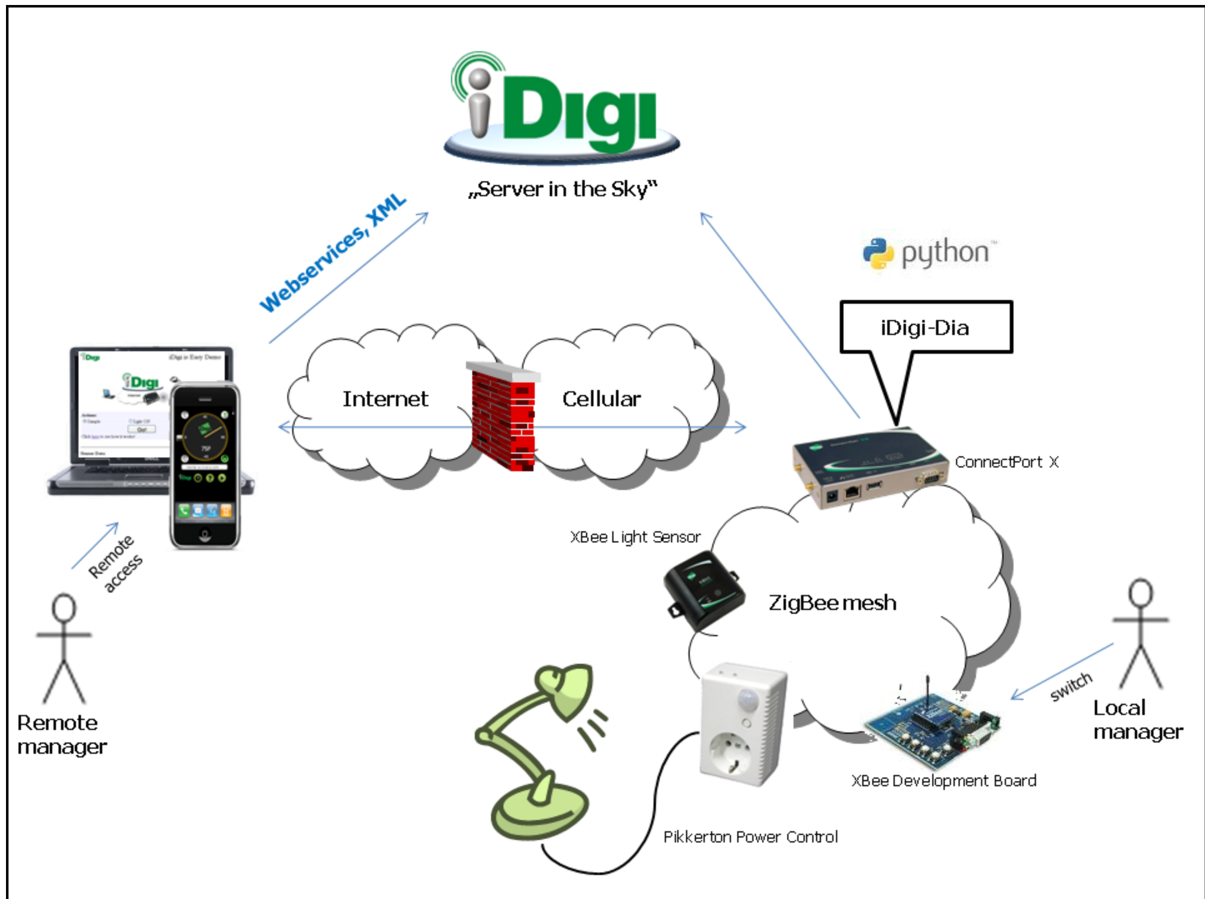
- It is also possible to connect to the device via a smartphone, showing this website.

Requirements

- ConnectPort X gateways (CPX) with cellular connectivity
- Power control device (here Third Party device: Pikkerton router)
- XBEE sensors
- XBee Development Board
- Sample power plugged device (e.g. lamp)

Introduction

The Demo configuration is shown as below:



Files

The ConnectPort X gateways (CPX) has to be equipped with these Python files:

- dia.zip (generated with make command)
- dia.py
- DigiXBeeDrivers.zip

- Python.zip
- zigbee.py

To generate the dia.zip shown above, you also need the complete DIA demo folder which can be downloaded in the last section of this page.

Getting started

1. first step...
2. second step...

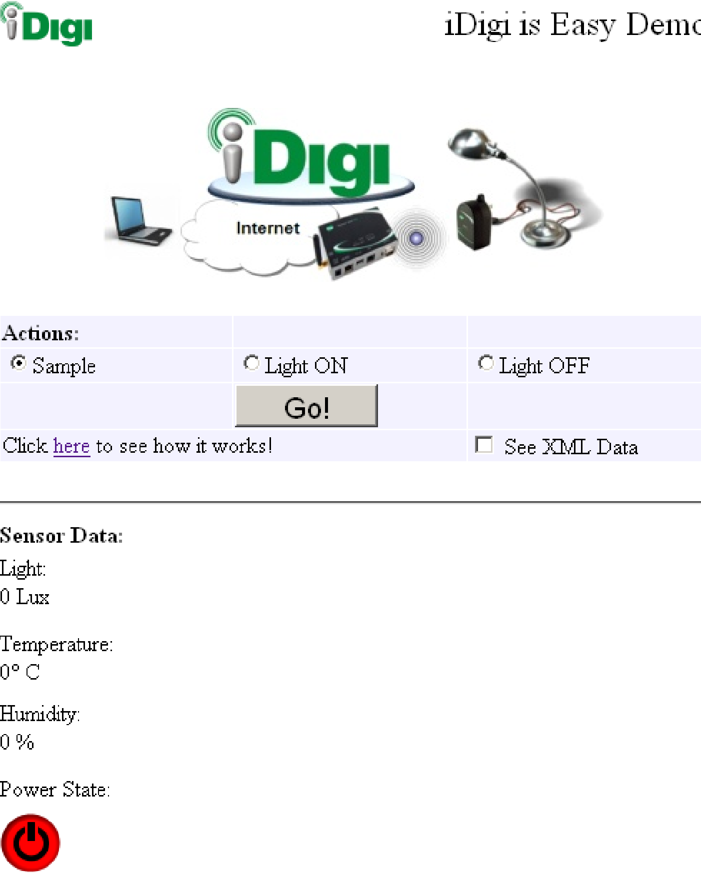
How it works / How it looks

1. Local intelligence
    - Switch 2 on the XBee Development Board turns on the power control adapter. => Light on.
    - Switch 3 turns it off
    - Switch 4 activates the "auto mode": If the /L/T/H Sensor Adapter measures a light value below a specific level, the power control is turned on, otherwise it is turned off.
    - LED 1 represents the state of the power control adapter.
    - LED 2 represents the state of the auto mode.

Please visit IDigi Easy Demo Details - Support for iDigi Easy Demo for details.

2. Remote Access
    - You can access to the power control information via the Device Cloud server and using any browser-enabled device.
    - A dedicated web site allows you to remotely change the state of the demo (switching on/off manually or (de-)activating the auto mode.

Here is a screenshot of the iDigi Easy Demo web interface:



Interesting code

We will give you a short introduction in the key source code, soon with more detail.

Beside the standard DIA library, you need three new pieces:

```
- device driver for the smartplug
- presentation that contains the local intelligence
- the .yml file, here is an example
```

Source

IDigiEnergyDemo_Extrafiles.zip

Device Cloud RPM demo

This topic is under construction

Purpose

This page is supposed to introduce you into a further sample Device Cloud application. These are the special points that are demonstrated in this demo:

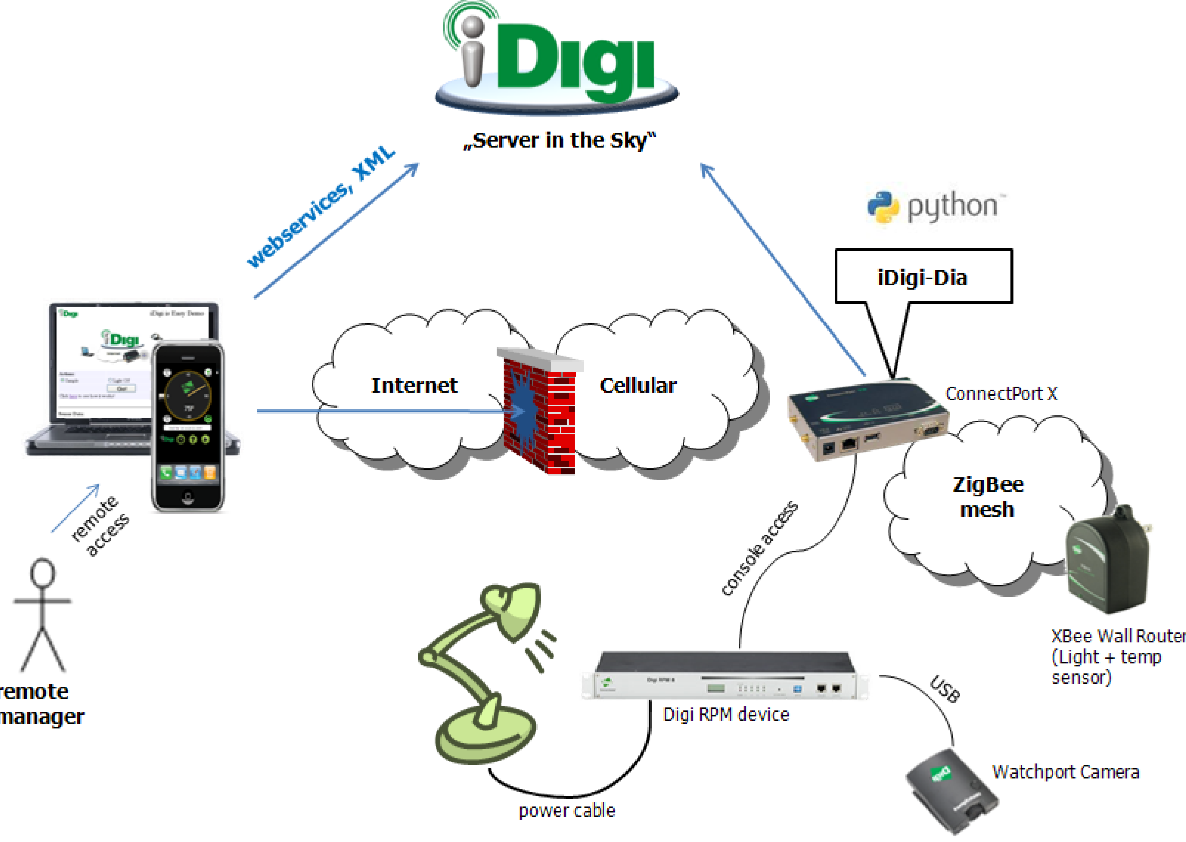
- The application is available via Device Cloud service although the ConnectPort X has a **private IP** in the cellular network.
- There is local intelligence that remotely toggles a Digi RPM device if some sensor values are dropping below a specific level.
- It is also possible to configure the power control via a website.

Requirements

- ConnectPort X gateways (CPX) with cellular connectivity
- Digi RPM device
- XBee Wall Router
- Module: camera
- Sample power plugged device (e.g. lamp)

Introduction

The Demo configuration is shown as below:



Files

The ConnectPort X gateways must be equipped with these Python files:

- dia.zip (generated with make command)
- dia.py
- DigiXBeeDrivers.zip
- Python.zip
- zigbee.py

To generate the dia.zip shown above, you also need the complete DIA demo folder which can be downloaded in the last section of this page.

Getting started

1. first step...

Please don't forget to make sure that the Serial port is free for use by Python. Otherwise you cannot open the COM1 in the Python code.

In the CLI the setting should be:

```
set term port=1 state=off
```

In the CPX website the Serial Config should show:

```
Port 1 None Custom 9600  8N1 or
Port 1 None Unassigned Custom 9600  8N1
```

Don't forget to reboot after changing the setting!

2. second step...

How it works / How it looks

1. Local intelligence

If the XBee Wall Router measures a light value below a specific level, the RPM power outlet is turned on, otherwise it is turned off.

Please refer to Remote Power Management Demo for details on how to configure an Digi RPM device via Python.

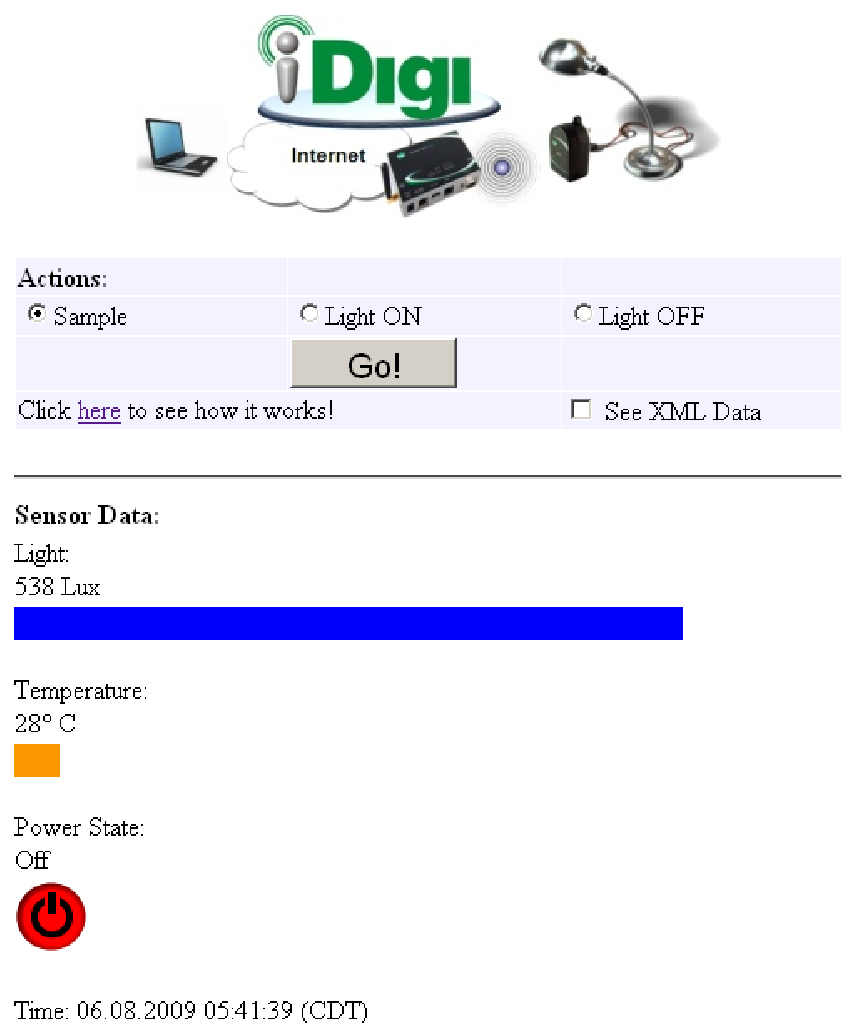
2. Remote Access

You can access to the power control information via the Device Cloud server and using any browser enabled device.

A dedicated web site allows you to remotely change the state of the demo (switching on/off manually or (de-)activating the auto mode.

Here is a screenshot of the iDigi Energy Demo web interface:

iDigi WEB Demo



Interesting code

We will give you a short introduction in the key source code, soon.

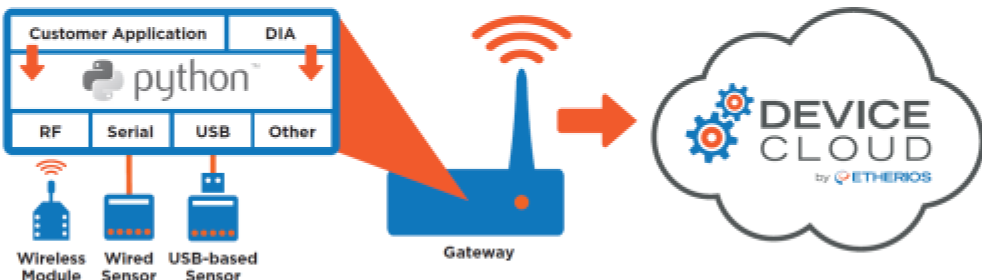
Source

There is no source, yet.

Device Cloud Wiki

Overview

DIA (Device Integration Application) is software that simplifies connecting devices (sensors, PLCs, etc.) to communication gateways. DIA includes a comprehensive library of plug-ins that work out-of-the-box with common device types and can also be extended to include new devices. DIA's unique architecture allows the user to add most of these new devices in under a day.



DIA is designed upon a tested architecture that provides the core functions of remote device data acquisition, control, and presentation between devices and information platforms. It collects data from any device that can communicate with a Digi gateway, and is supported over any gateway physical interface. Digi DIA presents this data to upstream applications in fully customizable formats, significantly reducing a customer's time-to-market.

Written in the Python programming language for use on Digi devices, DIA may also be executed on a PC for prototyping purposes when a suitable Python® interpreter is installed.

Drivers and presentations

See DIA Drivers and Presentations to download drivers and presentations for your own use and modification.

Education

DIA education materials are maintained at http://www.digi.com/products/cloud/digi-device-cloud. Create your own developer account and you'll have instant access to DIA training information as well as the ability to develop and test online with the Device Cloud. Look under "resources."

Reference information

- Python.org
- Digi Python programmers guide

Code examples

- Start with the great examples, included in DIA. Go to www.digi.com/products/cloud/digi-device-cloud and get started!

- Example XBee Serial Device
- Network-Time-Server-DIA-example

DigiMesh support in DIA

DigiMesh support added to DIA 2.0 (May 2012)

The next release of DIA will add expand XBee support to include DigiMesh for firmware 8062 or higher. Support is limited to DigiMesh XBee technologies which are supported by Digi gateways (see Existing DigiMesh gateways).

Supported DIA drivers

Support is limited to adapters released by Digi. see Existing DigiMesh gateways.

Instead of the traditional XBeeDeviceManager, you should add a DigiMeshDeviceManager:

```
driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
```

Supported DIA drivers

Support is limited to adapters released by Digi. see Existing DigiMesh adapters

- Both DigiMesh 900 Mhz and 2.4 Ghz can be seated in the XBIB adapter, which is supported by the xbee_xbib driver, such as:

```
driver: devices.xbee.xbee_devices.xbee_xbib:XBeeXBIB
```

- Both DigiMesh 900 Mhz and 2.4 Ghz have RS-232 and RS-485 adapters, which are supported by the xbee_serial driver and derivatives, such as:

```
driver: devices.xbee.xbee_devices.xbee_serial_
terminal:XBeeSerialTerminal
```

- Both DigiMesh 900 Mhz and 2.4 Ghz have digital adapters, which are supported by the xbee_dio driver, such as:

```
driver: devices.xbee.xbee_devices.xbee_dio:XBeeDIO
```

- Both DigiMesh 900 Mhz and 2.4 Ghz have analog adapters, which are supported by the xbee_aio driver, such as:

driver: devices.xbee.xbee_devices.xbee_aio:XBeeAIO

- Only DigiMesh 2.4 Ghz has a wall router (called a 'Range Extender'), which is supported by the xbee_xbr driver, such as:

```
driver: devices.xbee.xbee_devices.xbee_xbr:XBeeXBR
```

Unsupported DIA drivers

The following products are not supported by Digi.

- Smart Plug
- Light/Temperature (LT/LTH) Sensor
- WatchPort Sensor Adapters

Enable Modbus query of DIA devices

Overview

Modbus is a simple memory read/write protocol. It allows a remote SCADA or HMI host to read a block of words. What those words mean must be manually configured into the host software. The Modbus DIA server Module allows a Modbus master to read the data from a collection of XBee devices as if they were each a Modbus device. For example, twenty level sensors on twenty tanks would appear as 20 Modbus devices.

Mapping Modbus unit Id to DIA device

The Modbus server uses an ASCII text file in the FS/WEB/Python directory named mbus_map.txt, which you can edit manually as required. A reboot is required to activate any changes you made. Below is an example file. The CSV-like fields are:

- Modbus slave address / Unit Id
- DIA device name, which is used to locate the device for each Modbus poll
- The device type (name) returned by the device driver - the exact value is not important and is used primarily for debug trace information
- MAC address of the device (if applicable)

```
# DIA Modbus Server unit_id mapping as of 2010-02-08 15:55:41
1,'OUTDOOR','XBeeSensor_LT','[00:13:a2:00:40:4a:6e:6b]!'
2,'INDOOR','XBeeSensor_LT','[00:13:a2:00:40:4a:6a:1e]!'
```

Hard-coded mapping

The YML can be used to hard-code a DIA device to Modbus mapping. The YML example below assumes the devices named 'solar', 'outdoor', and 'indoor' exist. Polling any Unit Id but 1, 2 or 3 will result in a Modbus exception 0x0A.

```
- name: mbus_srv
  driver: presentations.modbus.mbdia_pres:MbDiaPresentation
  settings:
      mapping: "( (1,'solar'), (2,'outdoor'), (3,'indoor') )"
```

Auto-enumerated mapping

The YML below assumes the auto-enum driver named 'xbee_autoenum' exists. It allows automatic creation of Modbus slaves numbered 1 to 20. Note that the same mbus_map.txt file is created to make the auto-assignment consistent between gateway reboots. You can edit this file as desired. To remove a device previously mapped, delete the appropriate line of the file. New devices will be placed into the lowest Unit Id available, so if you delete Unit Id 3 when 10 are defined, the next device discovered will be placed as Unit Id 3.

```
- name: mbus_srv
  driver: presentations.modbus.mbdia_pres:MbDiaPresentation
  settings:
      mapping: "('auto', 1, 20)"
      auto_enum_name: xbee_autoenum
```

Forming Modbus responses

The Modbus server module requires the target Dia device driver to support the following methods:

get_mbus_device_type()

Returns a string name of the type. No particular meaning is assigned - it can be the class name, but does not need to be.

get_mbus_device_code()

Returns a numeric code, which is included as one of the registers in the register block response. No particular meaning is assigned - it can be the lower DD word or any other value.

export_device_id()

Returns a dictionary of values for use with the Modbus Read Device Identification command (not yet implemented in the Modbus code, but will be eventually). The dictionary keys are the numeric "Object Ids" defined by the standard, and the values are all ASCII strings. Here is an example:

```
    def export_device_id( self):
        """Return the Device Id response strings"""
        dct = { 0:'Digi', 1:'XS-B14-CB1RB', 2:'1.0', 3:'www.digi.com',
                4:'XBee Sensor', 5:'/L/T', 6:'Dia' }
        return dct
```

To summarize the Object Id items:

Object Id	Required	Description
0x00	Yes	Vendor Name
0x01	Yes	Product Code
0x02	Yes	Major/Minor Revision
0x03	Yes	Vendor URL
0x04	No	Product Name
0x05	No	Model Name
0x06	No	User Application Name

export_base_regs(req_dict)

Extract the Python dictionary formatted for the Modbus block. The input dictionary will include the actual Modbus request parsed, so you can tailor the output to fit the request. See the examples in src\devices\modbus. The caller expects you to return a list (an 'array') of 16-bit integers which custom fit the Modbus request exactly. If the Modbus request is coil/boolean based, then the return is a list of boolean True/False values.

A call to src.common.modbus.mbdia_block.mbblk_to_data() creates this array for the standard values of ['status', 'din', 'ain1', 'ain2', 'ain3', 'ain4', 'dot', 'aot1', 'aot2', 'aot3', 'aot4', 'volt', 'timestamp', 'dd', 'mac'], but nothing stops you from manually creating your own custom response without using mbblk_to_data(). See Modbus DIA block register map to understand the standard register mapping.

The req_dict will include at least keys:

- req_dict['protfnc'] = Modbus function code such as 3 or 6.
- req_dict['dst'] = Modbus Slave Address / Unit Id.
- req_dict['iofs'] = zero-based offset from the request.
- req_dict['icnt'] = object count from the request.

See the file src.common.modbus.mbus_pdu.py to review the other keys usable.

import_base_regs(req_dict)

Allows Modbus to write data - FUTURE FEATURE, SO NOT YET SUPPORTED

Error messages

Common error messages

Many error messages are misleading, as the messages are from secondary causes. Document such messages and the results here:

Bad local file header

```
#> py dia.py
Determining platform type...Digi Python environment found.
Traceback (most recent call last):
  File "<string>", line 161, in ?
  File "<string>", line 82, in main
zipimport.ZipImportError: bad local file header in WEB/Python/dia.zip
```

The most likely cause is that the zip file has already been opened, and you are trying to open a 'new version' a second time. You need to reboot after uploading a new copy of the ZIP file.

For example, if **dia.py is already running**, then you'll always get this error. Check the 'connections' or 'who' list. Perhaps you left it running before, or have it set to auto-run.

A secondary cause might be a bad ZIP image - HTTP/web upload is not totally reliable. At times large files timeout/abort during upload and you may end up with only 320K of a 350K file. So confirm the file size of the ZIP is as expected, and/or upload dia.zip again.

Error workaround

As a workaround for this error without needing to reboot, one can clear zipimport's cached file headers. As part of one's startup script (before zip files are loaded onto sys.path call the following):

```
def clear_zipimport_cache():
    """Clear out cached entries from _zip_directory_cache"""
    import sys, zipimport
    syspath_backup = list(sys.path)
    zipimport._zip_diretory_cache.clear()
    # load back items onto sys.path
    sys.path = syspath_backup
```

Exception while uploading

Literally, the message each time DIA tried to upload data was the following:

```
iDigi_DB(idigi_db1): exception while uploading: (-6, 'The name does not resolve
for the supplied parameters. Neither nodename  nor servname were supplied. At
least one of these must be supplied.')
```

The solution (or problem) was that the DNS IP addresses within the CPX4 were NOT set properly, thus the **remote management server name (sd1-na.idigi.com)** could not be resolved, thus Device Cloud was NOT connected. Other symptoms:

- Device Cloud (or connectware manager) listed the device as disconnected.
- The web UI connections page did not list the "connectware tcp" entry

Socket.error

Socket is already open

```
Traceback <most recent call last>:
File "<string>", line 23, in ?
File "<string>", line 1, in bind
socket.error: <22, 'invalid argument'>
```

Although this error could mean a badly formed IP or port value, it also can mean that some other Python script is running and holding that specific IP+port combination open. Navigate to your ConnectPort's web UI, click on Applications->Python->Auto-start Settings, uncheck whichever script may be set to autorun, click Apply and reboot your gateway.

Syntax error (in YML file)

A very common cause is the use of tab characters. YML requires spaces only, so either use a text editor which replaces tabs with spaces, or manually make sure only spaces are use.

ValueError: failed to parse request (in RCI call)

One cause for this is 'fancy quotes' - if you cut and paste the request from a PDF or other web document, at times the quotations used are the fancy 'angled' quotes. Thus the string <rci_request version="1.1"> is really seen as <rci_request version=ö1.1ö>, which causes the RCI call to fail.

List index out of range error

A node which was part of the "index" is no longer available

```
#> Python EmbeddedKitService.py
Starting up...
Ready for incoming requests!
Discovering nodes...
Exception in thread WPANSerialEndpoint:
Traceback <most recent call last>:
  File "WEB/Python/Python.zip/threading.py", line 442, in __bootstrap
  File "WEB/Python/EmbeddedKitManager.py", line 199, in run
    nodes = self.get_bindings_hash_list<>
  File "WEB/Python/EmbeddedKitManager.py", line 496, in get_bindings_hash_list
    hash_list = [self._bindings.bindings[k].to_hash<> \
  File "WEB/Python/EmbeddedKitManager.py", line 38, in to_hash
    return {
IndexError: list index out of range
-
```

This error typically means that a node which was once associated with the parent (CP-X?) where the Python script is running is no longer available. Common causes: node has been configured for sleep parameters and is now asleep, node is powered off, node is in an unknown state. Power off parent and all associated nodes to clear out routing and neighbor tables, then power them back on and re-try the script when the nodes are available.

Example Smart Plug

Below is an example configuration file that includes a XBee Smart Plug device. NOTE: The extended address field will have to be edited for customer use.

```
devices:
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager

  - name: smart_plug
    driver: devices.xbee.xbee_devices.xbee_rpm:XBeeRPM
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: 00:13:a2:00:40:0a:12:b9!
        sample_rate_ms: 1000
        default_state: Off

presentations:
  - name: console0
    driver: presentations.console.console:Console
    settings:
        type: tcp
        port: 4146
```

For more information regarding configuration and use of the Smart Plug with the DIA, look here: Device Cloud Wiki.

Example XBee Serial Device

Overview

This DIA example implements a device driver for a fictitious wind turbine controller. It is aimed at illustrating how to parse a stream of serial data transmitted wirelessly from an XBee RS-232 adapter, XBee RS-485 adapter, or an XBee module in a development board or custom embedded design.

The device presumes that a wind turbine controller is connected to an XBee module and streaming a serial data. The DIA driver's job is to receive and parse this stream of serial data into a set of channel properties. The wind turbine controller provides three data points of information that we wish to capture:

1. The number of revolutions per minute the turbine is turning
2. The number of kilowatts generated instantaneously by the turbine
3. The temperature in Celsius reported by a sensor at the top of the turbine

Each complete sentence of data is newline-delimited and each individual data point in a sentence is colon-delimited. All data points are represented as ASCII strings. An example of the data stream is the following:

```
72:8.3:15.00\n76:8.4:15.25\n:60:7.2:14.75\n
```

Pre-Requisites

You should register at https://www.digi.com/products/cloud/digi-device-cloud and download DIA. After that, read the Getting Started guide. You should have a basic working knowledge of the DIA (how to start and stop drivers and presentations) before proceeding.

Driver Design

The basic design methodology of a driver is as follows:

1. Decide what settings and channel preperties are needed, add them to the driver initialization section.
2. Review which XBee configuration blocks are being set by this driver, adjust them as needed.
3. Add custom DD values to `src/devices/xbee/common/prodid.py`

   Note Program remote device XBee DD values to match in order to allow your device to be automaticalyl enumerated by xbee_autoenum.py. The most-significant 16 bits should match your module type (0x0003 is ZigBee) while the least-nignificant 16-bits may be set to the range 0xff00-0xffff and are reserved for private use by customers.

4. Accumulate serial data into a receive parse buffer, parse sentences from this buffer.
5. When complete sentences are found, update the driver's channel properties.

Usage

Download the below Python source file. This file is intended solely for reference sake. Read over the file including all comments and Python doc strings.

Downloads

Example_serial_device.zip

GE Ventostat CO2 ZigBee monitor

Ventostat® Wall Mount CO2, Humidity and Temperature Transmitters

- Vendor Product Page: https://www.instrumart.com/products/18180/telaire-ventostat-series-co2-monitor
- Product Brochure: http://www.instrumart.com/assets/Ventostat-B_datasheet.pdf
- Model Number: T8100-HD-ZB (you won't find on web site)

GE Sensing Telaire Products sells a Ventostat variant with Digi ZigBee - with the S2C SMT Em357 module. You will need to talk to GE sales to obtain this because they have a variety of wireless variants from many sources, yet do not include on their web site to reduce customer-confusion.

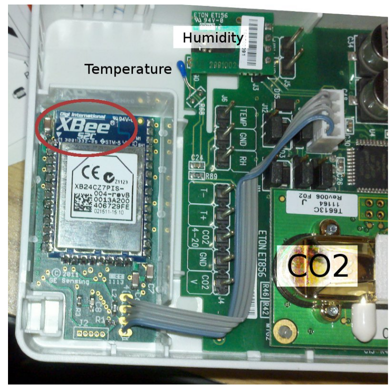
In a nutshell, it is like a thermostat which also measures the carbon-dioxide in the room, which allows the ventilation system to increase or decrease energy use based on room occupancy. Why is that important? Easy - consider my office cubical, which is probably much like your own working space. On weekends the CO2 level is about 400PPM, which is nearly outdoor quality. Great ... but in summer, that means my employer is paying too much to suck in fresh (hot) air which it must cooled down - just so that the people who ARE NOT AT WORK can feel better. In contrast, during the week the CO2 level climbs to 800-1000PPM, which perhaps means they do not intake enough fresh air.

The 'VentoStat' allows the ventilation system to increase fresh air intact when more people are 'breathing' inside, and reduce fresh air intact when less (or no) people are 'breathing' inside. Tests by the US DOT have shown this can cut an easy 17% off the energy costs related to ventilation, giving payback in months. Is CO2 dangerous? Not in levels under several tens of thousands of PPM. Some people might complain about 'stuffiness' when it climbs over 1500 or 2000 PPM, but in truth CO2 is more a 'canary gas' used as an indirect measure, in that as CO2 climbs, so does body-odor, humidity, perfume and food smells - all things which make employees feel the room is stuffy and unhealthy.

The GE VentoStat isn't cheap - expect it to be in the $350 range for low volume. And while you can buy 'air quality' sensors for as little as $50, I have been told there are some major difference between the GE Ventostat and such sensors:

- The $50 sensors tend to measure hydro-carbons, meaning vehicle fumes, new-carpet smells (etc) which is not directly related to human occupancy like CO2 level is.
- The $50 sensors tend to be catalyst-film based, so wear out or need 'refresh' in a few years, whereas the infrared method used by GE claims a minimum of 10 years maintenance-free use.

Ventostat Python code



Example usage

But enough of the sales fluff. Even in a home, one can make use of the Ventostat to manage the low-power fan-only mode in modern furnaces. For example, I have a 3-zone HVAC system at home with a 96% AFUE furnace. The fan-only mode is published as 75-watts - and since I have power monitoring in my home circuit panel, I can confirm that it really is about 75-watts. However, since my home has a fixed-open fresh-air intake, running the fan when not required can make rooms unnecessarily warm or cold in spring or fall. It also is a waste of money to run the fan-only mode when the windows are open.

So one can use a GE ventostat (or similar air-quality product) to feed a CO2 level into the Device/DIA, then use an alarm function to enable/disable the HVAC fan-only mode. The details of this are unfortunately outside the scope of DIA since you also need a DIA-driver for your HVAC system. My own furnace uses RCS TR16 the RCS TR16 thermostats based on RS-485, which I talk wirelessly to via Robust DataComm Xbee/485 adapters which support the 16vac supplied by my furnace to the thermostats.

Theory

The Ventostat uses a simple ASCII protocol. For example, to query all 3 readings at once, you send the string '$33$88\r\n' and you will receive a response such as '719, 35.3, 23.3\r\n', which is CO2, humidity and temperature respectively. Therefore the Digi XBee is in 'AT Transparent Router' mode, and the core DIA driver is based upon a common XBee serial driver which handles the details for sending and receiving strings.

Note: be careful when reflashing the XBee, for the Ventostat's serial port must be (19200,8,N,1), which is not the Xbee default. GE sells a special USB-serial cable which allows you to use XCTU directly with the internal S2C XBee module.

Actual code ZIP

This file contains the files which must be merged with the DIA 2.x.x code.

GE_Files_Dia_2_1.zip

File list

```
vento_readme.txt
REM These YML samples are the same, just two names
cfg\ventostat_ge.yml
cfg\dia.yml
```

```
REM The GE core files including the support for the Min/Max/Average
src\devices\vendors\GE_Energy\__init__.py
src\devices\vendors\GE_Energy\ut_minmaxavg.py
src\devices\vendors\GE_Energy\ventostat.py

REM My own 'DeviceBase' since I find the Digi DIA one to unhelpful & not very
object oriented
src\samples\annotated_sample.py
src\devices\vendors\robust\__init__.py
src\devices\vendors\robust\robust_base.py
src\devices\vendors\robust\robust_xbee.py
src\devices\vendors\robust\robust_xserial.py
src\devices\vendors\robust\prodid.py
src\devices\vendors\robust\xserial_util.py
src\devices\vendors\robust\avail_base.py
src\devices\vendors\robust\parse_duration.py
src\devices\vendors\robust\sleep_aids.py diff\src\devices\vendors\robust
```

Example YML fragment

This shows some of the power of my own device base. It creates a 'description' channel, plus as configured below I get 1 reading every 5 minutes synchronized to the clock, so on the hour, then at 5, 10, 15 (etc) minutes after the hour. Does that clean & fancy timing matter? Does seeing the value in DegF matter? Not in the cosmic sense, but it's what I want!

These settings are better described in the vento_readme.txt file.

```
  - name: Z1CO2
    driver: devices.vendors.GE_Energy.ventostat:Ventostat
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:52:94:b2!"
        dev_poll_rate_sec: '5 min'
        dev_poll_cleanly_min: True
        degf: True
        add_statistics: True
        dev_desc: "Living Room Zone CO2 sensor"
```

Example Channel_Dump

- The channels basic to the Ventostat class: 'co2', 'humidity', 'temperature', 'error', 'version'
- The optional Ventostat statistic channels enabled: 'co2_stats', 'hum_stats', 'tmp_stats' (the format is like "CO2, min=917, avg=926, max=1026")
- The channels from my own device & Xbee Base: 'availability', 'online', 'description' ('availability' and 'online' relate to XBee health and the percentage of lost heartbeat messages - in my Xbee base, all routerss send IO data by default every 60 seconds as a heartbeat of mesh health.)

These channels are better described in the vento_readme.txt file.

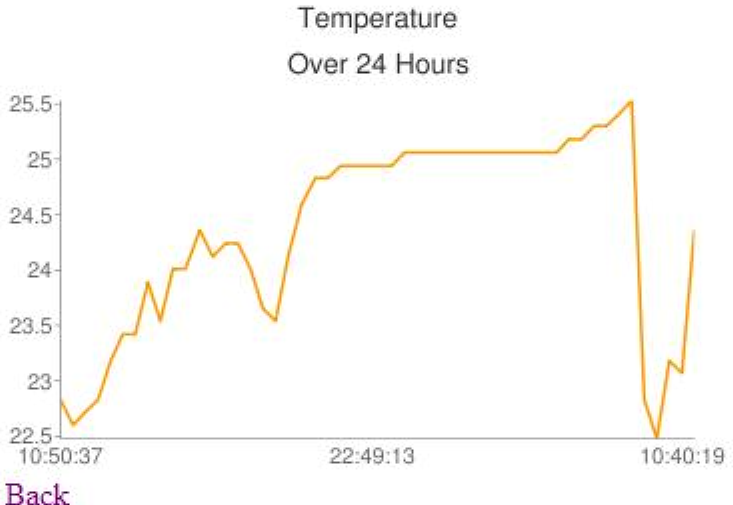
```
Channel                  Value              Unit     Timestamp
------------------------ ------------------ -------- -------------------
availability             99.8               %        2012-07-19 19:07:24
co2                      947                PPM      2012-07-19 19:11:00
co2_stats                CO2, min=917, avg=          2012-07-19 19:11:00
description              Co2 sensor by Desk          1970-01-01 00:00:00
error                    False                       2012-07-19 19:02:25
hum_stats                HUM, min=53.4, avg          2012-07-19 19:11:00
humidity                 53.4               RH       2012-07-19 19:11:00
```

online	True		2012-07-19 19:02:44
temperature	74.3	F	2012-07-19 19:11:00
tmp_stats	TMP, min=73.9, avg		2012-07-19 19:11:00
version	104, 2011/08/12 11		2012-07-19 19:02:25

Google App Engine Device Cloud Client

Purpose

The Google App Engine Device Cloud Client is one example of a client application that uses web service (XML) communication to a Digi gateway using Device Cloud. It is deployed to a Google Appspot account and runs on the servers provided by Google. The advantage of this hosted service is that there is no server infrastructure needed at the customer and applications can run 24/7 on the Google servers, making it an ideal solution if continuous sensor readings are required, e.g. for daily/weekly statistics of sensor data, displayed to the users in form of graphs accessible over the web:



Requirements

Hardware

Digi ConnectPortX ZB Gateway, Digi ZB Wall Router and one embedded XBee ZB module on development board. Recommended kits that include these pieces are

- Device Cloud Professional Development Kit ZB
- Device Cloud X4 Starter Kit ZB

Software

- Digi Python Development Environment (Digi ESP to be able to create and build DIA projects and execute them on the gateway
- Python 2.5
- Google App Engine SDK for Python (requires a Google Appspot account)

Usage

Installation

1. Create DIA project that includes drivers at least for the Digi Wall Router and the XBee ZB module on development board. Device names should be *xbr0* for the wall router and xbib0 for the development board. In addition to the drivers it is mandatory that the DIA project includes the RCI Handler presentation to enable the gateway to talk to the Device Cloud. Optional the *Embedded web* presentation can be included for local tests.

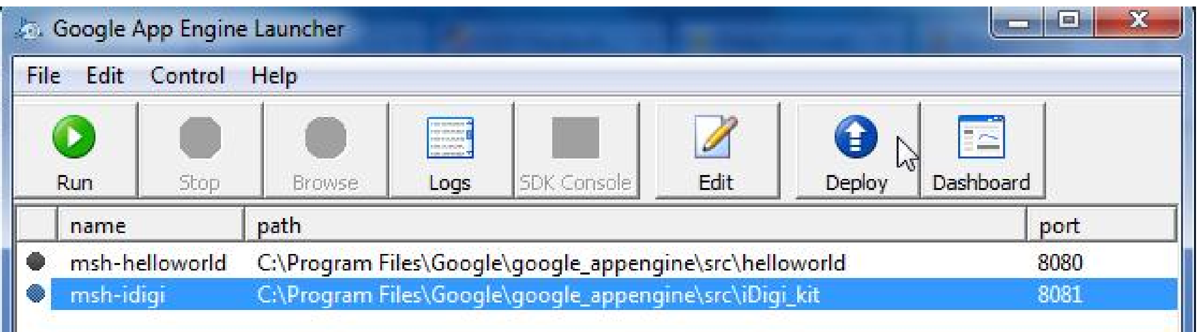
   Recommended is to start with the **iDigi Professional Development Kit** sample that can be found in the ESP by choosing File | New | iDigi DIA sample project. This sample comprises the drivers with correct device names as well as the necessary presentations mentioned above. For more information, consult the Device Cloud Professional Development Kit ZB Getting Started Guide (see steps 6 and 7)

- Run DIA project on the Gateway as explained in the Getting Started Guide.
- If available, test with embedded web page if wall router and development board are working as expected.
- Add Digi gateway to your developer.idigi.com account as explained in the Getting Started Guide (steps 1 and 2) and make sure it shows up as Connected.
- Install Python 2.5 and the Google App Engine SDK for Python.
- Extract the necessary source files to a subfolder of your Google App Engine SDK installation, by default \Program Files\Google\google_appengine.
- Use a text editor to open Config.py in the iDigi_kit folder. Change the username and password fields with the data of your Device Cloud developer account and modify the Device ID field with the Device ID of your gateway. Change the other fields if necessary and if you know what you are doing.

- Create a new application in your Google Appspot account, e.g. "<your name>-idigi" and choose any title for the application, e.g. "iDigiGoogleApp"

Deployment

1. Open the Google App Engine Launcher and
2. Add the Device Cloud client with **File | Add Existing Application** and choosing the iDigi_kit subfolder extracted previously, change the name of the application to match the application that you created on your Google Appspot account by selecting the application in the list, clicking Edit and modifying the application: entry in the file.
3. Deploy the application by selecting it in the list and clicking **Deploy**.



4. close the deployment window when instructed to do so.

These steps can be repeated as often as you change the application source code.
To check the status and output of the application.

1. Use the Current Version link in your Google Appspot application list to see the output of the application, click on the gauges to see the corresponding graphs.

2. After a few moments of operation, check the **Logs, Cron Jobs** and **Datastore Viewer** views in your application Dashboard view (accessible by clicking on the application name in the Google Appspot application list) to ensure the application operates as expected. **Cron Jobs** should list a job */update (sensor data update) running every 2 minutes (UTC)*, the **Datastore Viewer** should list sensor values collected from the wall router and development boards.

Current limitations

- Only Celsius temperature values are supported
- Device names need to be xbr0 and xbib0

Sources

Feel free to edit sources to your needs:

Google_appengine_iDigi_kit.zip

Google Gadget LTH Sensor Example

Overview

This simple web services example uses the Digi L/T/H (Light/Temperature/Humidity) sensor and displays the result of the most recently uploaded samples via a Google Gadget. The DIA configuration file samples the L/T/H sensor and periodically uploads the results to the Device Cloud cached storage facility. The Google Gadget, when requested (via the Fetch button) retrieves the sample collections (files) from the Device Cloud and finds the most recent (by timestamp). It then retrieves the contents of that most recent file and assigns variables to the contents (samples) of that file. The Google Gadget logic (written in Javascript, the language of Google Gadgets) then uses a combination of graphics and sensor samples to display the current settings.

Here's an example DIA configuration file (yml) that will work (included in the zip file):

Download the zip file Digi_LTH_Gadget1.zip, which includes the Google Gadget and associated image files, xml, etc. Google Gadgets are basically "zip" files which have been renamed "gg." All the JavaScript source and graphics are included. The yml file included below is a sample which you will need to modify with your "extended_address:" to match that of your own L/T/H sensor. Load the modified configuration file onto your Digi Gateway using the "Python" administrative menu. Unzip the Google Gadget file and then run the "Digi_LTH_Gadget.gg" file. Download Google Desktop first if you have not already done so.

```
# This configuration file is used to specify which components will be
# specified for use with the DIA Sensor Demo.
# Essentially, we will be selecting the sources of information, and how
# that information is presented.  This demo is intended to be used with
# the number of devices configured in this file.  If you wish to use a
# larger number of devices, you will need to change the configuration
```

```
# of the idigi_db presentation near the bottom of this file.

# The devices section specifies settings for devices devices:
# First we specify an XBee Device Manager.  Since the XBee is considered,
# a shared resource on our system, we need a device which can manage
# requests from other devices relating to the XBee.
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager

# Here we are including a Digi LTH (Light, Temperature, Humidity)
# XBee Sensor in our system.  Configuration is similar to the Smart Plug.
# The main addition is that since the LTH Sensor will likely be battery
# powered, we have set the XBee to sleep between samples.
  - name: lth_sensor_1
    driver: devices.xbee.xbee_devices.xbee_sensor:XBeeSensor
        settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:4a:ba:7f!"
        sleep: True
        sample_rate_ms: 1000
        awake_time_ms: 320

# The presentation section allows us to specify the ways in which
# information in our system is available.
presentations:

# Here we define a standard console method.  Using this,
# someone can connect (with telnet in Windows for example),
# to get a quick and simple look at the current state of the
# system.  They can view the status of all channels, and set
# any channels that support being configured.
  - name: console
    driver: presentations.console.console:Console
    settings:
        type: tcp
        port: 4146

# The idigi_db presentation layer is what allows us to
# interact with the Device Cloud Cache (temp) Storage Service.
# This presentation layer tracks when updates are made to any
# of the channels.  After a certain interval of time (defined by
# the interval setting), or after the number of new samples has
# exceeded our sample_threshold, we combine all of the sample
# information in an XML file, and then push it up to Device Cloud
# through an available network connection.
#
# Each time we push up to Device Cloud, we are essentially creating
# a file to store our information.  The file_count lets us choose the
# maximum number of files we will have at one time as a result of this
# presentation layer.  It will cycle through and overwrite old data.
#
# If you wish to increase the number of devices in this demo, remember
# that each device can potentially have multiple channels.  You will want
# to ensure that sample_threshold parameter is greater than the total
# number of channels that are actively configured or the idigi_db
# presentation will constantly attempt to combine all sample information
# and send to Device Cloud.
  - name: idigi_db
```

```
    driver: presentations.idigi_db.idigi_db:iDigi_DB
    settings:
        interval: 30
        sample_threshold: 100
        collection: sensor_demo
        file_count: 20
        filename: sensor_reading

# Create a web presentation instance.  When running on a PC this will
# start a new web-server on a default port
# When running on a Digi device this presentation will "extend" the
# web-server built in to the Digi device with a new page named
# "idigi_dia.html".  See the file src/presentations/web/web.py form more
# information.
  - name: web0
    driver: presentations.web.web:Web
    settings:
        page: idigi_dia.html

# The RCIHandler allows for channel dumps, querying a channel, and setting
# a channel.  The RCIHandler parses an incoming XML formatted message
# to determine what you wish to do.  The messages are shown below. If
# an error is encountered, the request you sent will have a child error
# element specifying the error information.
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
        target_name: idigi_dia
```

Ensure that you have an Device Cloud account and that your ConnectPort-X gateway is actively connected. For additional information, visit www.etherios.com/devicecloud and visit the "resources" page.

Restart your gateway and ensure that DIA is running. You can automate DIA startup via the "Pythons->Autostart" screen in the Digi ConnectPort X webUI administration utility.

Test your DIA configuration locally by pointing your laptop to the ConnectPort-X's local IP address. This means that you need to be locally attached for this particular test. type "IP_Address_of_ConnectPort-X/idigi_dia.html" into your web browser. You should receive a web-UI that will display the three samples available to the virtual connector/driver: prefix_string, suffix_string, xtended_string (a concatenation of the first two samples).

If you haven't already done so, install Google Gadget Desktop (visit Google for details), start, and then double-click on the "gg" gadget file to load onto your Google Desktop. Click the "Options" button and populate the default options with your own:

- Your Digi ConnectPort-X gateway address (just the last two octets - e.g. 00409DFF-FF382CF3)
- URL of the Device Cloud Connectivity Server (the URL where your Digi ConnectPort-X product is connected - found from the Device Cloud when you are logged in)

- Collection: sensor_demo maps to the yml file configuration. Search for it and you'll see how it is used to group the filenames which encompass the various samples within a sample period.
- Sensor: This maps to the XBee device name within the yml file configuration. This label/tag is how samples for this particular sensor are named.
- Username: Your Device Cloud username
- Password: Your Device Cloud password

That's it! Click the **Fetch** button to retrieve your sample collections. We recommend that you download the Google Gadget Design tool, which provides an IDE exclusively for Google Gadget creation.

Importing Modbus data from IO device

Many people use small Modbus I/O device to manage field sensors. There are many sources, including Acromag, Advantech, DataQ, Datanab, DGH, ICP DAS, Opto22, Phoenix Contact, and Wago to name just a few. The I/O device might have for example 4 analog inputs, 8 digital inputs and so on.

Generally, one can read a dozen Modbus registers and see all the input status in one command. This Wiki page covers how to take this raw Modbus data and upload it to Device Cloud as 'DIA Data' for access by web services.

Supported Products

The code explained on this page only works on Digi Connect devices, such as:

- Connect SP (single-port Ethernet to 232/485)
- Connect WAN Family (cellular to Ethernet or Serial)
- ConnectPort X2D (Industrial, metal-enclosure X2 Xbee mesh to Ethernet)
- ConnectPort X4, X4 IA, X4 H (Xbee mesh to Ethernet and cellular)

Specifically, it does not work on the following devices

- ConnectPort X2 A, B or C - the low-cast commercial, plastic-enclosure Xbee mesh to Ethernet
- Any ConnectPort X2 with only 8MB of RAM memory
- ConnectPort X2e Xbee mesh to Ethernet
- Any Transport model
- Older non-Python products

Install the Digi ESP for Python

You will need to configure and upload a Python application. This is not as difficult as it sounds, although the jargon and steps can be new and involves some learning curve. Fortunately, once you have a functioning set of file you can replicate the solution, bypassing many of the steps.

You start by downloading the ESP installer for Windows or Mac OS. It is a large file since it includes Python and lots of documentation. You should download the newest one (2.1.0 as of Sep-2012).

Explaining how to use is beyond the scope of this page, but in summary:

1. Run the ESP.
2. Close the **Package manager** that offers to check for updates
3. Run the **Device Manager** under Device Options. Select your gateway.

   Note The gateway must be online and accessible to create, build, and upload a configuration.

4. Create a new **iDigi Dia Project**.
5. ESP won't support smart\graphical creation of a configuration, but change to the source view allows you to cut and paste the text in (explained below).

Install the Modbus client add-on

The stock Digi ESP for Python does NOT include the Modbus Client code which will import the Modbus data. We need special Modbus code which can generate a Modbus poll (such as read 10 holding registers), parse the response and create the desired DIA channels for upload to Device Cloud.

Download the Modbus Dia Code Add-On, unzip it and manually copy into the DIA copy in the ESP Program Files directory. This adds both the [[Modbus_Dia_Client | Modbus Client] (import Modbus from Modbus devices) and the Modbus DIA server.

Configure and test the general Modbus IA engine

To maximize flexibility, the Python code assumes a Modbus bridge runs on localhost (127.0.0.1) UDP port 502. Therefore you must create a Modbus bridge table which defines a list of Modbus Unit Id mapped to a remote IP, XBee MAC address, or serial port. This use of the external bridge supports Modbus/TCP form in TCP or UDP, plus Modbus/RTU or Modbus/ASCII on serial, Xbee wireless, or encapsulated in TCP/IP or UDP/IP.

In this example, assume we have a serial Modbus/RTU I/O device, so when you create the **Message Destination**, select the serial port, not an XBee Device.

Assuming you activated the Modbus/TCP master as a **Message Source**, then the Digi gateway will respond to any remote Modbus/TCP master queries. Use an OPC server or a Modbus Master tools such as ModScan to confirm that you can read the data from your I/O device.

This test is critical - if you cannot read the I/O device by indirect Modbus/TCP through the Digi gateway, then the DIA Modbus client will not be able to read it either.

Do not go past this step until you can see your Modbus I/O device data via the Digi gateway.

Create the DIA configuration

Now things will start to get more interesting.

Example generic DIA Modbus import uploading raw Modbus data from an Acromag 951EN

Paste the text below into the 'source' view of the ESP dia.yml, overwriting all other text. As shown below, this create create DIA channels named (for example) as mbus.a1_r30001, which will hold the unsigned integer value 0-65535 which was read from the I/O device's input register 3x00001. As written, the Modbus data is pulled every 60 seconds, and uploaded to Device Cloud every 60 seconds. The Modbus client creates other channels - such as mbus_a1_error, which is True if the polls are failing. Read the [[Modbus_Dia_Client | Modbus Client Wiki Page] to learn more about channels created, as well as parse options which can scale or handle floating point data.

Of course you will need to edit the 'pollinfo' to match you I/O device. The Acromag 951EN uses Modbus function #4 to read the read-only input registers. Many other I/O devices put the same information in the holding-register memory accessed by Modbus function #3.

```
devices:
  # read inpit data from Acromag 951EN
  - name: mbus
    driver: devices.modbus.mbus_udp_device:MBusUDPDevice
    settings:
        poll_rate_sec: 60
        trace: 'debug'
        round: 3
        poll_list:
            - poll: a1
              pollinfo: { 'uid':1, 'fnc':4, 'ofs':0, 'cnt':10 }
              channels:
```

```
                - parse: { 'nam':'r30001', 'ofs':0, 'unt':'word' }
                - parse: { 'nam':'r30002', 'ofs':1, 'unt':'word' }
                - parse: { 'nam':'r30003', 'ofs':2, 'unt':'word' }
                - parse: { 'nam':'r30004', 'ofs':3, 'unt':'word' }
                - parse: { 'nam':'r30005', 'ofs':4, 'unt':'word' }
                - parse: { 'nam':'r30006', 'ofs':5, 'unt':'word' }
                - parse: { 'nam':'r30007', 'ofs':6, 'unt':'word' }
                - parse: { 'nam':'r30008', 'ofs':7, 'unt':'word' }
                - parse: { 'nam':'r30009', 'ofs':8, 'unt':'word' }
                - parse: { 'nam':'r30010', 'ofs':9, 'unt':'word' }
            - poll: d1
              pollinfo: { 'uid':1, 'fnc':2, 'ofs':0, 'cnt':6 }
              channels:
                - parse: { 'nam':'i10001', 'ofs':0, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10002', 'ofs':1, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10003', 'ofs':2, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10004', 'ofs':3, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10005', 'ofs':4, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10006', 'ofs':5, 'frm':'?', 'unt':'bit' }

  - name: edp_upload0
    driver: devices.edp_upload:EDPUpload
    settings:
        interval: 60
        filename: "acro"
        sample_threshold: 9999

presentations:

  # Create a new console instance on TCP port 4146.  It can be connected
  # two by using any telnet client.
  - name: console0
    driver: presentations.console.console:Console
    settings:
        type: tcp
        port: 4146
  # Enable a web interface available on http://<ip of ConnectPort>/idigi_dia
  - name: web0
    driver: presentations.embedded_web.web:Web
    settings:
        page: idigi_dia


tracing:
    default_level: "info"
    default_handler:
    - stderr
```

Getting creative

Looking into the Acromag 951EN manual shows that the DIA channel mbus.a1_r30007 is actually the CH0 analog input and will be 0-20000 representing 0ma to 20mA. We can edit the parse line to send up data samples up as floating tagged as mA. So for example, if the signal is now 12.0mA, the original YML would upload a sample "**mbus.a1_r30007 = 12000, units = word**". The modified YML would upload a sample "**mbus.a1_CHO = 12.000, units = mA**".

```
                - parse: { 'nam':'r30006', 'ofs':5, 'unt':'word' }
                - parse: { 'nam':'CH0', 'ofs':6, 'unt':'mA', 'typ':'float',
```

```
'expr':'%d/1000.0' }
                - parse: { 'nam':'r30008', 'ofs':7, 'unt':'word' }
```

Microsoft PowerShell with web services

What is PowerShell?

Powershell is a relatively new command line environment in Windows. It is intended to replace the common DOS command shell. It is a much more powerful programming environment than the old command shell, and rivals Linux command shells such as BASH in many ways. It has some very slick methods for dealing with XML, so makes it very useful for use with web Services.

On Windows 7, go to **programs/accessories/Windows Powershell/** and click **Windows PowerShell** to open a PowerShell shell window.

For an introduction to PowerShell, search with your favorite search engine for *Powershell tutorials*.

Overview

I wrote a few scripts that query Device Cloud using RESTful web services. They emulate the functionality of a PowerShell cmdlet, outputting the results of the query in a structured format. This allows you to pipe them to an output presentation such as export-csv, convertto-html, and my favorite, out-gridview. You can also filter and sort them also using one of many PowerShell's object manipulation functions.

I wrote example scripts to pull down data using the Device Core, DiaChannelDatFull, DiaChannelDatHistoryFull and XbeeAttributeDataCore APIs. All of them are near identical, with a few tweeks to the code to deal with slight differences in format. It would be trivial to modify them to parse data from any of the iDig web services calls.

Note that all of these calls are HTTP GETs. Doing POST, PUT and DELETE functions is more complicated, you will need to use some dotNET functions or execute an external program to do these.

Requirements

PowerShell 2.0

These scripts require PowerShell V2.0 or later. PS 2.0 is native on Windows 7 and Windows Server 2008 R2, but available for older Windows versions by googling 'windows powershell 2.0 download'. It is a very easy install, and completely free.

PowerShell Community Extensions 2.0

You also must install the PowerShell Community Extensions 2.0 or later. Download the PowerShell Community Extensionsd 2.0 here. Follow the simple instructions at that link to install these. The community extensions provide the get-httpresource cmdlet which makes this all super easy to do (at least for GETs).

.Net Infrastructure 3.5 Update 1

out-GridView (optional, but recommended) requires .Net Infrastructure 3.5 Update 1 or later. You probably already have this, and will find out very quickly if you do not. Try "dir | out-gridview" as a simple test. This is also easy to install, and out-gridview is well worth the effort, more information on out-gridview is in the applications section below.

Applications

So, you have the script running, and it spews out your data to the console. What good is that? PowerShell is all about "piping" the data to the next piece of code to either massage it, act on it, and/or present it.

out-gridview and other "Presentations"

out-gridview is the presentation I use the most. It is a simple GUI applet that displays grid data. It is like a slimmed down spreadsheet app. You can search, sort and filter data with ease. It is much simpler and easier than using a full spreadsheet program, It is great when I am tinkering with my DIA build, troubleshooting web apps, etc.

```
.\get-idigiDiaSnapshot.ps1 developer idigiUserName | out-gridview
```

If this is your primary application, you might even want to add " | out-gridview" to the end of the last line of any of the scripts below and you will not need to pipe it on the command line. Click here for a good help page on out-gridview.

There are quite a few other cmdlets built-in to PowerShell that might be useful, such as exportTo-html, export-csv, send-MailMessage, etc.

Sorting and Filtering

PowerShell has a lot of tools to sort, filter and manipulate data. The two I use the most often are:

Select-Object, which is used to choose what properties (columns) are output. For example, perhaps all you care about are a few attributes, try this:

```
.\get-idigiDiaSnapshot.ps1 developer idigiusername | select-object -property
ddInstanceName, dcChannelName, dcdUpdateTime, dcdStringValue
```

The above will output just those four columns to the console. Tip: if the output is four or less properties (columns), it will automatically show it in table (columnar) format. Five or more and it will show it in list format. You can override that behavior by piping the output to format-table, format-list, or out-gridview.

Sort-Object will sort the output. The following line will sort by update time, then instance name, then channel and then sends it to the grid view.

```
.\get-idigiDiaSnapshot.ps1 developer idigiUserName | sort-object -property
dcdUpdateTime, ddInstanceName, dcChannelName | out-gridview
```

If you look at the scripts below you will examples of both of these in action. There are quite a few other functions available which can be used to filter and manipulate your output.

The Scripts

All of these scripts have a common usage pattern:

```
scriptname <cloud> <userid>
```

For example:

```
get-idigiDiaSnapshot.ps1 developer jsmith
```

Both parameters are required. For security reasons, you will be prompted for the password after you run the script. If you want to run it in a non-interactive manner look into the "-credential" options in PowerShell.

Also note that several of the lines are extremely long, as is typical with PowerShell. I used a method to split the long lines of putting a space and backtick (" `") at the end of a line that is continued. PowerShell supports this method, so these scripts can be copy/pasted directly into your text editor.

get-idigiDiaSnapshot (DiaChannelDataFull)

This script will query /ws/DiaChannelDataFull. This provides the last sample for each device.

```
# Usage:  .\get-idigiSnapshot.ps1 <cloud> <username> for example: ".\get-
idigiDiaSnapshot.ps1 developer idigiUserName"
$Cloud = $args[0]
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/DiaChannelDataFull
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.DiaChannelDataFull | Select-Object -Property @
{Name="devConnectwareId";Expression={$_.id.devConnectwareId}}, `
@{Name="ddInstanceName";Expression={$_.id.ddInstanceName}}, @
{Name="dcChannelName";Expression={$_.id.dcChannelName}}, * -ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are
# shown using select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property devconnectwareid, ddInstanceName,
dcChannelName, dcdUpdateTime, dcdStringValue, dcdFloatValue, `
dcdIntegerValue, dcdBooleanValue, dcUnits | sort-object -property
devconnectwareid, ddInstanceName, dcChannelName
```

get-idigiDiaHistory (DiaChannelDataHistoryFull)

This script will query /ws/DiaChannelDataHistoryFull. This queries the last 1000 records on your account. Depending on timing, you may get more or less samples per sample point.

```
# Usage:  .\get-idigiDiaHistory.ps1 <cloud> <username> for example: .\get-
idigiDiaHistory.ps1 developer idigiUserName
$Cloud = $args[0]
$iDigiUserID = $args[1]
# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
# The "?orderby=dcdhId%20desc" ensure we get the most recent samples.
# WIthout it you may get a lot of samples for one device, and none for others,
due to the 1000 receord maximum
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/DiaChannelDataHistoryFull/?orderby=dcdhId%20desc
```

```
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties

$ResultTable = $XMLfile.result.DiaChannelDataHistoryFull | Select-Object -
Property @{Name="dcdhId";Expression={$_.id.dcdhId}}, `
@{Name="devConnectwareId";Expression={$_.id.devConnectwareId}}, @
{Name="ddInstanceName";Expression={$_.id.ddInstanceName}}, `
@{Name="dcChannelName";Expression={$_.id.dcChannelName}}, * -ExcludeProperty id

$ResultTable | select-object -property dcdhId, devconnectwareid, ddInstanceName,
dcChannelName, dcdUpdateTime, dcdStringValue, `
dcdFloatValue, dcdIntegerValue, dcdBooleanValue, dcUnits | sort-object -property
devconnectwareid, ddInstanceName, dcChannelName, dcdUpdateTime
```

get-idigiSmartEnergySnapshot (XbeeAttributeDataCore)

This script will query /ws/XbeeAttributeDataCore. This queries the last 1000 records on your account. Depending on timing, you may get more or less samples per sample point.

```
# Usage:  .\get-idigiSmartEnergySnapshot.ps1 <cloud> <username> for example:
".\get-idigiSmartEnergySnapshot.ps1 developer idigiUserName"
$Cloud = $args[0]
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/XbeeAttributeDataCore
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.XbeeAttributeDataCore | Select-Object -Property @
{Name="xpExtAddr";Expression={$_.id.xpExtAddr}}, `
@{Name="xeEndpointId";Expression={$_.id.xeEndpointId}}, @
{Name="xcClusterType";Expression={$_.id.xcClusterType}}, `
@{Name="xcClusterId";Expression={$_.id.xcClusterId}}, @
{Name="xaAttributeId";Expression={$_.id.xaAttributeId}}, * `
-ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are shown using
# select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property devConnectwareId, xpExtAddr, xeEndpointId,
xcClusterType, xcClusterId, xaAttributeId, `
cstId, xeProfileId, xeDeviceId, xeDeviceVersion, xaAttributeType,
xadAttributeStringValue, xadAttributeIntegerValue, xadUpdateTime `
| sort-object -property devconnectwareid, xeEndpointId, xcClusterId
```

get-idigiDeviceCore (DeviceCore)

This script will query /ws/DeviceCore. This provides basic information about each gateway, including status of the iDigi connection.

```
# Usage:  .\get-idigiDeviceCore.ps1 <cloud> <username> for example: ".\get-
idigiDeviceCore.ps1 developer idigiUserName"
```

```
$Cloud = $args[0]
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/DeviceCore
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.DeviceCore | Select-Object -Property @
{Name="devId";Expression={$_.id.devId}}, `
@{Name="devVersion";Expression={$_.id.devVersion}}, * -ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are
# shown using select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property  devConnectwareId, devId, devVersion,
devRecordStartDate, devMac, `
cstId, grpId, devEffectiveStartDate, devTerminated, dvVendorId, dpDeviceType,
dpFirmwareLevel, `
dpRestrictedStatus, dpLastKnownIp, dpGlobalIp, dpConnectionStatus,
dpLastConnectTime, `
dpContact, dpDescription, dpLocation, dpServerId, dpZigbeeCapabilities,
dpCapabilities, dpUserMetaData, dpTags `
| sort-object -property devConnectwareid dpConnectionStatus
```

Modbus DIA block register map

(Note: this feature is being tested, and will be included in a future release of DIA)

DIA Modbus register map (basic)

The block oriented Modbus server returns a fixed register map common for all supported devices. Registers which are unused (for example the analog outputs on a analog input device) shall be zero (0x0000).

Basic register map

All supported devices start with the basic register map, where any analogs are converted to fixed point integers.

Modbus	Offset	Mode	Name	Description
4x00001	0	Read-Only	--	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	status	Device Status as 16 bits, See below
4x00003	2	Read-Only	din	Digital Inputs as 16 bits, 0 if none
4x00004	3	Read-Only	ain1	Analog Input #1, 0 if none
4x00005	4	Read-Only	ain2	Analog Input #2, 0 if none
4x00006	5	Read-Only	ain3	Analog Input #3, 0 if none
4x00007	6	Read-Only	ain4	Analog Input #4, 0 if none
4x00008	7	Read-Write	dot	Digital Outputs as 16 bits, 0 if none
4x00009	8	Read-Write	aot1	Analog Output #1, 0 if none
4x00010	9	Read-Write	aot2	Analog Output #2, 0 if none
4x00011	10	Read-Write	aot3	Analog Output #3, 0 if none
4x00012	11	Read-Write	aot4	Analog Output #4, 0 if none

Modbus	Offset	Mode	Name	Description
4x00013	12	Read-Only	volt	Battery Voltage as Fixed Point, so 0 = 0.0v, 420 = 4.20v up to 65535 = 635v, 0 if none
4x00014	13	Read-Only	timestamp	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	--	High Word of UNIX style date/time of last data reading
4x00016	15	Read-Only	dd	Lower word of the XBee DD value
4x00017	16	Read-Only	mac	bytes 7-8 of IEEE MAC Address
4x00018	17	Read-Only	--	bytes 5-6 of IEEE MAC Address
4x00019	18	Read-Only	--	bytes 3-4 of IEEE MAC Address
4x00020	19	Read-Only	--	bytes 1-2 of IEEE MAC Address

Device status bits

The 16-bit device status are:

mask	bit	Description
0x0001	1	Driver Fault; if 1 then some DIA driver fault is true
0x0002	2	Device is off-line; if 1 then the device is not responding. Depending on config, normally there will be no Modbus response in this situation to simulate an unreachable slave.
0x0004	3	Low-battery signal; if 1 then device signals a low-battery condition
0x0008	4	Device Fault; if 1 then device is talking, but indicating internal error
0x0010	5	Event Sample; if 1 then the data in this sample was triggered by an event, such as alarm exception or the operator pressing the 'wake / sample' button. If 0, then this sample is a normal time cycle update.
0x00E0	6-8	Reserved
0x0100	9	If 1, then Digital Input register is valid
0x0200	10	If 1, then Analog Input registers are valid
0x0400	11	If 1, then Digital Output register is valid
0x0800	12	If 1, then Analog Output registers are valid
0xF000	13-16	Reserved

Supported devices

Digi XBee AIO Adapter (mbdia_xbee_aio driver)

Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Analog Input #1, see format below
4x00005	4	Read-Only	Analog Input #2, see format below
4x00006	5	Read-Only	Analog Input #3, see format below
4x00007	6	Read-Only	Analog Input #4, see format below
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

The fixed-point format for the registers shall be:

- type = 'raw', then the raw binary value
- type = 'off', then zero
- type = 'TenV', then voltage * 1000, so 4.23v is 4230
- type = 'CurrentLoop', then current * 1000, so 12.10mA is 12100
- type = 'Differential', then voltage * 1000, so -1.23v is 0xFB32 and +1.23v is 1230

Digi XBee /L/T/H sensor adapter (mbdia_xbee_sensor driver)

Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Temperature as Deg C, fixed point * 100, so 2300 = 23.00 DegC
4x00005	4	Read-Only	Light as direct value
4x00006	5	Read-Only	Humidity as fixed point * 100, so 7950 = 79.50 % RH, zero if this sensor does NOT have a humidity input
4x00007	6	Read-Only	Temperature as Deg F, fixed point * 100, so 7950 = 79.50 DegF

Modbus	Offset	Mode	Description
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Digi Smart Plug (mbdia_xbee_rpm driver)

Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00003	2	Read-Only	Digital Outputs: bit 0x0001 is plug status
4x00004	3	Read-Only	Current as Amps, fixed point * 100, so 230 = 2.3 Amps
4x00005	4	Read-Only	Light as direct value
4x00006	5	Read-Only	Temperature as Deg C, fixed point * 100, so 2950 = 29.50 DegC
4x00007	6	Read-Only	Always zero
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Massa M3 Ultrasonic level sensor (mbdia_massa_m3 driver)

Any registers not show below are returned as zero (0) - but that might change in the future.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Level as inches, fixed point * 100, so 1423 = 14.23 inches
4x00005	4	Read-Only	Temperature as Deg C, fixed point * 100, so 2315 = 23.15 DegC
4x00006	5	Read-Only	Target Strength, fixed point * 100, so 7500 = 75%
4x00007	6	Read-Only	Latest Event counter as word, so 1 to 65535
4x00013	12	Read-Only	Battery Voltage, fixed Point * 100, so 420 = 4.20v
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Note Target Strength is only 0%, 25%, 50%, 75% or 100%

Modbus DIA Client

Modbus DIA Client Driver

This driver polls remote Modbus servers/slaves for data (words or bits), which are then converted to standard DIA channels. So it imports Modbus data into DIA channels.

You can get more info and hints in the Modbus starting page.

Uses Digi IA/Modbus Engine

The normal Digi "IA Modbus Engine" is required, and the DIA will appear to it as a remote client using Modbus/TCP form in UDP (the UDP is important), so you need to enable at least the Modbus/TCP master on UDP port 502. TCP/IP could have been supported, but it consumes considerable more resources than UDP/IP without adding value.

You need to set up IA table destinations pointing at your slaves, so these can be any combination of Modbus/TCP, Modbus/ASCII or Modbus/RTU via TCP/IP, UDP/IP, serial or Zigbee mesh.

This design has the following advantages:

- Supports a wide variety of Modbus devices:
  - Modbus/TCP slaves/servers via TCP/IP or UDP/IP, via Ethernet or cellular
  - Modbus/RTU slaves/servers via TCP/IP, UDP/IP, direct serial port or XBee 232/485 Adapters
  - Modbus/ASCII slaves/servers via TCP/IP, UDP/IP, direct serial port or XBee 232/485 Adapters
- Allows transparent pass-through for remote Modbus Clients. For example, a remote Modbus/TCP client can read or write a thousand various registers in a complex Modbus device, yet the DIA configuration need only import three registers of interest
- The Digi IA/Modbus engine is a very mature product is supported by at least these Digi products:
- Digi ConnectPort X4, X4H, X8
- Digi ConnectPort TS8, TS16
- Digi Connect WAN IA, WAN VAN
- Digi ConnectPort WAN

This design has the following disadvantages:

- The Modbus DIA Driver cannot be used on Digi products unable to run the IA/Modbus engine concurrently with Python.
- Specifically, it cannot be used on:
  - Digi ConnectPort X2
  - Digi ConnectPort X3

Example YML Configuration

Below is an example YML configuration which reads two blocks of data from remote server(s)

```
  - name: mbus
    driver: devices.modbus.mbus_udp_device:MBusUDPDevice
    settings:
        poll_rate_sec: 30
        udp_peer: ('127.0.0.1',502)
        trace: 'debug'
        round: 3
        poll_list:
            - poll: in01
              pollinfo: { 'uid':1, 'fnc':3, 'ofs':0, 'cnt':20 }
              channels:
                - parse: { 'nam':'panel', 'ofs':3, 'frm':']H', 'unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }
                - parse: { 'nam':'battery', 'ofs':4, 'frm':']H', 'unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }
                - parse: { 'nam':'load','ofs':5,'frm':']H','unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }


            - poll: in02
              pollinfo: { 'uid':1, 'fnc':1, 'ofs':0, 'cnt':48 }
              channels:
                - parse: { 'nam':'DIN', 'ofs':8, 'frm':'?', 'unt':'valid', }
                - parse: { 'nam':'AIN', 'ofs':9, 'frm':'?', 'unt':'valid', }
                - parse: { 'nam':'DOT', 'ofs':10, 'frm':'?', 'unt':'valid', }
```

Base Device Settings

A single thread is spawned for each **MBusUDPDevice** device created. The thread sleeps and wakes on a cycle defined by the YML poll_rate_sec setting. When it wakes, it runs through a list of POLLS, running them in strict half-duplex sequential order. Modbus is not a high-speed protocol, so users must be reasonable in their selection of poll rates.

Specific settings for the base device:

```
    poll_rate_sec: 30
    udp_peer: ('192.168.196.140',502)
    trace: 'debug'
    round: 3
    poll_list:
    ...
```

poll_rate_sec

- Defines the poll rate. Note that the device attempts to prevent system drift by compensating for delays in response.
- Type integer, stated in seconds, minimum is once per 5 seconds
- Optional, default = once per 15 seconds

udp_peer

- Defines the remote host address as IP (or DNS name) plus UDp port number to send requests to
- Type string
- Optional, default = ('127.0.0.1',502) (localhost and the well-known Modbus/TCP port number, assumes links to Digi IA/Modbus Engine)

- Note that at present, the client does NOT time-out. It relies upon the server to always respond - which is true when the Digi IA/Modbus Engine is used.

trace

- Defines how chatty the driver should be on the trace output
- Type string or integer
- Optional, default = 0x0032 (fancy steady-state, start/stop events and field errors)
- See the ia_trace.py documentation for more options.

round

- Defines a global 'round()' for floating point value
- Type integer
- Optional, default = 999, magic number for ignore
- Example: a temperature with a +/- 1 deg accuracy should not be returned as '23.34876549'. Setting round: 2 will cause all floating points to rounded to 2 places, so '23.34876549' is ASCII encoded as '23.35'

poll_list

- Defines a collection (list) of Modbus block read/writes and now the data should be imported to DIA
- Required
- Example: see next section

Poll List Settings

Each **MBusUDPDevice** device can run a collection of Modbus polls. Although each MBusUDPDevice sends request to a single server, that server can interpret the Modbus Unit Id (or slave address) to obtain responses from multiple remote servers.

Specific settings for the poll objects:

```
- poll: in01
pollinfo: { 'uid':10, 'fnc':3, 'ofs':0, 'cnt':20 }
channels:
...
```

poll

- Define the name for this poll object. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness
- Example: in01 or motor or plc_b

pollinfo

- Define the Modbus parameters for this poll object. Only functions 1, 2, 3 and 4 are supported at present.

- Type Python dictionary
- Required, user is responsible to insure uniqueness
- Example: { 'uid':10, 'fnc':3, 'ofs':0, 'cnt':20 } means poll 20 holding regs starting at 4x00001 on slave unit-id 10. You should create the required destination in the Digi IA/Modbus table

pollinfo['uid']

Defines the Modbus Unit Id or slave address to use in the request

pollinfo['fnc']

Defines the Modbus function code to use - limited to function 1, 2, 3 and 4

pollinfo['ofs']

Defines the offset (zero-based). So reading Modbus register 4x00001 is offset 0, which reading coil 0x00007 is offset 6.

pollinfo['cnt']

Defines the number of registers (words) or coils (bits) to read

channels

- Defines a collection (list) of DIA channels to create from this Modbus poll object
- Required
- Example: see next section

Channel List Settings

Each Modbus poll block of data can be imported into multiple DIA channels.

Specific settings for the channel objects:

```
    - parse: { 'nam':'panel', 'ofs':3, 'frm':']H', 'unt':'vdc', 'typ':'float',
'expr':'(%d/1000.0)*3.25' }
    - parse: { 'nam':'totalEnergy', 'ofs':0, 'frm':'[L', 'unt':'Wh' }
    - parse: { 'nam':'load', 'ofs':5, 'frm':'[f', 'unt':'vdc', }
    - parse: { 'nam':'overload', 'ofs':8, 'frm':'?', 'unt':'valid', }
```

parse

- Define the import/creation of a single DIA channelect. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness
- Example: { 'nam':'totalEnergy', 'ofs':0, 'frm':'[L', 'unt':'Wh' } reads the first two registers of the poll block, treats them as a Modbus 32-int with LOW word first. It creates a DIA channel named mbus.in02_acTotalEnergy which will include a sample such as <Sample: "17294" "Wh" at "2009-10-28 15:36:30">

parse['nam']

- Defines the DIA channel name. So if the poll is named 'inv01' and channel named 'totalEnergy', then the final channel will be named 'inv01_totalEnergy'.

- Type string
- Required
- Example: four poll blocks named ['inv01','inv02','south_wing','ghouse'] could create four channels named inv01_totalEnergy, inv02_totalEnergy, south_wing_totalEnergy, and ghouse_totalEnergy.

parse['ofs']

- Defines the word or bit offset in the poll block. It is zero-based.
- Type integer
- Required

parse['frm']

- Defines how the Modbus data bytes are parsed to obtain the DIA sample.
- Type string
- Required
- Values are similar to the Python struct format
  - '?' for 1-bit coils - these can be parsed from register or coil response, but be aware the 'ofs' value is from the start of the poll block so reading the 3rd bit in the 4th register requires 'ofs' of 66 (67th bit, zero-based)
  - 'h' for 16-bit signed int import to DIA ('H' is unsigned)
  - '[i' for 32-bit signed int import to DIA ('[I' is unsigned), with the '[I' meaning LOW word is in first Modbus register, which ']I' or '>I' would be BIG word in first Modbus register
  - '[f' for 32-bit float import to DIA with '[' or ']' being used as in the 32-bit int

parse['unt']

- Defines the Unit of Measure string
- Type string
- Optional, default is (an empty string)

parse['typ']

- Over-rides the type implied by the format string
- Type string, values in ['float','int','long','bool']
- Optional, default is to use parse['frm'] to estimate channel type

parse['expr']

- An 'eval' expression to do simple value conversion of the data
- Type string, in print format such as '(%d/1000.0)*3.25'
- Optional, default is no conversion
- Example: { 'nam':'panel', 'ofs':3, 'frm':'H', 'unt':'vdc', 'typ':'float', 'expr':'(%d/1000.0)*3.25' } treats the Modbus register as a 16-bit signed integer, yet applies the formula '(value/1000.0)*3.25' to create a floating point channel sample

- Example: { 'nam':'cold', 'ofs':3, 'frm':'H', 'unt':'is_cold', 'typ':'bool', 'expr':'bool(%d<230)' } treats the Modbus register as a 16-bit unsigned integer, yet creates a boolean channel sample

Handling Sample Errors

If the Modbus server/slave did NOT respond, then the DIA sample will be an 'AnnotatedSample', which just means you'll find a new field named 'errors', which is a set[] containing the reason code which are:

- 'not_init' means the sample has never been updated even once.
- 'stale' means the last 3 or more polls have timed out.
- 'bad_calc' means some import/conversion failure occurred.

Modbus DIA server

Enabling the DIA Modbus server

The DIA Modbus server allows remote Modbus masters/clients to query your DIA devices as if they were Modbus devices. However, the data has been cached by the Digi gateway, so if the DIA device is sleeping and wakes only once per hour, then the Modbus data returned will be repeated (stale) for the entire hour.

Data models

Block Devices

Each device appears as a distinct Modbus destination with the I/O such as:

- 4 analog inputs (field to system)
- 4 analog outputs (system to field)
- 16 digital inputs (field to system)
- 16 digital outputs (system to field)

More information is on this Wiki page: Modbus DIA block register map.

How the Modbus Unit Id (or Slave Address) is mapped to DIA device is covered on this Wiki page: Enable Modbus query of DIA devices.

Channel Mapping

(Future work) You build custom Modbus maps on a channel-by-channel basis.

Robust Block Server

This version runs only on Digi gateways with the Modbus/IA Engine. DIA uses Modbus/UDP on localhost to the IA Engine, which provides a mature and robust multi-master solution.

It allows up to 32 incoming Modbus masters via:

- Modbus/TCP in TCP/IP or UDP/IP
- Modbus/RTU on gateway serial ports, or encapsulated in TCP/IP or UDP/IP
- Modbus/ASCII on gateway serial ports, or encapsulated in TCP/IP or UDP/IP
- Digi Realport on Windows set up for UDP mode, which enters the Modbus/IA Engine as a Modbus/RTU or Modbus/ASCII master encapsulated in UDP/IP

Gateway configuration

By web UI

To enable correct bridging of Modbus into the DIA, create a simple Industrial Automation configuration summarized by these steps:

- Click the Applications | Industrial Automation link on the left side of the display
- Confirm a Modbus Protocol table exists, or add one if required. The table name can be anything
- Click the Table Name to see the Table Settings page

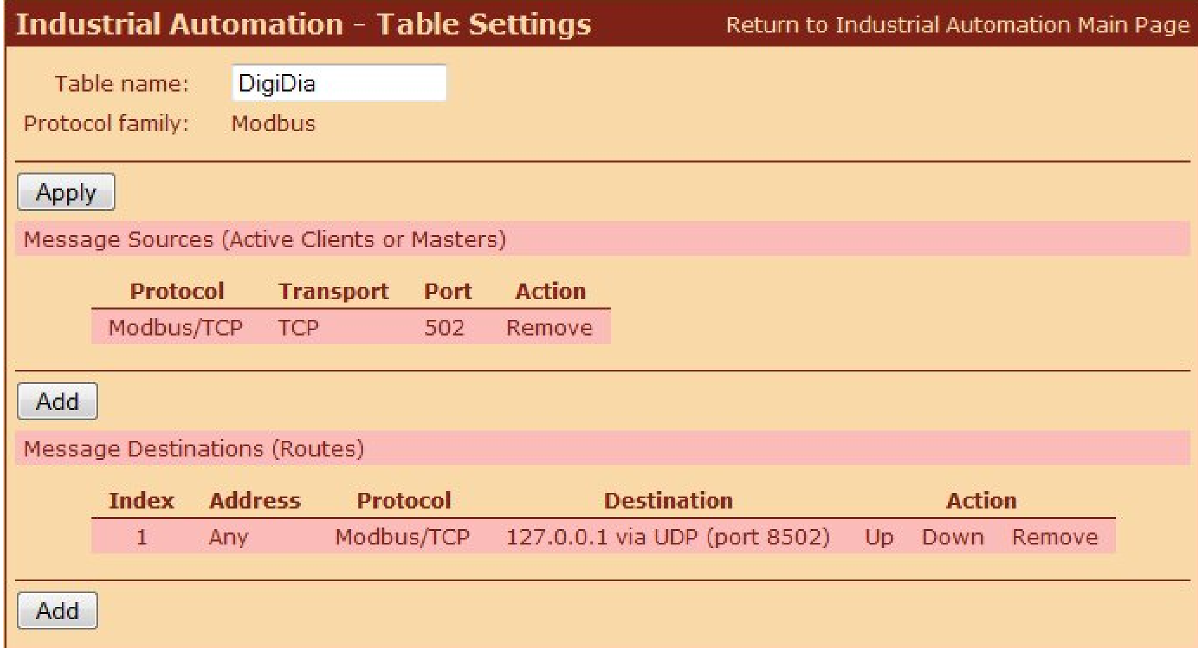
- Add at least one message source (an incoming Master/Client). This could be Modbus/TCP on Ethernet or any other selection. More than one can be created.
- Add a message destination to a Modbus/TCP in UDP/IP on IP 127.0.0.1, UDP port 8502. To start with, route all messages to this and place it in the first row (index #1).
- reboot the gateway

By CLI over Telnet or SSH

```
set ia table=1 state=on name=DigiDia family=modbus accessmode=multi
set ia table=1 ownerperiod=15000
set ia table=1 addroute=1
set ia table=1 route=1 active=on type=ip protaddr=0-255 protocol=modbustcp
set ia table=1 route=1 transport=udp connect=passive address=127.0.0.1
set ia table=1 route=1 ipport=8502 replaceip=off slavetimeout=1000
set ia table=1 route=1 chartimeout=50 idletimeout=0 lineturnmode=off
set ia table=1 route=1 fixedaddress=0 rbx=off
set ia master=1 active=on type=tcp ipport=502 protocol=modbustcp table=1
set ia master=1 priority=medium messagetimeout=2500 chartimeout=50
set ia master=1 idletimeout=0 lineturnmode=off errorresponse=on
set ia master=1 broadcast=replace
```

The Final Result

Your final IA Engine Configuration should look something like this (web colors may vary - this is a NDS web customization):



Configuring DIA

DIA YML Changes

Below is an example YML showing use of DIA 1.3 with auto-enumeration of XBee AIO and LTH sensors. Any device discovered will be automatically added to the Modbus server/slave list.

Important points:

1. You MUST use the special Modbus 'sub-class' for the DIA drivers - such devices.modbus.mbdia_xbee_aio:MBusXBeeAIO. These behave like the normal DIA drivers, but have the added Modbus calls to create the Modbus register maps
2. You must include the Modbus server presentation: presentations.modbus.mbdia_pres:MbDiaPresentation

```
- name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
  - name: xbee_autoenum
    driver: devices.xbee.xbee_devices.xbee_autoenum:XBeeAutoEnum
    settings:
        xbee_device_manager: xbee_device_manager
        short_names: True
        devices:
           - name: ain
            driver: devices.modbus.mbdia_xbee_aio:MBusXBeeAIO
            settings:
                sample_rate_ms: 60000
                power: "On"
                sleep: False
                channel1_mode: "CurrentLoop"
                channel2_mode: "CurrentLoop"
                channel3_mode: "CurrentLoop"
                channel4_mode: "CurrentLoop"
          - name: lth
            driver:     devices.modbus.mbdia_xbee_
sensor:MBusXBeeSensor
            settings:
                sleep: True
                sample_rate_ms: 15000
                awake_time_ms: 5000
presentations:
  - name: mbus_srv
    driver: presentations.modbus.mbdia_pres:MbDiaPresentation
    settings:
        mapping: "('auto', 1, 20)"
        auto_enum_name: xbee_autoenum
```

Seeing the Modbus 'Unit Id/Slave Address' assignment

The Modbus DIA server maintains a text file in the gateway's Python file systems named 'mbus_map.txt'. Every time it boots, it starts by reading this file (if it exists). Any new nodes seen are appended. You can edit this file at any time, reordering the lines.

This is a static example of mbus_map.txt, where the devices were manually entered by name in the YML as mapping: "((1,'solar'), (2,'outdoor'), (3,'indoor'))"

```
# DIA Modbus Server unit_id mapping as of 2010-03-24 15:01:56
1,'solar','XBeeAIO','00:13:a2:00:40:4b:90:24!'
2,'outdoor','XBeeSensor','00:13:a2:00:40:4a:6e:83!'
3,'indoor','XBeeSensor','00:13:a2:00:40:32:14:FA!'
```

This is a dynamic/autoenum example of mbus_map.txt

```
# DIA Modbus Server unit_id mapping as of 2010-03-26 21:46:48
1,'lth_14_c6','XBeeSensor_LT','[00:13:a2:00:40:32:14:c6]!'
2,'lth_15_1a','XBeeSensor_LT','[00:13:a2:00:40:32:15:1a]!'
```

Motion Detection with XBee

How to cub-class a DIA driver.

This page covers how to leverage an existing DIA driver without modifying the original file.

Motion detector example

In this example, I wish to connect a common Motion-Detector/Glass-Break device with relay contacts ('dry-contacts') via ZigBee to Device Cloud and DIA.

The motion-detector sensor requires 12vdc to run, so I used an older XBee DIO Adapter which runs on 9-30vdc. With a 12vdc supply, the entire setup can be powered including directly driving 12vdc relays and 12vdc low-voltage garden-style lighting.

Technically, the existing DIA DIO driver could be used to read the sensor contacts and drive the lamp output, however the motion detection event is very short-lived. It may be true for only a few seconds, and turning on a lamp for a few seconds is not what is required. Instead, a timer is required to turn the lamp on for a few minutes. In theory one could use the DIA transforms to do this, but at present they are stateless, so cannot function as timers.

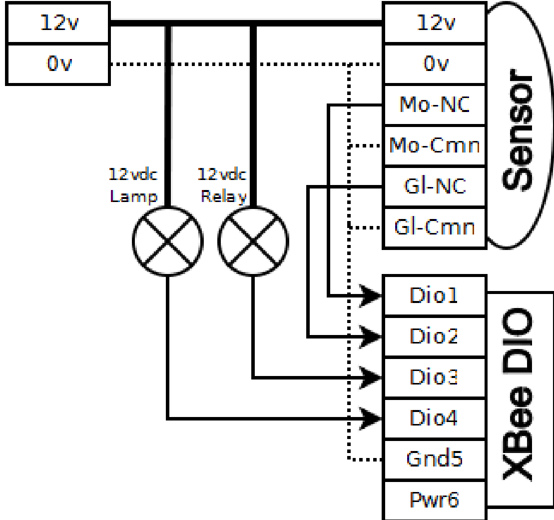
So one has two choices:

- Clone and rename the existing DIA DIO driver, and modify it directly.
- Create a new subclass of the existing DIA DIO driver which adds the new functionality.

In this demo example I chose to subclass. The new driver allows the existing DIO driver to manage incoming data and update the standard DIO data channels. Then it reads the input channels and adjusts the output channels as required.

Ideally, the time the lamp is held on should be a setting, which complicates the subclass slightly. That is left for a future TODO - including the support for the glass-break input and the 12vdc relay to drive brighter AC lights. If all goes well, my final design will have a programmable XBee to manage the light and relay control locally within the XBee adapter itself.

Hardware example



Wiring the system is quite strait forward. The items required:

- XBee Digital I/O Adapter preferably with 9-30vdc supply, but if you have a 3-6 vdc model, buy a PCB to step-down 12 vdc to 5 vdc. For example, ebay seller electronics-salon sells a nice assembled model for us$10 which I have used on several projects.
- Any 12 vdc supply large enough to power everything - anything larger than 1.5 Amp should be okay. As is, the 10-watt halogen garden lamp (at about 0.8 A) is the most power-hungry device.
- Motion Sensor, such as Bravo BV-500GB by DSC, sold by smarthome.com
- 12 vdc Low-Voltage Lamp, such as the Malibu brand in the image, purchased from Home Depo. Once the halogen lamp fails, they are easy to replace with LED versions from superbrightleds.com
- 12 vdc Power Relay with AC-rated contacts, such as Omron LY1F-DC12 from digikey.com which is rated at 15 Amp @ 110 vac. It requires 75m A @ 12 vdc to operate.

Notes

- The 12 vdc power ground link to the XBee DIO Adapter is optional if the XBee DIO Adapter is powered by the 12 vdc supply, but it is required if the XBee DIO Adapter is powered by (for example) the 5vdc wall-wart Digi will supply if you buy the 3-6 vdc XBee DIO Adapter
- The four DIO terminals of the XBee DIO Adapter when used as outputs can only sink current, so pull the 12 vdc low. That is why both the lamp and relay are tied to 12 vdc, with the XBee DIO Adapter acting as ground.
- The power-output (t erminal #6) of the XBee DIO Adapter can only supply 50 mA @ 12 vdc, which is not enough for either the landscape lamp or the power relay. However, it would be enough for direct LED lights or a smaller reed-relay which could power a larger lamp or relay. Moving an 'output' to this terminal would free up one of the XBee adapter IO for use as another input, such as a tamper contact, or perhaps even a sensor to detect if the lamp is really on and emitting light.

- The reason for having both a dim 12 vdc lamp and bright 120 vac lamp (via 12 vdc relay) is I plan to use the dim 12vdc lamp more as a night-light then as security. So based on time of day, sunlight level, and whether the security level is 'we are home' or 'we are out', either or neither lamp may be lit due to motion.
- XBee DIO Adapter DIP Switch settings: 2, 3, and 4 are on.
- For inputs, only terminals 1 & 2 have internal pull-up. To use terminals 3 & 4 as input, you may need to use resisters (for example 10K ohm) to pull the inputs up to 12vdc when floating/open.

YML code

This is just the fragment required for the custom DIO driver. We desire the DIO adapter to fresh the channel status every 30 seconds, plus the channels will be updated any time they change. The four IO are assigned like this:

- Channel 1 input is the Motion-Detect NC/normally-closed contact - it opens when motion is seen.
- Channel 2 input is the Glass-Break NC/normally-closed contact - it opens when the distinctive sound of break glass is heard. This input is for future use, and is ignored now.
- Channel 3 output is reserved for a 12 vdc relay which can drive 120 vac lamps, which would be appropriate for yard or full-room lights.
- Channel 4 output directly drives a 12 vdc garden-style lamp, which is used for demo purposes and night-lighting.

```
  - name: motion
    driver: devices.experimental.xbee_motion:XBeeMotion
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:33:4e:a9!"
        sample_rate_ms: 30000
        sleep: False
        power: Off
        channel1_dir: "In"
        channel2_dir: "In"
        channel3_dir: "Out"
        channel4_dir: "Out"
```

Source code

Here is the entire custom file. Since it uses the normal XBee DIO driver, there is little function required. Basically, this driver allows the XBee DIO driver to process the incoming data, then compares the Motion-Detect input to the current light state.

```
import traceback
import time

from devices.device_base import DeviceBase
from devices.xbee.xbee_devices.xbee_base import XBeeBase
from settings.settings_base import SettingsBase, Setting

from devices.xbee.xbee_devices.xbee_dio import *

class XBeeMotion(XBeeDIO):
```

```
    OFF_DELAY = 30

    def __init__(self, name, core_services):

        ## Initialize the base class
        XBeeDIO.__init__(self, name, core_services)

        # zero means off, else holds time.time turned on
        self.__light_on = 0

        return

    ## Locally defined functions:
    def sample_indication(self, buf, addr, force=False):

        ## allow base class to process data message
        XBeeDIO.sample_indication(self, buf, addr, force)

        # then do our reaction to the status
        now = time.time()

        motion = self.property_get( 'channel1_input').value
        if motion:
            if self.__light_on == 0:
                print 'Motion Seen, turning light on'
                self.set_output(Sample(now, True, "On"), 3)

            # else: print 'Motion Seen, light already on'

            # in all cases, bump light_on time to now
            self.__light_on = now

      else:
            # else no motion
            if self.__light_on != 0:
                # then light is on
                if (now - self.__light_on) > self.OFF_DELAY:
                    print 'No Motion, turning light off'
                    self.set_output(Sample(now, False, "Off"), 3)
                    self.__light_on = 0

                # else: print 'No Motion Seen, light is on, should stay on'

      return
```

Network-Time-Server-DIA-example

Overview

This simple DIA example is a NTP (Network Time Protocol) connector / driver. With it, you can easily direct your ConnectPort-X to update its system time via any publicly available time server.

Prerequisites

You should register at www.digi.com/products/cloud/digi-device-cloud and then download DIA. After that, read the Getting Started guide and begin looking at the simple drivers (found under ../src/devices in the DIA source code tree). You should have a basic working knowledge of DIA (how to start / stop & use basic drivers / presentations) before proceeding.

Download the Network_Time_Protocol_Dia_Driver.zip archive. The included "yml" file is the configuration is used when starting the DIA application. It's included here for reference.

```
## NTP Time Server DIA Configuration File
devices:
# The ntp_time_device device driver; creates a demonstration device
# containing a network time server address and timezone settings.
# Recommended update_rate for this driver is 86400 seconds (1 day).
- name: ntp_time
  driver: devices.ntp_time_device:NTPTimeDevice
  settings:
      Time_Server: "2.fedora.pool.ntp.org"
      Time_Zone: -6
      update_rate: 3540

presentations:
# Create a new console instance on TCP port 4146.  It can be connected
# to by using any telnet client.
- name: console0
  driver: presentations.console.console:Console
  settings:
      type: tcp
      port: 4146
  # Create a web presentation instance.  When running on a PC this will
  # start a new web-server on a default port
  # When running on a Digi device this presentation will "extend" the
  # web-server built in to the Digi device with a new page named
  # "idigi_dia.html".
  - name: web0
    driver: presentations.web.web:Web
    settings:
        page: idigi_dia.html
        port: 8081
```

Re-compile DIA to using this configuration file or incorporating the elements you require. You can do this via the command line (Python make.py ntp_time.yml) or using the Digi IDE for DIA.

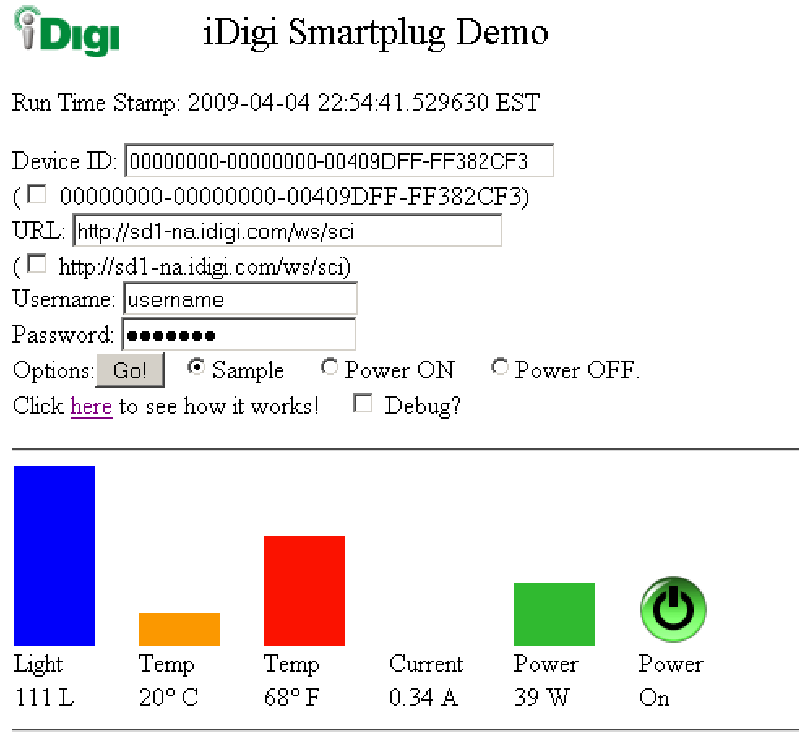
Test the "virtual" device with your browser (Gateway IP Address/idigi_dia.html) or via telnet (to port 4146 on your Gateway). You can adjust the Time Server, Time Zone and the update rate (in seconds).

Python-based SmartPlug sensor example

Overview

This simple Web Services example uses the Digi Smartplug (remote power management) sensor (light/temperature/current), is written in Python, and displays the result of the most recently available samples via html. The Digi Smartplug contains a relay, so this example includes both the ability to view samples, as well as the option to "set" the relay to a power-on or power-off state. The DIA configuration file, when used with the DIA application running on a Digi ConnectPort gateway, defines the RCI presentation. The RCI presentation is a web service method. When you provide the proper credentials and target information, DIA will listen for web services requests to sample or set the power relay on the sensor.

Even if you don't have a Digi Smartplug you can still easily adapt this code for other sensors. The principle is the same. You are simply sending a web service request to Device Cloud in order to get or set sensor channels. If a sample is available, a "channel_get" (or channel_dump) request will return the most recent value.



Instructions:

Download the zip file SmartPlug.zip, which includes the Python source and images. There are two (2) Python versions. One that provides a basic "channel_dump" of all attached sensors, and another version which very specifically requests (channel_get) only the specific sensor and channels (light, temperature, current, power_on). Place the Python code in your web server's "cgi-bin" directory and the images in the appropriate image location. You may need to change some of the image URL

references. You will also want to change the references to the deviceID and possibly the Device Cloud connectivity server URL.

Code Flow:

The code flow is simple, the smartplug.py code draws an html form. You enter the required information and click "Go." The same smartplug.py or smartplugV2.py program is called, this time reading the form variables. A web service call to Device Cloud is now created and submitted. Device Cloud receives the request, validates (checks permission) and sends to the target device. The target device (Digi gateway), running DIA, has a handler (callback) which is listening for a web service call (RCI). Upon receiving a request, it either provides the "channel_dump," "channel_get," or "channel_set" information and response back up to Device Cloud. Device Cloud then forwards the response to the requester (your smartplug.py or smartplugV2.py program), which then parses the XML response and displays the information on your browser.

DIA Configuration:

Here's an example DIA configuration file (yml) that will work with this example. Copy, modify and use with your own DIA application.

```
# This configuration file is used to specify which components will be
# specified for use with the DIA Smartplug Demo.
# Essentially, we will be selecting the sources of information, and how
# that information is presented.
# The devices section specifies settings for devices
devices:
# First we specify an XBee Device Manager.  Since the XBee is considered,
# a shared resource on our system, we need a device which can manage
# requests from other devices relating to the XBee.
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
# Here we specify that we have a Digi Smart Plug as a device.  In the
# settings, we specify the parent XBee Device Manager for this device,
# as well as the extended address of the actual node that is associated
# to our gateway.  We then specify that we would like the Smart Plug
# to send up sensor readings once a second, and that when DIA starts,
# the outlet on the Smart Plug should be turned on.
  - name: smart_plug_1
    driver: devices.xbee.xbee_devices.xbee_rpm:XBeeRPM
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:34:0c:6c!"
        sample_rate_ms: 2000
        idle_off_seconds: 120
        default_state: "Off"


# The presentation section allows us to specify the ways in which
# information in our system is available.
presentations:
# Here we define a standard console method.  Using this,
# someone can connect (with telnet in Windows for example),
# to get a quick and simple look at the current state of the
# system.  They can view the status of all channels, and set
# any channels that support being configured.
  - name: console
    driver: presentations.console.console:Console
```

```
    settings:
    type: tcp
    port: 4146

# Create a web presentation instance.  When running on a PC this will
# start a new web-server on a default port
# When running on a Digi device this presentation will "extend" the
# web-server built in to the Digi device with a new page named
# "digi_dia.html".  See the file src/presentations/web/web.py form more
# information.
  - name: web0
    driver: presentations.web.web:Web
    settings:
        page: idigi_dia.html


# The RCIHandler allows for channel dumps, querying a channel, and setting
# a channel.  The RCIHandler parses an incoming XML formatted message
# to determine what you wish to do.  The messages are shown below. If
# an error is encountered, the request you sent will have a child error
# element specifying the error information.
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
        target_name: digi_dia
```

Running the Demo

Call the Python code (smartplug.py) from your web browser. Fill in the blanks and click on "Go."

- Username: Your Device Cloud username
- Password: Your Device Cloud password

Conclusion

That's it! Pull the code apart. Experiment. Improve it. Everything's inside the one Python code module. You'll find the web services function, the html display (near the end) and the XML for parsing the Device Cloud response. There's even a very simple graph builder, using 1x1 pixel colors to illustrate the light, temperature and current sample values.

Simple RCI by HTTP

Reading DIA channels via web commands

Digi provides many detailed documents explaining web services and remote RCI calls, but most provide too much detail or partial examples. They assume you already know how to move the requests, so just want the core syntax.

Simple Python for use on a PC

Below is a simple script which allows using XML to query data in real-time from a gateway running DIA.

Sample YML

What this DIA system does isn't important. The RCI calls below will read or write the properties from the device named 'level', which are named **level.alert** and **level.config** respectively. The **RCIHandler** must be enabled, plus the setting **target_name** must match what you place into your RCI calls.

```
devices:
  - name: count
    driver: devices.template_device:TemplateDevice
    settings:
        update_rate: 10

  - name: level
    driver: devices.experimental.alert_output:AlertOutput
    settings:
        source: 'count.adder_total'
        rising: 12.0
        falling: 11.5

presentations:
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
        target_name: idigi_dia
```

Writing the RCI request

Detailed information on the RCI commands accepted by the DIA RCI presentation can be found in the DIA user documentation.

Those RCI commands, such as <channel_dump />, <channel_set name="..." value="..." />, and <logger_set name="..." /> are wrapped in the following syntax. Notice that the string **target="idigi_dia"** matches the target name in the YML file.

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
      </do_command>
</rci_request>
```

In DIA versions 2.2.0.1 and below, the RCI handler can only accept a single tag in the <do_command> block. A workaround is to wrap multiple commands in an arbitrary wrapper tag like the following:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
`                <channel_get name="level.alert"/>
                <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

The only thing to remember with this is that the <blob> tag will wrap the response code in a similar fashion.

For DIA versions **newer than 2.2.0.1**, the RCI handler does not need a wrapper tag, so the below syntax would work:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
                 <channel_get name="level.alert"/>
                 <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

However, this will generate an XML parse error with DIA versions 2.2.0.1 and below.

Source Code

Below is an actual script written and used under Python 2.4.3 on a Windows 7 PC.

```
            # Simple PC example to query the DIA device

import httplib, urllib

msg_dump = \
"""<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
       </do_command>
   </rci_request>"""

msg_get = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_get name="level.alert"/>
           </do_command>
           <do_command target="idigi_dia">
               <channel_get name="level.config"/>
           </do_command>
       </rci_request>"""

# notice that the VALUE this channel takes is complex - a list of 3 values.
# That is defined by the DRIVER involved.  Most channels will take a
# simple True/False, integer, floating point or string value.
# examine the channel_get response to discover what to return channel_set
msg_set = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_set name="level.config" value="[10,9,True]"/>
```

```
        </do_command>
    </rci_request>"""

if __name__ == '__main__':

    msg = msg_get

    conn = httplib.HTTPConnection("192.168.196.204:80")
    conn.request("POST", "/UE/rci", msg)

    response = conn.getresponse()
    print response.status, response.reason
    data = response.read()
    print data
    conn.close()
```

Sample Output

```
C:\py\dia\work>Python test_rci.py
200 OK
<rci_reply version="1.1"><do_command target="idigi_dia"><channel_get
name="level.alert"
value="False" units="alert" timestamp="Mon Jul 12 11:08:46 2010"></channel_
get></do_command>
<do_command target="idigi_dia"><channel_get name="level.config"
value="[12.000000,11.500000,False,'count.adder_total']"
units="" timestamp="Mon Jul 12 11:08:46 2010"></channel_get></do_command></rci_
reply>
```

See Also

RCI request` : This is geared more to using RCI to read/write values in the Digi hardware and not in DIA.

Look up the RCI handler module in the DIA HTML documentation, or see the doc strings directly in the DIA source file "src\presentations\rci\rci_handler.py". It supports the commands including, but not limited to:

- <channel_dump/>
- <channel_get name="..."/>
- <channel_refresh name="..."/>
- <channel_set name="..." value="..."/>
- <channel_info name="..."/>
- Plus there is a series of logger commands

Subscribing to a channel

One of the powerful advanced features within the DIA platform is the ability to have data automatically pushed from one device driver to another. This is called publish/subscribe.

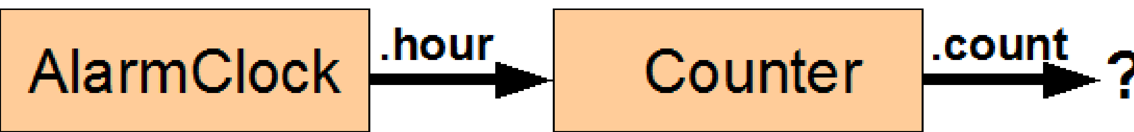
This article is intended for programmers writing their own device drivers - the devices described are NOT part of DIA.

Counting hours - simple example

Hypothetical YML

In this example we'll create two devices.

- The first is a thread-based **AlarmClockDevice** which wakes up periodically to do things. It has an output (meaning something which can be 'GET') named **hour** which pulses once per hour. The DIA Device Alarm device sample code is here: DIA Device - alarm clock.
- The second device is a passive object without any thread. Instead it subscribes to the AlarmClockDevice's once per hour pulse, and catches the act of publishing to do something once per hour - in this case, it is a counter which counts the input pulse, so indirectly counts the hours.



Here is a fragment of the YML file used for this.

```
devices:
  - name: tick_tock
    driver: devices.alarm_clock_device:AlarmClockDevice
  - name: count_hours
    driver: devices.blocks.counter_device:CounterBlock
    settings:
        input_source: tick_tock.hour
```

The most important line is the **input_source: tick_tock.hour**, which describes a setting named **input_source** within the counter object which takes a string value. **tick_tock** is the name of another device, which offers a channel named hour which can be obtained.

Publisher/Source Device Code

The AlarmClockDevice defines these GETTABLE values in the def __init__() function.

```
    ## Channel Properties Definition:
    property_list = [
            # gettable properties

            ChannelSourceDeviceProperty(name='minute', type=tuple,
                initial=Sample(timestamp=0, value=(0,None)),
                perms_mask=DPROP_PERM_GET),
```

```
        ChannelSourceDeviceProperty(name='hour', type=tuple,
            initial=Sample(timestamp=0, value=(0,None)),
            perms_mask=DPROP_PERM_GET),

        ChannelSourceDeviceProperty(name='day', type=tuple,
            initial=Sample(timestamp=0, value=(0,None)),
            perms_mask=DPROP_PERM_GET),
]
```

Then later in the thread.run() section of the code there is a fragment which sets the hour property when appropriate. In this design it is always set to True - it is never set False because it is the act of setting which triggers the desired reaction based on the publish/subscribe.

```
if( self.time_tup[TM_MIN] == 0):
  # publish hourly pulse only first time we see min=zero
  if( not self.push_hr):
    self.push_hr = True
    # now_tuple hold the time and time_tuple info
    self.property_set(PROP_HOUR, Sample(self.time_now, now_tuple ))
  else:
    self.push_hr = False
```

That is it for the source of the data - it merely sets a new sample named hour once per hour, and has no understanding of the external events this drives.

Subscriber/Consumer Device Code

The CounterBlock code is more involved. First we need some properties/settings in the def __init__() function.

- The first is a string setting called **input_source**, which defaults to None meaning there is no publish/subscribe link desired.
- The second is a gettable/settable property called input which includes a callback function to handle any sets - which will also handle the sample push as the source device publishes the new sample.

```
    ## Settings Table Definition:
    settings_list = [
        Setting(name='input_source', type=str, required=False,
default_value=None),
    ]

    ## Channel Properties Definition:
    property_list = [

        ChannelSourceDeviceProperty(name="input", type=tuple,
            initial=Sample(timestamp=0, value=(0,None)),
            perms_mask=DPROP_PERM_SET | DPROP_PERM_GET,
            set_cb=self.prop_set_input),
        ]
```

Now we need to actually create the link when the CounterBlock starts (in the *def start(self) routine*). It fetches the current setting named **input_source**, which should be either **None** or a string like **tick_ tock.hour** (or device.property).

Warning: *make sure you put this near the END of the self.start() routine as it could cause a premature callback if the channel is already active. Or said another way, make sure everything in your callback is initialized and ready to go before subscribing to the channel.*

```
    # wire up any inputs
    cm = self.__core.get_service("channel_manager")
    cp = cm.channel_publisher_get()
    cdb = cm.channel_database_get()

    try:
        source_name = SettingsBase.get_setting(self, 'input_source')
        if( source_name != None):
            source_chan = cdb.channel_get( source_name)
            # pre-load the starting value, which won't be published to
us
            self.my_input = source_chan.get().value
            cp.subscribe( source_name, self.prop_set_input )

    except:
        traceback.print_exc()
        self.my_input = True
```

Finally we need to create the actual callback to handle either a direct **set**, or the **publish set**. The real meat of the passive action is in the **self.process(sam.value)** call. In this example it merely counts the hours.

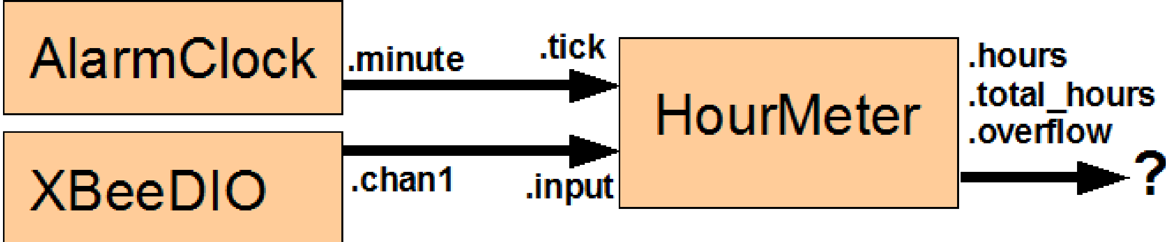
```
    def prop_set_input( self, sam):
        # someone pushing in new input - is either sample or channel

        if( not isinstance( sam, Sample)):
            # the publish/sub pushes in the channel, so convert to
sample
            sam = sam.get()

        self.process( sam.value)
        self.property_set("input", Sample(sam.timestamp, self.my_
input))
        return True
```

Counting Motor Run Hours

The first example was simple, but such a DIA-based uptime counter is not very practical.

So we can expand the example with real-world I/O to create a motor-run timer.

- **motor_01** is a **Digi XBee DIO Adapter** which reads the motor's run contact on its first input (motor_01.channel1_input).
- **motor_01_hours** is an HourMeterBlock, which is a special sub-class of the CounterBlock object which counts minutes that the input is true or false - not the number of times it is set. The motor_01_hours.hours output would show the resettable run total in hours (a float), while the motor_01_hours.total_hours output would show the non-resettable run total in hours (a float).
- Since a reset total of 250 is included, the motor_01_hours.overflow output (a bool) could be used to trigger some action like sending an email. Since motor_01_hours auto resets, motor_ 01_hours.hours would automatically drop back to zero after 250 hours where seen.

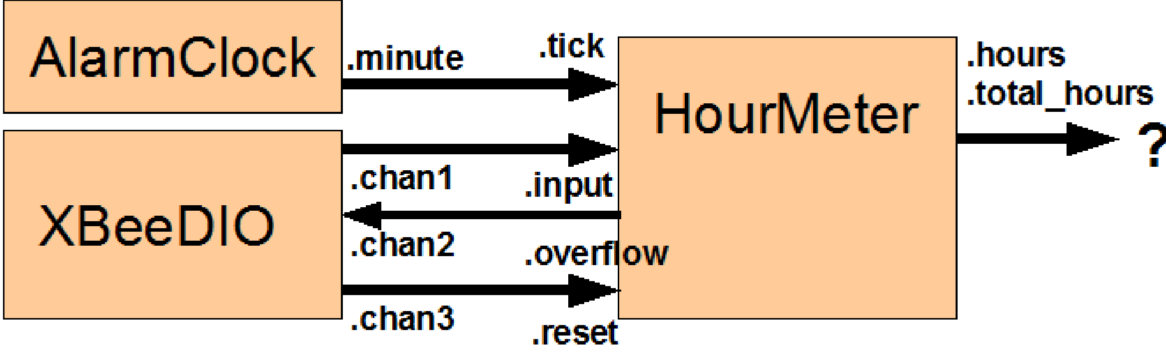



```
devices:
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
  - name: motor_01
    driver: devices.xbee.xbee_devices.xbee_dio:XBeeDIO
    settings:
       xbee_device_manager: xbee_device_manager
       extended_address: "00:13:a2:00:40:0a:49:7a!"
       sample_rate_ms: 1000
       channel1_dir: "in"
  - name: tick_tock
    driver: devices.alarm_clock_device:AlarmClockDevice
  - name: motor_01_hours
    driver: devices.blocks.counter_device:HourMeterBlock
    settings:
      auto_reset: 250
      input_source: motor_01.channel1_input
      tick_source: tick_tock.minute
```

Counting motor run hours with lamp

This example can be expanded even further:

- **motor_01** is a **Digi XBee DIO Adapter** with the follwoing config:
  - **motor_01.channel1_input** reads the motor run contact. It is used as the motor_01_hours.input, enabling totalizing running hours in minute increments.
  - **motor_01.channel2_source** sends **motor_01_hours.overflow** to a red lamp the maintenance people can see
  - **motor_01.channel3_input** reads a push button the maintenance people can push, which resets the motor_01_hours block and turns the red lamp off.
  - **motor_01_hours** is an HourMeterBlock, which as before counts the hours the motor runs using minute ticks provided by the AlarmClockDevice.
- Since a reset total of 250 is included, the **motor_01_hours.overflow** output (a bool) can also be used to trigger some action like sending an email. In this design motor_01_hours doesn't auto resets, so **motor_01_hours.hours** keeps counting above 250 hours, and only resets when the user pushes the button on **motor_01.channel3_input**.




```
  devices:
    - name: xbee_device_manager
      driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
    - name: motor_01
      driver: devices.xbee.xbee_devices.xbee_dio:XBeeDIO
      settings:
          xbee_device_manager: xbee_device_manager
          extended_address: "00:13:a2:00:40:0a:49:7a!"
          sample_rate_ms: 1000
          channel1_dir: "in"
          channel2_dir: "out"
          channel3_dir: "in"
          channel2_source: motor_01_hours:overflow
    - name: tick_tock
      driver: devices.alarm_clock_device:AlarmClockDevice
    - name: motor_01_hours
      driver: devices.blocks.counter_device:HourMeterBlock
      settings:
          manual_reset: 250
          input_source: motor_01.channel1_input
          reset_source: motor_01.channel3_input
          tick_source: tick_tock.minute
```

Twitter DIA example

Overview

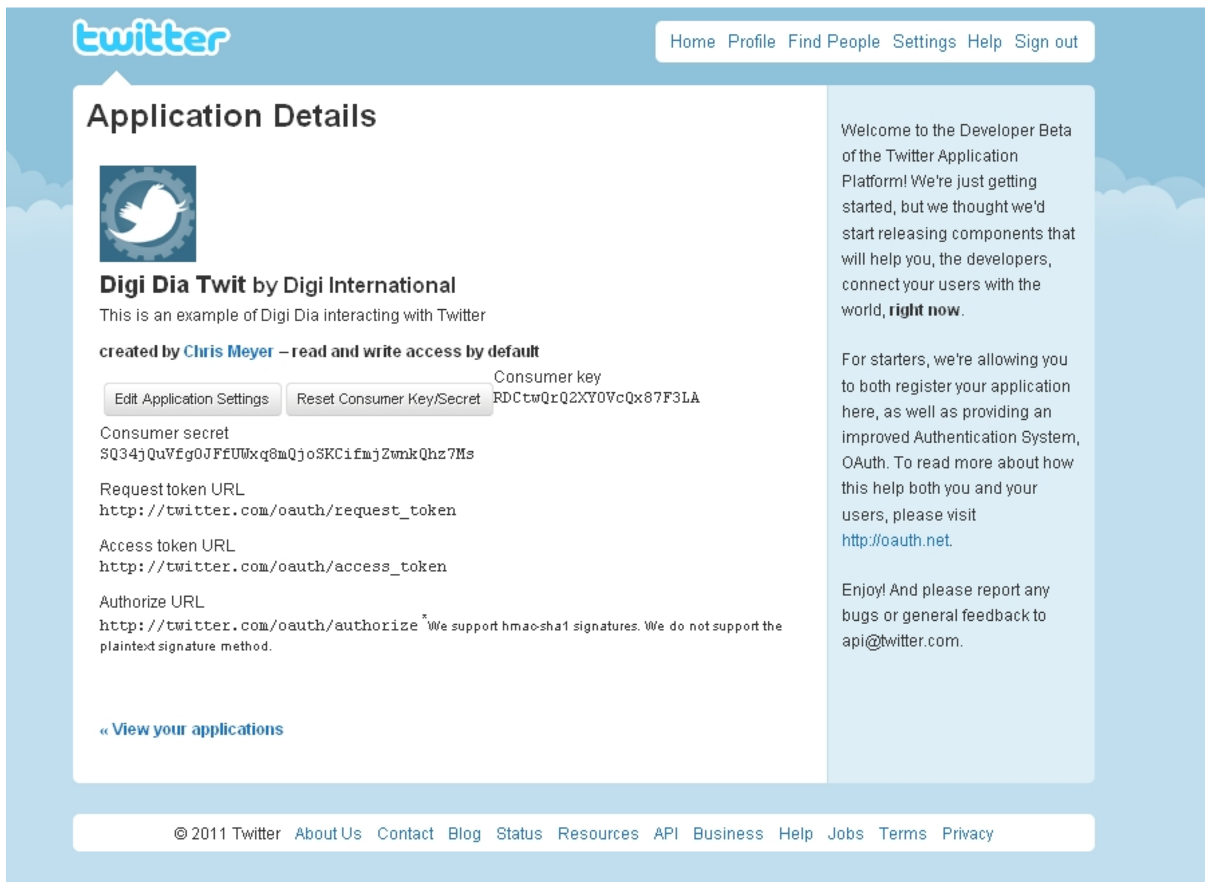
This simple DIA example interacts with the Twitter - API.

Prerequisites

You should register at www.etherios.com/devicecloud and download DIA. After that, read the Getting Started guide and begin looking at the simple drivers (found under ../src/devices in the DIA source code tree). You should have a basic working knowledge of DIA(how to start / stop & use basic drivers / presentations) before proceeding.

This demo requires the tweepy Python library. Tweepy can be found here: https://github.com/joshthecoder/tweepy.

Register a new twitter application by going to http://twitter.com/apps/new. When you submit the form, you will be presented with a page that shows your consumer key and consumer secret. Take note of these.



Download the following archive Media:Twitter_Dia.zip, unzipping its contents within the src/presentations directory of the DIA. Add the following to your yml config file (by default this will be the *cfg/dia.yml*) and change the settings accordingly, or remove the optional settings.

```
  presentations:
    - name: twitter
      driver: presentations.twitter.updater:Updater
      settings:
         interval: 60
         status_template: "uploaded from the iDigi Dia: %s"
         sample_template: "%s=%s"
         channels: [template.counter, template.adder_reg1]
```

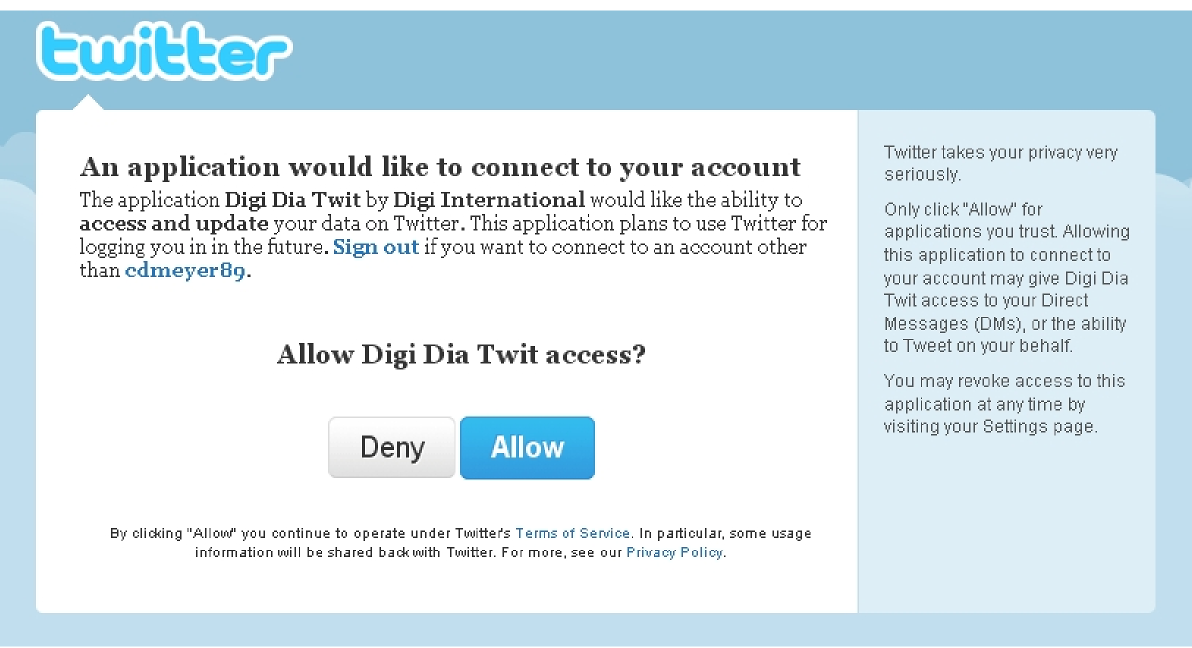
Here is a breakdown of the different options:

```
interval optional
  How often to "tweet". If this setting does not exist it will default to 180
seconds
status_template optional
  Format of the tweet, with a %s where to put the sample data.  If left blank
this will default to "iDigi Dia Update: %s"
sample_template optional
  Format of the samples, two %ss required, first for the channel name, second for
sample value.  defaults to "%s:%s"
channels optional
  List of channels the module is subscribed to. defaults to all channels
```

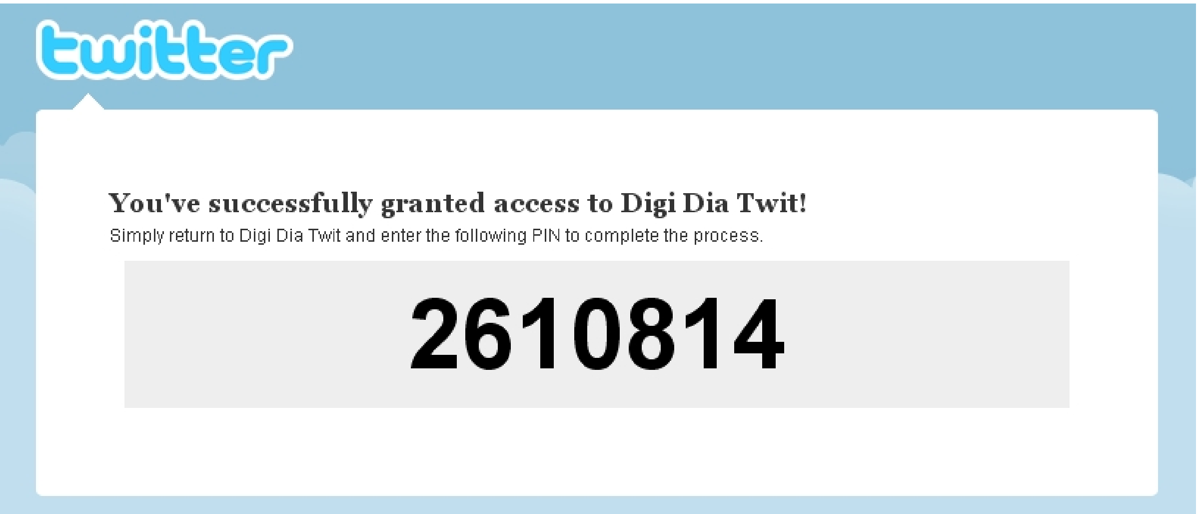
Re-compile DIA incorporating the above elements under presentations of the cfg/dia.yml file. You can do this via the command line (Python make.py dia.yml) or using the Digi IDE for DIA.

Run the DIA When it's running, the first thing you will be asked for is your consumer key. Enter the consumer key that was shown when you registered the twitter app. You will then be asked to enter your consumer secret. Do the same as before.

Next, you will see a prompt that says P*lease verify URL*: followed by a long URL. Type that URL into your browser and search for it. You will see a twitter page that asks whether you want to allow the application to connect to your twitter account. Click **Allow**.



You will then be presented with a PIN. This is the PIN you must enter into the command prompt.



Once you enter the PIN, DIA will begin tweeting to your account!

Hint

```
  If DIA fails with an error message "No module named gopherlib" followed by
"global name 'twitter' is not defined"
  The reason could be that the module gopherlib is excluded in the DIA build
process.
  Please adjust the file tools\digi_build_zip.py and comment out the gopherlib
line like:

  # Other things that can get in the way, and don't seem likely
```

```
    #"gopherlib",
    #"ftplib","pydoc",
```

Understanding XBee EndPoints

The way Digi implimented the XBee/ZigBee end-points in the XBee extensions to the Python socket API implies that they function like TCP or UDP source and destination port numbers. However unlike TCP or UDP which treats each src/dst port pair as unique conversations, XBee/ZigBee assigns each distinct 'destination end-point' to a single application (or handler) for message delivery. The handler is free to manually assign a meaning to the 'source end-point' in received packets, but it is NOT likely unique.

For example, a Modbus/TCP server can bind on TCP port 502. When five clients connect, each uses a destination port of 502 and a unique 'source port' (per IP address pair). Five 'sockets' may be created, and each of the five server-tasks spawned is blissfully unaware that other clients are active.

In contrast, an XBee server can bind on an end-point such as 0xE8, but five remote devices sending to destination end-point 0xE8 will appear to use the same 'socket'. Additionally, they all may claim a source end-point of 0xE8. The Xbee server code must manually process and distribute messages based on extended address or other criteria.

Which XBee technologies support end-points?

You can use the upper word of the DD setting to distinguish XBee which support end-points and those which do not.

Upper DD Word	Name	End-Point Support	Description
0x0000	Unspecified	Unknown	This means the XBee is mis-configured
0x0001	802.15.4	No	Non-mesh star/peer configured nodes on 2.4GHz
0x0002	ZNet 2.5	Yes	Digi's older pre-ZigBee firmware
0x0003	Digi Zigbee	Yes	Digi's firmware supporting Zigbee 2007
0x0004	Digi Mesh 900	Yes	Digi's proprietary mesh on 900Mhz
0x0005	Digi Mesh 2.4	Yes	Digi's proprietary mesh on 2.4Ghz
0x0006	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 868Mhz
0x0007	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 900Mhz

Notes:

- Use the DD value in the Digi gateway's XBee - you cannot trust that the remote XBee has a valid DD setting!
- Even if the XBee does NOT support end-points, Python may allow you to bind on a specific end-point like 0xE8. However, you will never receive any responses on 0xE8 because all responses appear to be targeted at end-point 0x00.

Managing end-points in XBee

Under XBee AT-transparent mode

So a concrete example, sending a packet with (src=0xE9 dst=0xE8) will NOT automatically result in return-responses being received as (src=0xE8 dst=0xE9) - you need to explictely change the AT settings in the remote XBee to return serial data to destination end-point 0xE9.

Therefore, to move your Python script bind end-point away from 0xE8 (the default), then do the following:

- Your Python script binds on an alterative end-point such as 0xE9, 0x01, or 0x41.
- Manually set all required remote XBee with an AT setting DE=0xE9 (or as required). You can do this by:
  - Under XCTU, relevant Xbee firmwares (such as ZigBee Router AT 0x2264) will have a closed/collapsed sub-option named "ZigBee Addressing" under Adressing. Click on it to open and you'll see the DE/SE/CI options.
  - Newer 2.9.x Digi gateway firmwares expose the DE command under the advanced settings for remote XBee nodes.
  - Use standard "remote AT command" API frames or ddo_set_param() calls to force DE to the desired value.

Note that changing the XBee's SE setting does NOT allow moving serial encapsulation away from 0xE8 - it only changes the source end-point cliamed by AT-Transparent responses. The remote XBee in AT-transparent mode will only forward data out the serial port which is targetted at 0xE8.

Under XBee API mode

When running XBee firmware in API mode, then most source/destination end-points are passed through transparently. It is up to the serial-attached CPU to parse and assign meaning to the end-points. Therefore, an external CPU might gracefully and automatically return responses to requests with (src=0xE9 dst=0xE8) as (src=0xE8 dst=0xE9).

End-point numbers to avoid

Although no exact list is available, you should avoid using these end-points:

End-Point	Description
0x00	ZigBee Device Object end-point - reserved by ZigBee stack
0xDC to 0xEE	Reserved for Digi use
0xEF to 0xF0	Reserved for other vendor use
0xF1 to 0xFE	Reserved by ZigBee Alliance / for other uses
0xFF	Digi treats as 'any end-point' (a wild-card) during Python socket operations

End-points 0x01 to 0xDB should be free to use. However, the rules for picking end-point numbers is much like that for TCP/UDP port numbers - be flexible and ready to change if required. For example, you might use TCP port 2101 to tunnel data to a Digi device server. It works, but TCP port 2101 is officially assigned for use as "RTCM-SC104", which is a Radio Technical Commission for Maritime

Services standard used to move GPS data via TCP/IP. Microsoft also overloads this port for use with RPC-based MQIS and Active-Directory Lookups. The same may be true of XBee end-points - even if you reuse a preassigned end-point it is only a problem if you require some other tool or function with a conflicting use.

The examples above used 0xE9, which Digi claims is reserved, so might be used for some other purpose in the future. If you select end-point 0x77, some other vendor might create a product requiring a Python script binding on end-point 0x77. So design your system to allow the end-point number selected to be easily changed as required.

EndPoints in the Modbus/IA Engine

You can use the Digi Modbus/IA engine to route Ethernet-based Modbus/TCP requests to remote Modbus/RTU serial devices via XBee RS-232 adapter and XBee RS-485 adapter. Configuring the XBee module is covered in Modbus Example Serial Adapter.

However, the default Modbus/IA behavior is to use ZigBee endpoint 0xe8, which will conflict with most sample Python programs and the Device Cloud/DIA framework. This conflict is because only 1 task can bind on (or register to receive) incoming XBee packets on endpoint 0xe8.

Therefore you should move the Modbus/IA traffic away from endpoint 0xe8 and use another, such as 0xe9. This is easier than trying to change the behavior of a Python application in an unknown number of places:

- In the remote XBee adapter, set DE as explained above to E9 (or to your desired value)
- In the IA Modbus unit id/slave address mapping table, add the new Xbee endpoint after the '!', so MAC looks like as 00:13:a2:00:40:30:de:cd!E9. Note that this does NOT change the 'destination' endpoint in the outgoing Modbus request - this remains 0xe8 for XBee module reasons. Instead, it causes the Modbus/IA engine to bind on incoming endpoint 0xE9 instead of 0xE8.

XBee Analog I/O DIA Example

Below is an example configuration file that includes a XBee Analog Adapter. NOTE: The extended address field will have to be edited for customer use.

```
devices:
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager

  - name: aio_voltage
    driver: devices.xbee.xbee_devices.xbee_aio:XBeeAIO
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:0a:12:ab!"
        sample_rate_ms: 1000
        power: "On"
        channel1_mode: "TenV"
        channel2_mode: "TenV"
        channel3_mode: "TenV"
        channel4_mode: "TenV"

presentations:
  - name: console0
    driver: presentations.console.console:Console
    settings:
        type: tcp
        port: 4146
```

Adding Sleeping Behavior

Here is the AIO portion of a YML which sleeps, waking every 5 minutes and sending a sample. New settings you'll see here include:

- **sleep** = True/False (default = False)
- **awake_time_ms** = how many msec to remain awake AFTER sending the first sample. Note that the Power output will remain on during this time.
- **sample_predelay** = how many msec to delay after power output is enabled, before the sample is taken. Default is 1000 msec. The value requires depends on the sensor. Some output valid signals instantly, while others require from 2 to 15 seconds for the sensor controller to boot, and/or for it to take several readings before outputing a stable voltage or mA signal.

```
  - name: itank01
    driver: devices.xbee.xbee_devices.xbee_aio:XBeeAIO
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:52:94:D4!"
        sleep: True
        sample_rate_ms: 300000
        awake_time_ms: 5000
        sample_predelay: 25
        power: "On"
        channel1_mode: "TenV"
```

For more information regarding configuration and use of the Analog Adapter with the DIA, look here: Device Cloud Wiki.

Data Tunneling

TCP to Zigbee dynamic name mapping

Introduction:

This sample is a simplistic design to allow TCP traffic to a Digi Gateway product be routed to the Gateway's Mesh network and vice versa using a naming system defined by the user.

Requirements:

- Digi Gateway product
- XBee Endpoint device associated with the Gateway (Control of device via serial recommended)
- TCP access to the Digi Gateway product
- Standard Python.zip

Overview:

The following guide will describe the application in several steps, with the completed application listed at the end. The goal of this application is to demonstrate how to form a mapping between the 64 bit hardware addresses and the user defined name of the node, the enforcement of the naming scheme for the client connection, and the queuing of data for the respective interfaces.

Code walkthrough

Declarations

```
##################################################################################
##################
# Import statements
##################################################################################
##################

import socket
import select
import struct
import zigbee
import errno
import table   #User defined file

##################################################################################
##################
# declarations
##################################################################################
##################

MAX_TCP_PACKET_SIZE = 8192  #Maximum size of tcp packet we will receive or push
at once
MAX_ZIG_PACKET_SIZE = 100   #Maximum size of zigbee packet we will read or send
at once

tcp_port = 20000            #TCP Port the application sends and receives on
quit_port = 30000           #TCP Port if connected to throws a keyboard exception
```

```
end_point  = 0x00           #address information that we will bind and send to
profile_id = 0x0000
cluster_id = 0x00

zig_addr_name_dict = table.table  #The dictionary of zigbee 64 bit hardware
address mapped to names
#This could be generated or typed in by the user

zig_queue = []                    #queue of data we send out the zigbee socket
tcp_queue = []                    #queue of data we send out the TCP socket
```

The first section *Import statements* declares the libraries we intend to use. socket, select, struct, zigbee, and errno are all libraries you should have encountered before. The *table* library is a user defined library intended to be used only in this application. It's purpose is to define the 64 bit hardware address to node name mapping scheme.

The second section 'declarations *declares several constants that are used throughout the application*.

MAX_TCP_PACKET_SIZE

Describes the size of the maximum read and write we will perform on the TCP client socket.

MAX_ZIG_PACKET_SIZE

Describes the size of the maximum read and write we will perform on the Zigbee socket.

tcp_port

Defines which port the client socket will be expected upon.

quit_port

Defines which port the application will terminate on if we see a connection. Note: this is not necessary but helpful for sample purposes.

end_point, profile_id, cluster_id

Declare the address information we will use to bind and send when interacting with the mesh network.

zig_addr_name_dict

The dictionary object that provides the 64 bit hardware address and node name look up. This information is retrieved from the user defined table module.

zig_queue

The list object that acts as a queue for all information that will be sent out the zigbee socket.

tcp_queue

The list object that acts as a queue for all information that will be sent out the TCP client socket.

User defined procedures

```
#################################################################################
##################
# cleanUp - removes the client socket from the read/write lists, closes and sets
to None
#################################################################################
##################

def cleanUp(client_sock):
  try:
    read_list.remove(client_sock)
    write_list.remove(client_sock)
    client_sock.close()
```

```
    client_sock = None
  except Exception, e:
    print e
```

This code defines a procedure that inputs a socket, removes the socket from the read_list and write_list, closes it and sets it to value None. This is necessary for our application because we must be capable of having the client TCP connection close unexpectedly without exiting our application.

Initialization

```
#################################################################################
##################
# Init the dictionary, declare the sockets
#################################################################################
##################

# We provide reversal lookup to this dictionary. So we can go from the name -> 64
bit addr or
# 64 bit addr -> name.

  for item in zig_addr_name_dict.keys():
    zig_addr_name_dict[zig_addr_name_dict[item]] = item
    #Reversal lookup now available!

  # Declare the sockets
  listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  listen_sock.bind(("", tcp_port))
  listen_sock.listen(1)

  zig_sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
  zig_sock.bind(('', end_point, profile_id, cluster_id))
  zig_sock.setblocking(0)

  quit_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  quit_sock.bind(("", quit_port))
  quit_sock.listen(1)

  client_sock = None

  read_list = [listen_sock, zig_sock, quit_sock]
   write_list = [zig_sock]
```

Here we provide a reversal lookup between the address keys and name values in the zig_addr_name_dict. For example previously we could retrieve the name via:

```
  name = zig_addr_name_dict[addr]
```

Now we can retrieve the address via:

```
  addr = zig_addr_name_dict[name]
```

This works provided that the address and name values cannot be the same. This is highly unlikely unless a user defines the name of a node to be the 64 bit address of an existing node.

Also defined here are the applications 3 static sockets. The listen socket which is the one which clients will connect to, the zigbee socket which we will communicate with the Mesh network with and the quit socket, which we use to terminate the application by request.

We also define the client socket as **None**, though we intend to make it a socket object later.

For use in the upcoming select call, we create a list consisting of all the sockets we expect to receive from, and a list of sockets we expect to write to.

Main Loop

```
################################################################################
##################
# Main loop

################################################################################
##################

print "Entering main loop"

while 1:

  rl, wl, el = select.select(read_list, write_list, [])
```

We are now entering the main loop of the application, where we will stay until termination.

We now perform a select call on the sockets we listed in the last step. We now have to define the behavior of each of those sockets if they appear in the appropriate list. The next few steps will cover that topic.

Zigbee socket in the read list

```
################################################################################
##################
# Zigbee present in read list, we have new data

################################################################################
##################

if zig_sock in rl:
  try:
    data, addr = zig_sock.recvfrom(MAX_ZIG_PACKET_SIZE)
    print "Read %d bytes from address: %s" %(len(data), addr)
  except Exception, e:
    print e
  else:
    try:
      name = zig_addr_name_dict[addr[0]] ## Get the name from the dictionary
    except KeyError, e:                  ## If that address doesn't have a name
      print e                            ## print and go no further here
    else:
      tcp_queue.append("%s:%s" %(name, data)) ## Append the 'name:data' format to
the queue
```

If the Zigbee socket is in the read list, it means we have incoming date from the mesh network. We retrieve the data and address of that packet and perform a name lookup on the address. If we can perform that lookup successfully, we append the data to the TCP queue in the 'name:data' format.

Note that the address information is a tuple object structured like (Hardware_address, end_point, profile_id, cluster_id). We only need the hardware address portion to perform the lookup. While performing the name lookup, we can potentially receive a KeyError exception. This is because we could receive data from the mesh network that we don't have a 64 bit hardware address specified for.

Zigbee socket in the write list

```
##################################################################################
##################
# Zigbee present in write list AND we have data to write

##################################################################################
##################


if (zig_sock in wl) and (len(zig_queue) != 0):
  name, data = zig_queue[0].split(":", 1)    ## Retrieve the 'name:data' datoms
  name = name.strip()                        ## Strip excess unprintable
characters
  try:
    addr = zig_addr_name_dict[name]          ## Get the addr dervied from the
name
  except KeyError, e:
    #An node with an undefined address-&gt;Name mapping has contacted us
    print e

  else:
    if len(data) &gt; MAX_ZIG_PACKET_SIZE:
      segment = len(data)
    else:
      segment = MAX_ZIG_PACKET_SIZE

    try:
      sent_data = zig_sock.sendto(data[:segment], 0, (addr, end_point, profile_
id, cluster_id))
      data = data[sent_data:]
    if len(data) == 0:                     ## If all data has been sent, pop it
      zig_queue.pop(0)
    else:                                  ## Otherwise store the remaining data
      zig_queue[0] = "%s:%s" %(name, data)
    except Exception, e:
      print e
```

If the zigbee socket is present in the write list and we have data to send. The latter part of the condition is critical to avoid errant sends. All the data in the zig_queue is structured in the 'name:data' format. This allows us to make a few assumptions with our code. We copy the name and data from the first element of the queue, making sure to split only on the first ':' to avoid splitting the data incorrectly.

We use the name to retrieve the hardware address of the node we will send to. We then send the maximum amount of data we can and use the return value of the sendto(...) function to determine how much was actually sent. If we determine that all the data was sent, we pop the first element out of the queue, otherwise we return the remaining data to the queue in the 'name:data' format.

Note that we never send the name portion of the data, yet we must keep it in case we do not completely send, so when the next time the select call returns the zigbee socket in the write list, we can perform the same operation on it without making adjustments that we have processed it once before. In short, we want to keep the data in a single state until we have finished it.

Listen socket in read list.

```
##################################################################################
##################
# listening socket in read list, means we have a client!
```

```
####################################################################################
##################

if listen_sock in rl:
  client_sock, addr = listen_sock.accept()
  client_sock.settimeout(0)                ## disable blocking
  tcp_queue = []                           ## Remove all current data in queues
  zig_queue = []
  read_list.append(client_sock)            ## put the socket into a list so the
select will cover
  write_list.append(client_sock)           ## it when we next get back there.
```

The listen socket we defined is now in the read list. We accept the socket, set the new socket to non-blocking IO, clear our data queues to avoid old data, and append the client socket to the read and write list the select call uses.

The cleanUp function defined earlier will only be called on the client socket. It's intended to be used when the client connection errors out or disconnects.

Client Socket in the read list

```
####################################################################################
##################
# client socket in read list, we have data

####################################################################################
##################

if client_sock in rl:

  try:
    data = client_sock.recv(MAX_TCP_PACKET_SIZE)


  except socket.error, e:           #We have a socket exception
    if (e.args[0] == errno.EAGAIN): #If it's a blocking related exception
      pass                          #come back again next select call
    else:                           #If it's NOT a block related exception
      print e                       #Clean it up
      cleanUp(client_sock)
    except Exception, e:
      print e
      cleanUp(client_sock)
    else:
      print "Read %s bytes from client" %len(data)
      if len(data) == 0:              ## If 0 bytes read, we clean up the
connection
        cleanUp(client_sock)

      pack =  data.split(":", 1)      ## split it by the first ':'
      if len(pack) != 2:              ## If the split item doesn't have 2 parts
        tcp_queue.append("ERROR: Invalid format, must follow 'NAME:DATA' format")
      else:                           ## Send back a message saying they didn't
follow the format
        name = pack[0].strip()
        tail = pack[1]

        ## If we don't have an address associated with the key
```

```
        ## Send back a mesage saying it's unknown
        ## or if the item does not have anything after the ':'
        ## send back a message saying you sent no data

        ## if passing the above, queue up the message


        if zig_addr_name_dict.has_key(name) == False:
          tcp_queue.append("ERROR: %s is an unknown name" %name)
        elif len(tail) == 0:
          tcp_queue.append("ERROR: Cannot send messages of 0 bytes in length")
        else:
          zig_queue.append(data)
```

We now process any incoming data from the TCP client socket. In this situation it means we have incoming data. While receiving the data, we watch for the EAGAIN socket exception. This is an indicator that this operation would block, and we'll come back next select call. All other exceptions on the socket will cause us to disconnect the client_socket by call ing the cleanUp(...) function on it. We will also call the cleanUp(...) function if we receive 0 bytes in data.

Next we split the incoming data once by ':' to determine if it's in the proper format. If that is not the case, we queue up a response to the client, stating the error and the correct format to use.

We then check to see if the 'name' portion of the data maps to a hardware address we know about. If it doesn't we queue up a response to the client, stating the error.

If the data passes all the criteria, we can then append it to the zigbee queue.

Client socket in the write list

```
#################################################################################
##################
# client socket in write list AND we have data to write

#################################################################################
##################

if (client_sock in wl) and (len(tcp_queue) != 0):
  name = tcp_queue[0].split(":", 1)[0]  ## Retrieve the name from the item
  data = tcp_queue[0]                   ## Make a copy of the complete item

  if len(data) &gt; MAX_TCP_PACKET_SIZE:
    segment = len(data)
  else:
    segment = MAX_TCP_PACKET_SIZE

  try:
    sent_data = client_sock.send(data[:segment])
  except socket.error, e:           #We have a socket exception
    if (e.args[0] == errno.EAGAIN): #If it's a blocking related exception
      pass                          #come back again next select call
    else:                           #If it's NOT a block related exception
      print e                       #Clean it up
      cleanUp(client_sock)
  except Exception, e:
    print e
    cleanUp(client_sock)
  else:
    data = data[sent_data:]         ## Make the copy shorter by the amount we
sent
```

```
    if len(data) == 0:              ## If that's all the data, pop the item from
the queue
      tcp_queue.pop(0)
    elif sent_data &lt; len(name) + 1: ## If we only sent enough to cover the
'name:' portion
      pass                          ## Don't store the changes, we didn't do
anything
    else:
      tcp_queue[0] = "%s:%s" %(name, data)  ## Otherwise store the remaining
chunk.
```

If the client socket is in a writable state and we have data to write, we attempt to send the data. Note that when we send the data we keep the 'name' portion of the data. This is to help identify who the response is coming from, as it is possible that the client code have many outstanding requests at once. With that said, this may not be the response they are looking for with that name.

While performing the send, if we have the EAGAIN exception, we come back next select call. Any other exception will cause us to assume the client socket is now bad and call the cleanUp(...) function on it.

A key part in us sending this data is that when the sent the data, we sent more than the 'name' + ':'. If we didn't, we must preserve the structure of the data, and keep the data in the same structure it was before the select call. If we sent all the data, we pop the element out of the queue. If we sent more then 'name' + ':', we return the remaining in the 'name:data' format.

Quit socket in read list

```
#################################################################################
##################
# quit socket - convience to quit from the app

#################################################################################
##################

if quit_sock in rl:
  raise KeyboardInterrupt("Quitting the application")
```

If the quit socket is in the read list, we throw a KeyboardInterrupt and cause us to exit the main loop.

The address and name table in table.py

```
table = {
     "[00:13:a2:00:40:01:4c:e1]!": "Node_1",
     "[00:13:a2:00:40:01:51:a3]!": "Node_2",
     "[00:13:a2:00:40:01:e9:20]!": "Node_3"
}
```

The table that we use to provide the address to name lookup. We keep it separate from the rest of the application because we may want to have it generated. The only limitation is that the address must be a valid 64 bit address in the above format, and the name of the node must be smaller than MAX_TCP_PACKET_SIZE - 1. This mapping also has the potential to send broadcast transmissions by having a user input an entry: "[00:00:00:00:00:00:FF:FF]!": "Broadcast".

Known limitations and notes

The code above has a few known limitations. The first being we do not keep the data in the same format when it arrives from the client all the way to the mesh nodes. This is because of the limitation

of the mesh network's maximum transport size. In 802.15.4, the maximum packet size is 100 bytes, in Znet2.5, it's 72 bytes. ZB architecture depends on the options, but is generally in the same range.

If we were to maintain the data in the same format throughout the application, we could run into a issue where packets being sent to mesh nodes that are mapped to a large name would suffer throughput. In addition, the mesh nodes should be able to assume that all packets received are intended for them. Broadcast transmissions (which could be defined in the name table) could also be taken into account because the addressing scheme of the packets would indicate that it was a broadcast being sent rather then a directed packet.

In the case of the client sending multiple requests to the same node, it is possible the responses to those requests would return in a different order then when we sent them. A potential solution would be to add a sequence number as part of the data to indicate the response, however both the TCP client and the mesh nodes would have to respect that structure, and that is beyond the capabilities of this application.

The source is located here: Tcp_zig_dynamic_mapping.zip

TCP to Zigbee port binding

Introduction

This page is to document a demonstration application that operates on a Digi Gateway product to tunnel data between a TCP network and a XBee network.

WARNING This script is **deprecated!** for an up to date script, use xbee_transport.py

Requirements

- A Python enabled Digi Gateway
- A XBee device associated with the Gateway (Preferably with a method to monitor traffic sent to it)
- CLI access to the Digi product

Overview

The application works to tunnel data from the TCP network and the XBee network by defining a TCP port XBee address pair. The TCP port is bound to the Gateway any any client connection will have its data forwarded to the XBee address specified. Correspondingly, the data received by the Gateway from that XBee address will be forwarded to the TCP client connection.

To help define the TCP port and XBee addresses is a helper script called 'table_generator.py'. After uploading the script to the Gateway, run it. This performs a discovery on the XBee network, identifies the unique addresses, and outputs a TCP port to XBee address mapping to a configuration file. By default the TCP port starts at port 4000, and increments by one per additional XBee address found. The output file is placed into the WEB/Python directory named as 'bind_table.py'. To edit it, download it from the Gateway and manually change the parts desired.

The main application 'tcp_zig_binding.py' takes one parameter optionally. The parameter represents the amount of time to wait before sending a message received from the XBee network to the corresponding TCP port. The usefulness of this is to potentially send fewer messages over the TCP network, or to group related messages into a single TCP packet. The parameter represents the amount of time in seconds, for example, .2 represents 200 ms delay. By default the application has a delay of .3 ms. Use a value of 0 to disable.

Code

This is the code from the main application 'tcp_zig_binding.py'.

```
"""
This script is intended to act as a data tunneling application between the TCP
network and zigbee network.
It assigns multiple TCP sockets to listen on specific ports on the gateway
device, this is defined in
bind_table.py, which can be hand made or generated. Each of the TCP sockets
corresponds to a specific Zigbee
address on the zigbee network.  This allows an application to send data to port
4000 and always have it be in
turn sent to the same zigbee address.
```

```
There are several limitations and liabilities.  First the Series 2 radios have a
maximum payload size of 72
bytes of data, with no internal ability to understand fragmented data.  The
Series 1 radios have a maximum payload
of 100 bytes, again with no internal ability to understand fragmented data.
While this application can perform
some packet reassmebly on this end, the zigbee endpoint would not be able to do
so, and whatever device or
application attached to said zigbee device will have to reassemble the data as
needed.

"""

import socket
import select
import bind_table
import sys
import zigbee
import errno
import struct
import traceback
from time import clock

tcp_conn_dict = {}
tcp_port_dict = {}
tcp_data_dict = {}  ##Data associated with tcp port

zig_port_dict = bind_table.node_list  ##Where we obtain the zigbee to address
lookup
zig_data_queue = []

listen_list = []
client_list = []

MAX_ZIGBEE_PACKET_SIZE = 100
MAX_TCP_PACKET_SIZE    = 8192
segment    = 0
end_point  = 0
profile_id = 0
cluster_id = 0

"""
This value should be changed to suite the data tunneling needs.  Factors such as
the remote device's baud rate,
packetization time, and size of data all come into play, test and retest as
needed.
"""

TCP_BUFFERING_TIME = .300 #300 milliseconds is out timeout period

#################################################################################
##################
# Parse Args
#################################################################################
##################

#We only have one argument, the TCP_BUFFERING_TIME
try:
```

```
  if len(sys.argv) > 1:
    TCP_BUFFERING_TIME = float(sys.argv[1])
except Exception, e:
  print get_exception_info()
  print "TCP_BUFFERING_TIME requires a floating number, it's default is .3 (300
ms)"
  sys.exit(0)

################################################################################
##################
# Detect our radio type and assign address information accordingly
################################################################################
##################

"""
Due to the differences between series 1 and series 2 radios, we much be able to
determine which type of radio
is currently onboard our gateway, and set the end_point, profile_id, and cluster_
id appropriately.  To do this,
we use the ddo_get_param(local_radio, 'HV'), which is the AT command to retrieve
the hardware version.

Once we have the hardware version, we have to unpack the binary string into a
tuple, and take the first
(and only) element from that tuple, and assign it to hw_version.  We test that
number returned, if it's
over 6400, it must be a Series 2 radio, under 6400, series 1.
"""

try:
  hw_version = zigbee.ddo_get_param(None, "HV")
  hw_version = struct.unpack("=H", hw_version)[0]
except Exception, e:
  hw_version = None
  print get_exception_info()
  print "Failed to retrieve hardware version from local radio"
  print "Assuming it's a series 1 device"

if hw_version != None:
  if hw_version > 6400: #If the hardware version is greater then 6400, it must be
a series 2 radio
    print "Detected Series 2 radio in gateway, configuring zigbee socket
appropriately"
    end_point  = 0xE8 #232
    profile_id = 0xC105
    cluster_id = 0x11 #Out the UART
    MAX_ZIGBEE_PACKET_SIZE = 72   #Packet sizes are at maximum 72 bytes

  else:
    print "Detected Series 1 radio in gateway, configuring zigbee socket
appropriately"

################################################################################
##################
#  cleanUpConn()
################################################################################
##################
def cleanUpConn(conn):
  try:
```

```
    client_list.remove(conn)   #We remove it from our listening list
    sock = tcp_conn_dict[conn] #We locate the associated listener
    tcp_conn_dict[sock] = None #We set the listener's lookup to None
    tcp_data_dict[sock] = []   #We reset the data queue
    del tcp_conn_dict[conn]    #We delete the key value pair for the reverse
lookup
    conn.close()               #We close the instance
    conn = None                #We set the instance to None
  except Exception, e:
    print get_exception_info()

def _format_exception_info(max_tb_level=20):

    e_type, e_value, e_traceback = sys.exc_info()
    e_name = e_type.__name__
    e_args = []

    try:
        for item in e_value.args:
          e_args.append(str(item))
    except AttributeError:
        pass

    try:
        for item in e_value.message:
          e_args.append(str(item))
    except AttributeError:
        pass

    tb_string = traceback.format_tb(e_traceback, max_tb_level)
    return (str(e_name), e_args, str(tb_string))

def get_exception_info():
    """Formats the last raised exception, and returns a string that
       contains the name of the exception, any info associated with
       the exception, and a call trace for the exception."""

    e_name, e_args, e_tb = _format_exception_info()
    e_string = "".join(["Exception ", e_name, ":\n",
                        "".join(e_args), "\n",
                        "Traceback:\n", "".join(e_tb)])

    return e_string

#################################################################################
##################
# Init
#
#################################################################################
#################

"""
We loop through the current zigbee address to tcp port dictionary, create a
socket per entry, bind
the socket to the specified tcp port, and map the various dictionaries to provide
fast lookups
"""

##Note we are doing keys(), so we only generate the list once at the start,
```

```
otherwise
##It will change due to our assignment of port -> address lookups

for addr, port in zig_port_dict.items():

    ##Create socket, bind to localhost:port, set it to listen and append it to
the listening socket list
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(("", port))
    sock.listen(1)
    listen_list.append(sock)

    tcp_conn_dict[sock] = None  ##Init the listener to connection dictionary

    tcp_port_dict[sock] = port  ##Assign listener to port lookup
    tcp_port_dict[port] = sock  ##Assign port to listener lookup

    tcp_data_dict[sock] = []    ##Assign listener to tcp_data_queue lookup

    zig_port_dict[port] = addr  ##Assign port to zigbee_address lookup
                                ##The lookup between address and port already
exists by this point

#end for key in zig_port_dict

## Creating the zigbee socket, binding it to a local endpoint, set it to not
block
zig_sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
zig_sock.bind(('', end_point, profile_id, cluster_id))
zig_sock.setblocking(0)

################################################################################
##################
# Mainloop
################################################################################
##################
print "starting mainloop"
while True:

  """
  We listen for reads on the listening sockets for new connections the client
sockets for new data
  from the tcp side and the zigbee socket for incoming zigbee data

  We listen for write ability on the client sockets so we can write the received
and queued zigbee
  data on them, and we listen for write ability on the zigbee socket so we can
write the received
  and queued tcp data on it.
  """

rl, wl, el = select.select(listen_list + client_list + [zig_sock],  #Reading
sockets
                           client_list + [zig_sock],                #Writing
sockets
                           [],
                           .5)                                      #Not used,
(error lists, see man select)
```

```
####################################################################################
##################
# Handle zigbee reading
####################################################################################
##################

if zig_sock in rl:
  data, addr = zig_sock.recvfrom(MAX_ZIGBEE_PACKET_SIZE)
  print "Zigbee read: %d bytes from address " %(len(data)), addr

  if len(data) != 0:                 #if we've received data
    try:
      port = zig_port_dict[addr[0]]    #We find the matching tcp port
      sock = tcp_port_dict[port]       #we find the socket matching our port

      if len(tcp_data_dict[sock]) != 0:  #If the queue isn't empty
        (old_data, packet_time) = tcp_data_dict[sock][-1]  #Copy the last packet
off

        if clock() - packet_time >= TCP_BUFFERING_TIME:  #Check the time of the
last packet

          #If the difference in time is greater then our buffer time
          #We should just append the data to the queue
          tcp_data_dict[sock].append((data, clock()))
      else:
        #The difference in time ISN'T greater then our buffer time
        #We should append our new data to the old data

        lump_data = old_data + data
        tcp_data_dict[sock][-1] = (lump_data, packet_time)

        # We have to be careful of how we append the data, and to assign the
older
        # time back into the tuple.  Otherwise we may continue to receive data
        # append it out of order, and never send it anyways because the buffer
time
        # may keep reseting

      else:
        tcp_data_dict[sock].append((data, clock()))

    # We can have a valid key error here, in this case, if we receive a zigbee
packet from an
    # Address that is NOT in the bind table, we will throw a key error.  An
alternative is
    # Receiving a packet from the local zigbee radio, which will have the 0000
address, something
    # That is not normally in the bind table if automatically generated.

    except KeyError, e:
      pass

    except Exception, e:
      print "We received an unknown error, ", e
      print "Please report this!"
      sys.exit(0)

#end if zig_sock in rl
```

```
###################################################################################
##################
# Handle zigbee writing

###################################################################################
##################

if zig_sock in wl:
  if len(zig_data_queue) != 0:
    (addr, data) = zig_data_queue[0]

    if len(data) < MAX_ZIGBEE_PACKET_SIZE:
      segment = len(data)
    else:
      segment = MAX_ZIGBEE_PACKET_SIZE
    try:
      sent_data = zig_sock.sendto(data[:segment], 0, (addr, end_point, profile_
id, cluster_id))

      data = data[sent_data:]

      if len(data) == 0:  ##If empty, pop element, if not, return modified
element to position in queue
        zig_data_queue.pop(0)
      else:
        zig_data_queue[0] = (addr, data)

    except socket.error, e:          #If its a socket error
      if e.args[0] == errno.EAGAIN:  #If its a blocking error
        pass                         #Do nothing, come again next select call
      else:
        print get_exception_info()                      #Not a blocking error,
print and exit
        sys.exit(0)

    except Exception, e:             #Unknown error, print and exit
      print get_exception_info()
      sys.exit(0)

#end if zig_sock in wl


###################################################################################
##################
# Handle listener sockets

###################################################################################
##################

for sock in listen_list:
  if sock in rl:  #if it is receiving
    conn, addr = sock.accept()      # accept conn
    print "New connection from: ", addr

    conn.setblocking(0)             # No blocking on connection
    tcp_conn_dict[sock] = conn      # assign sock to conn lookup
    tcp_conn_dict[conn] = sock      # assign conn to sock lookup
    tcp_data_dict[sock] = []        # Clean any queued data for new connection
```

```
    client_list.append(conn)        # append conn to client_list for select
statement

#endfor sock in listen_sock


#################################################################################
##################
# Handle read client sockets

#################################################################################
##################

for conn in client_list:
  if conn in rl:
    try:
      data = conn.recv(MAX_TCP_PACKET_SIZE)          #Receive our data
      print "Receiving %d bytes from tcp socket" %len(data)
    except socket.error, e:           #We have a socket exception
      if (e.args[0] == errno.EAGAIN): #If it's a blocking related exception
        continue                      #come back again next select call
      else:                           #If it's NOT a block related exception
        print get_exception_info()                      #Clean it up
        cleanUpConn(conn)
        continue

      except Exception, e:              #An unknown exception that's not socket
related
        print get_exception_info()                        #clean it up
        cleanUpConn(conn)
        continue

      if len(data) == 0:                #Did we receive a clean break?
        cleanUpConn(conn)               #Clean it up
        continue
      else:
        sock = tcp_conn_dict[conn]   #Find the associated socket with connection
        port = tcp_port_dict[sock]   #Find the associated port with socket
        addr = zig_port_dict[port]   #Find the associated zigbee address with
port
        zig_data_queue.append((addr, data))

#endfor conn in client_sock


#################################################################################
##################
# Handle write client sockets

#################################################################################
##################

for conn in client_list:
  if conn in wl: ##We can write
    sock = tcp_conn_dict[conn] ##Find the associated socket with connection

    if len(tcp_data_dict[sock]) != 0:  ##Find the associated data list with
socket, check if length of list is 0
```

```
      (data, packet_time) = tcp_data_dict[sock][0]
      if clock() - packet_time >= TCP_BUFFERING_TIME:
        #This packet has exceeded the time buffering, it is ready to send!
        try:
          if len(data) < MAX_TCP_PACKET_SIZE:
            segment = len(data)
          else:
            segment = MAX_TCP_PACKET_SIZE

          sent_data = conn.send(data[:segment])
          print "TCP: Sent %d bytes of %d total" %(sent_data, len(data))
          data = data[sent_data:]

          #If all data was sent, pop element out of queue, else restore modified
element to queue
          if len(data) == 0:
            tcp_data_dict[sock].pop(0)
          else:
            tcp_data_dict[sock][0] = (data, packet_time)

          except socket.error, e:
            if (e.args[0] == errno.EAGAIN):
              continue
            else:
              print get_exception_info()
              cleanUpConn(conn)
              continue

            except Exception, e:
              print get_exception_info()
              cleanUpConn(conn)
              continue

  #end for conn in client_list

#end main loop
```

Downloading the source

Tcp_zig_binding.zip

UDP to XBee network

Introduction

This page describes a simple application that forwards UDP traffic to a Gateway to a XBee destination while forwarding XBee traffic from that destination to the UDP network.

Requirements

- Gateway product that supports the Python interpreter.
- XBee device associated with the Gateway
- UDP access to the Gateway

Script

```
""" This script is intended for demonstration purposes.  It meets the minimum
requirements needed to move the data from the UDP network to the XBee network.

A more robust application may have additional error checking, command line
argument support or another features beyond this.

The basics of this application are to create a number of UDP sockets that
are logically connected to a like number of XBee nodes connected to the
Gateway product.  Traffic received on the UDP socket will be sent over the
XBee network to the paired XBee node, and likewise traffic received from the
XBee node will be sent to the paired socket.

The UDP sockets use the 'last known address'(LKA) to know the destination to
send the XBee data to.  The LKA is the last received UDP packet on that socket
source address, and the script will blindly send data back to that address,
regardless of whether or not it is listening.

Initially, the UDP socket does not have a LKA, and any received data that would
normally be forwarded to the LKA is dropped.  To set the LKA of a UDP socket to
stop forwarding data from the XBee network, send it a UDP packet of length 0.

NOTE:  There is no limit to how much the application will attempt to queue up
to send to the XBee network or the UDP network.  If too much is queued up,
unexpected errors may occur, resulting in strange Python behavior, device
panicing, or loss of data.
"""

import socket
import zigbee
import select
import bind_table

##############################################################################
# We need to map several pieces of data from the UDP socket.
# These dictionaries will provide a inexpensive look up table for that.
# Declarations
##############################################################################
```

```
udp_port_dict  = {} ##Udp socket to port it was bound to
udp_lka_dict   = {} ##Udp socket to its LKA it received data from
udp_queue_dict = {} ##Udp socket to its queued up data
udp_socks      = [] ##List of all Udp sockets

zig_port_dict  = bind_table.node_list ## list of XBee address to port numbers
zig_data_queue = []                   ## Data/address queued for the XBee socket

MAX_UDP_SIZE = 8192 ## Maximum UDP packet size to read/write
MAX_ZIG_SIZE = 84   ## Maximum XBee packet size to read/write

end_point  = 0xe8   ## Endpoint to bind the XBee socket to
profile_id = 0xc105 ## Profile ID to bind the XBee socket to
cluster_id = 0x11   ## Cluster ID to bind the XBee socket to

################################################################################
# To populate the above dictionaries with relevent data
# All data is based off the address to port dictionary in bind_table
################################################################################

for item,port in zig_port_dict.items():
  zig_port_dict[port] = item # provide the reverse lookup from port to address

  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  sock.bind(("", port))  #Create and bind a UDP socket

  udp_port_dict[sock] = port  #provide a udp socket to port and vise versa
  udp_port_dict[port] = sock

  udp_lka_dict[sock] = None   #Provide a udp socket to LKA
  udp_queue_dict[sock] = []   #Create a list object to act as a data queue

  udp_socks.append(sock)      #Append the socket to the UDP socket list


#Create and bind the XBee socket
zig_sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
zig_sock.bind(("", end_point, profile_id, cluster_id))

#Create a list of the udp_sockets plus XBee socket to monitor
sock_list = udp_socks + [zig_sock]

################################################################################
# For the main portion of the script, we perform a select call on the list of
# sockets, and must handle 4 distinct results from the select call.
#  - XBee data to read
#  - XBee data to write
#  - UDP data to read
#  - UDP data to write
################################################################################

print "Starting main"
while True:
  rl, wl, el = select.select(sock_list, sock_list, [])

  ##############################################################################
  # XBee data to read
```

```
  # Read the data, if the address read from is mapped to a UDP socket,
  # Queue the data up for that socket's data queue
  ###############################################################################

  if zig_sock in rl:
    data, addr = zig_sock.recvfrom(255)
    print "Received %d bytes from address: " %len(data), addr
    if addr[0] in zig_port_dict:
      port = zig_port_dict[addr[0]]
      udp_queue_dict[udp_port_dict[port]].append(data)
    else:
      print "Unknown zigbee node contacted us!"

  #############################################################################
  # XBee data to write
  # If we have data to write, send data to the address specified in the queue's
  # tuple.  Save any data that wasn't sent, and if empty, pop the element off
  #############################################################################

  if zig_sock in wl and len(zig_data_queue) != 0:
    data, addr = zig_data_queue[0]
    segment = ((len(data) > MAX_ZIG_SIZE) and MAX_ZIG_SIZE) or len(data)
    sent = zig_sock.sendto(data[:segment], 0, addr)
    print "Wrote %d bytes to address: " %sent, addr
    data = data[sent:]
    if len(data) == 0:
      zig_data_queue.pop(0)
    else:
      zig_data_queue[0] = (data, addr)

  #############################################################################
  # UDP data to read
  # Get the data and address, if the data is of length 0, remove the LKA and
  # move on.  If the LKA address doesn't match the source address of this
  # packet, the LKA address becomes the source address of this packet.
  # Find the UDP socket's associated XBee address, and queue it up in the
  # XBee data queue
  #############################################################################

  for sock in udp_socks:
    if sock in rl:
      data, addr = sock.recvfrom(MAX_UDP_SIZE)
      print "Read %d bytes from address: " %len(data), addr
      if len(data) == 0:
        udp_lka_dict[sock] = None
        continue

      if udp_lka_dict[sock] != addr:
        udp_lka_dict[sock] = addr
        udp_queue_dict[sock] = []

      port = udp_port_dict[sock]
      node_addr = zig_port_dict[udp_port_dict[sock]]
      zig_data_queue.append((data, (node_addr, end_point, profile_id, cluster_
id)))

  #############################################################################
  # UDP data to write
  # If we have data to write and have a LKA address, send that LKA address
```

```
  # the data we have queued for it.  If we have sent it all, pop off the data
  # or write back the remainder to the queue
  ##############################################################################

  for sock in udp_socks:
    if sock in wl and len(udp_queue_dict[sock]) != 0 and udp_lka_dict[sock] is
not None:
      data = udp_queue_dict[sock][0]
      segment = ((len(data) > MAX_UDP_SIZE) and MAX_UDP_SIZE) or len(data)

      sent = sock.sendto(data[:segment], 0, udp_lka_dict[sock])
      print "Wrote %d bytes to address: " %sent, udp_lka_dict[sock]
      data = data[sent:]

      if len(data) == 0:
        udp_queue_dict[sock].pop(0)
      else:
        udp_queue_dict[sock][0] = data
```

Files

This script makes use of a configuration generator called "table_generator.py" to create the initial table located in bind_table.node_list. See the TCP to Zigbee page for more details. The below archive contains the source of the UDP_zig_tunneling script and the "table_generator.py" file.

Udp_zig_tunneling.zip

Understanding XBee EndPoints

The way Digi implimented the XBee/ZigBee end-points in the XBee extensions to the Python socket API implies that they function like TCP or UDP source and destination port numbers. However unlike TCP or UDP which treats each src/dst port pair as unique conversations, XBee/ZigBee assigns each distinct 'destination end-point' to a single application (or handler) for message delivery. The handler is free to manually assign a meaning to the 'source end-point' in received packets, but it is NOT likely unique.

For example, a Modbus/TCP server can bind on TCP port 502. When five clients connect, each uses a destination port of 502 and a unique 'source port' (per IP address pair). Five 'sockets' may be created, and each of the five server-tasks spawned is blissfully unaware that other clients are active.

In contrast, an XBee server can bind on an end-point such as 0xE8, but five remote devices sending to destination end-point 0xE8 will appear to use the same 'socket'. Additionally, they all may claim a source end-point of 0xE8. The Xbee server code must manually process and distribute messages based on extended address or other criteria.

Which XBee technologies support end-points?

You can use the upper word of the DD setting to distinguish XBee which support end-points and those which do not.

Upper DD Word	Name	End-Point Support	Description
0x0000	Unspecified	Unknown	This means the XBee is mis-configured
0x0001	802.15.4	No	Non-mesh star/peer configured nodes on 2.4GHz
0x0002	ZNet 2.5	Yes	Digi's older pre-ZigBee firmware
0x0003	Digi Zigbee	Yes	Digi's firmware supporting Zigbee 2007
0x0004	Digi Mesh 900	Yes	Digi's proprietary mesh on 900Mhz
0x0005	Digi Mesh 2.4	Yes	Digi's proprietary mesh on 2.4Ghz
0x0006	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 868Mhz
0x0007	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 900Mhz

Notes:

- Use the DD value in the Digi gateway's XBee - you cannot trust that the remote XBee has a valid DD setting!
- Even if the XBee does NOT support end-points, Python may allow you to bind on a specific end-point like 0xE8. However, you will never receive any responses on 0xE8 because all responses appear to be targeted at end-point 0x00.

Managing end-points in XBee

Under XBee AT-transparent mode

So a concrete example, sending a packet with (src=0xE9 dst=0xE8) will NOT automatically result in return-responses being received as (src=0xE8 dst=0xE9) - you need to explictely change the AT settings in the remote XBee to return serial data to destination end-point 0xE9.

Therefore, to move your Python script bind end-point away from 0xE8 (the default), then do the following:

- Your Python script binds on an alterative end-point such as 0xE9, 0x01, or 0x41.
- Manually set all required remote XBee with an AT setting DE=0xE9 (or as required). You can do this by:
  - Under XCTU, relevant Xbee firmwares (such as ZigBee Router AT 0x2264) will have a closed/collapsed sub-option named "ZigBee Addressing" under Adressing. Click on it to open and you'll see the DE/SE/CI options.
  - Newer 2.9.x Digi gateway firmwares expose the DE command under the advanced settings for remote XBee nodes.
  - Use standard "remote AT command" API frames or ddo_set_param() calls to force DE to the desired value.

Note that changing the XBee's SE setting does NOT allow moving serial encapsulation away from 0xE8 - it only changes the source end-point cliamed by AT-Transparent responses. The remote XBee in AT-transparent mode will only forward data out the serial port which is targetted at 0xE8.

Under XBee API mode

When running XBee firmware in API mode, then most source/destination end-points are passed through transparently. It is up to the serial-attached CPU to parse and assign meaning to the end-points. Therefore, an external CPU might gracefully and automatically return responses to requests with (src=0xE9 dst=0xE8) as (src=0xE8 dst=0xE9).

End-point numbers to avoid

Although no exact list is available, you should avoid using these end-points:

End-Point	Description
0x00	ZigBee Device Object end-point - reserved by ZigBee stack
0xDC to 0xEE	Reserved for Digi use
0xEF to 0xF0	Reserved for other vendor use
0xF1 to 0xFE	Reserved by ZigBee Alliance / for other uses
0xFF	Digi treats as 'any end-point' (a wild-card) during Python socket operations

End-points 0x01 to 0xDB should be free to use. However, the rules for picking end-point numbers is much like that for TCP/UDP port numbers - be flexible and ready to change if required. For example, you might use TCP port 2101 to tunnel data to a Digi device server. It works, but TCP port 2101 is officially assigned for use as "RTCM-SC104", which is a Radio Technical Commission for Maritime

Services standard used to move GPS data via TCP/IP. Microsoft also overloads this port for use with RPC-based MQIS and Active-Directory Lookups. The same may be true of XBee end-points - even if you reuse a preassigned end-point it is only a problem if you require some other tool or function with a conflicting use.

The examples above used 0xE9, which Digi claims is reserved, so might be used for some other purpose in the future. If you select end-point 0x77, some other vendor might create a product requiring a Python script binding on end-point 0x77. So design your system to allow the end-point number selected to be easily changed as required.

EndPoints in the Modbus/IA Engine

You can use the Digi Modbus/IA engine to route Ethernet-based Modbus/TCP requests to remote Modbus/RTU serial devices via XBee RS-232 adapter and XBee RS-485 adapter. Configuring the XBee module is covered in Modbus Example Serial Adapter.

However, the default Modbus/IA behavior is to use ZigBee endpoint 0xe8, which will conflict with most sample Python programs and the Device Cloud/DIA framework. This conflict is because only 1 task can bind on (or register to receive) incoming XBee packets on endpoint 0xe8.

Therefore you should move the Modbus/IA traffic away from endpoint 0xe8 and use another, such as 0xe9. This is easier than trying to change the behavior of a Python application in an unknown number of places:

- In the remote XBee adapter, set DE as explained above to E9 (or to your desired value)
- In the IA Modbus unit id/slave address mapping table, add the new Xbee endpoint after the '!', so MAC looks like as 00:13:a2:00:40:30:de:cd!E9. Note that this does NOT change the 'destination' endpoint in the outgoing Modbus request - this remains 0xe8 for XBee module reasons. Instead, it causes the Modbus/IA engine to bind on incoming endpoint 0xE9 instead of 0xE8.

Using Digi Realport with Python

Digi Realport is a set of operating system drivers which make remote IP-based serial ports appear as local physical ports. Traditionally Digi Realport uses TCP/IP only and talks to a very special low-level driver in Digi products. Unfortunately, at present these low-level drivers in products such as the X4 and X8 gateways literally expects to talk to the hardware serial ports. Thus there is no way to connect standard Digi Realport to a Python script.

However the latest versions of Digi Realport for Windows has added a UDP mode which ONLY moves serial data, emulating a 3-wire RS-232 cable. The data it sends is well packed into a single UDP packet and works very well with protocols such as Modbus or Rockwell DF1. Since none of the Digi Realport protocol is included, any Python script can wait on the UDP socket and interact with a remote Windows-based host application.

Supported products

Digi Realport for Windows (such as P/N 40002549_xx.zip) Older versions or versions for a different OS may not have UDP support.

Digi products which support Python Zmatrix.

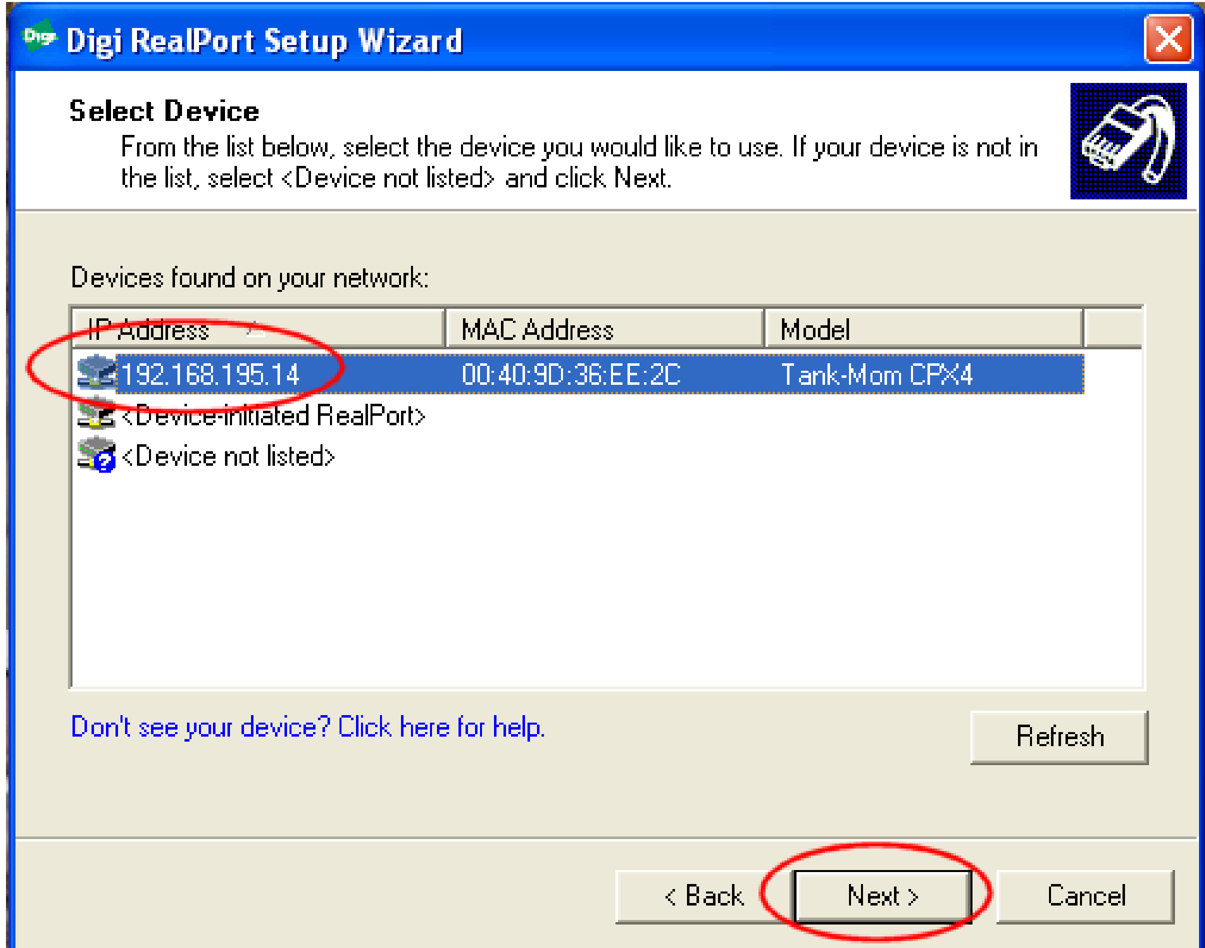
Setting up Digi Realport under Windows

Download the latest version of Digi Realport. This March_2010_ver.4.4.365.0_Realportdriver.zip is for Microsoft Windows XP/2003/Vista/2008 (both 32/64 bit arch).

CLICK HERE to find Realport for another Operating System or verify you have the latest Microsoft Windows version.

Installation

Unpack (unzip) the files in a suitable directory. Run the SETUP.EXE and you should see this this display:

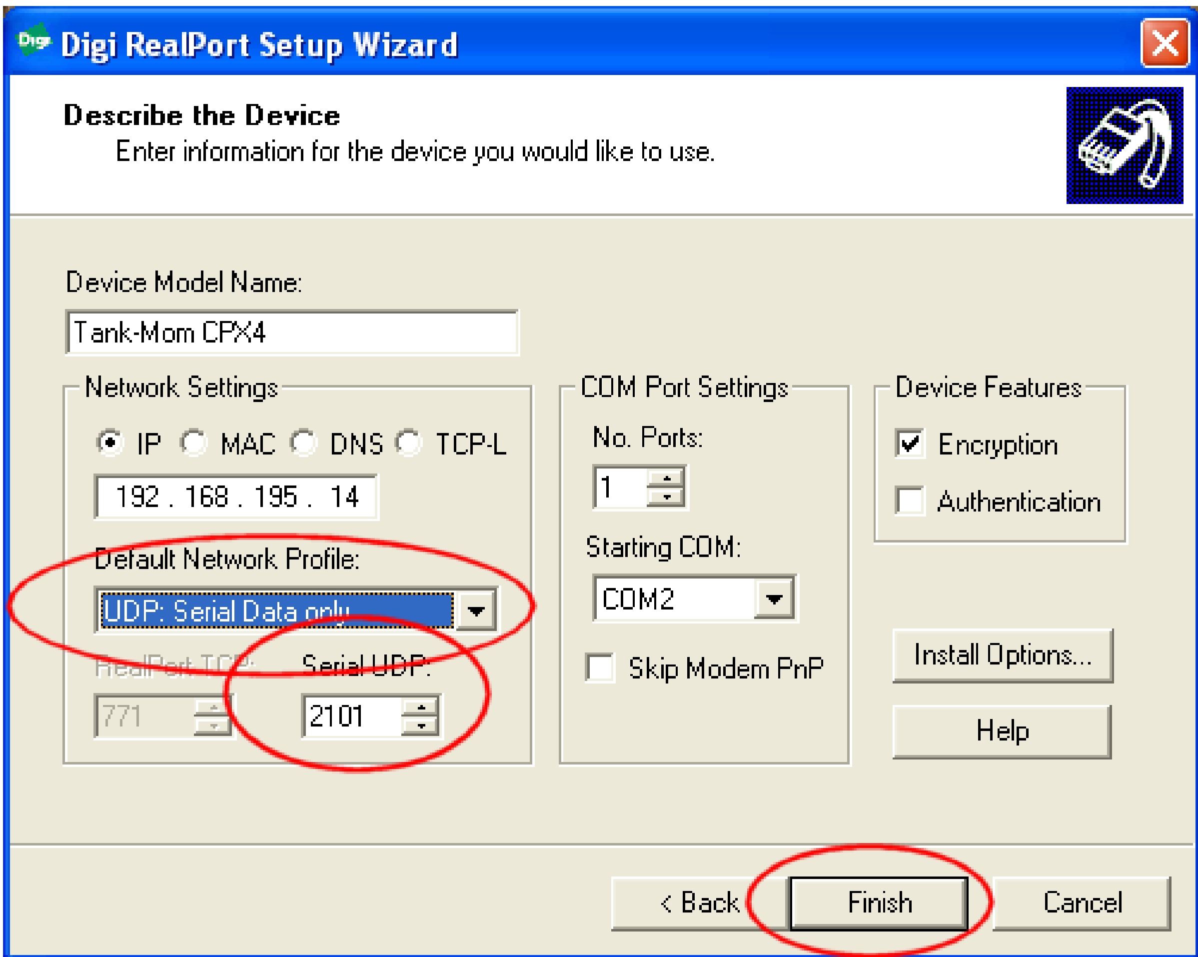


It is easiest to install Digi Realport with your device (or one of the same model) sitting next to you on your local Ethernet. This way the setup program should find it by browsing, plus learn all of the correct capabilities automatically - once installed, it is easy to change the IP address if your device is remotely located over wide-area-network such as cellular. In the example above, we'll be enabling Realport to a Digi ConnectPort X4 gateway. If you don't see your product listed, it might not have a proper IP address configured or your Windows firewall is blocking the UDP multi-cast being used. Always temporarily disable your Windows firewall when looking for LOST Digi devices, since they will reply to the Wizard with unreachable or even NULL (0.0.0.0) IP addresses and firewalls will always discard such 'mal-formed' UDP packets.

When you see this display, then:

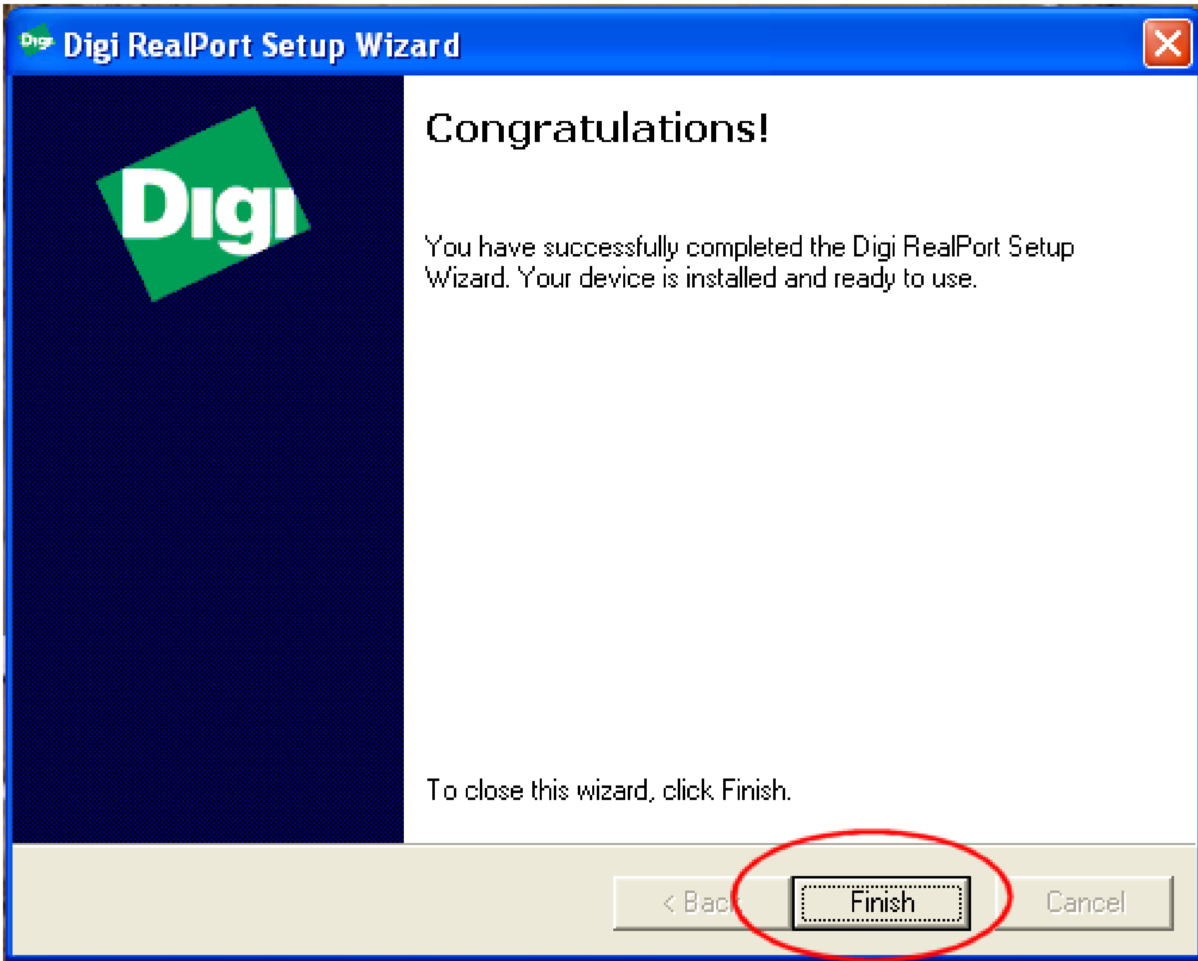
1. Select the device to configure.
2. Click **Next**.

   You should the see this display, where you can configure the basic features to use:



3. Change the Default Network Profile to **UDP: Serial Data only**. This changes Digi Realport to use UDP/IP instead of TCP/IP, plus the Serial Data only warning means all control signals, the ability to change baud rate (etc) is lost in this mode. Digi Realport in UDP mode literally mimics a 3-wire RS-232 line with only Txd, Rxd and signal ground lines.
4. Set the appropriate UDP port number to target as destination - the default of 2101 is likely okay.
5. Tweak other settings as desired. In this display we are configuring COM2 - you could move this to COM6 or other values. Also, with Realport in UDP mode things like Encryption and Authentication are NOT usable, even if the device supports it.
6. Click **Finish**.

   You should see the setup now do some work, run through a few progress displays and finally show this display:

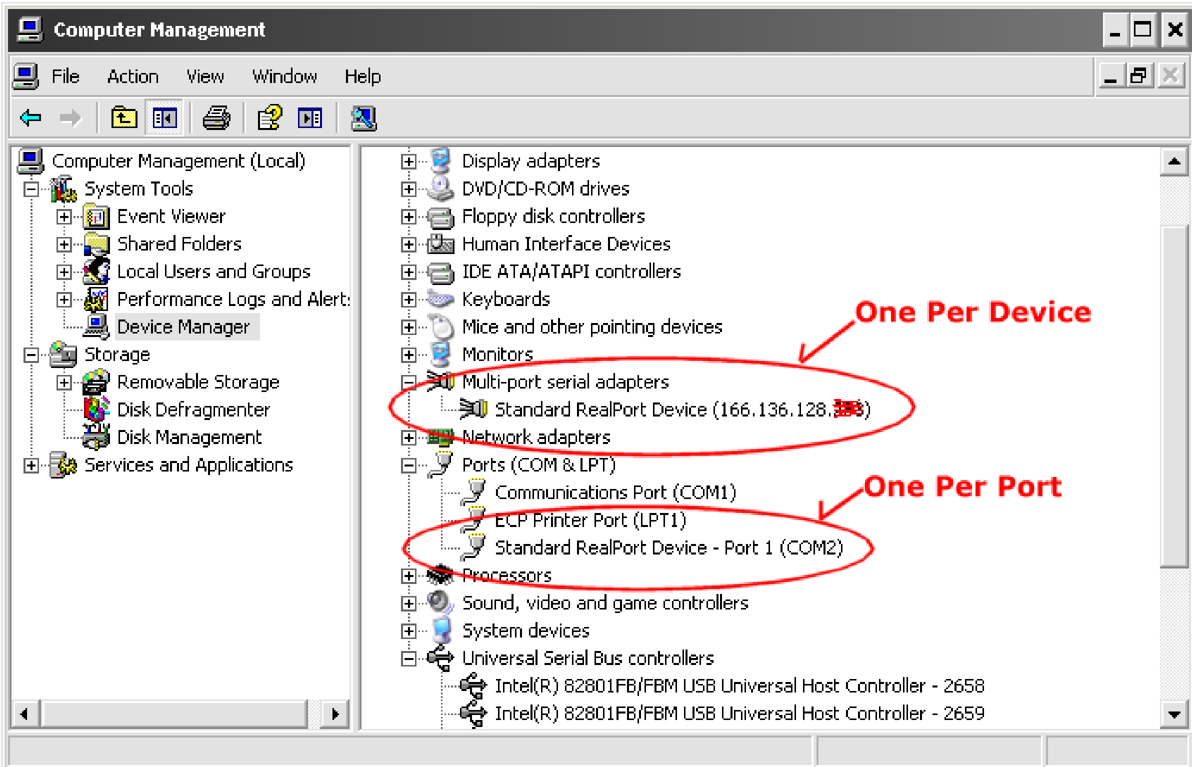


At this point, the computer will send any serial data written by a Windows application to COM2 to the IP address 192.168.195.14 in UDP packets to port 2101.

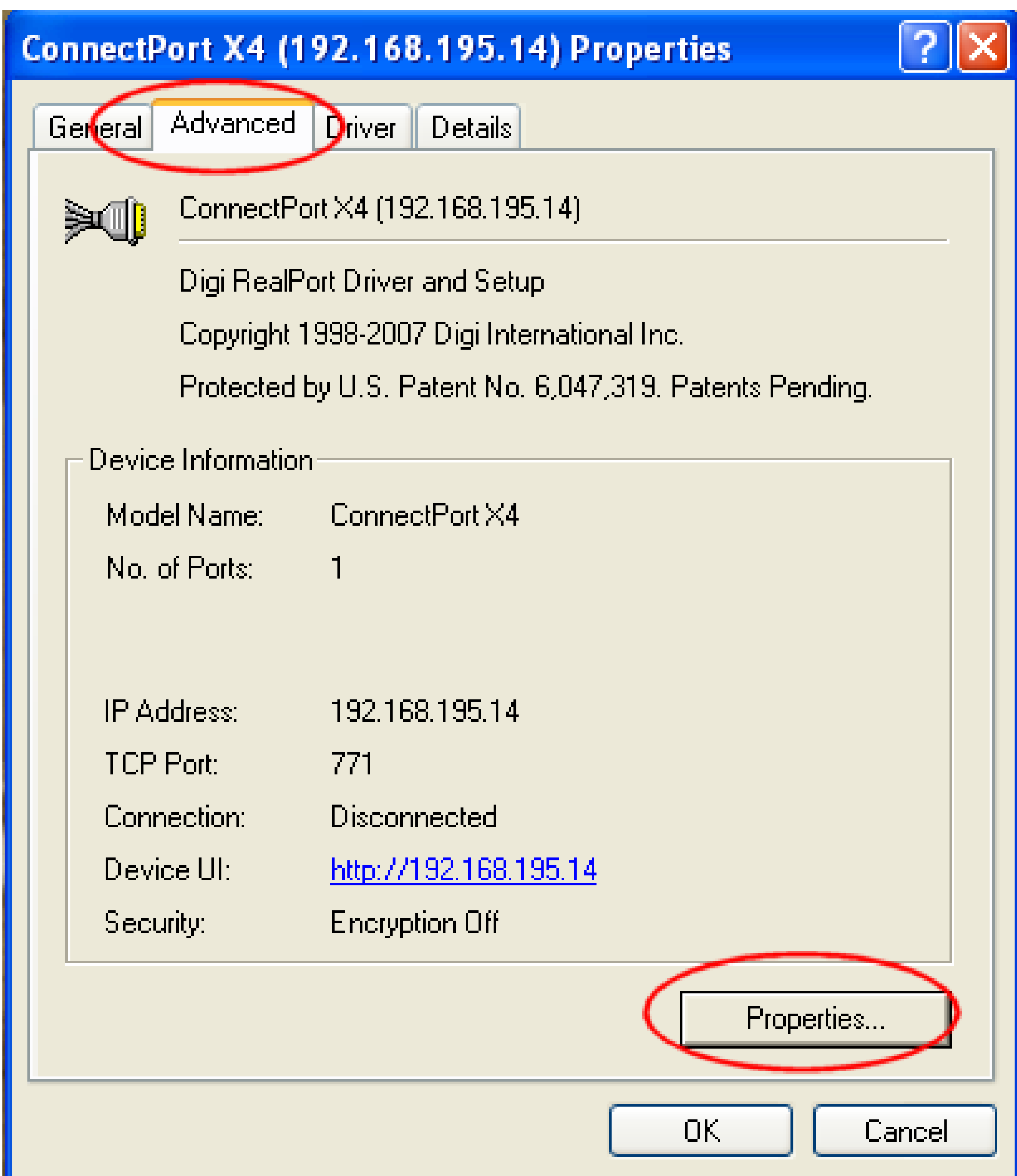
Changing settings within Realport in UDP mode

After you've installed Digi Realport, you can change setting through the Device Manager. You can open it several ways:

1. Right click My Computer, click Manage, Click Device Manager
2. Open System Properties, click the Hardware Tab, click the Device Manager



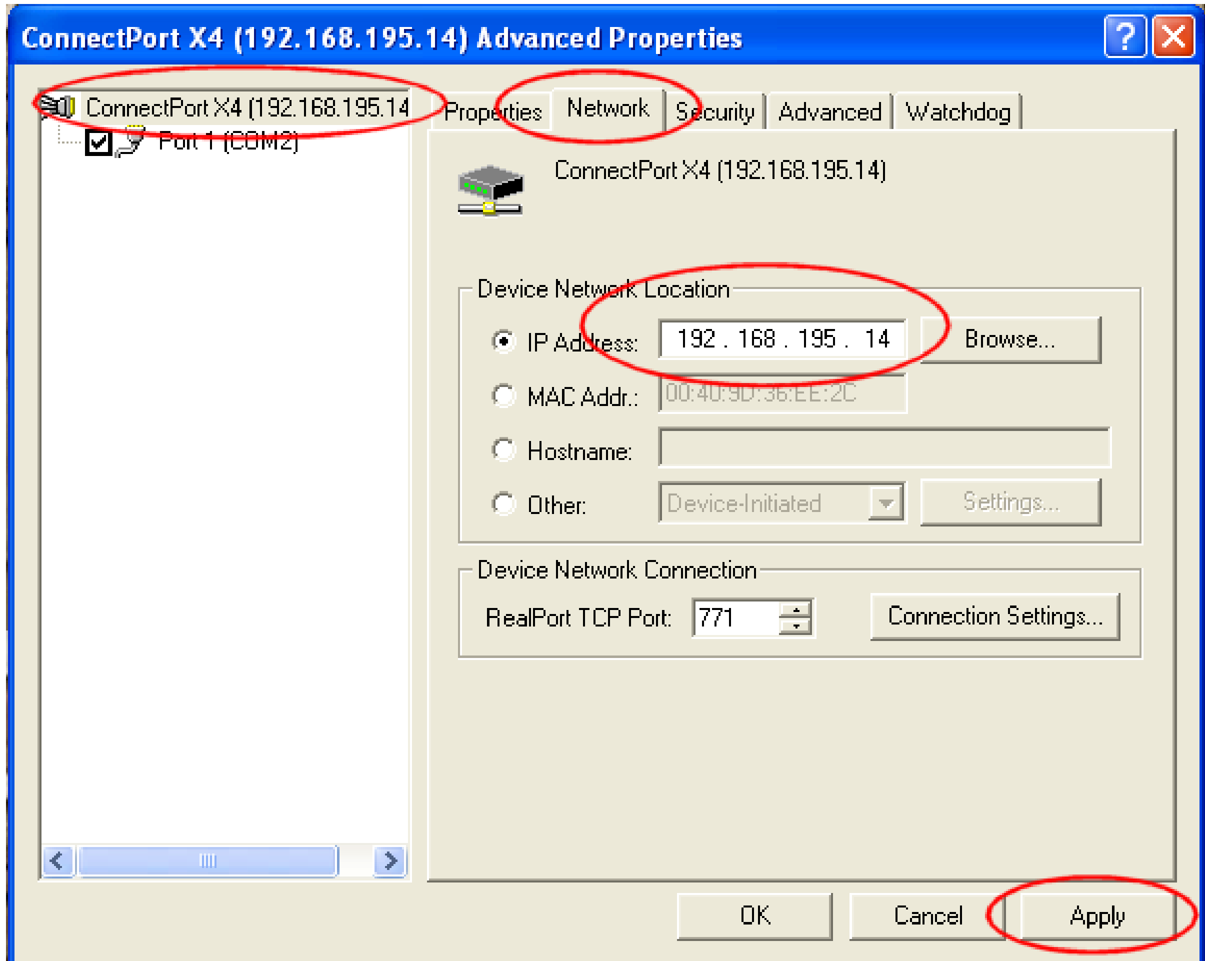
The Digi Realport device installs itself as a Multi-Port Adapter, right click it and select Properties. You will see this display:



Select the Advanced Tab, and select the Properties button. You will notice the Connection status is Disconnected - this is true since this is UDP/IP, not TCP/IP.

To Change the IP address

Select your Device (in this case ConnectPort X4) in the left panel and not the Port, then select the Network tab. Here you can select to use and change an IP address, or you can change to use a DNS Hostname. So while in this case the device was installed locally with a non-routing IP of 192.168.195.14, you could change it here to be the actual public IP such as 166.x.x.x. Remember to hit Apply when done.

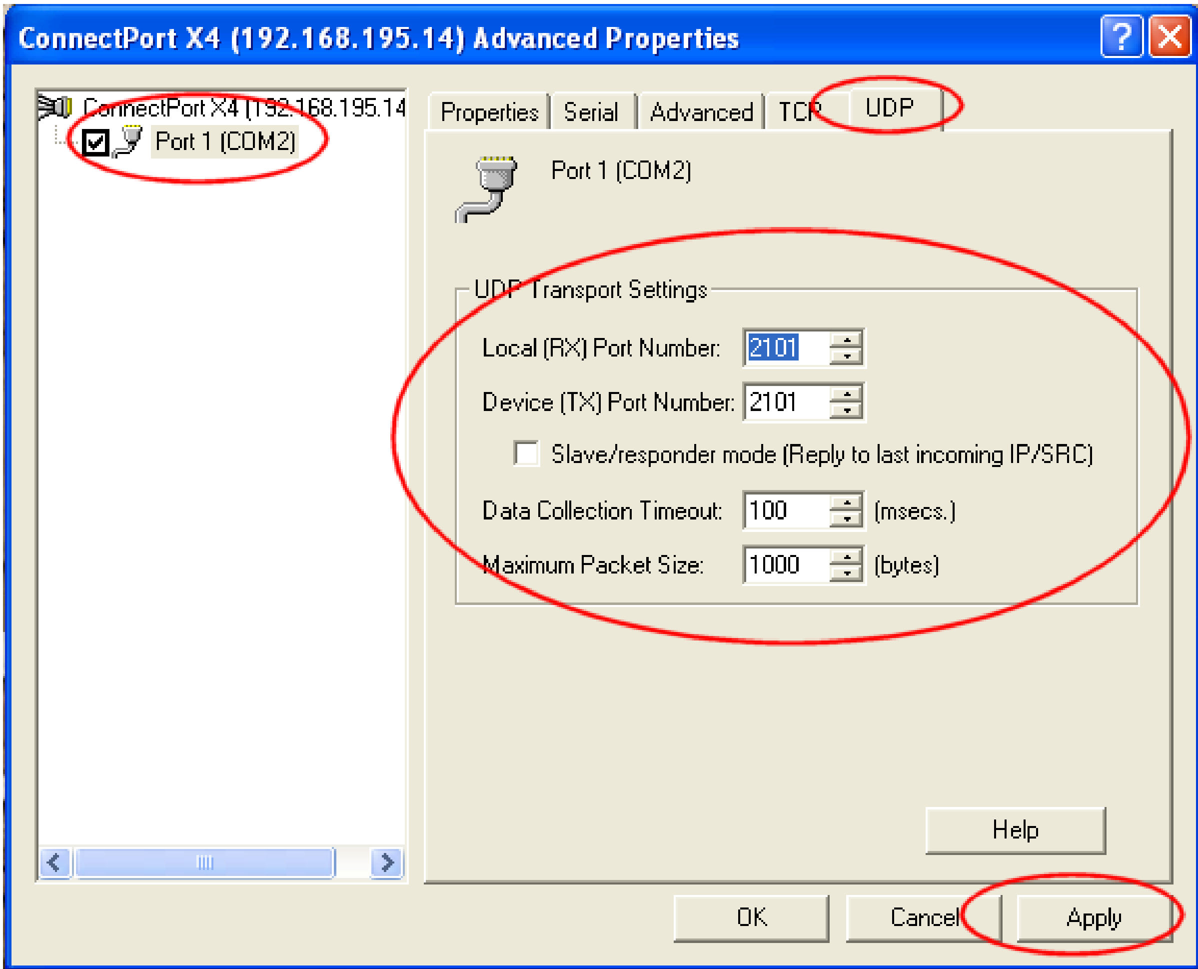


To Change the UDP settings

1. Select the **Port 1**, then the tab. The COM2 shown here will match the port you installed Realport on. Pressing the HELP button will - surprise - show you some fairly complete help information.

   The Local (RX) Port is the UDP source port within UDP packets, while the Device (TX) Port is the UDP destination port. In this example, the two numbers of 2101 match, but they can be any valid port numbers. The default is ideal for Windows-based clients polling the remote Digi product configured in UDP Sockets serial port profile. It sends data from the application to the Digi, and it must return it to the Local (RX) Port number.

   If the Digi product is a WANIA, CPX4 or DOIAP with the IA Engine active sending requests to the Windows host, then check the box labeled as Slave/responder mode, which disables the Device (TX) Port. In this mode, Digi realport remembers the UDP source/destination information to send the responses back to the Digi product.



2. Click **Apply** when you are done.

Xbee transport

Introduction

This page is to describe a method of transporting data from a TCP IP network to a XBee network on a Digi Gateway product. The goal of the script included on this page is to provide a simple way of setting up the transportation service without needing to know the peculiarities of the XBee network. Once the service is setup, that it will provide best-effort* delivery of data between the two networks.

Best effort varies between network protocols. In protocols that support network layer ACKs and retries, they are used and respected where possible. Consult the specifications of the protocol for more information.

Overview of the code

Included on this page are three separate code pieces:

- xbee_info.py - Contains particular information about the XBee protocol such as bind args and maximum packet size. Used as a library for other scripts and not called directly.
- xbee_generator.py - A script that creates a table of information mapping the TCP IP network to the XBee network. It uses repeated discoveries coupled with protocol specific information to generate a dictionary between the 64 bit MAC address of XBee nodes and a TCP port number.
- xbee_transport.py - The script that implements the data transfer service between the TCP IP network and the XBee network. It pulls in information from xbee_info.py for the XBee socket binding arguments and maximum packet size to be used over the XBee network. It imports the dictionary produced from xbee_generator.py to use as the mapping.

For the sake of briefness, this page will only go into depth on xbee_transport.py.

xbee_transport.py

```
Update: Using only the updated code. Includes fix for NDS migration from 2.8 and
below to 2.9+.
from socket import *
from select import *
from time import clock
from bind_table import node_list
import xbee_info

bind_args, xb_psize = xbee_info.get_xbee_info()

sock_port   = {}
sock_client = {}
sock_queue  = {}
client_list = []
listen_list = []

def cleanup(client):
  client_list.remove(client)
  sock = sock_client[client]
  sock_client[sock] = None
  del sock_client[client]
```

```
  client.close()
  client = None

def main():
  print "Creating lookup table"
  for dest, port in node_list.items():
    node_list[port] = dest

    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(("", port))
    sock.settimeout(0)
    sock.listen(1)

    sock_port[sock] = port
    sock_port[port] = sock

    sock_client[sock] = None

    listen_list.append(sock)

  print "Creating xbee socket"
  print "Bind args are: ", bind_args
  print "Packet size is: ", xb_psize
  xb_sock = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)
  xb_sock.bind(bind_args)
  xb_sock.setsockopt(XBS_SOL_EP, XBS_SO_EP_SYNC_TX, 1)
  xb_sock.settimeout(0)

  TCP_BUFFERING_TIME = .300 #Time to buffer TCP packets, helps with zigbee
fragmentation
  #and may reduce TCP cost to send data

  quit_sock = socket(AF_INET, SOCK_STREAM)
  quit_sock.bind(('', 40001))
  quit_sock.listen(1)

  xb_queue = []

  print "Entering mainloop"
  while True:
    read_list = [xb_sock] + listen_list + client_list + [quit_sock]
    write_list = [xb_sock] + client_list
    rl, wl, el = select(read_list, write_list, [], 1.0)

    if xb_sock in rl:
      data, addr = xb_sock.recvfrom(8192)
      print "Received %d bytes from address: " %len(data), addr

      new_addr = (addr[0], addr[1], addr[2], addr[3])

    if new_addr in node_list:
      print "addr in node_list dict"
      port = node_list[new_addr]
      sock = sock_port[port]
      if sock_client[sock] is not None:
        print "Queueing data to be sent out TCP port"
        print "Clock time is: %f" %clock()
        if len(sock_queue[sock]) == 0:
```

```
          print "Queue is empty, appending data"
          sock_queue[sock].append((data, clock()))
      else:
        packet_data, packet_time = sock_queue[sock][-1]
        if clock() - packet_time <= TCP_BUFFERING_TIME:
          print "Queue has item, meets TCP_BUFFERING_TIME condition, appending
data to previous packet"
          sock_queue[sock][-1] = (packet_data + data, packet_time)
        else:
          print "Queue has item, but is too old, not appending data to previous
packet, appending to queue instead"
          sock_queue[sock].append((data, clock()))


    if xb_sock in wl and len(xb_queue) != 0:
      data, addr = xb_queue[0]
      send_size = (len(data) < xb_psize) and len(data) or xb_psize
      print "Writing %d bytes to dest: %s" %(send_size, dest)
      try:
        sent = xb_sock.sendto(data[:send_size], 0, addr)
      except Exception, e:
        print "Exception occured sending to address: %s" %addr
        xb_queue.pop(0)
        port = node_list[addr]
        sock = sock_port[port]
        if sock_client[sock] is not None:
          print "Sending error message to tcp client"
          sock_queue[sock].append(("ERROR: Destination %s could not be sent %s
for reason: %s" %(addr, data, e), 0))

      else:
        if sent == len(data):
        xb_queue.pop(0)
      else:
        xb_queue[0] = data[sent:], addr

    for sock in listen_list:
      if sock in rl:
        client, addr = sock.accept()
        print "Received TCP connection from address: ", addr
        sock_client[sock] = client
        sock_client[client] = sock

        sock_queue[sock]   = []
        client_list.append(client)

    for client in client_list:
      if client in rl:
        print "Reading from client"
        try:
          data = client.recv(8192)
          print "Received %d bytes from TCP connection" %len(data)
        except Exception, e:
          print e
          cleanup(client)
          continue

      if len(data) == 0:
        cleanup(client)
```

```
        continue

      sock = sock_client[client]
      port = sock_port[sock]
      xb_queue.append((data, node_list[port]))

    for client in client_list:
      if client in wl:
        sock = sock_client[client]
        if len(sock_queue[sock]) != 0:
          data, packet_time = sock_queue[sock][0]

          if clock() - packet_time >= TCP_BUFFERING_TIME:
            ##Data is ready to send
            print "Sending data, clock time is: %f" %clock()
            try:
              sent = client.send(data)
              print "Wrote %d bytes out TCP connection" %sent
            except Exception, e:
              print e
              cleanup(client)
              continue

            if sent == len(data):
              sock_queue[sock].pop(0)
            else:
              sock_queue[sock][0] = (data[sent:], packet_time)

    if quit_sock in rl:
      raise KeyboardInterrupt("Stopping script, quit sock activated")


if __name__ == '__main__':
  main()
```

Source code

Xbee_transport.zip

Zmatrix

Zigbee/802.15.4 to TCP/UDP Transfer Application: Zmatrix

This application moves data back and forth between Ethernet and Zigbee/802.15.4. The Ethernet sockets can be either TCP server or TCP Port. They can also be a variety of UDP sockets: UDP nocast, unicast, multicast, broadcast, and special cast.

This application is configured through a user specified configuration file. It is marked with examples to give you an idea how to configure it. You will however need to customize it to get it working in your environment. Specifically, you'll need to enter the hardware addresses of your Zigbee/802.15.4 modules into this configuration file.

The application will bind a particular Zigbee/802.15.4 end node to a particular behavior. For example, you may want your Zigbee/802.15.4 end device to talk to a TCP server on the network through the gateway. You will then need to setup the configuration file to have the application make a network connection to the TCP server when it starts.

Once this application is running, it can be terminated by making a tcp connection to the gateway on it's port number 47285. This is easily done with the telnet command in windows. For example: telnet 192.168.1.100 47285

... will accomplish this.

Here are the 7 possible type of connections to the zigbee side: TCP client TCP server UDP1 UDP2 UDP3 UDP4 UDP5

Connection Type Description

TCP client When the script executes, it will attempt to make an outbound TCP connection on the network side (as opposed to Zigbee/802.15.4 side). Once that connection is established, it will then pass data back and forth between the Zigbee/802.15.4 device and the device on the network that the TCP connection is connected to. When the program terminates (see above), this socket will be closed.

TCP Server: The script will setup a TCP socket and listen for an incoming TCP connection. Once the connection comes in the TCP connection is setup. Data will then be passed and forth between the Zigbee/802.15.4 device and the incoming TCP connection. When the program terminates (see above), this socket will be closed.

UDP1: (no cast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. However, any data received from the Zigbee/802.15.4 side is dropped

UDP2: (unicast) The script will listen on a user specified port for incoming UDP data. IF it receives any, it sends it to the Zigbee/802.15.4 side. If it receives any data from the Zigbee/802.15.4 side it will forward that to a specified IP address and port number via UDP.

UDP3: (special cast - sticky UDP) The script will listen on a user specified port for incoming UDP data. IF it receives any, it sends it to the Zigbee/802.15.4 side. It will also remember the last place it received network data (IP address and port number). When it receives Zigbee/802.15.4 data it will forward it to IP address and port number of last device that sent it UDP data on that port.

UDP4: (broadcast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. Data coming from the Zigbee/802.15.4 side will be sent over the network as a UDP broadcast.

UDP5: (multicast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. Data coming from the Zigbee/802.15.4 side will be sent over the network as a UDP multicast.

Downloading the source

Zmatrix.zip

Device Cloud

Archiving data files from Device Cloud

When experimenting with XBee sensors, solar power and other fun technologies, it may be useful to archive your old data files on your own server in raw XML form. For example, someone testing use of solar power for a remote sensor could archive data for a full year, then run a report comparing the solar panel voltage verse battery voltage verse temperature verse month of the year.

Using Device Cloud presentations such as idigi_db (included in DIA) or idigi_upload (on this page DIA event uploader), you'll have a circular collection of N files which eventually overwrite themselves. For example, if you upload every hour and have at most 25 files, then your Device Cloud account will hold only 1 days worth of data.

Running by CRON

The Python script below can be run by a task scheduler every few hours to copy the data into a local archive. For example CRON can be used on a Unix/Linux server.

Assuming the user which runs this job runs at low permission level, you should create a shell script which runs this Python script, saving the output into a log location. User ownership and permissions can be tricky, but with care it will work. For example, you can archive the files to a SAMBA share drive under a special SAMBA user.

Python code

This Python application runs under Windows or Linux - it is not designed to run on a Digi gateway.

ZIP archive

Here is the Python code as a ZIP file: Archive.zip.

Full code

If you wish to browse the file for ideas, it is included inline below

```
            # Archive.py - download the XML files from the Device Cloud server

import os
import sys
import time
import urllib, urllib2
from urllib2 import URLError

# put your idigi account info here
IDIGI_URL = "http://developer.idigi.com"
USERNAME = "my_nama"
PASSWORD = "my_passa"

# your archive builds below this - if you use CRON to run this script,
# then you'll want the full path name
ARCHIVEBASE = '/home/idigi/archive'

# archive files are named with the modified-date from the Device Cloud storage
# in a form 'YYYYMMDD_HHMM' such as '20100812_1045'

# set to True to append the seconds, False to truncate them
ADD_SECS = False
```

```
# set True to force the seconds to be '00', ignored if ADD_SECS = False
ZERO_SECS = True

            # tuple of device id and storage directory tag name
DEV_LIST = [ ("00010000-00000000-03560210-13650987", 'x3_aio'),
             ("00000000-00000000-00409DFF-FF36EE2C", 'demo_solar'),
             ]

# In this example, the archive will store files as as:
#   /home/idigi/archive/x3_aio/x3_aio_20100812_1045.xml
#   /home/idigi/archive/x3_aio/x3_aio_20100812_1100.xml
#   /home/idigi/archive/demo_solar/demo_solar_20100812_1045.xml
#   /home/idigi/archive/demo_solar/demo_solar_20100812_1100.xml
# and so on

def log_into_idigi(username, password, idigiurl):
    '''Open the HTTP socket with login to idigi.com'''
    pwdManager = urllib2.HTTPPasswordMgrWithDefaultRealm()
    pwdManager.add_password(None, idigiurl, username, password)
    authHandler = urllib2.HTTPBasicAuthHandler(pwdManager)
    opener = urllib2.build_opener(authHandler)
    urllib2.install_opener(opener)
    return

def get_file_list(idigiurl, devid):
    '''Fetch a list of files in this device storage, which
    are returned as a list of tuples such as:
    [('data02.xml', '20100812_1045'),
     ('data03.xml', '20100812_1100')]
    These represent the file name in idigi and modified time
    '''
    dataurl = "%s/ws/data/~/%s/" % (idigiurl, devid)

    try:
        f = urllib2.urlopen(dataurl)
        data = f.read()
        # print data
        f.close()

    except URLError, e:
        print e.code
        print e.read()
        f.close()
        sys.exit(1)

    data = data.split('<')
    lst = []
    for ln in data:
        x = parse_fileinfo( ln)
        if x is not None:
            lst.append(x)

    return lst

def parse_fileinfo( ln):
    '''Return a tuple of file name and string of modified date,
    such as ('data02.xml', '20100812_1045')
    '''
```

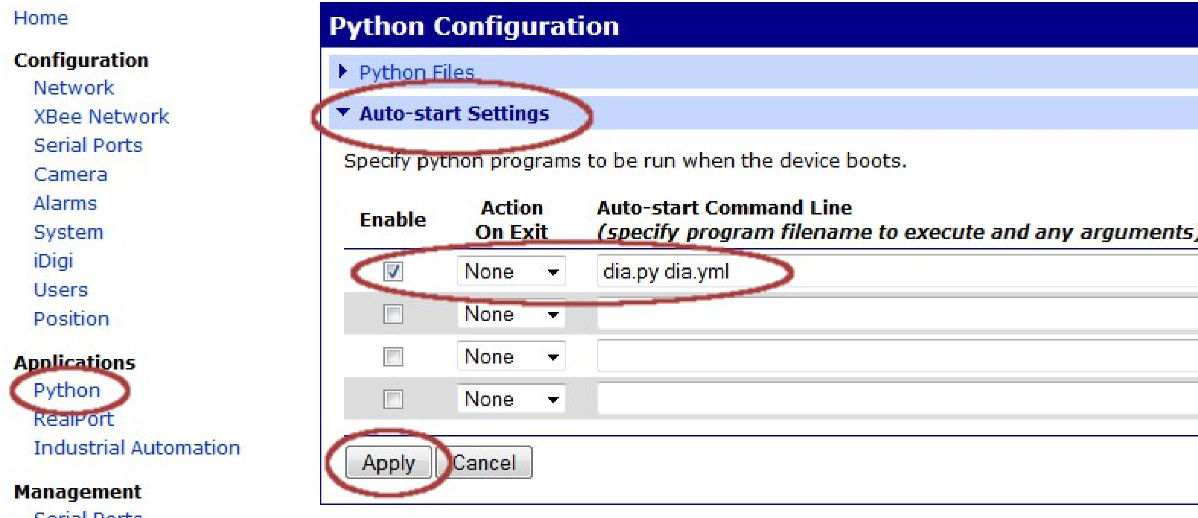
```
    # you could expand this to use SAX or another XML parser, but
    # since we just extract 2 pieces of info per line, this brute
    # force method works and is likely faster

    if ln.startswith('exist:resource'):
        # print 'ln:%s' % ln
        x = ln.find('name=')
        if x >= 0:
            st = ln[x+6:].split('"')
            name = st[0]
            x = ln.find('modified=')
            if x >= 0:
                st = ln[x+10:].split('"')
                date = st[0]
                date = date[0:4] + date[5:7] + date[8:10] + '_' + \
                       date[11:13] + date[14:16]
                if ADD_SECS:
                # then user wants the seconds in the name
                if ZERO_SECS:
                    # then just treat as seconds
                    date += '00'
                else: # else save actual
                    date += date[17:19]
            return (name, date)

    return None

def get_one_file(idigiurl, devid, filnam):
    '''Download one file as a string'''
    dataurl = "%s/ws/data/~/%s/%s" % (idigiurl, devid, filnam)

    try:
        f = urllib2.urlopen(dataurl)
        data = f.read()
        f.close()
    except URLError, e:
        print e.code
        print e.read()
        f.close()
        sys.exit(1)

   return data

def main():

   for dev_id, dev_nam in DEV_LIST:
        # verify the archive sub-directory exists
        arch = os.path.join(ARCHIVEBASE, dev_nam)
        if not os.path.exists(arch):
            print "Creating archive dir <%s>" % arch
            os.mkdir(arch)
        else:
            print "Device <%s> archive dir <%s> exists" % (dev_id,arch)

        # get the file list for this device
        log_into_idigi(USERNAME, PASSWORD, IDIGI_URL)
        files = get_file_list(IDIGI_URL, dev_id)
        # print files
```

```
        for fil_nam, fil_tim in files:
            save_file = os.path.join(arch, dev_nam + '_' + fil_tim + '.xml')

            if os.path.exists(save_file):
                print "Archive file <%s> already exists, skip download" % arch

            else:
                data = get_one_file(IDIGI_URL, dev_id, fil_nam)
                print data
                try:
                    print 'Saving file %s ...' % save_file,
                    f = open(save_file,'wb')
                    f.write(data)
                    f.close
                    print 'done.'

                except URLError, e:
                    print e.code
                    print e.read()
                    f.close()

if __name__ == '__main__':
     main()
```

Auto-start Python on a Digi gateway

By web interface

The most direct way to enable auto-start is via the device's web interface. Select the Python link on the left, then the Auto-Start Settings. You will be shown the four scripts you can auto-start. Enter the program name, including the ".py" extension. *Do not include the command "Python"!*



The On-Exit actions default to None, but can be set to restart the script, or reboot the gateway. **However a warning - never enable a restart/reboot option while debugging your code**, for having a simple typo in your main routine can cause the gateway to reboot in a tight loop which

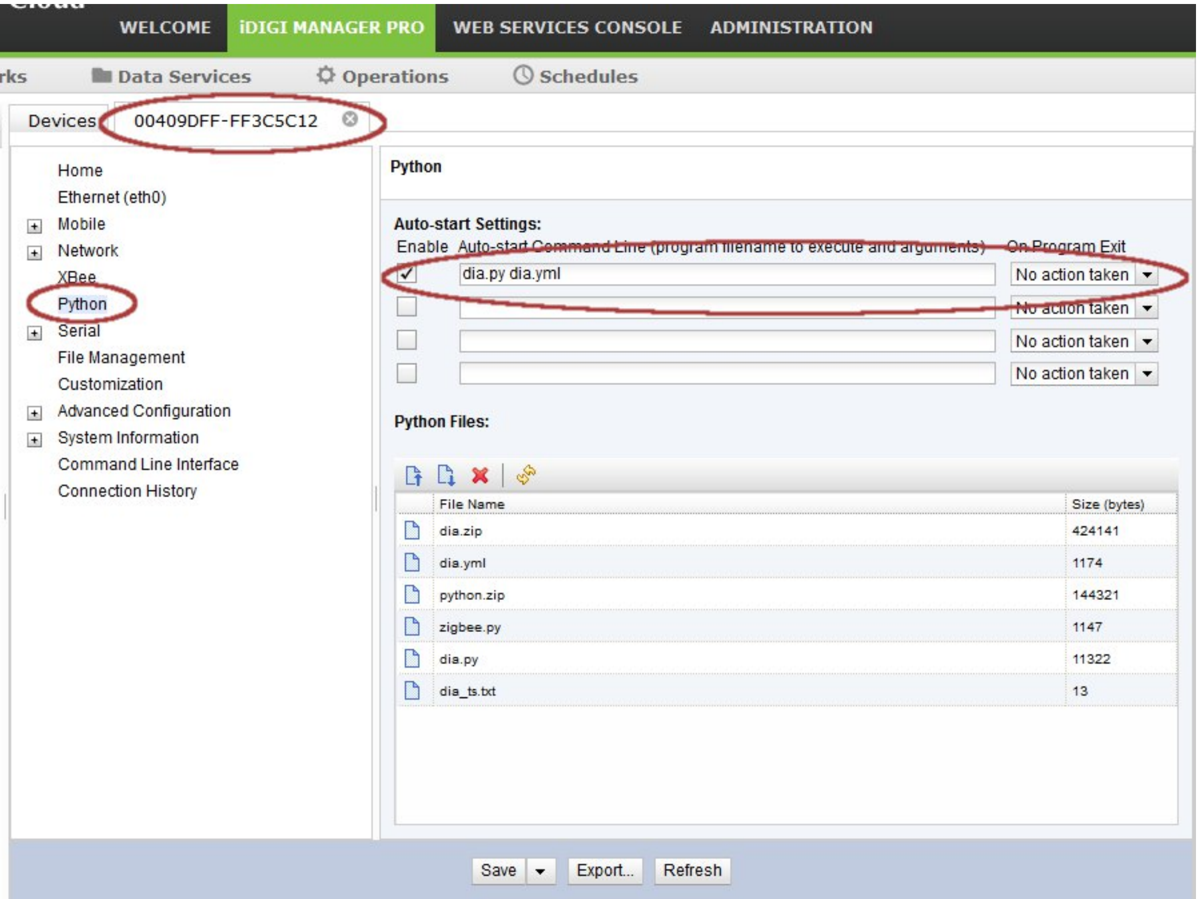
makes recover difficult (impossible by remote Device Cloud access). Users of the DIA framework should review this helpful change to dia.py: Rapid Reboot Detection.

Notes on ConnectPort X2e ZB

- Selecting the Python link on the left directly shows you the Auto-Start settings. Unlike on the ConnectPort X2/X4, Python files are NOT shown on this page. The Python files are accessed under the File Management link.
- The ConnectPort X2e ZB has built in Rapid Reboot Detection. The faster a reboot occurs, the longer the X2e/Linux kernel delays before auto-starting Python scripts.

By device manager

Setting auto-start via the Device Manager web console is much like setting it via the web interface. Note the above warning about the restart/reboot setting! if the device is rebooting every few seconds due to a Python error or simple typo, you will not be able to recover by Device Cloud access.



By RCI/SCI

The Python Auto-Start settings are held within the 'Python' settings group. See Digi's RCI and SCI documentation to understand how to read and write settings.

Request

```
<rci_request version="1.1">
 <query_setting>
  <Python />
 </query_setting>
</rci_request>
```

Response

```
<rci_reply version="1.1">
 <query_setting source="current" compare_to="none" encrypt="none">
  <Python><state>on</state><command>dia.py
dia.yml</command><onexit>none</onexit></Python>
  <Python
index="2"><state>off</state><command></command><onexit>nosne</onexit></Python>
  <Python
index="3"><state>off</state><command></command><onexit>none</onexit></Python>
  <Python
index="4"><state>off</state><command></command><onexit>none</onexit></Python>
 </query_setting>
</rci_reply>
```

Digi DIA notes

The DIA framework requires at least 2 special files to run - **dia.py** and **dia.zip**. You run the dia.py script to start the framework, and this file handles the diverse platform differences, mounting the dia.zip and running the framework from within. By default the dia.yml configuration is embedded in the dia.zip file. Some users prefer to manually keep a copy of dia.yml outside of the dia.zip to simplify remote browsing and changes. If this is done, you need to auto-start "dia.py dia.yml".

As of DIA version 2.0, creating a simple text file in the Python area named "nospin.txt" will enable dia.py to watch for a rapid reboot situation, which means the gateway has been rebooted more than 10 times in 20 minutes. This causes dia.py to delay launching the dia.zip files. See Also: Rapid Reboot Detection.

Digi ESP notes

The Digi ESP (Eclipse for Python) IDE use a default start file name of **dpdsrv.py**. If you use ESP to upload your Python code, you can safely put dpdsrv.py into your auto-start setting instead of dia.py.

Generally, dpdsrv.py does nothing more than run your script, which means you can often safely bypass dpdsrv.py and run your own main script. However, opening the dpdsrv.py script in an editor allows you to see what it is doing. Upon some platform (like a Windows PC), the dpdsrv.py *DOES* create a corrected search path so cannot be bypassed.

Command line build of DIA projects

We highly recommend building DIA projects in Digi ESP (our Eclipse-based development environment), but there are times where that is not optimal:

- When you have a DIA driver that does not have the supporting files to integrate into ESP
- When you would like to automate the process in a script
- You just don't roll that way ;)

Doing a build at the command line is simple and efficient, and very flexible.

Python

You will need Python installed on your system to do the make process.

- For NDS gateways, such as the ConnectPort X4, X5, X2D, etc. you will need Python 2.4.3.
- For Linux based X2e Zigbee Gateways, you will need Python 2.7.1.

NOTES:

- Python 2.4.3 has been deprecated, so can be difficult to find and install for many PC platforms. The best way to get it from the ESP installation, you can even choose during install to only install Python, not the full Eclipse environment.
- Python 2.7.1 is readily available for most platforms, including Windows, Mac and Linux.

Windows or Mac

The easiest and most reliable way to get the correct Python environment on your build PC is to install ESP. It will install a compatible version of Python 2.4, Python 2.6 and Python 2.7.

ESP can be downloaded from the this page on our support website ESP download is under Development Tools.

Tip At least in the Windows installer, you will have the option to not install ESP, just the Python binaries. If you are short on space, this is a good idea. I do recommend doing the full installation, if nothing else to get the DIA documentation (part of the Help system in ESP). Who knows, you might also find ESP a good way to do your projects.

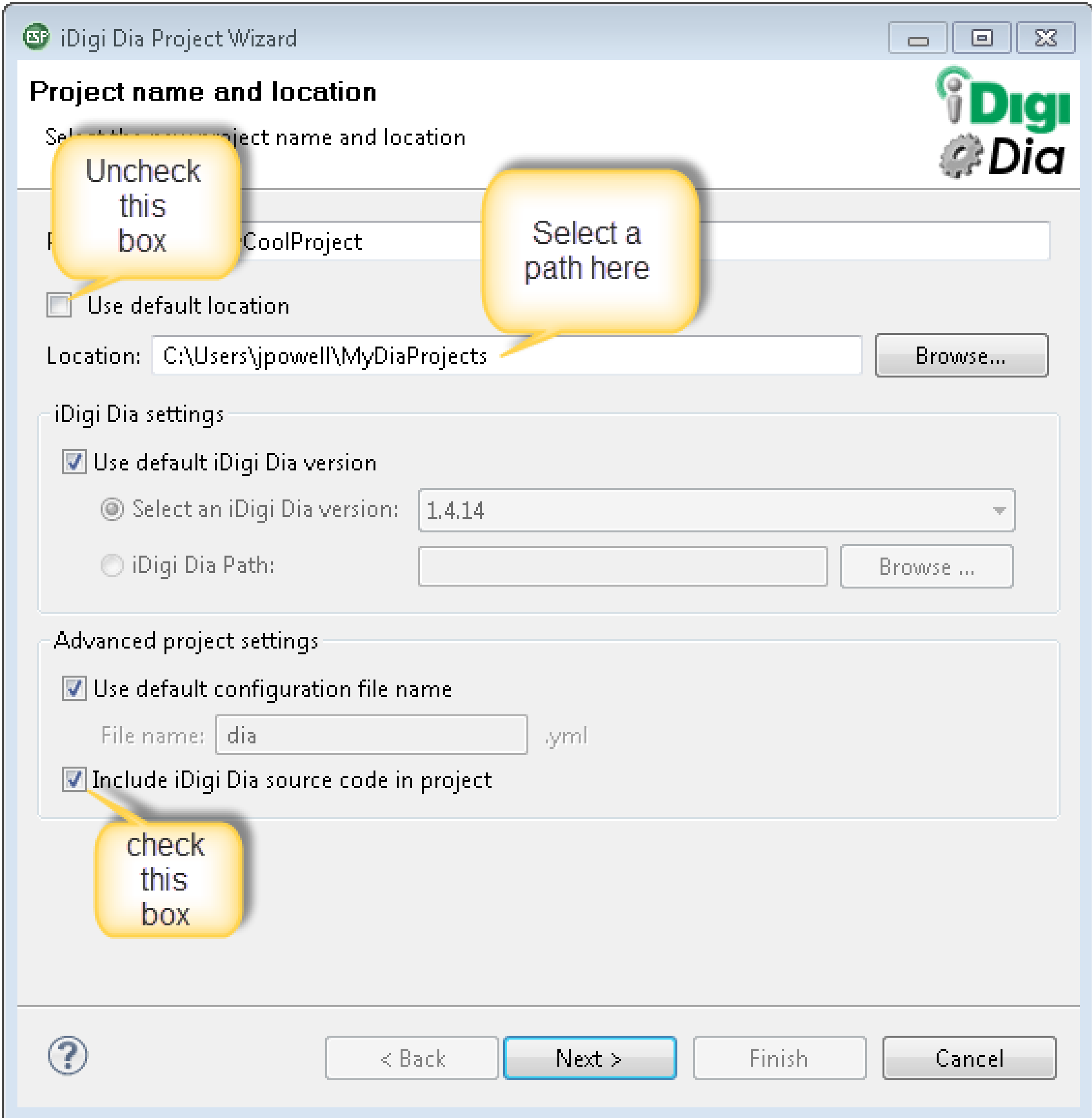
Linux or Unix

This is a bit trickier, as most distributions no longer have Python 2.4 in their repositories. Use your favorite search engine and search for instructions ("Ubuntu Python 2.4" for example). I was able to easily find instructions for Ubuntu, and there are several resources on the web as this is a common request.

If you are only developing for the X2e platform, you should have no problem getting Python 2.7.1.

How to get the build tree

The easiest way is to create a new "DIA Project" in ESP. In the first window, give your project a name, uncheck default location, enter a path that is easy for your to access, and check the "include DIA source code in project" checkbox.



Doing the build

Now, the easy part!

From the root directory of the tree (you will see subdirectories such as src\, lib\, doc\, etc.) type:

```
C:\Python24\Python.exe .\make.py cfg\MyCoolProject.yml
```

For X2e *builds, use c:\Python27 instead*.

The path to Python.exe may need to modified to match your PC, but this is normally the correct path for Windows.

Tip Though it is normal convention to name the file "dia.yml" it is not necessary when building this way. You can have a whole bunch of .yml files for different projects, point to them during the build

process, and it will work just fine. The yml (no matter what the source yml filename is) is serialized and named "dia.pyr" in dia.zip during the make process (see the 6th line of output when running the make).

You will hopefully see something like this:

```
Project make started.
Analyzing files...
Compiling files...
Zipping files...
Finished writing archive 'bin\dia.zip'
Transforming settings file 'cfg\MyCoolProject.yml' to 'pyr' format...
Adding transformed settings as 'dia.pyr' to 'bin\dia.zip'...
Project make completed successfully.
```

Your dia.zip (the built project) will be in the bin\ directory. Transfer that to the /WEB/Python/ directory on your ConnectPort/ConnectCore gateway. If you have not already done an ESP build from this version of DIA on your gateway, you will also need to copy dia.py (resides in the root directory of the build tree) to the same directory.

ConnectPort X3 - GPRS, 232 interface - demo application

The ConnectPort X3 COM Port can be used to get the LOGs or **Debug Prints**. The Demo Application Code Describes that Debug Prints from ConnectPort X3 COM Port can be uploaded to the Device Cloud Data Storage every half an hour in the folder name as ConnectPort X3 Device ID. And whenever the data stored in the Device Cloud Data Storage is to be archived, another script can be run on the PC to generate LOG.txt in the C Drive.

To run this Demo Application two types of Python Scripts are required:

1. Python Script runs on the ConnectPort X3 and Upload the Debug Prints from CPX3 COM Port to Device Cloud Data Storage in the .txt file format every half an hour.
2. Python Script runs on the PC and Archive data from Device Cloud Data Storage and Generate LOG.txt in the C Drive.

Python Script runs on ConnectPort X3

The Python Script below runs on the Gateway in the Auto Start Mode. Here two files need to be uploaded to the CPX3:

1. CPX3_G.PY : Script that runs on the CPX3.
2. CPX3_IMP.ZIP : httplib.pyc is imported from this ZIP file.

This Script will upload the **Debug Prints** from the CPX3 COM Port will upload to Device Cloud Server Platform every half an hour. Unlike PC COM Port, Any Device can be connected to the ConnectPort X3 COM port, with prior Baud rate settings and AT Commands to be sent on the TOP of the Script. The AT Commands can be added in ATCMD variable in the LIST format and each AT command will be given after every 2 second interval. This application will only Work when Auto Start is enabled on ConnectPort X3. Auto Start can be enabled either by **Digi X3 Dashboard** or from **Device Cloud Server Platform**.

1. After clicking **Enable**, In the Auto Start command line */WEB/Python/CPX3_G.PY* is to be given to enable Auto Start mode through "Digi X3 Dash Board".

2. After clicking **Enable**, In the Auto Start command line *CPX3_G.PY* is to be given to enable Auto Start mode through Device Cloud Server Platform.

ZIP CPX3_Gateway

Here is the Python Script CPX3_G.PY in a ZIP file: CPX3_G.zip.
Here are the library files in a ZIP file: CPX3_IMP.ZIP.

Full script

If you wish to browse the file for ideas, it is included inline below.

```
            # CPX3_Serial: Uploading LOGs from CPX3 COM Port to Device Cloud Data
Storage

# Note: To run the Script on CPX3, CPX3_IMP.ZIP needs to be uploaded on CPX3

import os
import sys
import errno
import time
import termios

sys.path.append("WEB/Python/CPX3_IMP.ZIP")
import httplib

import base64

ATCMD = ["at+xxxx\r\n", "at+xxxx\r\n"] #Here AT command to send through COM
                                       #Port is to be given in the list.
                                       #"xxxx" need to be changed with the
                                       #AT command.

port_num = 0  #Select Gateway COM Port
# Baud Rate Settings
ispeed = 115200
ospeed = 115200

collection = "00000000-00000000-xxxxxxxx-xxxxxxxx" # Device ID
IDIGI_USERNAME  = "user_name" #Username for developer.iDigi.com
IDIGI_PASSWORD  = "password"  #Password for developer.iDigi.com
auth = base64.b64encode("%s:%s" % (IDIGI_USERNAME, IDIGI_PASSWORD))

def port_open(num):
    try:
        serial_open = os.open('/com/%d' %num, os.O_RDWR | os.O_NONBLOCK)
        print "Port Open %s" % serial_open
    except:
        print "... port in use or invalid name ..."
    return serial_open

# Get the AT command from the List ATCMD
def AT_Command(count):
    return ATCMD[count]

def current_time():
    local = list(time.localtime())
    year = local[0]
    month1 = local[1]
```

```
    date = local[2]
    minut = local[4]

    # Convert to Indian Standard Time
    if minut>=30 :
        hour = local[3]+6
        minute = local[4]-30
    else:
        hour = local[3]+5
        minute = local[4]+30

    #Convert Numerical Month to Month Name
    mth = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', \
           'Oct', 'Nov', 'Dec']
    cnt=0
    for M in mth:
        if cnt==(month1-1):
           month = M
        cnt = cnt + 1
    return "%dhr_%dmin_%dth_%s_%d.txt" %(hour, minute, date, month, year)

#Get the Current Time and return only Hour and Minutes.
def time_cal():
    local = list(time.localtime())
    minut = local[4]

    # Convert to Indian Standard Time
    if minut>=30 :
        hr = local[3]+6
        min = local[4]-30
    else:
        hr = local[3]+5
        min = local[4]+30
    return (hr,min)


def put_data(msg, file):
    # open a connection to the host and get the specified file
    url = 'developer.idigi.com'
    addr = httplib.HTTP(url)
    addr.putrequest('PUT', '/ws/data/~/%s/%s' % (collection, file))
    addr.putheader("Authorization", "Basic %s" % auth)
    addr.putheader('Host', url)
    addr.putheader('User-agent', 'Python-httplib')
    addr.putheader("Content-type", 'text/xml; charset="UTF-8"')addr.putheader
("Content-length", "%d" % len(msg))
    addr.endheaders()
    addr.send(msg)

    # get the reply to the request
    returncode, returnmsg, headers = addr.getreply()

    # if successful, print out the contents of the .txt file
    if returncode == 201:
        return "data written into the file %s" % file
    # else print the error code and error message
    else:
        return returncode

# Open the Port and Serial COM Port settings
```

```
fd = port_open(port_num) #Open Port 0

old = termios.tcgetattr(fd)
old[4] = ispeed # ispeed
old[5] = ospeed # ospeed

# Print Serial Port settings
for b in old:
print b

try:
    termios.tcsetattr(fd, termios.TCSANOW, old)
except:
    print "... handle a port not accepting your settings ... "

cnt = len(ATCMD) #Count the Length of the List ATCMD


### Entry Point###
while 1:
    write_data=""""""
    file = current_time() # File name/ Time Stamp to upload the file
    Start_time_hr,Start_time_min  = time_cal()

    while 1:
        received_data=""""""
        # Read from serial Port
        try:
            received_data = os.read(fd,1000)
            print "%s " %received_data

        except OSError, e:
            if( e.errno == errno.EAGAIN): # EAGAIN just means NO data ready,
                                          #try again
                print " ...  NO data received or response timeout here ... "
            else: # other errors
                print " ...  port faults - likely fatal ... "
                traceback.print_exc() # make sure you show user what error was

        # temporary Store LOG in the Variable
         write_data = """%s
%s
""" % (write_data, received_data)

        time.sleep(2) # 2 Second Sleep

        if cnt < 0:
            cnt = len(ATCMD)-1
        else:
            cnt = cnt - 1

        #write to serial Port
         try:
             count = os.write(fd, AT_Command( cnt))
         except:
             print " ...  port faults - likely fatal ... "

        # Calculate Time for the LOG Generation
         if  time_cal() == (Start_time_hr + 0, Start_time_min + 30):
```

```
            break

    #write Data to iDigi
    try:
        put_msg = put_data(write_data, file)
        print put_msg
    except:
        print "Data Could not be writeen at %s" % current_time()
```

Python script to be run on PC

This Python script will get list all the .txt files from the Device Cloud Data storage from the folder name as CPX3 Device ID. Then it will read the data from all .txt files from the list in the ascending order from the time of generation of the file and write them to **LOG.txt** in the local C Drive. Note: This Script only need to run when the LOG at the local Drive need to be generated as **LOG.txt**.

Here Depending upon CPX3 Device ID the value of the variable named as **collection** need to be changed. And Device Cloud Server Platform Username and password need to be changed.

ZIP CPX3_GENERIC_PC

Here is the Python Script CPX3_GENERIC_PC.PY in a ZIP file: CPX3_GENERIC_PC.zip.

Full script

If you wish to browse the file for ideas, it is included inline below.

```
            # LOG file on PC: Getting .txt files from Device Cloud Data Storage 
and generate LOG.txt
#                 in the C Drive.

# Note: This Application only need to be run when user need to generate
#       LOG.txt from the data fetched from Device Cloud Data Storage.

import os
import sys
import time
import httplib
import base64

collection = "00010000-00000000-xxxxxxxx-xxxxxxxx" # Device ID
IDIGI_USERNAME  = "user_name" #Username for developer.iDigi.com
IDIGI_PASSWORD  = "password"  #Password for developer.iDigi.com
auth = base64.b64encode("%s:%s" % (IDIGI_USERNAME, IDIGI_PASSWORD))

os.getcwd()         #Get the Path to make the LOG file
os.chdir('C:/')     #Change the Path to make the LOG file
os.getcwd()         #Get the Path to make the LOG file


def parse_fileinfo(rs):
    if rs.startswith('exist:resource'):
    x = rs.find('name=')
    if x >= 0:
        st = rs[x+6:].split('"')
        name = st[0]
        return name

def get_file_name():
```

```
    # open a connection to the host and get the specified file
    url = 'developer.idigi.com'
    addr = httplib.HTTP(url)
    addr.putrequest('GET', '/ws/data/~/%s' % collection)
    addr.putheader("Authorization", "Basic %s" % auth)
    addr.putheader('Host', url)
    addr.putheader('User-agent', 'Python-httplib')
    addr.endheaders()
    name = """"""

    # get the reply to the request
    returncode, returnmsg, headers = addr.getreply()

    # if successful, print out the contents of the .txt file
    if returncode == 200:
        msg2 = addr.getfile()
        msg1 = msg2.read()
        data = msg1.split('<')
        lst = []
        for rs in data:
            x = parse_fileinfo(rs)
            if x is not None:
                lst.append(x)
        return lst

    # else print the error code and error message
    else:
        return returncode


def get_data(file):
    # open a connection to the host and get the specified file
    url = 'developer.idigi.com'
    addr = httplib.HTTP(url)
    addr.putrequest('GET', '/ws/data/~/%s/%s' % (collection, file))
    addr.putheader("Authorization", "Basic %s" % auth)
    addr.putheader('Host', url)
    addr.putheader('User-agent', 'Python-httplib')
    addr.endheaders()

    # get the reply to the request
    returncode, returnmsg, headers = addr.getreply()

    # if successful, print out the contents of the .txt file
    if returncode == 200:
        msg2 = addr.getfile()
        msg1 = msg2.read()
        return msg1
    # else print the error code and error message
    else:
        return returncode

# Creating LOG file in the C Drive
try:
    try:
        data = open('LOG.txt','w')
        list_file_name = get_file_name()
        msg_write = """"""
```

```
        for file in list_file_name:
            get_msg = get_data(file)
            msg_write = msg_write + get_msg

        data.writelines(msg_write)
    except IOError:
        print 'File error:'
finally:
    if 'data' in locals():
        data.close()

#Read the LOG.txt file from C Drive.
try:
    try:
        data = open('LOG.txt','r')
        print data.read()
    except IOError:
        print 'File error:'
finally:
    if 'data' in locals():
        data.close()
```

Delete a file in Device Cloud storage

Below is Python sample code developed for a Linux or Windows host to delete a file known to exist within your device's Device Cloud data store. Your application paradigm may be to:

- Read a list of filenames
- Download any desired files
- Delete files

```
...
import httplib2
...

def delete_one_file(idigiurl, devid, filnam):

    # idigiurl is like "http://developer.idigi.com"
    # devid    is like "00000000-00000000-00409DFF-FF111111"
    # filnam   is like "my_data10.xml"
    # USERNAME and PASSWORD are strings as expected

    dataurl = "%s/ws/data/~/%s/%s" % (idigiurl, devid, filnam)

    try:
        h = httplib2.Http()
        h.add_credentials(USERNAME, PASSWORD)
        resp, content = h.request(dataurl, method='DELETE')
        #print resp, content
        return True

    except:
        traceback.print_exc()
        return False
```

Device Cloud Alarms

Introduction

The Alarm web service is used to retrieve information about the alarms within your Device Cloud account. You have the option of retrieving information about all of the alarms within your account, or a single alarm. The alarms feature allows you to configure, manage, and react to alarm conditions in Device Cloud. Alarms can be defined to monitor various activities within the platform such as when a device disconnects and fails to reconnect within a specified amount of time, or monitoring XBee Nodes with an excessive number of activation attempts. The alarms feature only supports the monitoring of Devices and XBee Nodes. Alarms cannot be created to monitor Data Services or any other Device Cloud feature at this time. Once you have created an alarm targeting a single device or group of devices, you can gather information about that alarm using the Alarm web service.

Alarm

The Alarm web service is used to retrieve information about all of the alarms within your Device Cloud account.

URI: http://<hostname>/ws/Alarm HTTP operations supported:

GET: Get a list of your existing alarms

Elements include:

- almId - alarm ID of the alarm
- cstId - the automatically generated ID assigned to a customer
- almtId - the ID of the alarm template used to create the alarm
- grpId - the automatically generated ID assigned to a customer's group
- almName - name of the alarm
- almDescription - basic alarm description
- almEnabled - a boolean indicating whether or not the alarm is enabled
- almPriority - the user defined alarm priority (high, medium, or low)
- almScopeConfig - information about how the alarm is scoped
- almRuleConfig - information about how alarm rules are configured

Example #1: To get a list of all alarms:

```
GET /Alarm
```

This will return a list of all of the alarms configured within your Device Cloud account.

Example #2: To get the details of a specific alarm ID:

```
GET /ws/Alarm/{almId}
```

AlarmStatus

The AlarmStatus web service is used to retrieve the current status of an alarm. URI: http://<hostname>/ws/AlarmStatus HTTP operations supported:

GET: Get a list of current alarm statuses

Elements include:

- id
- almId - alarm ID of the alarm
- almSourceEntityId - the specific device or source entity of this alarm status
- almsStatus - current status of the alarm (0: reset, 1: triggered, 2: acknowledged)
- almsTopic - the alarm topic
- cstId - the automatically generated ID assigned to a customer
- almsUpdateTime - the time the status was last updated
- almsPayload - the payload associated with the status change of this alarm in XML format. This can be

Any event in the system that caused the status of the alarm to change. As a result this value is highly varied, but almost always is some other object already represented and documented in web services.

Example #1: To get a list of all current alarm statuses:

```
GET /ws/AlarmStatus
```

This will return a list of all of the alarms statuses for all of your configured alarms.

Example #2: To get the status details of a specific alarm ID:

```
GET ws/AlarmStatus/{almId}
```

AlarmStatusHistory

The AlarmStatusHistory web service is used to retrieve a record of alarm statuses over time. URI: http://<hostname>/ws/AlarmStatusHistory HTTP operations supported:

GET: Get a list of alarm statuses over time.

Elements include:

- id
- almId - alarm ID of the alarm
- almSourceEntityId - the specific device or source entity of this alarm status
- almsStatus - current status of the alarm
- almsTopic - the alarm topic
- cstId - the automatically generated ID assigned to a customer
- almsUpdateTime - the time the status was last updated
- almsPayload - the payload associated with the status change of this alarm in XML format. This can be

Any event in the system that caused the status of the alarm to change. As a result this value is highly varied, but almost always is some other object already represented and documented in web services.
Query parameters:

- size - number of resources to return in a GET request (default: 1000, max: 1000)
- pageCursor - page cursor returned from a previous request that can be used to retrieve the next page of

data

- startTime - the start time (inclusive)
- endTime - the end time (
- timeline - which timestamps (client or server) to use in the request (default is client). Note that roll-ups

Are only supported for the client timeline (see interval, aggregate, and time zone).

order - ascending or descending (actually checks if parameter value starts with 'asc' or 'desc')

Example #1: To get a list of all alarm statuses over time:

```
GET ws/AlarmStatusHistory
```

This will return a list of all of the alarm statuses for all of your configured alarms, over time.

Example #2: To get the alarm status history details of a specific alarm ID:

```
GET ws/AlarmStatusHistory/{almId}
```

Device Cloud Data

Introduction

The data web service provides a temporary repository for storing files for later retrieval by either the gateway or web services application. The most common usage is for gateways to post data to the data store autonomously so that a web services client application may check periodically to retrieve any new or updated contents.

GET requests

Get a file or a folder listing from storage.

GET URI format is: /{data path}[?_recursive={yes | no}] for example:

```
/ws/data/~/test?_recursive=yes
/ws/data/~/test/test.xml
```

Request content: none

Return codes:

- 200 (OK)
- 400 (Bad request) if the request is invalid
- 403 (Forbidden) if the request is an invalid path for the user
- 404 (Not found) if the path is not found
- 501 (Not implemented) if the request can not be handled because the

Response header: sets Content-Type, Last-Modified, CacheControl, Expires. For file, also sets content-length

Return content: List for a folder, contents for a file

POST requests

Put a file into storage.

POST URI format is: /{data path} for example:

```
/ws/data/~/test/test.xml
```

Request content: Multi-part file contents for a file. The first part can be a _responseformat of json if json return content is requested.

Return codes:

- 201 (Created)
- 400 (Bad request) if the request is invalid including not a multipart file
- 403 (Forbidden) if the request is an invalid path for the user
- 503 (Service unavailable) if the storage service is not available

Response header: content-type

HEAD requests

Get a file or a folder information from storage.

HEAD URI format is: /{data path}[?_recursive={yes | no}] for example:

```
/ws/data/~/test?_recursive=yes
/ws/data/~/test/test.xml
```

Request content: none

Return codes:

- 200 (OK)
- 400 (Bad request) if the request is invalid
- 403 (Forbidden) if the request is an invalid path for the user
- 404 (Not found) if the path is not found
- 501 (Not implemented) if the request can not be handled because the query parameter value is not implemented.
- 503 (Service unavailable) if the storage service is not available

Response header: sets Content-Type, Last-Modified, Content-Length

Return content: None Return content: json or xml message data

PUT requests

Put a file or a folder to storage.

PUT URI format is: /{data path}[?_type={file | folder}] for example:

```
/ws/data/~/test?_type=folder
/ws/data/~/test/test.xml?_type=file
/ws/data/~/test/test.xml
```

Request content: Ignored for folder, file contents for a file

Return codes:

- 201 (Created)
- 400 (Bad request) if the request is invalid
- 403 (Forbidden) if the request is an invalid path for the user
- 503 (Service unavailable) if the storage service is not available

Response header: No added data

Return content: None

DELETE requests

Delete a file or a folder information from storage.

DELETE URI format is: /{data path} for example:

```
/ws/data/~/test?_recursive=yes
/ws/data/~/test/test.xml
```

Request content: none

Return codes:

- 200 (OK)
- 400 (Bad request) if the request is invalid
- 403 (Forbidden) if the request is an invalid path for the user
- 404 (Not found) if the path is not found
- 503 (Service unavailable) if the storage service is not available

Response header: No added data

Return content: None

Device Cloud data streams

Format for uploading data

Attribute explanation

name (required)

The traditional DIA framework use a **{instance}.{channel}** format, so for example a gateway with 2 LTH sensors might upload samples named "indoor.temperature" and "outdoor.temperature". As a Data Stream, these will have a form such as "dia/channel/00000000-00000000-00409DFF-FF6A72F6/indoor/temperature".

value (required)

The value can be anything within quotes or as a valid XML text value. Device Cloud attempts to auto-detect the type, which can lead to problems when a string of digits is misinterpreted as a number.

type (optional)

To explicitly force a type, add the type attribute:

- bool (becomes Data Stream **Integer**)
- str (becomes Data Stream **String**, as UTF-8)

- int (becomes Data Stream **Integer**, big-endian 32-bit two's compliment)
- long (becomes Data Stream **Long**, big-endian 64-bit two's compliment)
- float (becomes Data Stream **Float**, big-endian 32-bit IEEE754 floating point)
- double (becomes Data Stream **Double**, big-endian 64-bit IEEE754 floating point)

unit (optional)

Attaches a unit of measure to the value sample.

timestamp (optional)

The timestamp should be of the simple, non-duration forms in ISO8601. The simplest is the "2011-10-03T00:05:20Z", where the trailing Z means UTC time. If the timestamp is omitted or of an invalid format, then the Device Cloud will use the server system time as of processing.

Compact format example

Compact formatting treats all data as attributes, making the XML size smaller. White-space is optional and ignored.

```
  <idigi_data compact="True">
    <sample name="home.M_AC_Power_2" value="0.2" unit="W" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_AC_Power_1" value="0.2" unit="W" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_Imported_1" value="0.04690" unit="KWh" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.offline" value="False" unit="" timestamp="2011-10-03
00:04:54Z"/>
    <sample name="home.M_Imported_2" value="0.01738" unit="KWh" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.M_AC_Voltage" value="125.2092" unit="VAC" timestamp="2011-
10-03T00:05:20Z"/>
    <sample name="home.M_AC_Current_1" value="0.06" unit="A" timestamp="2011-10-
03T00:05:20Z"/>
    <sample name="home.M_AC_Current_2" value="0.0" unit="A" timestamp="2011-10-
03T00:05:20Z"/>
  </idigi_data>
```

Full format example

Full formatting creates a larger XML size. White-space is optional and ignored.

```
  <idigi_data>
    <sample>
      <name>home.M_AC_Power_2</name>
      <value>0.2</value>
      <unit>W</unit>
      <timestamp>2011-10-02T23:58:01Z</timestamp>
    </sample>
    <sample>
      <name>home.M_AC_Power_1</name>
      <value>1.2</value>
      <unit>W</unit>
      <timestamp>2011-10-02T23:58:01Z</timestamp>
```

```
    </sample>
</idigi_data>
```

Device Cloud web services

About

Device Cloud offers a number of web services that allow users and their applications to interact with Device Cloud and their devices.

/ws/DataStream

This web service allows for a user to store time series data and retrieve it for specific time ranges, do rollups, and with the ability to apply basic aggregates.

See Device Cloud data streams.

/ws/sci

This web service allows for a user to interact with their device. They can send commands to the device, or they can query/set settings and other information. Some managements capabilities are also available, such as disconnecting a device, or updating firmware.

See SCI.

/ws/data

Devices have the ability to push up information to a temporary data store without Device Cloud. This web service is the public interface for retrieving this information from Device Cloud.

See Device Cloud Data

/ws/DeviceCore

Devices can be provisioned or removed from Device Cloud using this interface. Additionally, a list of devices available and information about them can be retrieved, or queried.

See DeviceCore.

Device Cloud creation of IA configuration

Creating IA configurations from within Device Cloud

Under the **Advanced Configuration** section of the Device Cloud device, you will find four settings groups:

- Industrial Automation Master
- Industrial Automation Routes
- Industrial Automation Serial
- Industrial Automation Table

The order is alphabetical, unrelated to how you need to enter the data. Without rehashing the entire theory of the Digi IA Engine, it consists of a series of **Masters** (acting as *servers*) which accept, time-stamp and queue requests from remote Masters (acting as *clients*). The Master task needs to locate an entity which can answer the request. It uses protocol knowledge (the Unit ID in the case of Modbus) to search **Table** consisting of **Routes**, with each route pointing to an external resource. Routes may point to onboard serial ports, IP or DNS addresses, or XBee/ZigBee nodess.

The easiest way to create a valid IA configuration is via the web UI since special wizard-like programming forces you to create the required entries in the correct order. However, you can create an IA configuration via Device Cloud. Below is an example which enables Modbus/TCP and Modbus/UDP Masters (accepting Modbus requests via IP), and routes everything to a localhost port (127.0.0.1) where a Python program acts as a Modbus slave to answer the requests.

Start under the advanced configuration

Create industrial automation table 1

We make this first to make sure any created Masters/Routes can validate their protocol is compatible with their peers. For example, a protocol X master trying to obtain responses from a protocol Y slave would fail. So the IA table acts as the mediator - if it is assigned to protocol Y, then the IA Master speaking protocol X will refuse to attach to it.

Set the following settings:

- Enable = On
- Name = SunSpec (or any text desired - it has no use other than as UI feedback)
- Family = Modbus
- Route List = 0 (0 means Industrial Automation Routes 1)
- (Save this to download to the device)

Create industrial automation master 1

Set the following settings:

- Enable = On
- Active = On
- Protocol = Modbus/TCP
- Type = TCP
- Table Index = 0 (0 means Industrial Automation Table 1)
- IP Port = 502
- (Save this to download to the device)

Create industrial automation master 2

Repeat the settings for Industrial Automation Master 1, except:

- Type = UDP
- (Save this to download to the device)

Create industrial automation routes 1

Set the following settings:

- Enable = On
- Active = On
- Protocol = Modbus/TCP
- Type = directip
- Transport = UDP (the Python code assumes Modbus/UDP to reduce resources)
- Table Index = 0
- Min/Max Protocol address = 0 and 255 - leave as is or restore
  - (Note: these 2 fields are broken - setting them works, but you will NOT see the correct values when read from the device. In the near future a single string field named 'protaddr' will be seen, in which you'll put the range as 0-255. This allows scattered ranges such as 1-5,9,20-29 as well.)
- IP Address = 127.0.0.1 (which is local-host, where Python is waiting)
- Port = 8502 (we cannot use 502 since that port is already used.
- (Save this to download to the device)

Fix the industrial automation routes 1

Hit refresh, and you'll see that the Transport and Port settings did not take properly - this is a sequence issue since the Modbus/TCP code re-initializes those values during creation of a new route. So re-enter the data:

- Transport = UDP
- Port = 8502
- (Save this to download to the device, it works this take as the route already exists)

DeviceCore

Introduction

A web service is used to add and delete devices and to get information about devices.

GET requests

Get is used to retrieve information about devices.

Request headers

Name	Value	Effect
Accept	application/json	Returns a JSON view of the data
Accept	application/xml	Returns an XML view of the data (default)
Accept	application/vnd.ms-excel	Returns an excel view of the data
Authorization	Basic {Base 64 encoded password}	Authorizes DB access

The GET URI format is:

- /DeviceCore - gets a list of all devices in the account matching the authorization credentials
- /DeviceCore/.json - gets a list of all devices in JSON format
- /DeviceCore/.xls - gets a list of all devices in Excel format
- /DeviceCore/[ID] - gets a resource for the specified ID
- /DeviceCore/[ID].json - gets a resource for the specified ID in JSON format
- /DeviceCore/[ID].xls - gets a resource for the specified ID in Excel format

Query parameters:

- start - the record number to start the results from
- size - the number of records to return
- condition - a query where condition to use to filter the results
- orderby - a column used to sort the results

Examples

http://login.etherios.com/ws/DeviceCore - will return all devices in the account matching the authorization credentials.

http://login.etherios.com/ws/DeviceCore/32 - will return the device information matching the device where ID=32 (ID is an auto-generated number in the Device Cloud database).

http://login.etherios.com/ws/DeviceCore?start=201&size=200 - will return 200 records starting with record 201.

http://login.etherios.com/ws/DeviceCore?condition=devRecordStartDate>'2010-01-17T00:00:00Z' - will return all devices added after midnight Jan 17th, 2010

http://login.etherios.com/ws/DeviceCore/?condition=devConnectwareId='00000000-00000000-00409DFF-FF3C1E0F' - will return the record for device ID "00000000-00000000-00409DFF-FF3C1E0F".

http://login.etherios.com/ws/DeviceCore/.xls - Will return device information for all devices in a spreadsheet format.

http://login.etherios.com/ws/DeviceCore/.json - Will return device information for all devices in JSON format.

More Complex Examples

http://login.etherios.com/ws/DeviceCore/?condition=dpConnectionStatus=0 - will return an XML document with the device information for all devices in a disconnected state.

http://login.etherios.com/ws/DeviceCore/.xls?condition=dpConnectionStatus=0 - will return a spreadsheet with the device information for all devices in a disconnected state.

http://login.etherios.com/ws/DeviceCore/?condition=dpTags like '%25,production%25,' - will return an XML document with the device information for all devices with the device tag "production".

http://login.etherios.com/ws/DeviceCore/.xls?condition=dpTags like '%25,production%25,' - will return a spreadsheet with the device information for all devices with the device tag "production" .

Note "%25" is the URLencoded version of the % character. The % character in this query is used as a wildcard in Java-style pattern matching and must be URLencoded. The result of "%25,production%25" is delivered (after decoding the URL) to the matching engine as "%,production%" which is a proper java match pattern looking for any records with a substring match of ",production".

---

http://login.etherios.com/ws/DeviceCore/?condition=dpTags like '%25,production%25,' and dpConnectionStatus=0 - is a compound query that will return an XML document with the device information for all devices that have the device tag "production" AND are in a disconnected state.

http://login.etherios.com/ws/DeviceCore/.xls?condition=dpTags like '%25,production%25,' and dpConnectionStatus=0 - is a compound query that will return a spreadsheet with the device information for all devices that have the device tag "production" AND are in a disconnected state.

POST request

POST is used to add devices to your inventory.

POST URI format is: /DeviceCore

Request content

XML representation of a new device OR a list of resources in the format <list>...</list> Example request content: <DeviceCore> <devMac>00:40:9D:3C:1E:4F</devMac> </DeviceCore>

Return codes

- 201 (Created) - A new resource (or list of resources) was created
- 207 (Multi-status) - A list was passed in and not all were created
- 400 (Bad request) - if the request is invalid
- 500 (Internal server error) - if the request cannot be handled due to a server error

Response header

Location contains the URI for a created resource (last resource created for a list)

Return content

XML document with a root result element containing a location element for each resource created and any errors encountered.

DELETE request

Delete is used to delete a device from your inventory.

DELETE URI format

/DeviceCore/{ID}

Example

/DeviceCore/1

Return codes

- 200 (OK) -
- 400 (Bad request) - if the request is invalid
- 500 (Internal server error) - if the request cannot be handled due to a server error

DeviceVendor

Introduction

A web service is used to add and view registered vendor ids.

GET requests

Retrieve Vendor IDs that are available to you. The use of /DeviceVendorSummary can also be used to see device types under specified vendor ids.

Request headers

Name	Value	Effect
Accept	application/json	Returns a JSON view of the data
Accept	application/xml	Returns an XML view of the data (default)
Accept	application/vnd.ms-excel	Returns an excel view of the data
Authorization	Basic {Base 64 encoded password}	Authorizes DB access

The GET URI format is:

- /DeviceVendor- gets a list of all vendor ids in the account matching the authorization credentials
- /DeviceVendor/.json - gets a list of all vendor ids in JSON format
- /DeviceVendor/.xls - gets a list of all vendor ids in Excel format
- /DeviceVendor/[ID] - gets a resource for the specified ID
- /DeviceVendor/[ID].json - gets a resource for the specified ID in JSON format
- /DeviceVendor/[ID].xls - gets a resource for the specified ID in Excel format

Query parameters:

- start - the record number to start the results from
- size - the number of records to return
- condition - a query where condition to use to filter the results
- orderby - a column used to sort the results

POST request

Register for a Vendor ID to use for device development (currently limited to one).

POST URI format is: /DeviceVendor

Request content

Elements including:

- dvVendorId – integer representation of this vendor ID
- dvVendorIdDesc – hex representation of this vendor ID

- cstId – iDigi id of customer owning this vendor ID
- dvDescription – Information describing this vendor ID
- dvRegistrationDate – When the vendor ID was registered

Return codes

- 201 (Created) - A new resource (or list of resources) was created
- 207 (Multi-status) - A list was passed in and not all were created
- 400 (Bad request) - if the request is invalid
- 500 (Internal server error) - if the request cannot be handled due to a server error

DeviceVendorSummary

Introduction

Provides a quick view of device types existing under your vendor ID.

GET requests

Can be used to see device types under specified vendor ids.

Request headers

Name	Value	Effect
Accept	application/json	Returns a JSON view of the data
Accept	application/xml	Returns an XML view of the data (default)
Accept	application/vnd.ms-excel	Returns an excel view of the data
Authorization	Basic {Base 64 encoded password}	Authorizes DB access

The GET URI format is:

- /DeviceVendorSummary- gets a list of all devices in the account matching the authorization credentials
- /DeviceVendorSummary/.json - gets a list of all devices in JSON format
- /DeviceVendorSummary/.xls - gets a list of all devices in Excel format

Query parameters:

- start - the record number to start the results from
- size - the number of records to return

- condition - a query where condition to use to filter the results
- orderby - a column used to sort the results

Elements include:

- dvVendorId – integer representation of this vendor ID
- dmDeviceType – a device type existing under this vendor ID
- dvVendorIdDesc – hex representation of the vendor ID
- cstId – iDigi id of customer owning this vendor ID
- dvDescription – Information describing this vendor ID
- dmUiDescriptorCount – Number of view descriptors that exist for this device type

List of Device Cloud Disconnect Reasons

This is not an exhaustive list, additional internal ones exist that should never be reported but exist and new ones may appear without this list being updated.

Reason Given	Explanation
*	There was a disconnect reason recorded but its become "stale" so set to this instead of empty string
Stale connection found	There is a new tcp connection to a device reporting the same device id so this connection was booted
Timed out sending firmware request to device	Timed out sending firmware request to device
Invalid Vendor ID	The vendor id provided was not 4 bytes, packet corruption likely
Vendor ID	Vendor id reported by device does not match any registered
Unexpected data in Security Layer	Unexpected opcode in security form, packet corruption likely
Invalid credentials	Device password set and device reported incorrect token
Supplied encryption form not supported	Supplied encryption form not supported
Closed after reboot	Device rebooted
Firmware update complete	Firmware update has completed so we disconnect device to reboot
RCI timeout for device	After an RCI timeout the EDP connection is closed

Reason Given	Explanation
Connection reset by peer	An RST occurred, meaning the TCP connection was severed remotely from Device Cloud. This could be from when a device or some intermediate networking equipment is killing the connection. Ex. ▪ A NAT device whose translation table expired the TCP connection from lack of sufficient keep alives, very common in cellular devices whose EDP keep alive is set to high. ▪ Device was reconfigured to a new server and a boot action=reset was executed (Digi Connect product specific)
SSL handshake failed	Bad SSL handshake, potentially man in the middle attack or bad cert
No keep-alives in X seconds	Keep alive threshold was met so device is disconnected
Invalid Device ID supplied	Device id provided in security layer did not meet expected format
Disconnect job submitted	RCI disconnect job was submitted either by SCI or webui
Reboot job submitted	RCI reboot job was submitted either by SCI or webui
Reset sent	Firmware reset command sent from SCI or webui
Redirect sent	Connection control reset sent from SCI or webui
... OR Session closed	Device was disconnected by Device Cloud server but a disconnect reason was not given, common to see around same time as a Server Reset
Server Reset	Server device connected to became unavailable. This can happen during maintenance windows or server failure but should not be a problem, the device can connect back in to another system.

Microsoft PowerShell with web services

What is PowerShell?

Powershell is a relatively new command line environment in Windows. It is intended to replace the common DOS command shell. It is a much more powerful programming environment than the old command shell, and rivals Linux command shells such as BASH in many ways. It has some very slick methods for dealing with XML, so makes it very useful for use with web Services.

On Windows 7, go to **programs/accessories/Windows Powershell/** and click **Windows PowerShell** to open a PowerShell shell window.

For an introduction to PowerShell, search with your favorite search engine for *Powershell tutorials*.

Overview

I wrote a few scripts that query Device Cloud using RESTful web services. They emulate the functionality of a PowerShell cmdlet, outputting the results of the query in a structured format. This allows you to pipe them to an output presentation such as export-csv, convertto-html, and my favorite, out-gridview. You can also filter and sort them also using one of many PowerShell's object manipulation functions.

I wrote example scripts to pull down data using the Device Core, DiaChannelDatFull, DiaChannelDatHistoryFull and XbeeAttributeDataCore APIs. All of them are near identical, with a few tweeks to the code to deal with slight differences in format. It would be trivial to modify them to parse data from any of the iDig web services calls.

Note that all of these calls are HTTP GETs. Doing POST, PUT and DELETE functions is more complicated, you will need to use some dotNET functions or execute an external program to do these.

Requirements

PowerShell 2.0

These scripts require PowerShell V2.0 or later. PS 2.0 is native on Windows 7 and Windows Server 2008 R2, but available for older Windows versions by googling 'windows powershell 2.0 download'. It is a very easy install, and completely free.

PowerShell Community Extensions 2.0

You also must install the PowerShell Community Extensions 2.0 or later. Download the PowerShell Community Extensionsd 2.0 here. Follow the simple instructions at that link to install these. The community extensions provide the get-httpresource cmdlet which makes this all super easy to do (at least for GETs).

.Net Infrastructure 3.5 Update 1

out-GridView (optional, but recommended) requires .Net Infrastructure 3.5 Update 1 or later. You probably already have this, and will find out very quickly if you do not. Try "dir | out-gridview" as a simple test. This is also easy to install, and out-gridview is well worth the effort, more information on out-gridview is in the applications section below.

Applications

So, you have the script running, and it spews out your data to the console. What good is that? PowerShell is all about "piping" the data to the next piece of code to either massage it, act on it, and/or present it.

out-gridview and other "Presentations"

out-gridview is the presentation I use the most. It is a simple GUI applet that displays grid data. It is like a slimmed down spreadsheet app. You can search, sort and filter data with ease. It is much simpler and easier than using a full spreadsheet program, It is great when I am tinkering with my DIA build, troubleshooting web apps, etc.

```
.\get-idigiDiaSnapshot.ps1 developer idigiUserName | out-gridview
```

If this is your primary application, you might even want to add " | out-gridview" to the end of the last line of any of the scripts below and you will not need to pipe it on the command line. Click here for a good help page on out-gridview.

There are quite a few other cmdlets built-in to PowerShell that might be useful, such as exportTo-html, export-csv, send-MailMessage, etc.

Sorting and Filtering

PowerShell has a lot of tools to sort, filter and manipulate data. The two I use the most often are:

Select-Object, which is used to choose what properties (columns) are output. For example, perhaps all you care about are a few attributes, try this:

```
  .\get-idigiDiaSnapshot.ps1 developer idigiusername | select-object -property
ddInstanceName, dcChannelName, dcdUpdateTime, dcdStringValue
```

The above will output just those four columns to the console. Tip: if the output is four or less properties (columns), it will automatically show it in table (columnar) format. Five or more and it will show it in list format. You can override that behavior by piping the output to format-table, format-list, or out-gridview.

Sort-Object will sort the output. The following line will sort by update time, then instance name, then channel and then sends it to the grid view.

```
  .\get-idigiDiaSnapshot.ps1 developer idigiUserName | sort-object -property
dcdUpdateTime, ddInstanceName, dcChannelName | out-gridview
```

If you look at the scripts below you will examples of both of these in action. There are quite a few other functions available which can be used to filter and manipulate your output.

The Scripts

All of these scripts have a common usage pattern:

```
  scriptname <cloud> <userid>
```

For example:

```
  get-idigiDiaSnapshot.ps1 developer jsmith
```

Both parameters are required. For security reasons, you will be prompted for the password after you run the script. If you want to run it in a non-interactive manner look into the "-credential" options in PowerShell.

Also note that several of the lines are extremely long, as is typical with PowerShell. I used a method to split the long lines of putting a space and backtick ("`") at the end of a line that is continued. PowerShell supports this method, so these scripts can be copy/pasted directly into your text editor.

get-idigiDiaSnapshot (DiaChannelDataFull)

This script will query /ws/DiaChannelDataFull. This provides the last sample for each device.

```
# Usage:  .\get-idigiSnapshot.ps1 <cloud> <username> for example: ".\get-
idigiDiaSnapshot.ps1 developer idigiUserName"
$Cloud = $args[0]
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
```

```
https://$Cloud.idigi.com/ws/DiaChannelDataFull
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.DiaChannelDataFull | Select-Object -Property @
{Name="devConnectwareId";Expression={$_.id.devConnectwareId}}, `
@{Name="ddInstanceName";Expression={$_.id.ddInstanceName}}, @
{Name="dcChannelName";Expression={$_.id.dcChannelName}}, * -ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are
# shown using select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property devconnectwareid, ddInstanceName,
dcChannelName, dcdUpdateTime, dcdStringValue, dcdFloatValue, `
dcdIntegerValue, dcdBooleanValue, dcUnits | sort-object -property
devconnectwareid, ddInstanceName, dcChannelName
```

get-idigiDiaHistory (DiaChannelDataHistoryFull)

This script will query /ws/DiaChannelDataHistoryFull. This queries the last 1000 records on your account. Depending on timing, you may get more or less samples per sample point.

```
# Usage:  .\get-idigiDiaHistory.ps1 <cloud> <username> for example: .\get-
idigiDiaHistory.ps1 developer idigiUserName
$Cloud = $args[0]
$iDigiUserID = $args[1]
# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
# The "?orderby=dcdhId%20desc" ensure we get the most recent samples.
# WIthout it you may get a lot of samples for one device, and none for others,
due to the 1000 receord maximum
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/DiaChannelDataHistoryFull/?orderby=dcdhId%20desc

# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties

$ResultTable = $XMLfile.result.DiaChannelDataHistoryFull | Select-Object -
Property @{Name="dcdhId";Expression={$_.id.dcdhId}}, `
@{Name="devConnectwareId";Expression={$_.id.devConnectwareId}}, @
{Name="ddInstanceName";Expression={$_.id.ddInstanceName}}, `
@{Name="dcChannelName";Expression={$_.id.dcChannelName}}, * -ExcludeProperty id

$ResultTable | select-object -property dcdhId, devconnectwareid, ddInstanceName,
dcChannelName, dcdUpdateTime, dcdStringValue, `
dcdFloatValue, dcdIntegerValue, dcdBooleanValue, dcUnits | sort-object -property
devconnectwareid, ddInstanceName, dcChannelName, dcdUpdateTime
```

get-idigiSmartEnergySnapshot (XbeeAttributeDataCore)

This script will query /ws/XbeeAttributeDataCore. This queries the last 1000 records on your account. Depending on timing, you may get more or less samples per sample point.

```
# Usage:  .\get-idigiSmartEnergySnapshot.ps1 <cloud> <username> for example:
".\get-idigiSmartEnergySnapshot.ps1 developer idigiUserName"
$Cloud = $args[0]
```

```
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/XbeeAttributeDataCore
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.XbeeAttributeDataCore | Select-Object -Property @
{Name="xpExtAddr";Expression={$_.id.xpExtAddr}}, `
@{Name="xeEndpointId";Expression={$_.id.xeEndpointId}}, @
{Name="xcClusterType";Expression={$_.id.xcClusterType}}, `
@{Name="xcClusterId";Expression={$_.id.xcClusterId}}, @
{Name="xaAttributeId";Expression={$_.id.xaAttributeId}}, * `
-ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are shown using
# select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property devConnectwareId, xpExtAddr, xeEndpointId,
xcClusterType, xcClusterId, xaAttributeId, `
cstId, xeProfileId, xeDeviceId, xeDeviceVersion, xaAttributeType,
xadAttributeStringValue, xadAttributeIntegerValue, xadUpdateTime `
| sort-object -property devconnectwareid, xeEndpointId, xcClusterId
```

get-idigiDeviceCore (DeviceCore)

This script will query /ws/DeviceCore. This provides basic information about each gateway, including status of the iDigi connection.

```
# Usage:  .\get-idigiDeviceCore.ps1 <cloud> <username> for example: ".\get-
idigiDeviceCore.ps1 developer idigiUserName"
$Cloud = $args[0]
$iDigiUserID = $args[1]

# This will import the PowerShell Community Extensions
import-module Pscx
# First, we download the XML file and save it as an XML typed variable.
# XML typing does some magical things as far as structuring the data
[xml]$XMLfile = get-httpresource -credential $iDigiUserID
https://$Cloud.idigi.com/ws/DeviceCore
# Import it as an array, with the properties from the ID subelement in the array
at the same level as other properties
$ResultTable = $XMLfile.result.DeviceCore | Select-Object -Property @
{Name="devId";Expression={$_.id.devId}}, `
@{Name="devVersion";Expression={$_.id.devVersion}}, * -ExcludeProperty id

# You can also manipulate it like any other PS Object.  For example, you might
want to limit which properties are
# shown using select-object and sort that with sort-object.
# I am using select-object here to eliminate a bunch of junk properties that PS
adds because of the XML typing.
$ResultTable | select-object -property  devConnectwareId, devId, devVersion,
devRecordStartDate, devMac, `
cstId, grpId, devEffectiveStartDate, devTerminated, dvVendorId, dpDeviceType,
```

```
dpFirmwareLevel, `
dpRestrictedStatus, dpLastKnownIp, dpGlobalIp, dpConnectionStatus,
dpLastConnectTime, `
dpContact, dpDescription, dpLocation, dpServerId, dpZigbeeCapabilities,
dpCapabilities, dpUserMetaData, dpTags `
| sort-object -property devConnectwareid dpConnectionStatus
```

PHP to Device Cloud

PHP To Device Cloud Example

Below is a simple PHP script performing a POST then a GET Request to Device Cloud and displaying the response obtained.

```
<?php
    $username = 'YourUsername'; # enter your username
    $password = 'YourPassword'; # enter your password

    $url = "http://login.etherios.com/ws/DataPoint";
    $myStreamID ='device1/temp';

    $myData = 42;
    $message ='
    <DataPoint>
      <data>'. $myData .'</data>

      <!-- Everything below this is optional -->
      <streamId>' . $myStreamID . '</streamId>
      <description>Temperature at device 1</description>
      <location>0.0, 0.0, 0.0</location> <!-- lat, long, elevation -->
      <quality>99</quality>
      <!-- <timestamp>[epoc long or ISO8601, if not set defaults to
now]</timestamp>  -->

      <!-- The following sets the data stream the data point belongs to (also
optional) -->
      <streamType>float</streamType>
      <streamUnits>Kelvin</streamUnits>
      <streamForwards>allDevices/temp, allDevices/metrics</streamForwards>
    </DataPoint>
    ';

    // check http://php.net/manual/en/function.curl-setopt.php for more options

    // create a new cURL resource
    $ch = curl_init();

    //======================== POST - Example====================
    // set URL and authentication
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_USERPWD, "$username:$password");

    // settings for a regular HTTP POST
    curl_setopt($ch, CURLOPT_POST, 1);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $message);
```

```
    // process request
    $output = curl_exec($ch);
    echo $output;  // output result from POST

    //======================= GET - Example===================

    // changing the url to make a get request
    curl_setopt($ch, CURLOPT_URL, $url.'/'. $myStreamID);
    curl_setopt($ch, CURLOPT_HTTPGET, 1);

    // process request
    $output = curl_exec($ch);
    echo $output; // output result from Get

    // close cURL resource, and free up system resources
    curl_close($ch);
?>
```

RCI Descriptor

Overview

Remote Command Interface (RCI) is a mechanism designed to allow remote configuration, control, and information exchange between an RCI client (typically a web services client acting via Device Cloud) and an RCI target (typically a Digi device implementing the RCI specification). RCI consists of a transport mechanism (such as the Device Cloud device protocol, EDP) and an XML based request and reply document specification. RCI allows a user to:

- Inspect and configure device settings
- Inspect and configure device state (such as inspecting network statistics or setting the level of a GPIO pin)
- Reboot a device
- Configure XBee networks via Digi gateways
- Device file system operations (list, get, put, delete)
- Send requests and retrieve replies from dynamically registered agents such as Python programs

The primary documentation for RCI is the RCI Specification (part number: 90000569).

RCI descriptors are XML documents that describe the content of RCI request and response documents. They describe all of the commands and parameters that RCI offers, as well as all of the configurable settings and state.

RCI descriptors exist for a particular device. This allows descriptors to be exact for a given RCI target. Descriptors are constant given a certain set of variables, so they can be cached on a more coarse scale than per device. Ideally, only the EDP device type and version would determine a unique descriptor.

The intention of the RCI descriptor is that it should stand on its own. That is, the contents of the descriptor needs to be complete enough such that a program walking the descriptor can identify all valid commands and options and all allowed configurable settings and state and all of the options available for the allowed configurable settings and state. The goal is that this program should be able to display a generic GUI to a user using the info in the descriptor. This generic GUI should be complete

in RCI function and be able to configure all supported settings and state in a reasonably user friendly way.

This generic GUI will be referred to throughout this document as the prototypical RCI descriptor consumer. The generic GUI is not part of this specification. It is assumed that a production quality GUI built from descriptors will have additional customized features not found in the descriptors. Furthermore, there may be exceptions to the above because of practical considerations (for instance, some settings may be too complex to allow the descriptors to be fully self-describing. In this case, extra knowledge will be needed by the consumer of the descriptor. These exceptions will be made obvious though, by for instance, use of “custom” types.

Note: the goal of RCI descriptors is not to replicate the device’s Web interface (if it has one). The look and feel and content of a generic GUI will be different than the device’s web UI.

In addition to the goals above, other goals of descriptors:

- Keep the design as simple as possible. Do not attempt to capture the outlier cases.
- Make exceptions to rules obvious, for instance by use of “custom” types
- Structure descriptors so that generating descriptors is relatively straightforward. Firmware engineers will be the primary writers of the descriptors. Descriptors need to be friendly to the embedded environment.

Practical Descriptors

RCI descriptors will be retrievable directly from the target device. This is highly desirable since no a priori knowledge of a target device is needed. Also, often the allowed parameters for a command may be determined in firmware, so the same logic can be used to generate the descriptor thus eliminating the effort in creating a separate descriptor and the possibility of getting the descriptor and actual implementation out of sync.

However, due to practical considerations on the device such as RAM and/or Flash availability, it is not required that descriptor be dynamically accessible from the target. If a device does not allow for dynamic retrieval of descriptors, the descriptor for that target must be separately accessible and provisioned into Device Cloud (by EDP device type and firmware level). RCI descriptors are required for Deivce Cloud support.

The following discussion assumes that a device supports dynamic retrieval of its RCI descriptor. A user may determine if a device supports dynamic retrieval of descriptors by issuing the following RCI request:

```
<rci_request version="1.1">
  <query_descriptor/>
</rci_request>
```

If the device responds with an error (<error>) the device does not support descriptors. For instance, the expected reply for the above command on a device that does not support descriptors is:

```
<rci_reply version="1.1">
  <error id="3" desc="Unknown command" hint="query_descriptor"/>
</rci_reply>
```

An example response by a device that does support retrieving descriptors is:

```
<rci_reply version="1.1">
  <query_descriptor>
    <descriptor element="rci_request" desc="Remote Command Interface request">
      <attr name="version" desc="RCI version of request.  Response will be
```

```
returned in this versions response format" default="1.1">
        <value name="1.1" desc="Version 1.1"/>
      </attr>
      <descriptor element="query_setting" dscr_avail="true"/>
      <descriptor element="set_setting" dscr_avail="true"/>
      <descriptor element="set_factory_default" dscr_avail="true"/>
      <descriptor element="reboot" dscr_avail="true"/>
      <descriptor element="set_state" dscr_avail="true"/>
      <descriptor element="query_state" dscr_avail="true"/>
      <descriptor element="do_command" dscr_avail="true"/>
      <error_descriptor id="1" desc="Request not valid XML"/>
      <error_descriptor id="2" desc="Request not recognized"/>
      <error_descriptor id="3" desc="Unknown command"/>
    </descriptor>
  </query_descriptor>
</rci_reply>
```

Using the above as a model, one can see that descriptors supply the following information:

- Information about an RCI element level (“rci_request” in this case), including:
  - A short description of the element (the desc=”” attribute)
  - All attributes supported for that element along with descriptions for the attributes and any valid values.
- Elements that are supported as children of the current element. These will either be a full descriptor themselves or will be marked with a dscr_avail=”true” attribute to indicate that a descriptor is available for that sub-element.
- All errors and warnings (“error_descriptor”) that can be received for that element along with the error id and description for that error

With this information, the RCI consumer can fully interact with the rci_request element level. The next step is to request any descriptors that are missing (ie. For any dscr_avail=”true” descriptors). For instance, the following request would be made next:

```
<rci_request version="1.1">
  <query_descriptor>
    <query_setting/>
  </query_descriptor>
</rci_request>
```

The device responds with:

```
<rci_reply version="1.1">
  <query_descriptor>
    <descriptor element="query_setting" desc="Retrieve device configuration">
      <attr name="source" desc="Source of content of reply" default="current">
        <value name="current" desc="Current device settings"/>
        <value name="stored" desc="Stored device settings"/>
        <value name="defaults" desc="Device defaults"/>
      </attr>
      <error_descriptor id="1" desc="Setting group unknown"/>
      <error_descriptor id="2" desc="Element not allowed under field element"/>
      <error_descriptor id="3" desc="Invalid setting group, index combination"/>
      <error_descriptor id="4" desc="Invalid setting group, name combination"/>
      <descriptor element="static_position" desc="GPS static position
configuration">
        <element name="state" desc="GPS static setting enable" type="on_off"
```

```
default="off"/>
        <element name="latitude" desc="Latitude" type="float" min="-90" max="90"
default="0"/>
        <element name="longitude" desc="Longitude" type="float" min="-180"
max="180" default="0"/>
        <error_descriptor id="1" desc="Internal error (save failed)"/>
        <error_descriptor id="2" desc="Not enough permissions for specified
field"/>
        <error_descriptor id="3" desc="Field specified does not exist"/>
        <error_descriptor id="4" desc="Field specified is read-only"/>
        <error_descriptor id="5" desc="Invalid state value"/>
        <error_descriptor id="6" desc="Invalid latitude value"/>
        <error_descriptor id="7" desc="Invalid longitude value"/>
      </descriptor>
    </descriptor>
  </query_descriptor>
</rci_reply>
```

This example shows the descriptor for the query_setting command (with all settings group omitted except one example). Note:

- This descriptor contains no <descriptor> elements with dscr_avail="yes" attributes, so there are no other descriptor available making this the last descriptor the client needs to retrieve for this branch of the descriptor tree. Note, the choice to return dscr_avail="yes" and support independently retrievable descriptors is an implementation choice on the target. Descriptor clients must be prepared to receive the entire descriptor tree from the initial request (<rci_request version="1.1"><query_descriptor/></rci_request>).
- Just as <error> elements are context sensitive, the <error_descriptor> elements location in the xml hierarchy is significant. That is, errors appearing within a descriptor are found in the corresponding level in RCI. For example, if an error id="1" was received as a child of static_position:

  ```
  <rci_reply version="1.1">
    <query_setting>
      <static_position>
        <error id="1"/>
      </static_position>
    </query_setting>
  </rci_reply>
  ```

The error id="1" would refer to: Internal error (save failed), not Setting group unknown.

Assembling a set of descriptors into a single descriptor tree

When a descriptor contains dscr_avail="yes", the client needs to make multiple requests to gather the entire descriptor recursively until no more dscr_avail="yes" attributes are encountered. The resulting set of documents can be formed into a single descriptor by replacing each <descriptor dscr_avail="yes"> with the full descriptor retrieved, or by adding the child descriptor as a child of elements that are not <descriptor>.

Descriptor Use Notes

- As is the case with the rest of RCI, clients need to be prepared to parse responses that contain more information than the minimum asked for. For instance, if a caller requests the descriptor for do_command

```
<rci_request version="1.1">
  <query_descriptor>
    <do_command type="descriptor"/>
  </query_descriptor>
</rci_request>
```

  it is acceptable for the device to return: the descriptor for do_command only, do_command and all of its sub-commands, do_command and all of its peer commands (set_setting, reboot, etc), the entire device descriptor, or an error indicating the device does not support specific command descriptors. It is the responsibility of the client to parse the response for the specific descriptor desired in the response. However, the correct client behavior is to query the root descriptor and only recurse if dscr=”true” is returned. If a client requests a specific command descriptor and the device does not declare it supports it (with dscr_avail=”true”) it is recommended that the device return an error.

- There are cases where settings validation is complex (for instance, when one field’s valid values change based on another field’s settings). Although it is possible to make descriptors flexible enough to handle arbitrarily complex settings, the trade-off is clarity and simplicity of descriptors. The intention of descriptors is not to attempt to describe the exact behavior of firmware validation. Instead descriptors need to describe enough so that:
- All operations, commands, settings/state groups, fields, and values are described.
- When the device performs complex validation that is not practical to describe in the descriptor, the descriptors must describe the full set of values possible so that users have available to them the full range of configuration, even if some values are further restricted beyond what the descriptor mentions. When this happens, descriptions in the descriptor that note this are recommended.
- Ultimately, the device must reject invalid values and the RCI client must be written to handle this correctly by not relying exclusively on descriptors.

RCI request

Remote Command Interface (RCI) is a method for remote clients to control, configure, and gather statistics from Digi Connect devices. RCI is a stateless, request/response protocol. RCI uses XML and HTTP to exchange data between clients and Digi devices.

RCI over HTTP

RCI requests are sent to the device using an URI of UE/rci. For example, if the Digi Device’s IP address is 192.168.1.1, then RCI requests are sent to http://192.168.1.1/UE/rci.

RCI requests are sent as an HTTP POST with the XML request of the form specified in this document. Note, due to space limitations on the device, the largest request that can be processed is 32KB. If a

request is larger than this, it must be split into multiple RCI requests. RCI replies from the device are not subject to this limit.

Security is handled in the usual HTTP mechanism. The username and password must be passed to the device in the header of each HTTP request.

RCI over serial

RCI requests can also be sent over the serial port. This is useful in scenarios where a master processor is connected to the Digi Device through a serial port. This allows the master processor to configure the Digi Device as part of its configuration process, so that a separate manual configuration step for the Digi Device is eliminated. You must enable 'RCI over Serial' in either the Web Interface or the Command Line Interface before the Digi Device will accept RCI requests and return replies. The RCI over Serial option is available only on the primary port. RCI over Serial uses the DSR (Data Set Ready) serial signal. Verify that the serial port is not configured for autoconnect, modem emulation, or any other application which is dependent on DSR state changes. Note: When the Digi Device sees its DSR raised, it will set the serial port settings to 9600 baud, 8 data bits, no parity, and 1 stop bit. When DSR is lowered, the Digi Device will restore the previous serial settings.

Configure using the Command Line Interface (CLI)

1. Access the CLI using telnet or rlogin and the module's IP address. Ex:

```
telnet 192.168.1.2 -or-
rlogin 192.168.1.2
```

2. At the command prompt type:

```
 #> set rciserial state=on
```

Configure using the web user interface

1. Access the web interface by entering the module’s IP address in a browser’s URL window.
2. Choose Serial Ports from the Configuration menu.
3. If the device has more than one port, select Port 1.
4. If a port profile has not been selected, select Custom and click Apply.
5. Select Advanced Serial Settings.
6. Select Enable RCI over Serial (DSR) and click Apply.

RCI request/reply

An RCI XML document is identified by the XML elements rci_request and rci_reply. An RCI request specifies the XML element “rci_request” optionally with a version number. The version should match the version of RCI the client expects. The current RCI version is 1.1. If a version is not specified, the RCI version of the device is used to form the reply. Not specifying a version can cause problems when communicating with devices at different RCI versions, if the client code is not written in a version independent way. Therefore, it is highly recommended to always supply the version of RCI in requests, unless the client code has been designed to be version independent. Example of a request element:

```
<rci_request version="1.1">
```

The device will respond to requests with the element "rci_reply" along with the version number as an attribute. Example reply:

```
<rci_reply version="1.1">
```

rci_reply errors

Errors that occur at the request level will result in an error element as a sub-element of the <rci_reply>. Errors and warnings are explained below <rci_reply> errors: Error ID Description

1. Request not valid XML
2. Request not recognized
3. Unknown command

Command

The command section of the protocol indicates the action requested (or action performed in replies).

Commands are specified as sub-elements to <rci_request> and <rci_reply>.

This example requests all configuration settings:

```
    <rci_request version="1.1"> <!—Identifies the protocol and whether this is a
request or a response --
        <query_setting/> <!— request config of device -->
    </rci_request>
```

This example requests the configuration information for just boot settings and serial settings.

```
    <rci_request version="1.1">
        <query_setting>
            <boot/>
            <serial/>
        </query_setting>
   </rci_request>
```

Supported commands

COMMAND	REQUEST DESCRIPTION	RESPONSE DESCRIPTION
query_setting	Request for device settings. May contain setting group elements to subset query (only setting group subset supported. Subsetting below this level not supported).	Returns requested config settings. Requests specifying no settings groups (eg. <query_setting/>) return all settings.
set_setting	Set settings specified in setting element. Settings data required.	Empty setting groups in reply indicate success. Errors returned as specified below.

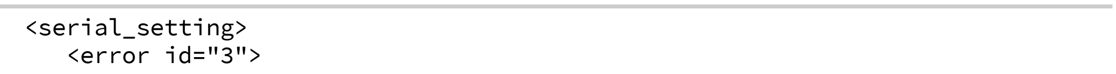
COMMAND	REQUEST DESCRIPTION	RESPONSE DESCRIPTION
query_state	Request current device state such as statistics and status. Sub-element may be supplied to subset results.	Returns requested state. Requests specifying no groups (eg. <query_state/> return all state.
set_factory_default	Sets device settings to factory defaults. Same semantics as set_setting.	Same semantics as set_ setting.
reboot	Reboots device immediately.	None
do_command	see RCI do command	see RCI do command

Errors and warnings

Response documents may contain an element as a child of the command or data element that indicates the result of the request. More than one error or warnings may be present. Error and Warning elements:

error	An error occurred.	Attribute id: A numeric id specified by the parent element (the command or the data element). An error element id="0" is equivalent to no error.	Children Elements **name** / **desc** desc: Optional - Text description of the error. hint: Optional - Used to indicate to the client the source of the error. This will typically be set to the field name that the error.
warning	Command executed, but a warning was issued.		

Example:



```
<serial_setting>
   <error id="3">
```

```
            <hint>baud</hint>
            <desc>Value out of valid range.</desc>
        </error>
         </serial_setting>
```

Errors are required to have an id. <hint> and <desc> are optional and more than one are allowed.

Notes

RCI XML must be well-formed XML

The device parses incoming RCI requests in a sequential manner. Each XML element is parsed and acted upon as it arrives. This is not ideal behavior, but is necessary because of the inherent resource limitations of a device. Ideally, the entire XML request would be read into memory, validated, parsed and acted upon only after validation.

XML structure errors may be found after actions have been taken. For instance:

```
    <rci_request version=”1.0”>
        <set_factory_default/>
    </rci_requestBADENDTAG>
```

This request will result in an XML parse error, but since the parse error occurs after the set_factory_ defaults, the device will be set to factory defaults. Therefore, it is highly recommended that RCI requests be validated with an XML parser before being sent to the device. Using any standard parsers, such as the XML parsing in the Java SDK, to form RCI requests accomplishes this.

XML structure characters must not be sent as character data

Care must be taken to avoid accidental badly formed XML in RCI requests because of including XML structure characters, such as "<", as user entered data. Any field that accepts character data must be checked to ensure that "<" and ">" are not present (fields such as the email body of an alarm are common places this can happen). It is recommended that all instances of "<" and ">" in character data be converted to "<" and ">", which is the standard XML representation(entities) of these characters.

To use RCI to Query DIA device/channel Information

Reading device/channel information by direct HTTP to a DIA device requires a different do_command set. See Simple RCI by HTTP for working code examples.

References

https://www.digi.com/search/results?q=900005www.digi.com/search/results?q=900005690069

RCI do command

Python

You can add callbacks to unhandled do_commands target via the rci Python module.

File System

The file_system commands are accessed via the do_command of an rci request.

ls

attributes: dir

Reports all files in a given directory

dir: the path in which to list available files

Example:

```
<rci_request version="1.1">
  <do_command target="file_system">
    <ls dir="/WEB/Python"/>
  </do_command>
</rci_request>
```

returns

```
<rci_reply version="1.1">
  <do_command target="file_system">
    <ls dir="/WEB/Python">
      <file name="Python.zip" size="144321"/>
      <file name="digi_daq.zip" size="458980"/>
      <file name="digi_daq.yml" size="5270"/>
      <file name="digi_daq.py" size="6387"/>
      <file name="zigbee.py" size="1147"/>
    </ls>
  </do_command>
</rci_reply>
```

get_file

attributes: name

returns the base 64 encoded raw data from a file in the file system denoted by the name attribute

name: name of the file including path

Example:

```
<rci_request version="1.1">
    <do_command target="file_system">
        <get_file name="/WEB/Python/Python.zip"/>
    </do_command>
</rci_request>
```

returns

```
<rci_reply version="1.1">
    <do_command target="file_system">
      <get_file name="/WEB/Python/Python.zip">
         <data>UEs...KYmwaR</data>
      </get_file>
    </do_command>
</rci_reply>
```

put_file

attributes: name

uploads a base 64 encoded file onto the device

name: path and destination filename for the file

Example:

```
<put_file name="/WEB/destination.txt">
  <data>BASE64DATA</data>
</put_file>
```

rm

attributes: name

removes a given file

Example:

```
<rm name="/WEB/Python/somefile.py"/>
```

Zigbee

The zigbee rci command interacts with a xbee module.

ZigBee discover (<discover/>)

Optional attributes: start, size, clear

Returns back a list of discovered nodes, with the first indexed node being the node in the gateway.

- start: says the rci should return the nodes whose index is >= start. For some reason, if start > 1, the Gateway will return this list from cache, and not perform an actual discovery.
- size: Determines number of nodes to return
- option: If this is set to "clear", it forces a clearing of the device’s cache, and will always perform a discover to get fresh results

Example:

```
<do_command target="zigbee">
  <discover start="1" size="10" option="clear"/>
</do_command>
```

ZigBee query setting (<query_setting/>)

Optional attributes: addr

Returns back a detailed list of settings for a given radio

addr: 64 bit address of the node to retrieve settings for. If omitted, defaults to gateway node

Example:

```
<do_command target="zigbee">
  <query_setting addr="00:13:a2:00:40:34:0c:88!"/>
</do_command>
```

ZigBee query state (<query_state/>)

This is identical to query_setting, except it returns back different fields.

ZigBee set setting (<set_setting/>)

Optional attributes: addr

Basically the reverse of query_setting, so you can set settings for a particular node

addr: 64 bit address of node to set settings for. If omitted, defaults to gateway node

Example:

```
<do_command target="zigbee">
  <set_setting addr="00:13:a2:00:40:34:0c:88!">
      <radio>
          <field1>value1</field1>
          …
          <fieldn>valuen</fieldn>
      </radio>
  </set_setting>
</do_command>
```

ZigBee firmware update (<fw_update/>)

Required attribute: file

Updates the firmware of the radio in the gateway

file: Path to a firmware file which must already exist on the gateway

Example:

```
<do_command target="zigbee">
  <fw_update file="/WEB/firmware_file"/>
</do_command>
```

SCI

Introduction

SCI (Server Command Interface) is an interface of Device Cloud that allows users to access information and perform commands that relate to their device. Examples of these requests include retrieving and setting configuration parameters, updating firmware, retrieving and updating files, working with XBee devices, and others.

The operations can be performed synchronously or asynchronously. Synchronous requests are useful if you would like to execute a short request to the server and block until the operation is completed. Asynchronous requests are useful when you want to execute a request that has the possibility of taking a while to complete, or you simply want to send the request off and return immediately. With asynchronous requests, you will receive an id that you can later use to check on the job status and retrieve results.

An SCI request is composed of XML that is POSTed to http(s)://{idigi-platform-url}/ws/sci.

In addition to the information provided in this article, a useful reference will be the Web Services Reference under the documentation section of http://developer.idigi.com.

Example request

The following examples will be based on URLs that are appropriate if you are using Device Cloud server at http://developer.idigi.com

Synchronous

Sending the request and getting the response

POST the following to http://developer.idigi.com/ws/sci.

```
<sci_request version="1.0"> <!-- common to every sci request -->
  <send_message> <!-- indicates we want to send an rci request -->
```

```
      <targets> <!-- preparing us for the list of who to operate on -->
        <device id="00000000-00000000-00409DFF-00000000"/> <!-- we will send it
to this device -->
      </targets>
      <rci_request version="1.1"> <!-- the payload for the send_message command,
an rci request -->
        <query_state><device_stats/></query_state>
      </rci_request>
    </send_message>
  </sci_request>
```

which will return when the operation has completed (or timed out) and the body of the response will be:

```
  <sci_reply version="1.0"> <!-- start of the sci response -->
    <send_message> <!-- the "operation" of our sci_request -->
      <device id="00000000-00000000-00409DFF-FF374CD3"> <!-- contains the
response for this device -->
        <rci_reply version="1.1"> <!-- the rci response for the particular device
-->
          <query_state>
            <device_stats>
              <cpu>36</cpu>
              <uptime>152</uptime>
              <datetime>Thu Jan 1 00:02:32 1970 (based on uptime)</datetime>
              <totalmem>8388608</totalmem>
              <usedmem>5811772</usedmem>
              <freemem>2576836</freemem>
            </device_stats>
          </query_state>
        </rci_reply>
      </device>
    </send_message>
  </sci_reply>
```

Asynchronous

Sending the request and getting a job id

POST the following to http://developer.idigi.com/ws/sci

```
  <sci_request version="1.0"> <!-- common to every sci request -->
    <send_message synchronous="False"> <!-- the "operation" with an attribute
specifying it will be asynchronous -->
      <targets> <!-- preparing us for the list of who to operate on -->
        <device id="00000000-00000000-00409DFF-00000000"/> <!-- we will send it
to this device -->
      </targets>
      <rci_request version="1.1"> <!-- the payload for the send_message command,
an rci request -->
        <query_state><device_stats/></query_state>
      </rci_request>
    </send_message>
  </sci_request>
```

```
which will return without waiting for the request to complete, and return an in
the body as:
```

```
<sci_reply version="1.0">
  <send_message>
    <jobId>883492</jobId>
  </send_message>
</sci_reply>
```

Using the id to check on status

Perform an HTTP GET at http://developer.idigi.com/ws/sci/883492.

which returns the job status and available results:

```
<sci_reply version="1.0">
  <status>complete</status>
  <send_message>
    <device id="00000000-00000000-00409DFF-FF374CD3">
      <rci_reply version="1.1">
        <query_state>
          <device_stats>
            <cpu>36</cpu><uptime>152</uptime>
            <datetime>Thu Jan  1 00:02:32 1970 (based on uptime)</datetime>
            <totalmem>8388608</totalmem>
            <usedmem>5811772</usedmem>
            <freemem>2576836</freemem>
          </device_stats>
        </query_state>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

Anatomy of an SCI request

Every SCI request looks like the following:

```
<sci_request version="1.0">
  <{operation_name}>
    <targets>
      {targets}
    </targets>
    {payload}
  </{operation_name}>
</sci_request>
```

operation_name is either send_message, update_firmware, or disconnect

targets contains one or more elements that look like: <device id="{device_id}"/>

payload is dependent on the operation

Available operations

Three main operations are currently available. This include send_message, update_firmware, and disconnect. send_message allows an RCI request to be sent to the device (or the server cache). update_firmware updates the firmware of the device. disconnect sends a request to the device to disconnect from the server.

There are a few attributes that can be specified for an operation that can tweak the behavior.

They include:

<{operation_name} **reply=**"*all|errors|none*">

<{operation_name} **synchronous=**"*true|false*">

<{operation_name} **syncTimeout=**"*xxx*">

<{operation_name} **cache=**"*true|false|only*">

<{operation_name} **allowOffline=**"*true|false*">

<{operation_name} **waitForReconnect=**"*true|false*">

reply determines how much information should be saved in the response to a request. *all* means that everything should be saved. (default) *errors* implies that only errors in the response should be kept, while success messages should be stripped. none means that result data for the operation should be stripped.

errors is useful if you are performing an operation and only want to be error information that occurred, such as when setting settings, or performing firmware update. none is useful when you aren't concerned with the reply at all. If you are performing a synchronous request because you want to wait until the operation is complete, but do not want to receive a bunch of data in the reply, this would accomplish that.

synchronous determines whether the request should be synchronous (default), or asynchronous (if false)

syncTimeout is applicable for a synchronous request and determines how long to wait for the request to complete (in seconds) before an error is returned

cache determines if the request should be processed on the server if possible, or always sent to the device. *true* means that if possible, the request will be processed on the server without being sent to the device (default). If it cannot, it will be sent to the device. *false* means that the request will bypass the server and be sent on to the device. only means that the request should only be processed by the server, and will never be sent to the device, even if the server is not capable of servicing the request.

waitForReconnect mode requests are a way to allow the completion status of an SCI command to depend on a device connect.

For instance, performing a reboot command without specifying waitForReconnect="true" causes an immediate return of success or failure. This is counter-intuitive since we may expect the completion status to depend on the device fully reconnecting.

Use with caution, as many commands do not directly result in a disconnect/reconnect to occur.

allowOffline requests enable you to send a request to a disconnected device via an SCI request. If the device is already connected, it will process the request right away. If the device is not connected, it will process the request as soon as the device connects.

send_message

This is used to send an RCI request to a device. For more information, see RCI request.

One of the main uses of RCI requests are to interact with the settings and state of a device. The Device Cloud keeps a cache of the latest settings and state that it has received from a device, and this makes it possible to retrieve information about the state or settings of a device without having to go to the device.

The format of the send_message command is as follows:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      {targets}
    </targets>
    <rci_request version="1.1">
      <!-- actual rci request -->
```

```
    </rci_request>
  </send_message>
</sci_request>
```

update_firmware

This is used to update the firmware of one or more devices.

The data for the firmware image is base64 encoded and included with the request. There are optional attributes filename, and firmware_target, which are included with the update_firmware element.

filename needs to be specified if your target device supports multiple targets that can be updated in order to choose which to upgrade. These will match patterns specified by the device which can be discovered using the query_firmware_targets command.

firmware_target can be used to bypass the filename matching and force an upgrade on a particular target.

A request looks like:

```
<sci_request version="1.0">
  <update_firmware>
    <targets>
      {targets}
    </targets>
    <data>{base64 encoded firmware image}</data>
  </update_firmware>
</sci_request>
```

and the reply looks like:

```
<sci_reply version="1.0">
  <update_firmware>
    <device id="00000000-00000000-00409DFF-000000">
      <submitted/>
    </device>
  </update_firmware>
</sci_reply>
```

disconnect

Disconnect is used to indicate that a device should disconnect from the server. Based on the device's configuration, it will likely reconnect.

A request follows this format:

```
<sci_request version="1.0">
  <disconnect>
    <targets>
      {targets}
    </targets>
  </disconnect>
</sci_request>
```

and a response looks like:

```
<sci_reply version="1.0">
  <disconnect>
    <device id="00000000-00000000-00409DFF-FF374CD3">
      <disconnected/>
    </device>
```

```
  </disconnect>
</sci_reply>
```

query_firmware_targets

Query Firmware Targets is used to retrieve information about the upgradeable firmware targets of a device. It will return the target number, name, version, and code size. A pattern may also be returned in the response which indicates a regular expression that is used to determine the appropriate target for a given filename.

A request follows this format:

```
<sci_request version="1.0">
  <query_firmware_targets>
    <targets>
      {targets}
    </targets>
  </query_firmware_targets>
</sci_request>
```

and a response looks like:

```
<sci_reply version="1.0">
  <query_firmware_targets>
    <device id="00000000-00000000-00409DFF-FF374CD3">
      <targets>
        <target number="0">
          <name>Firmware Image</name>
          <pattern>image\.bin</pattern>
          <version>7.5.0.11</version>
          <code_size>2162688</code_size>
        </target>
        <target number="1">
          <name>Bootloader</name>
          <pattern>rom\.bin</pattern>
          <version>0.0.7.5</version>
          <code_size>65536</code_size>
        </target>
        <target number="2">
          <name>Backup Image</name>
          <pattern>backup\.bin</pattern>
          <version>7.5.0.11</version>
          <code_size>1638400</code_size>
        </target>
      </targets>
    </device>
  </query_firmware_targets>
</sci_reply>
```

file_system

The file system command is used to interact with files on a device. This interface is for use with devices supporting the file system service (as opposed to other devices which support file system interaction through RCI requests).

Commands have the following general format:

```
<sci_request version="1.0">
  <file_system>
```

```
    <targets>
      {targets}
    </targets>
  <commands>
    {one or more file_system commands}
  </commands>
  </file_system>
</sci_request>
```

Support file system commands are as follows.

put_file

The put_file command is used to push new files to a device, or optionally write chunks to an existing file.

path is a required attribute giving the file to write to.

offset is an optional attribute specifying the position in an existing file to start writing at.

truncate is an optional attribute indicating the file should be truncated to the last position of this put.

The payload is specified in one of two ways:

child element **data** with the payload base64 encoded

child element **file** with a path to a file in storage to send

Example

A put file operation using a file on the server as the source for the data. The contents will be inserted into the file /path_to/write1.ext, as offset 200. It is set to not truncate the file if it extends beyond the length of written data.

```
<sci_request version="1.0">
  <file_system>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <commands>
      <put_file path="/path_to/write1.ext" offset="200" truncate="false">
        <file>~/referencedfilename.xml</file>
      </put_file>
    </commands>
  </file_system>
</sci_request>
```

A put file with the data base64 encoded and embedded in the request under the data element. Offset and truncate are not specified, so this example will create a new file if one does not exist, or overwrite an existing one.

```
<sci_request version="1.0">
  <file_system>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <commands>
      <put_file path="/path_to/write2.ext">
        <data>ZmlsZSBjb250ZW50cw==</data>
      </put_file>
    </commands>
```

```
    </file_system>
  </sci_request>
```

get_file

The get_file command is used to retrieve a file from the device, either in its entirety or in chunks. There is a currently a restriction such that the maximum retrieval size is 512KB. As a result, files greater than this size will have to be retrieved in chunks.

path is a required attribute giving the file to retrieve.

offset is an optional attribute specifying the position to start retrieving from.

length is an optional attribute indicating the length of the chunk to retrieve.

Example

The get file in this example will retrieve 64 bytes starting at offset 100 from the file /path_to/file.ext. Leaving off offset and length would cause the full file to be retrieved.

```
<sci_request version="1.0">
  <file_system>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <commands>
      <get_file path="/path_to/file.ext" offset="100" length="64"/>
    </commands>
  </file_system>
</sci_request>
```

ls

The ls command is used to retrieve file listings and details.

path is a required attribute specifying where to get file details for.

hash is an optional attribute which indicates a hash over the file contents should be retrieved. Values include none, any, md5, and crc32. any is used to indicate the device should choose its best available hash. md5 or crc32 may be specified but the device may not support them (or possibly any hash mechanism at all).

Example

This ls request will return a listing of the contents of /path_to_list. By specifying hash="any", the response will include the most optimal hash available, if any. Leaving off the hash attribute will default it to none.

```
<sci_request version="1.0">
  <file_system>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <commands>
      <ls path="/path_to_list" hash="any" />
    </commands>
  </file_system>
</sci_request>
```

rm

The rm command is used to remove files.

path is a required attribute specifying the location to remove.

Example

This rm request will attempt to delete /path_to/file.ext

```
<sci_request version="1.0">
  <file_system>
    <targets>
      <device id="00000000-00000000-00000000-00000000"/>
    </targets>
    <commands>
      <rm path="/path_to/file.ext" />
    </commands>
  </file_system>
</sci_request>
```

reboot

Reboot is used to issue a reboot for a device. The majority of devices may not support this operation, and will instead support reboot through RCI. This option exists as an alternative for devices that may not support RCI.

A request follows this format:

```
<sci_request version="1.0">
  <reboot>
    <targets>
      {targets}
    </targets>
  </reboot>
</sci_request>
```

data_service

A data service command is used to send a data service request to a device. The target on the device is specified, and the payload can be specified as plain text or base64 encoded. If it is base64 encoded and the format flag is set appropriately, the payload will be decoded before it is sent on to the device. This allows for binary data to be sent without the device having to decode it.

The command consists of a **requests** element with a child **device_request**, with the text of the element indicating the payload. The following attributes apply to **device_request**:

target_name is required and indicates the target on the device which will receive this request.

format is optional, and if set to *base64* indicates that the payload should be base64 decoded before being sent to the device.

A request follows one of the following formats:

```
<sci_request version="1.0">
  <data_service>
    <targets>
      {targets}
    </targets>
    <requests>
      <device_request target_name="myTarget">
        my payload string
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

```
<sci_request version="1.0">
  <data_service>
    <targets>
      {targets}
    </targets>
    <requests>
      <device_request target_name="myBinaryTarget" format="base64">
        YmluYXJ5ZGF0YS4uLg==
      </device_request>
    </requests>
  </data_service>
</sci_request>
```

Asynchronous operations

Overview

SCI Requests that are asynchronous return without waiting for the request to finish, and return a job id that can be used to retrieve the status and results later.

Performing an asynchronous request

An synchronous request is performed by specifying **synchronous="false"** in the element specifying the operation in the request, e.g. <send_message synchronous="false">

the response then has the form:

```
<sci_reply version="1.0">
  <send_message>
    <jobId>{job_id}</jobId>
  </send_message>
</sci_reply>
```

where **job_id** identifies the request you submitted.

Retrieve status

You can retrieve the status for a particular request, or retrieve information about submitted requests overall.

Status for a particular job

Do an HTTP GET on http://developer.idigi.com/ws/sci/{job_id}

and you will get a response like:

```
<sci_reply version="1.0">
  <status>{current_status}</status>
  <{operation_name}>
    {available results}
  </{operation_name}>
</sci_reply>
```

where

current_status can be *new*, *in_progress*, *complete*, *canceled*.

and the rest looks like it would for any other sci_reply, with any available results included.

Overall status

Do an HTTP Get on http://developer.idigi.com/ws/sci

and you will get a response that looks like:

```
<result>
  <resultTotalRows>1</resultTotalRows>
  <requestedStartRow>0</requestedStartRow>
  <resultSize>1</resultSize>
  <requestedSize>1000</requestedSize>
  <remainingSize>0</remainingSize>

  <Job>
     <jobId>601358</jobId>
     <cstId>0</cstId>
     <usrId>0</usrId>
     <jobType>0</jobType>
     <jobSyncMode>0</jobSyncMode>

     <jobReplyMode>0</jobReplyMode>
     <jobTargets>00000000-00000000-0004F3FF-00000000</jobTargets>
     <jobRequestPayload>&lt;rci_request&gt;&lt;query_setting/&gt;&lt;/rci_
request&gt;</jobRequestPayload>
     <jobDescription>query_setting</jobDescription>
     <jobStatus>2</jobStatus>

     <jobTargetCount>1</jobTargetCount>
     <jobProgressSuccess>1</jobProgressSuccess>
     <jobProgressFailures>0</jobProgressFailures>
     <jobSubmitTime>2010-03-02T15:36:22Z</jobSubmitTime>
     <jobCompletionTime>2010-03-02T15:36:22Z</jobCompletionTime>
  </Job>
</result>
```

where **jobId** is the id for the request

jobType is the type of the job (0: send_message, 1: update_firmware, 2: disconnect)

jobSyncMode indicates if the job is synchronous (0: synchronous, 1: asynchronous)

jobReplyMode indicates the reply mode (0: all, 1: none, 2: only), where only means only return errors

jobStatus is the current status of the job (0: new, 1: in_progress, 2: complete, 3: canceled)

jobTargetCount is the number of devices the job is targeted for

jobProgressSuccess is the number of devices that have completed the operation successfully

jobProgressFailures is the number of devices that have completed the operation with an error

Cancel a request

Do an HTTP DELETE on http://developer.idigi.com/ws/sci/{job_id}.

This will attempt to cancel the request. Some parts of the request may have already completed, and parts of the request that are in progress may continue to completion, but it should prevent any operations that have not yet begun stopstarting.

Sms

Introduction

This web service is used to send SMS messages to control devices

Shoulder tap

Send an SMS shoulder tap message to the specified devices.

Return codes

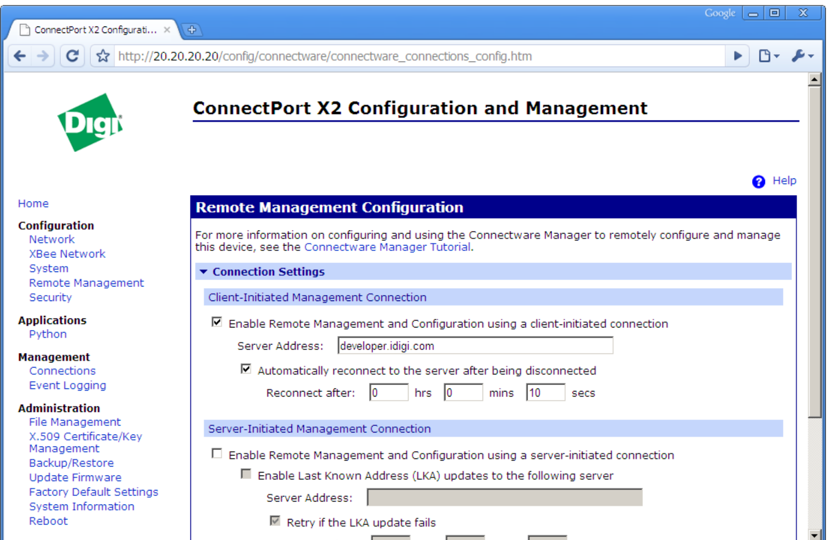
- 200 (OK) -
- 400 (Bad request) - if the request is invalid
- 500 (Internal server error) - if the request cannot be handled due to a server error

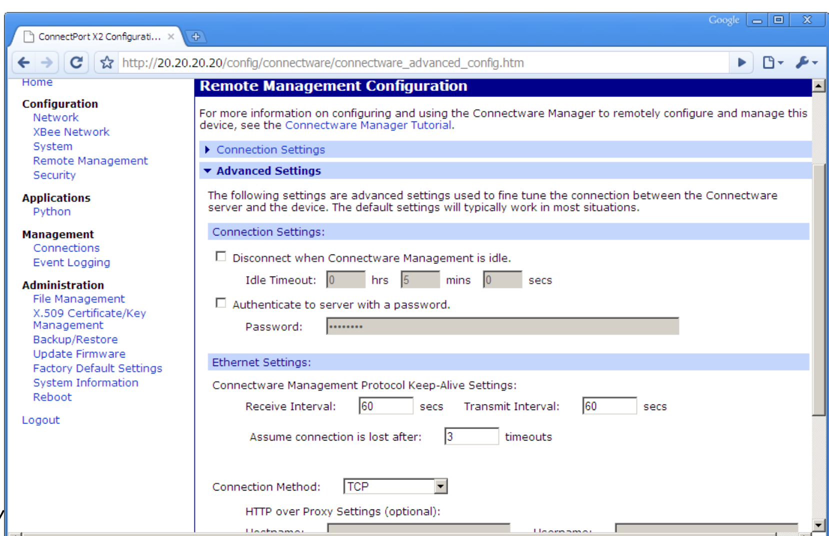
Troubleshoot Device Cloud connection

Device configuration

First, verify that your device is configured to connect to iDigi. To do this, go to the Web UI of the device. From there, click on the remote management tab. Make sure the the box is checked to enable remote management and that the correct server is configured. You will probably also want to make sure the box is checked for the device to automatically reconnect. You also have a few options to check under the Advanced Settings heading on this page. On some devices, there is the option to use SSL as the Connection Method. Initially, you may want to make sure this is set to TCP, and then change it to SSL once you have successfully verified your device is connected.

Images



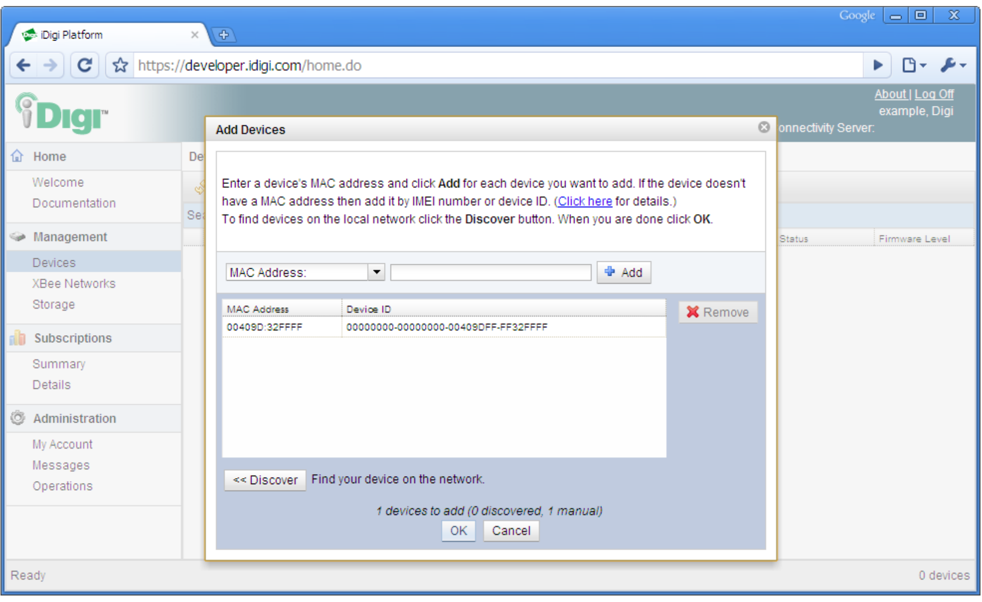


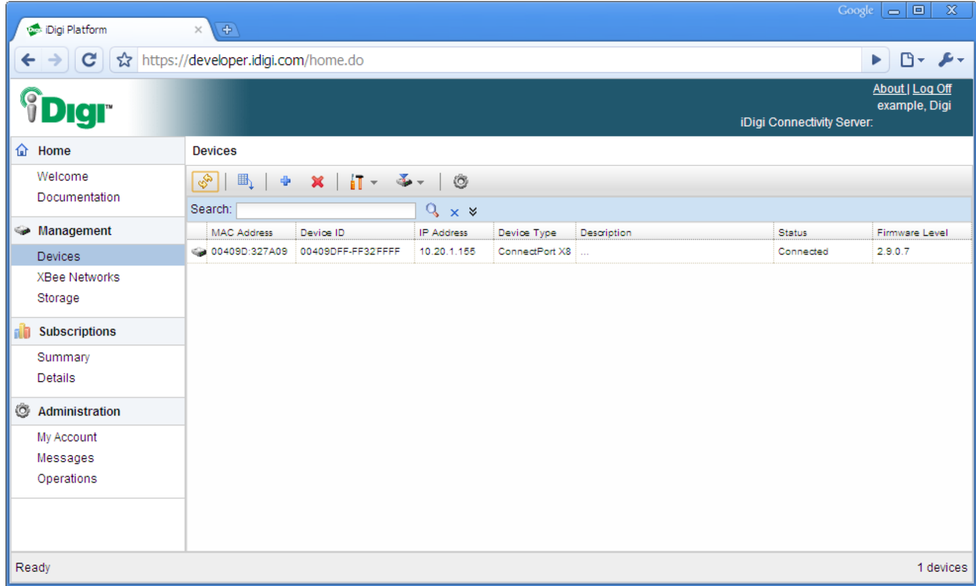
Adding device to Device Cloud

Next, we will make sure your device is added correctly to Device Cloud. Login to your account. If you have not yet added your device, do so by using the dialog to add a device. If your device is on your local LAN, you'll be able to use the automatic discovery mechanism to add it, or if that doesn't work, you can use the manual mode and just type in mac address or device id of the device.

After your device is added, it should show up in the list of devices. After a short amount of time, is the device is configured correctly, you should be able to refresh the device list and see the device is connected. If it doesn't connect, you may want to reboot the device just to make sure any previous connections the device have made are severed, so it will initiate the new connection. If it is not connected at this point, we'll move on to checking if DNS resolution is working for the device.

Images



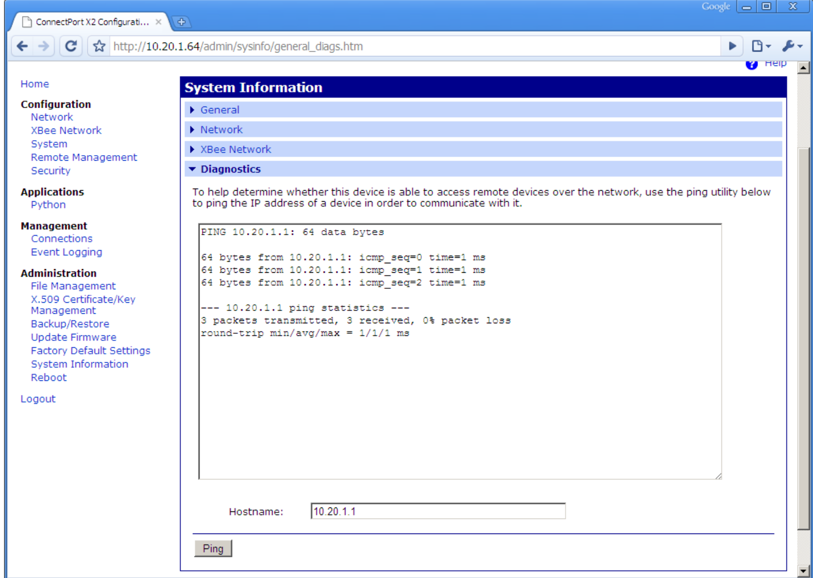


Verify DNS is working

Go to the Web UI of the device. Go to the System Information page, and click on the Diagnostics heading. Enter the Device Cloud server you are trying to connect the device to in the hostname field. Click ping and see if the device is able to communicate with the Device Cloud server by hostname. If this fails, trying pinging the Device Cloud server from your computer to determine the IP address of the Device Cloud server. Enter this IP address in the hostname field on the diagnostics page of the device, and see if the device is now able to communicate with the server. If it can, your device may not be configured correctly to handle DNS. This often happens when you assign a static address to the device without specifying a DNS Server.

If the device is still not able to ping the Device Cloud server by IP address, you may want to try pinging another hostname such as www.google.com, to determine if your device is capable of pinging a public system.

Images



Diagnostics page	A successful ping

Check for firewall

Open up the command prompt (Window key + R, type cmd into the text box and hit enter). enter the command "telnet developer.idigi.com 3197".

```
C:\>telnet developer.idigi.com 3197
```

If you receive the following:

```
Connecting To developer.idigi.com...Could not open connection to the host, on port 3197: Connect failed
```

Contact your network administrator about adding an exception into your networks firewall. Or else if the screen turns blank you can close the window, there is no issue.

Check device trace

When the connection problem is not immediately apparent, it is helpful to check the device's trace output to determine if the device is actually trying to connect, and to take a look at any helpful information that may be provided in the trace output. To turn on trace and see the output, you will need to telnet to the IP address of your device. Once you are in, execute the command set trace **state=on mask=edp:*** On Windows XP, you can open a command prompt and the telnet command is available. On other operating systems the process may be different.

On each device connection attempt, you should now see some output that may provide some information as to why the device is failing to connect.

Images

```
C:\WINDOWS\system32\cmd.exe
C:\>telnet 10.20.1.64
```

```
Telnet 10.20.1.64
#> set trace state=on mask=edp:*
#>
#> 70812491:edp:DEBG:EDP mgmt: Attempting connection to server (30) ***
70812492:edp:DEBG:before EDP CIC connect: 0 free, 0 min
70812500:edp:DEBG:Initiating client-initiated connection
70812504:edp:DEBG:Finding Service
70812505:edp:DEBG:, URL=en://developer2.idigi.com
70812506:edp:DEBG:Trying TCP
70812544:edp:DEBG:TCP failed
70812545:edp:DEBG:Connection to server failed
70812546:edp:DEBG:, server=developer2.idigi.com
70812570:edp:DEBG:Client-initiated connection failed, return=-2
70812571:edp:DEBG:Service Down - Retry or Call Customer Care: 1 (Can't Connect t
o Server)
70812576:edp:DEBG:after EDP CIC disconnect: 0 free, 0 min
70812601:edp:DEBG:EDP mgmt: Will reconnect to server in 10 seconds...
```

Telnet to the Device	Turn on Trace and Get Output

UI descriptor

Introduction

UI Descriptors allows customer devices to have custom device property pages in Device Cloud.

Reference

Menu templates

Under the root (ui) element a navigation element contains the menu template. Each menu is composed of a unique id, name (display text), and associated page template name. The page and data and data groups that the page handles are optional and are typically not supplied if the menu has sub-menus. If the data groups the page handles is not listed and it has a page that is user defined it will parse the page contents and generate the field itself.

```
<ui>
  <navigation>
    <menu id='test1' name='Test' page='test_page'
data='settings:mgmtconnection/*'/>

    <menu id='test2' name='Test page 2' required='true'>
      <menu id='test2child' name='Test Child'
        page='default_properties_page' dataRootDefault='settings'
        required='false' data='doesNotExist/*/desc'>
      </menu>
    </menu>

    <menu id='advanced_cfg' name='Advanced Configuration' page=''
      dataRootDefault='settings' data='' required='false'
      organizeByGroup='true' readonly='false' indexBy=''>

      <automenu page='' dataRootDefault='settings'
        data='settings:*' readonly='false'>
      </automenu>
    </menu>

  </navigation>
</ui>
```

Menu element

The menu element represents a menu item. The menu hierarchy will be generated in the same parent/child relationship as xml menu elements recursively. For example:

```
<ui>
  <navigation>
    <menu id="example_menu1" name="Single root level item" required="true" />

    <menu id="example_menu2" name="Parent root level item" required="true">
      <menu id="example_menu3" name="Child menu" required="true">
        <menu id="example_menu3" name="Sub-Child menu" required="true" />
```

```
      </menu>
    </menu>
  </navigation>
</ui>
```

Renders as:

Single root level item
Parent root level item
Child menu
Sub-Child menu

Single root level item

id (*required*)

The id attribute is required and must be unique. This is used to reference this menu item.

data

The data reference for a menu determines what property groups the associated page is responsible for. Property groups are, in RCI terms, settings or state groups. They are specified in the menu as a comma separated list of groups. Each group name can have an index (or dictionary name) specified in a slashed notation. For example: 'serial/1' OR 'tcp_echo,udp_echo,http,https'. A special data value of '*' is used to automatically generate menus for all property groups not already specified explicitly in the other menu items. This should be the last menu item specified and is typically placed under an 'Advanced' parent menu item.

name (*required*)

The name attribute is the label for the menu. It will be displayed in the menu and at the header of the property page when the menu is selected.

page

The page attribute is optional and used to specify what to render in the properties page when the menu item is selected. This may be either an Devie Cloud provided page like file management or a custom page defined later in this document. If this is left blank then the property page will either be blank itself or will list any children menu items. If you want a page that lists all settings designated by the "data" attribute set this value to "default_properties_page" which is a pre-defined Device Cloud properties page. This is the equivalent of creating a page with the contents being just an "<unprocessed/>" element (see below in Page Contents section).

required

Boolean defining if this menu should be displayed even if the data listed in the data attribute does not exist. If this is set to false (default) and the query_settings of the device does not contain any of the properties listed in the data this menu will be removed.

dataRootDefault

Default root for the data fields when not explicitly specified (optional, settings is default)

organizeByGroup

Determines if page information is organized by group or in the order specified in data

indexBy

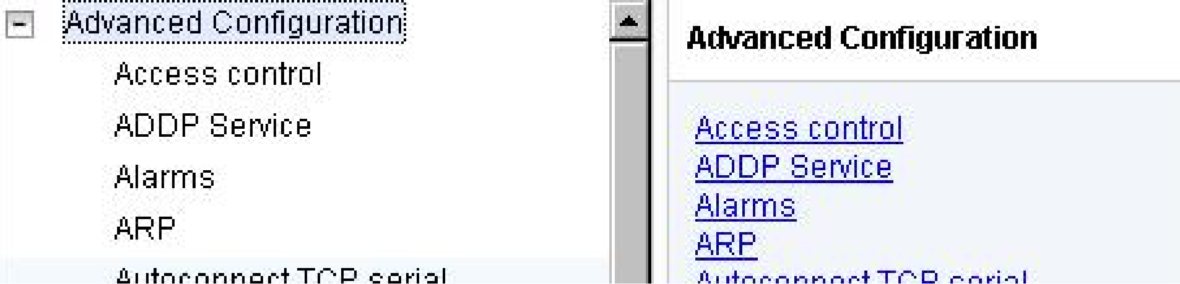
Default root for the data fields when not explicitly specified (optional, settings is default)

Automenu

The automenu will render all settings for a device that has not been reserved by a different menu item via the data attribute. In example:

```
<ui>
  <navigation>
        <menu id='advanced_cfg' name='Advanced Configuration' >
             <automenu data='settings:*' readonly='false' />
        </menu>
  </navigation>
</ui>
```

Renders as:



id

An optional unique identifier for this menu.

dataRootDefault

Default root for the data fields when not explicitly specified (optional, settings is default)

data

Comma separated list of the page's data fields (optional)

readonly

If all pages should render read-only.

Page templates

HTML templates Example for reference:

```
<ui>
  <content>
    <page id='test_page' help='test_page_help'><b>My IP setting:</b>
      <property rciId='settings:mgmtconnection/1/serverAddress' />
      <hr />
      <h1>Advanced:</h1>
      <unprocessed>
        <exclude rciId='settings:mgmtconnection/1/timedConnectionPeriod' />
      </unprocessed>
    </page>
  </content>
</ui>
```

Attributes

There are two attributes you can define for the page element in the content

id

The id attribute is required and must be unique. This is used as a reference in the menus.

help

A reference to the id attribute of a help element shared within the content parent.

Page contents

The page contains xhtml and allows the following tags: b, p, i, s, a, img, table, thead, tbody, tfoot, tr, th, td, dd, dl, dt, em, h1, h2, h3, h4, h5, h6, li, ul, ol, span, div, strike, strong, sub, sup, pre, del, code, blockquote, strike, br, hr, small, big, property, unprocessed, exclude. Most of these tags are standard HTML and will be rendered accordingly in the page area of the device properties when its corresponding menu is selected.

Property tag

The property tag is a special element that will be replaced with a UI field for a setting given by the rciId attribute. The type(text box, drop down, etc.) of the field would be determined by the RCI descriptor for the setting. If no descriptor is available it will be a text box.

Unprocessed tag

All data that is reserved by the menu pointing to this page that has not already been displayed by a property tag will be listed. There is an optional “exclude” element as a child which will remove specific setting from this list

Help templates

HTML templates Example for reference:

```
<ui>
  <content>
    <help id='test_page_help'>
        <b>Help</b>
      <h1 style="color:red">lots of HTML options!</h1>
      </table>
    </help>
  </content>
</ui>
```

Help templates contain an id attribute that are used as reference. The contents are xhtml that will be displayed in popup when the help is clicked in the properties page.

XBIB display LEDs

Gateways (ConnectPort X series Devices) talk to the Device Cloud Server Platform through RCI Request and RCI Response. The RCI Request is sent to the Gateway in the xml string format.

To send continuously different RCI Request commands to the Gateway two types of **Python Code** are required :

- Python Code run on Gateway and talks to Device Cloud Server Platform through RCI
- Python Code run on PC and talks to Device Cloud Server Platform to send RCI Request and get RCI Response

Python code runs on Gateway

The Python Script "led_gateway.py", runs on Gateway to display LEDs on XBIB-U-Dev Board or XBIB-R-Dev Board in following manner.

1. Turn ON/OFF LEDs as switch pressed along with RCI Request command as "LED" or "SWITCH" .e.g. Switch 1 is pressed then LED1 will turn ON.
2. LED Randomly blinks with RCI Request command as "LED_DANCE".

Detail description of LED display code

The Python script below, runs in following different way.

1. If RCI request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">SWITCH</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

   Then if for example **Switch 3** is pressed and on Device Cloud Send is pressed together, the LED 3 turns ON and in *RCI Response Window* Switch 3 is shown ON and other three switches are shown OFF. and if again example **Switch 3** is pressed and on Device Cloud Send is pressed together, the LED 3 turns OFF.

   The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">SWITCH1 : OFF ; SWITCH2 : OFF
; SWITCH3 : ON ; SWITCH4 : OFF </do_command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

2. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">LED</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

   Then if for example **Switch 2** is pressed and on Device Cloud Send is pressed together, the *LED 2* turns ON and in *RCI Response Window* LED 2 is shown ON and other three LEDs are shown OFF.

   The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
```

```
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">LED1 : OFF ; LED2 : ON ; LED3
: OFF ; LED4 : OFF </do_command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

3. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">LED_DANCE</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

Here by sending RCI Request command as **LED_DANCE**, LEDs on XBIB Dev Board turns ON/OFF in a predefined manner.

The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">LED Dance Finished</do_
command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

4. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">Swi</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

Then if for example switch 3 is pressed and on Device Cloud Server Platform **Send** is pressed the LED 3 turns ON but in RCI response Window *Wrong Choice Entered* will be shown. As if either **SWITCH** or **LED** is typed in RCI request then only the result will be displayed on RCI Response.

The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">Wrong Choice Entered</do_
command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

ZIP LED_GATEWAY

Here is the Python code as a ZIP file: Led_gateway.zip.

Here is the DigiXBeeDrivers.zip : DigiXBeeDrivers.zip.

Full Code

If you wish to browse the file for ideas, it is included inline below

```
            # led_gateway.py - RCI Callback functionality run on the gateway to
make the LEDs
#                  turn on and off based on the complimentary code and XBIB board
#                  switch presses

import sys
import zigbee
import rci
sys.path.append("WEB/Python/DigiXBeeDrivers.zip")
from sensor_io import parseIS

# User Entered Extended Address of the remote XBee
DEVICE_ID = "xx:xx:xx:xx:xx:xx:xx:xx"

#RCI Call Back Function
def rci_callback(xml):
    SW = zigbee.ddo_get_param(DEVICE_ID, 'IS')    #Read IO Samples
    SWDIR = parseIS(SW)

    #Check for Switch and LED Status
    cnt=0
    SA = ['DIO0','DIO1','DIO2','DIO3']
    for port in SA:
        SB = "%s" % port
        cnt=cnt+1

        if SWDIR[SB]==0:
            if cnt==1:
                if SWDIR["DIO12"]==0:
                    zigbee.ddo_set_param(DEVICE_ID,'P2',5)
                else:
                    zigbee.ddo_set_param(DEVICE_ID,'P2',4)
                elif cnt==2:
                    if SWDIR["DIO11"]==0:
                        zigbee.ddo_set_param(DEVICE_ID,'P1',5)
                    else:
                        zigbee.ddo_set_param(DEVICE_ID,'P1',4)
                elif cnt==3:
```

```
                if SWDIR["DIO4"]==0:
                    zigbee.ddo_set_param(DEVICE_ID,'D4',5)
                else:
                    zigbee.ddo_set_param(DEVICE_ID,'D4',4)
            elif cnt==4:
                if SWDIR["DIO5"]==0:
                    zigbee.ddo_set_param(DEVICE_ID,'D5',5)
                else:
                    zigbee.ddo_set_param(DEVICE_ID,'D5',4)

    SW = zigbee.ddo_get_param(DEVICE_ID, 'IS')
    SWDIR = parseIS(SW)

    #Check for valid xml string
    if (xml=='LED'):
        cnt=0
        LED_INFO = ['DIO12','DIO11','DIO4','DIO5']
        for port in LED_INFO:
            L_NO = "%s" % port
            cnt=cnt+1
            if SWDIR[L_NO]==0:
                SWDIR[L_NO]='ON'
            else:
                SWDIR[L_NO]='OFF'
            return "LED1 : %s ; LED2 : %s ; LED3 : %s ; LED4 : %s" %\
               (SWDIR['DIO12'],SWDIR['DIO11'],SWDIR['DIO4'],SWDIR['DIO5'])

    elif (xml=='SWITCH'):
        cnt=0
        SWITCH_INFO = ['DIO0','DIO1','DIO2','DIO3']
        for port in SWITCH_INFO:
            S_NO = "%s" % port
            cnt=cnt+1
            if SWDIR[S_NO]==0:
                SWDIR[S_NO]='ON'
            else:
                SWDIR[S_NO]='OFF'
        return "SWITCH1 : %s ; SWITCH2 : %s ; SWITCH3 : %s ; SWITCH4 : %s " %\
                (SWDIR['DIO0'],SWDIR['DIO1'],SWDIR['DIO2'],SWDIR['DIO3'])

    elif (xml=='LED_DANCE'):
        LED_D = ['P2','P1','D4','D5']
        for port in LED_D:
            zigbee.ddo_set_param(DEVICE_ID,port,4)
            LED_C = ['D4','P1','D5','P2']
            for port in LED_C:
                zigbee.ddo_set_param(DEVICE_ID,port,5)
                LED_B = ['D5','P2','D4','P1']
                for port in LED_B:
                    zigbee.ddo_set_param(DEVICE_ID,port,4)
                    LED_A = ['P1','D5','D4','P2']
                    for port in LED_A:
                        zigbee.ddo_set_param(DEVICE_ID,port,5)
        return "LED Dance Finished"

    else:
        return "Wrong Choice Entered"

# valid Extended Address
```

```
def valid_Extended_Address(DEVICE_ID):
    split_addr = DEVICE_ID.split(':')

    if len(split_addr) != 8:
        return False

    for char in split_addr:
        try:
            char = int(char, 16)
        except:
            return False


### Entry Point ###
if valid_Extended_Address(DEVICE_ID) == False: # Verify Length of Extended
Address
    print "Invalid Extended address given: %s" % DEVICE_ID
else:
    DEVICE_ID = '[%s]!' % DEVICE_ID

#Enable LEDs to DO High Configuration
led = ['P2', 'P1', 'D4', 'D5']
for at in led:
    try:
        zigbee.ddo_set_param(DEVICE_ID,at,5)
    except:
        print "parameter: %s can not be executed at address: %s" %(at, DEVICE_ID)
        print "command failed!"

#Enable Switches to DI Configuration
Switch = ['D0', 'D1', 'D2', 'D3']
for at in Switch:
    try:
        zigbee.ddo_set_param(DEVICE_ID,at,3)
    except:
        print "parameter: %s can not be executed at address: %s" %(at, DEVICE_ID)
        print "command failed!"

#Listening RCI message
rci.add_rci_callback("LED_SWITCH", rci_callback)
```

Python code runs on PC

Python scripts can be written to send standard HTTP requests to the server. These scripts use Python libraries to handle connecting to the server, sending the request, and getting the reply.

The Python Script “led_pc.py” runs on PC to send RCI callback requests through Device Cloud Server Platform to the Gateway and to show SCI Response on Console Window in Digi ESP for Python. Here HTTP POST of an SCI request written in Python.

ZIP LED_PC

Here is the Python code as a ZIP file: Led_pc.zip.

Full Code

If you wish to browse the file for ideas, it is included inline below

```
            # led_gateway.py - RCI Request and Response through Digi ESP for
Python

import httplib
import base64
import time



IDIGI_USERNAME  = "username"     #Username for developer.iDigi.com
IDIGI_PASSWORD  = "password"     #Password for developer.iDigi.com
IDIGI_ADDR      = "developer.idigi.com"
GATEWAY_ID      = "xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx"     #Target Gateway
Device ID


def rci_post_cmd(cmd):
    auth = base64.b64encode("%s:%s" % (IDIGI_USERNAME, IDIGI_PASSWORD))

    scimsg = '<sci_request version="1.0"><send_message><targets><device id=\"' +
GATEWAY_ID + \
             '\"/></targets><rci_request version="1.1"><do_command target="LED_
SWITCH">' + \
             cmd + '</do_command></rci_request></send_message></sci_request>'

    webservice = httplib.HTTP(IDIGI_ADDR,80)
    webservice.putrequest("POST", "/ws/sci")
    webservice.putheader("Authorization", "Basic %s"%auth)

    webservice.putheader("Content-type", 'text/xml; charset="UTF-8"')
    webservice.putheader("Content-length", "%d" % len(scimsg))
    webservice.endheaders()
    webservice.send(scimsg)

    webservice.getreply()       #Get SCI Response
    print webservice.getfile().read()


### Entry Point ###
while 1:
    rci_post_cmd("SWITCH")
    time.sleep(60)   #60 sec Delay

    rci_post_cmd("LED")
    time.sleep(60)   #60 sec Delay

    rci_post_cmd("LED_DANCE")
    time.sleep(60)   #60 sec Delay
```

XBee to Device Cloud - DataPoint Creation

Read data from any XBee modules on your Zigbee network and push that data up to the Digi Device Cloud via the DataPoint interface

Platform

This should run on either X2, X4 or X2e type gateways.

Code

```
##############################################################################
#                                                                            #
# Copyright (c)2014, Digi International (Digi). All Rights Reserved.         #
#                                                                            #
# Permission to use, copy, modify, and distribute this software and its      #
# documentation, without fee and without a signed licensing agreement, is    #
# hereby granted, provided that the software is used on Digi products only   #
# and that the software contain this copyright notice,  and the following    #
# two paragraphs appear in all copies, modifications, and distributions as   #
# well. Contact Product Management, Digi International, Inc., 11001 Bren     #
# Road East, Minnetonka, MN, +1 952-912-3444, for commercial licensing       #
# opportunities for non-Digi products.                                       #
#                                                                            #
# DIGI SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED     #
# TO, THE IMPLIED WARRANTIES OF MERCHANT ABILITY AND FITNESS FOR A           #
# PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY,   #
# PROVIDED HERE UNDER IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND.  #
# DIGI HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,           #
# ENHANCEMENTS, OR MODIFICATIONS.                                            #
#                                                                            #
# IN NO EVENT SHALL DIGI BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,        #
# SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,      #
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF     #
# DIGI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.                  #
#                                                                            #
##############################################################################

'''
XBEE DATA TO DEVICE CLOUD DATA POINTS SAMPLE APPLICATION

This application gathers data from a Digi Gateways zigbee interface and then
pushes it up to the Digi Device Cloud.

Platforms:    Digi Connect Port: X2, X4, X2e
Date:         4/22/14

Assumption(s):
- The gateway is tied to a user account and is showing up as 'connected'.
- The gateway has at least one other Zigbee node than itself.

Details:

DataPoints are talked about in the Digi Device Cloud Programmers Guide.  This
document is available at https://login.etherios.com.  You'll need to login and go
to the:
'Documentation -> Resources -> Device Cloud Programmers Guide'

Zigbee:
The Digi Gateway will have a Zigbee capability.  This allows it to communicate
on it's 'mesh network'.


Data Streams Defined:
The Data Streams API allows data to be stored in a time series for long
periods of time. Data streams represent time series data, a sequence of data
```

```
points. You can create a data streams with web services by uploading data
points, or using the DIA or Smart Energy frameworks with Device Cloud. You can
query the data by time ranges, roll-up intervals, and perform basic aggregates.

Time series data involves two concepts: data points and data streams.

Data points are the individual values which are stored at specific times.
Data streams are containers of data points. Data streams hold meta data
bout the data points held within them. The data streams and the data points
they hold are addressed using hierarchical paths (much like folders).



'''

import sys
import os
import socket
import select
import binascii
import idigidata
import xbee
import time

def fmt_dp(data, streamid, timestamp=None, datatype=None, units=None):
    fmt_str = "<DataPoint><data>" + str(data) + "</data><streamId>" + streamid +
"</streamId>"
    if datatype: fmt_str += "<dataType>" + datatype + "</dataType>"
    if units: fmt_str += "<units>" + units + "</units>"
    if timestamp: fmt_str += "<timestamp>" + str(int(timestamp)) + "</timestamp>"
    fmt_str += "</DataPoint>"
    return fmt_str

print "Starting Application"

# Check platform of gateway.  it's either NDS or Digi Embedded Linux
if sys.version_info < (2, 6):
    print "Running NDS"
    sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
else:
    print "Running DBL"
    sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.XBS_PROT_
TRANSPORT)

# Determine if the xbee modem in the Digi gateway is using 'Zigbee' or
# 'DigiMesh' firmware.  Then bind accordingly.
if ord(xbee.ddo_get_param(None, 'VR')[0]) == 0x10:
    print "Binding to Digi Mesh"
    sock.bind(("", 0x00, 0, 0))
else:
    print "Binding to Zigbee"
    sock.bind(("", 0xe8, 0, 0))

# start our loop listening for xbee data
while True:
    rs, ws, es = select.select([sock], [], [], 2)
    for r in rs:
        payload, src_addr = sock.recvfrom(200)
```

```
        output = "src_addr:"
        for item in src_addr:
            try: output += " [0x%0X]" % item
            except Exception, e: output += " [" + str(item) + "]"
        print output

        print "payload len: " + str(len(payload))

        output = "\npayload:"
        count = 0
        for item in payload:
            output += " [%0X]" % ord(item)
            if count >= 10:
                count = 0
                output += "\n"
            else: count += 1
        print output

        upload_data = fmt_dp(binascii.b2a_base64(payload)[:-1], "serial_data",
time.time() * 1000, "String", "b6 encoded")
        status, number, error_msg = idigidata.send_to_idigi(upload_data,
"DataPoint/stream.xml")
        if not status: print "DC Upload Error: " + str(number) + " " + error_msg
        else: print "Successfully uploaded data!"
```

Xbee Command to Device Cloud Cross Reference

Xbee experts and documentation generally use AT parameters to explain configuration parameters. Device Cloud uses verbose descriptions. The list below provides a cross-reference.

UPDATE: *Device Cloud now shows both the verbose command and the AT command in the Xbee properties screen, so this cross-refence is no longer necessary. Do still be aware that Device Cloud operates in decimal, rather than hex, in most, but not all, fields.*

Note Most of the parameters on Device Cloud use decimal, while most direct Xbee operations are in hex. Keep this in mind.

A1 - End device association

A2 - Coordinator association

AR - Aggregation route notification

BD - Serial interface data rate

BH - Broadcast radius

CA - CCA threshold

CC - Command sequence character

CE - Coordinator enable

CH - Operating channel

CI - Cluster identifier

CT - Command mode timeout

D0 - AD0/DIO0 configuration

D1 - AD1/DIO1 configuration

D2 - AD2/DIO2 configuration

D3 - AD3/DIO3 configuration
D4 - AD4/DIO4 configuration
D5 - DIO5/Assoc configuration
D6 - DIO6 configuration
D7 - DIO7 configuration
D8 - DIO8/SleepRQ configuration
D9 - DIO9/ON_SLEEP configuration
DE - Destination endpoint
DP - Disassociated cyclic sleep period
EE - Encryption enable
EO - Encryption options
FT - Flow control threshold
GT - Guard times
HP - Hopping sequence
IA - I/O input address
IC - DIO change detect
ID - Extended PAN identifier
ID - PAN identifier
IF - I/O sample from sleep rate
II - Initial PAN identifier
IR - I/O sample rate
IT - I/O samples before transmit
JN - Join notification
JV - Join verification
KY - Link encryption key
LT - Associate LED blink time
M0 - PWM0 output level
M1 - PWM1 output level
MM - MAC mode
MR - Mesh network retries
MT - Broadcast retries
MY - Network address
NB - Serial interface parity
NK - network_key
NH - Maximum hops
NJ - Node join time
NN - Network delay slots
NI - node_id
NT - Node discovery timeout
NW - Network watchdog timeout
P0 - DIO10/PWM0 configuration
P0 - PWM0 configuration

P1 - DIO11/PWM1 configuration
P1 - PWM1 configuration
P2 - DIO12/CD configuration
P3 - DIO13/DOUT configuration
PL - Transmit power level
PM - Power mode
PO - Polling rate
PR - Pull-up resistor enable
PT - PWM output timeout
RN - Random delay slots
RO - Packetization timeout
RP - RSSI PWM timer
RR - MAC retries
RR - XBee retries
SB - Stop bits
SC - Scan channels
SD - Scan duration
SE - Source endpoint
SM - Sleep mode
SN - Peripheral sleep count
SO - Sleep options
SP - Cyclic sleep period
ST - Time before sleep
SW - Sleep early wakeup
T0 - D0 output timeout
T1 - D1 output timeout
T2 - D2 output timeout
T3 - D3 output timeout
T4 - D4 output timeout
T5 - D5 output timeout
T6 - D6 output timeout
T7 - D7 output timeout
V+ - Supply voltage high threshold
WH - Wake host delay
ZA - ZigBee addressing enable
ZS - ZigBee stack profile

Digi API Frames

For updated information on API frames, see the appropriate XBee documentation.

API escape characters

Setting the DDO/AT command AP = 2 causes certain bytes to be escaped. This can be useful over serial links, but is less useful over TCP/IP or UDP/IP.

The escape bytes are added after the normal API frame (including CRC) has been formed, and the CRC bytes is also subject to escaping. Similarly, the escapes must be removed before the frame length or CRC can be known easily. Below are two simple Python routines to add/remove the escapes.

Adding the escapes

Assuming you have created a valid API frame as a binary string, this routine uses brute force to create a new string which has any required escape sequences added. Even if there are no escapes, a new copy of the string is created and returned.

```
API_START_DELIM = 0x7E
API_ESCAPE = 0x7D
API_XON = 0x11
API_XOFF = 0x13

def api_add_escape( binst):
    st = ""
    n = 0
    for ch in binst:
        chn = ord( ch)
        if( chn == API_START_DELIM):
            if( n > 0):
                # we don't escape the very first bytes if 0x7E
            st += '\x7D\x5E'
            else:
                st += ch
        elif( chn == API_ESCAPE):
            st += '\x7D\x5D'
        elif( chn == API_XON):
            st += '\x7D\x31'
        elif( chn == API_XOFF):
            st += '\x7D\x33'
        else:
            st += ch
        n += 1
    return st
```

Removing the escapes

This routine uses brute force to create a new string which has any escape sequences removed. Even if there are no escapes, a new copy of the string is created and returned. The routine does not enforce any API rules other than any 0x7D byte must be part of a valid two-byte sequence.

```
def api_remove_escape( binst):
    st = binst[0] # copy first 0x7E byte verbatim
    n = 1
    while( n < len(binst)):
        if( binst[n] == '\x7D'):
            if( binst[n+1] == '\x5E'):
                st += chr(API_START_DELIM)
            elif( binst[n+1] == '\x5D'):
                st += chr(API_ESCAPE)
```

```
            elif( binst[n+1] == '\x31'):
                st += chr(API_XON)
            elif( binst[n+1] == '\x33'):
                st += chr(API_XOFF)
            else:
                raise 'bad API escape'
            n += 2
        else:
            st += binst[n]
            n += 1

    return st
```

Differences between API frame 0x10 and 0x11

Sending API frames

If your serial device sends a payload using API frame 0x10 (**Transmit Request**), then these defaults are assumed:

- The endpoints are 0xE8 and 0xE8
- The cluster id is 0x0011
- The profile id is 0xC105

Sending the API frame 0x11 (**Explicit Addressing Command Frame**) allows your device to override these defaults. If you do not wish to override them, then there is *NO* advantage to using API frame 0x11 over 0x10. The same information moves over the RF channel regardless.

The Digi gateways always sends API frames as 0x11.

Receiving API frames

Although the API documentation might lead you to believe your device receives API frames 0x90 when the peer uses 0x10, and 0x91 when the peer uses 0x11, this is not true. What your device receives is defined by the AT Command "AO" (API Output).

- Setting AO=0 means your device receives API frames 0x90 (**Receive Packet**) regardless of whether the peer sent the frame to its local Xbee via API frame 0x10 or 0x11.
- Setting AO=1 means your device receives API frames 0x91 (**Explicit Rx Indicator**) regardless of whether the peer sent the frame to its local Xbee via API frame 0x10 or 0x11.

The Digi gateways always receives API frames as 0x91, and will see the default endpoints, cluster and profile id if the remote node sent the data via the 0x10 API frame.

Format API frame string

Once you create the API-specific frame, the code below wraps it in the final API form with delimiter, length and checksum. This code ASSUMES the escape format is **NOT** used.

```
def form_api( frame):
    """Given a binary-string as frame, create an API formatted message.
    """
    try:
        bcc = 0
        for ch in frame:
            # loop through and create checksum
            bcc += ord(ch)
            bcc &= 0xFF
        bcc = ((0xFF - bcc) & 0xFF)

        n = len( frame)
        api = chr(0x7E) + chr((n>>8) & 0xFF) + chr(n & 0xFF) + frame + chr(bcc)

    except:
        # print 'error - bad frame form?'
        api = None

    return api
```

Raw API over ethernet to CPX2

Talking raw API frames over ethernet to the connectPort X2

The Digi ConnectPort X2 offers low-cost Ethernet access into your mesh. The CPX2 runs two very different firmwares, either of which you can flash in at will.

ConnectPort X2 ethernet Python firmware (82001596)

This firmware provides a reduced-memory platform for your Python applications. It does not include the normal Digi IA engine for Modbus/TCP to Modbus serial bridging, nor does it offer raw API frame access to the XBee. Officially, these firmwares support XBee running the 802.15.4, ZNet and ZigBee (ZB) firmwares. The Wi-Fi version of the CPX2 with Python (so WiFi to Mesh) uses firmware number 82001630.

General comments:

- The CPX2 has one-half the normal RAM of an X4/X8, plus runs at less than half the CPU speed. Therefore it is not a perfect 'low-cost version' of the X4; it does introduce limitations.
- In general, your own Python code will run fine on either X2 or X4, however many stock public Python libraries (like HTTP or SMTP servers) assume RAM is virtual and free. So users expecting to make heavy use of external Python modules will find the CPX2 more challenging.
- You will find developing on the CPX2 frustrating, so you should develop on an X4 with an eye to keeping memory usage down, then 'port' your code to the X2 for testing and implementation.
- Some people get burned thinking 'My application is so functionally simple - it will run on the CPX2'. However, 'simple' does not equal 'low memory usage'. Some very complex applications require very little memory, while some very simple applications (like parsing XML) can consume many MB of RAM.

ConnectPort X2 Ethernet firmware w/ API Access (82001631)

Notice there is NO word 'Python' - this firmware offers the normal Digi IA engine for Modbus/TCP to Modbus serial bridging, or it can be configured for raw API frame access to the XBee. The Modbus bridging functions fully for ZNet and ZigBee, and partially for 802.15.4. The raw API frame support works for ALL versions of the XBee (including the point-to-multipoint models like the 868Mhz versions for Europe). The Wi-Fi version of the CPX2 with Python (so WiFi to Mesh) uses firmware number 82001597 - the numbers look swapped, but they are correct.

General comments:

- The Modbus/TCP bridge is documented here: Modbus bridge on CPX4 and Modbus Serial Over Mesh.
- You must select either the IA Engine/Modbus Bridge, or the raw API frames - the CPX2 does not support both functions concurrently
- The CPX4 and X8 have the same IA Engine/Modbus Bridge integrated in the normal firmware, however neither the X4 nor X8 support the raw API frames.
- The raw-API frames can be sent via TCP/IP, UDP/IP, SSL/TLS or Digi Realport.

Serial encap by API

How to talk directly by Digi-Maxstream API frames

If you have a Digi Xtend or mesh radio box attached to a serial port or remote device server, here is an easy way to encapsulate serial packets to send to remote serial devices. This page documents a simple, lowest common denominator method - it does not show all possible solutions. It avoids being fancy and powerful; it is a method which does the basics.

Digi mesh/rf-network types

Here is an enumeration of possible Digi mesh/rf-network types. At this simplest level, many types will share the same behavior for your encapsulated serial data code, yet programming in this knowledge will allow you to add fancier features in the future. To query the type of your mesh, use the AT Command DD which is documented on the XBee Product Codes page.

Notes on the table below:

- Name: a name or tag used by this page to refer to an rf-network type.
- Address Size: some rf-networks support other address forms, however the size listed is the one this example will use.
- Payload: the maximum serial message that can be moved - yet this is only a useful default. If options such as security or source-routing have been enabled, this payload size drops. On a few rf-networks, the size can be larger. You should allow customers/users to manually override this default at least on a full-network basis.

Codes	Name	Address Size	Payload	Description
XB24-ZB	Digi Zigbee	64-bit	84	Digi's 3.x firmware supporting Zigbee 2007
XB24-B	Digi ZNet	64-bit	72 or 75	Digi's 2.x firmware
XB24-DM	Digi Mesh	64-bit	210	Digi's proprietary mesh on 900Mhz and 2.4Ghz
XB24-A	Digi 802.15.4	64-bit	100	Non-mesh star configured nodes on 2.4GHz
XB09-DM	Digi XTend Mesh	64-bit	210	Digi Mesh on Xtend hardware
XB09-DP	Digi XTend	16-bit	2048	Non-mesh star configured nodes on 900Mhz

ASCII notations for Digi address types

- 64-bit addresses are normally coded as text like 00:13:a2:00:40:3e:19:c4!, with the '!' used to tag this as a 64-bit address for Mesh systems.
- 16-bit addresses are normally coded as hex like 0x1234

Full mesh RF-metworks

This includes the types “Digi Zigbee", "Digi Znet" and "Digi Mesh” (Find the newest Product Manual for your XBee here)

TRANSMIT; Client sends serial data

- API Identifier 0x10. API frame 0x11 can also be used. See also: Differences between API frame 0x10 and 0x11.
- Frame Id Byte = 0 unless client desires a Status report, then 0x01 to 0xFF
- 64-bit address of destination (USER MUST ENTER)
- 16-bit dest - set to 0xFFFE since in this example we only use the full 64-bit address
- Broadcast Radius set to 0
- Options set to 0
- Payload of serial protocol

TRANSMIT STATUS; Client receives

- API Identifier 0x8B
- Frame Id Byte = to match up to request TRANSMITTED above
- 16-bit dest - see manual, is ignored in this simple example
- Transmit retry Count - see manual
- Delivery Status - see manual
- Discovery Status - see manual

RECEIVE; Client receives serial response

- API Identifier 0x90. Setting the AT Command AO=1 allows your device to always receive 0x91 frames instead - see also: Differences between API frame 0x10 and 0x11.
- 64-bit address of source of response
- 16-bit dest - see manual, is ignored in this simple example
- Options - see manual
- Payload of serial protocol

Limited Mesh-like RF-Networks

This includes the types “Digi 802.15.4" and "Digi XTend Mesh” (Find the newest Product Manual for your XBee here)

TRANSMIT; Client sends serial data

- API Identifier 0x00
- Frame Id Byte = 0 unless client desires a Status report, then 0x01 to 0xFF
- 64-bit address of destination (USER MUST ENTER)

- Options set to 0
- Payload of serial protocol

TRANSMIT STATUS; Client receives

- API Identifier 0x89
- Frame Id Byte = to match up to request TRANSMITTED above
- Status - see manual

RECEIVE; Client receives serial response

- API Identifier 0x80
- 64-bit address of source of response
- RSSI signal strength - see manual
- Options - see manual
- Payload of serial protocol

Digi XTend, star network with 16-bit addressing

This includes the types “Digi XTend” (Find the newest Product Manual for your XBee here)

TRANSMIT; Client sends serial data

- API Identifier 0x01
- Frame Id Byte = 0 unless client desires a Status report, then 0x01 to 0xFF
- 16-bit address of destination (USER MUST ENTER) 0xFFFF is broadcast
- Options set to 0
- Payload of serial protocol

TRANSMIT STATUS; Client receives

- API Identifier 0x89
- Frame Id Byte = to match up to request TRANSMITTED above
- Status - see manual

RECEIVE; Client receives serial response

- API Identifier 0x81
- 16-bit address of source of response
- RSSI signal strength - see manual
- Options - see manual
- Payload of serial protocol

Simple serial app quick index

Quick Index of relevant pages for an Xbee serial product

Overview

A customer has a simple sensor with an integrated XBee module, which is pin-sleep controlled by a PIC serially connected to the XBee. The sensor PIC wakes up periodically, takes a reading, and send out a string. It expects either an ACK, or an ACK plus a new configuration string in response. The vendor wishes the Digi gateway to actively open a TCP socket and push information to a remote server.

For such a customer, this wiki page gives a quick index of relevant pages.

Pages to review

The XBee module

Since the PIC actively configures and manages the XBee, running with the **API firmware** simplifies the design - trying to switch between API and AT mode is not worth the timing complexities. Some newer firmware also split API and AT into two different firmwares, so your PIC cannot switch on the fly.

Even though you pin-sleep the **XBee, the XBee's SN and SP settings must be valid** or the parent node hangs up on the sleeping end-device. For example, to sleep for one hour the setting SP=0x0AF0 and SN=0x0081 would be correct. This means the parent will buffer requests for the sleeping end-device for 28 seconds, and the expected wake-up time of the end-device is 3612 seconds, so about once per hour. If the end-device does NOT contact the parent within 3 hours (3 x SP x SN), then it will drop the sleeping Xbee from the network.

This Digi support page lists the most recent XBee product manuals.

Your sensor sending data

When the sensor wakes, it creates its data as a binary or ASCII string. Binary is best for the mesh as it packs more information into less space, however if the overall message is less than 60 bytes your design is safe on all normal Xbee - even if security and other options enabled. You do NOT need to include the MAC address of your XBee or any other slave information. The remote device receiving your data will know which MAC the data came from.

Send your data as API code 0x10, targeting MAC 00:00:00:00:00:00:00:00, network address 0xFFFE. On Znet and ZB this moves the message to the Digi gateway.

Do NOT use Broadcast. This creates a non-scalable solution which will fail to work correctly in the field. You have been warned. Broadcast and modern wireless do not mix.

Your sensor receiving the data

If you have set the XBee's AO setting set to zero (0), then **your PIC will see any data responses as API code 0x90**. The MAC address included will be the Digi gateway's address (or the source of the message). You should normally ignore this information. Although the XBee manuals indicate up to 72 bytes will be received, this varies by XBee technology. Common sizes are 72, 75, 84, 100, and over 200. Your PIC doesn't need to accept more data then desired, however it should gracefully handle seeing too much by either truncating or rejecting the message. The API frame length is used to calculate the actually bytes received.

Python on the Gateway talking to your sensor

For a simple single-function loop, your code would use the UDP-like **socket()** function. You bind on the AF_ZIGBEE socket, which makes your code owner of the entire mesh. Your code then uses recvfrom() to receive the messages from your sensor, and the actual MAC of the sensor is in the **addr** return value. Optionally, your code sends any response by sendto(), reusing the **addr** to unicast. It is that simple.

If you are using the 802.15.4 firmware, you would socket.bind on endpoint 0x00, not 232/0xE8.

See XBee extensions to the Python socket API for details regarding the use of the Python socket functions.

Be warned that this dedicates the Digi gateway for your hardware, so your customers cannot use other Zigbee products - unless you expand your application to handle those products. Two Python programs cannot share access to the mesh. If you expect your customers to mix vendors equipment on a single mesh, then you should look at the Digi DIA platform which allows configurable drivers to be mixed from different sources. See www.etherios.com/devicecloud.

If you desire your Python application to manage the XBee settings, you can use the get_ddo/set_ddo functions. However, if the Xbee sleeps for hours at a time, these will always fail after 16 to 28 seconds. Since the XBee remains awake for the SP setting each time your PIC wakes it, you can hook the need to read/write XBee setting to the socket recvfrom() above. Quickly **sending set_ddo/set_ddo commands would only succeed when the Xbee is awake**.

See Module: xbee for an explanation of how to use of the get_ddo/set_ddo functions.

Python talking outwards or upstream

This is normal TCP or UDP sockets functions. You can find simple client examples on the public internet.

This page Handling socket error and Keepalive explains common error handling, which many web examples ignore.

Robustness issues

Error handling

Make extensive use of try/except at a low level. Avoid putting one big try/except at the highest level, plus always print something if the except is unexpected. A common beginner mistake is put a few very high level try/excepts which hide even common typos at lower levels.

Watch Dog

You can enable a simple watch dog to hard-reset the gateway. If you sensors sleep for hours at a time, you probably want the reset time to be quite long - 3 hours perhaps (or 10,800 seconds). You do not want the gateway rebooting every 5 minutes - especially if the problem is missing sensors.

See Module: digiwdog for an explanation of use of the watch dog functions.

Memory management

Make sure your application **manually forces garbage collection at least once per day**. While Python garbage collection is 'automatic', the algorithm used can be complex and not optimized for small embedded systems. If your code send out a report once or twice per day, calling gc.collect() after the outgoing client socket closes is ideal.

See Python garbage collection for an explanation of Python garbage Collection.

Local diagnostic web pages

Even if the totality of your application is 1) collect data, 2) forward daily as HTTP or email, adding a few status and diagnostic web pages is a valuable addition. The Digi Web Interface normally consumes port 80, so you do not want to create your own web server. Instead, see Module: digiweb, which allows your code to register a callback with the web ui. This would allow a user (or you) to open a web page like http://192.168.0.10/status and pull up a user-friendly web page. You need to understand how to manually build a raw page, but the effort will be worth it.

See Module: digiweb for an explanation of the Digi Web UI callback to pass unknown URL to your Python program.

Digi Demo Applications

ADC values

Program to receive ADC values from ZigBee

(ZigBee routers and sensors) Receive and parse the ADC values coming from the ZigBee routers and sensors.

How does it work?

This application receives data packets from routers and end devices and prints the parsed output to the console. These data packets include Light, Temperature and Humidity values.

Test files

This sample program contains two files, Get_Router_Sensor_reading.py and RouterSensor_reading_application_notes.doc. The program file is Get_Router_Sensor_reading.py.

ADC value test sample application

The ADC Value sample application can be found here: Get_Router_Sensor_reading.zip.

Basic usage

See Application Note here: RouterSensor_reading_application_notes.zip.

Sample of Get_Router_Sensor_reading.py file:

##

```
# Copyright (c)2012, Digi International (Digi). All Rights Reserved.      #
#                                                                         #
# Permission to use, copy, modify, and distribute this software and its   #
# documentation, without fee and without a signed licensing agreement, is #
# hereby granted, provided that the software is used on Digi products only #
# and that the software contain this copyright notice,  and the following #
# two paragraphs appear in all copies, modifications, and distributions as #
# well. Contact Product Management, Digi International, Inc., 11001 Bren  #
# Road East, Minnetonka, MN, +1 952-912-3444, for commercial licensing    #
# opportunities for non-Digi products.                                    #
#                                                                         #
# DIGI SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED  #
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A         #
# PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, #
# PROVIDED HEREUNDER IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND. #
# DIGI HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,        #
# ENHANCEMENTS, OR MODIFICATIONS.                                         #
#                                                                         #
# IN NO EVENT SHALL DIGI BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,     #
# SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,  #
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF  #
# DIGI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.               #
############################################################################
```

```
import os
import sys
import socket
import struct
```

```
import time
import zigbee
import datetime
import traceback
from socket import *

humidity = 0
light = 0
temp = 0

def parse_packet(payload):
    print "parser function"

def calc_light(value):
    mv = float(value * 1200) / 1023
    return mv

def calc_temp(value):
    mv = float(value * 1200) / 1023
    degc = ((mv - 500.0) / 10.0) - 4.0
    #degf = (degc * 1.8) + 32.0
    return degc

def calc_humidity(value):
    mv = (value / 1023.0) * 1200
    humidity = (((mv * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
    return humidity

try:

    sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)
    ''' Bind to ("", 0xe8, 0xc105, 0x92)to receive all incoming 'IS' responses
    and decode them per the XBee documentation
    '''
    sd.bind(("", 0xe8, 0xc105, 0x92))
    print "socket is bound"

    while 1:
        try:
            payload, src_addr = sd.recvfrom(255)
            len_payload = len(payload)

            print datetime.datetime.now()
            src = src_addr[0][1:24]
            print "Source Address: %s" %src

            #print len_payload
            if len_payload == 8:
                fixed_byte, digital_bits, analog_set, lig, tmp = struct.unpack
(">bhbhh", payload[:8])
                light = calc_light(lig)
                temp = calc_temp(tmp)
                print "light - %s" %light
                print "temperature - %s" %temp

            elif len_payload == 10:
                b1,b2,b3,b4,b5,b6,lig,tmp = struct.unpack(">bbbbbbhh", payload
[:10])
```

```
                print b1
                print b2
                print b3
                print b4
                print b5
                print b6
#                print b7
#                print b8
#                print b9
#                print b10
#                print "fixed_byte is %d" %fixed_byte
#                print "digital_bits is %d" %digital_bits
#                print "analog_set is %d" %analog_set
                print "lig is %d" %lig
               print "temp is %d" %tmp
               light = calc_light(lig)
               temp = calc_temp(tmp)
               print "light - %s" %light
               print "temperature - %s" %temp


           elif len_payload == 12:
               print "sensor"
               print len_payload
               fixed_byte, digital_bits, analog_set, dont_know_bit1, dont_know_
bit2, lig, tmp, humidity = struct.unpack(">bhbbbhhh", payload[:12])
               light = calc_light(lig)
               temp = calc_temp(tmp)
               humidity = calc_humidity(humidity)
               print "light is %d" %light
               print "temp is %d" %temp
               print "humidity is %d" %humidity
           else:
               print "invalid payload"
               print len_payload

           print " "

        except Exception, e:
            print "Exception %s" %e
            traceback.print_exc()

except socket.error, s:
    print "socket exception: %s" %s

sd.close()
print "socket is closed"
```

ASM assembly code

Program to execute assembly code in NET+OS

ASM TEST (For NET+OS 7.4.2 - 7.5.2 modules) To showcase how to execute Assembly code in NETOS.

How does it work?

Test files

This sample program contains two files, root.c and appconf.h. The main function is in root.c.

ASM test sample application

The ASM Test sample application can be found here: ASMSample.zip.

Basic usage

Compile, load and run program using NET+OS environment.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

void assembly_delay();

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;

}
#define LOOP_MAX (25)
void applicationStart (void)
{
           int loopIdx = 0;
            printf ("Hello World!\n");

           // continually call a routine that will execute one instruction 4
times in one asm call
           for (loopIdx = 0; loopIdx < LOOP_MAX; loopIdx++)
           {
               assembly_delay();
           }

           // now execute the same assembly directly
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");

          // now do it in one call
          asm volatile(
          "mov        r0, r0\n\t"
          "mov        r0, r0\n\t"
          "mov        r0, r0\n\t"
          "mov        r0, r0\n\t"
          );


          printf("Test done\n");


          tx_thread_suspend(tx_thread_identify());
```

```
    }

        void assembly_delay()
              {

                 asm volatile("mov   r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t");
              }
  }
```

Advanced Device Discovery Protocol (ADDP)

What is ADDP?

ADDP (Advanced Device Discovery Protocol) is a proprietary protocol developed by Digi International that allows devices on a local network to be found regardless of their network configuration.

How does it work?

ADDP uses a client/server model. The client is the application that is searching for devices. The server is the device that is being search for.

In the simplest terms, the client application sends out a specially formatted UDP broadcast packet on the network. ADDP servers listening for the packet, will receive it, and send an ADDP response back to the client. Once this process is complete, the client can then send configuration requests to the device. These can include things like network settings, and reboot requests.

Java library

A subset of the protocol has been implemented in Java. You can find the jar file here: ADDP Library.

The associated javadoc documentation can be found here: ADDP Java doc.

This library allows you to search synchronously, and asynchronously for devices on the network. You can then use it to reconfigure the device's network settings, or reboot the device.

Java sample application

A simple discovery sample application can be found here: AddpSample(r2010).zip

Basic usage

First, instantiate the AddpClient object.

```
AddpClient addpClient = new AddpClient();
```

Next, call SearchForDevices() and check the return value. Then get the devices, and walk the hashtable.

```
if (addpClient.SearchForDevices()) {
    AddpDeviceList deviceList = addpClient.getDevices();
```

```
    Enumaration<AddpDevice> e = deviceList.elements();
    while(e.hasMoreElements()) {
       AddpDevice device = e.nextElement();

       // do something with the device here
       System.out.println(device.toString());

       // if device is not configured for DHCP, then turn it on and reboot.
       if (device.getDHCP() == 0) {
          addpClient.setDHCP(device, true, "dbps");
          addpClient.rebootDevice(device, "dbps");
       }
    }
}
```

Android animation

Android sample animation test

(Android modules, i.MX51 and i.MX53) Android program, when this application runs on the android device, it will change the background images for every 1/2 second, we can start and stop the animation using the button displayed on the application.

Test files

This sample program contains several files, the /src folder contains the source files.

Animation test sample application

The Android Animation Test sample application can be found here: Animation2.zip

Basic usage

Compile, load and run program using Android environment.

Sample of Animation2Activity.java file:

```
package animation2.test;

//import canvas.paint.CanvasActivity.DemoView;
import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.widget.*;

public class Animation2Activity extends Activity {
    /** Called when the activity is first created. */

    Button b;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.setupButton();

    }


    private void setupButton() {
                b = (Button) this.findViewById(R.id.startFAButtonId);
                b.setOnClickListener(new Button.OnClickListener() {
                        public void onClick(View v) {
                                parentButtonClicked(v);
                        }
                });
        }
```

```
private void parentButtonClicked(View v) {
            animate();
    }

private void animate() {
            ImageView imgView = (ImageView) findViewById(R.id.animationImage);
            // imgView.setVisibility(ImageView.VISIBLE);
            imgView.setBackgroundResource(R.drawable.ani);

            AnimationDrawable frameAnimation = (AnimationDrawable)
imgView.getBackground();


            if (frameAnimation.isRunning()) {
                    frameAnimation.stop();
                    b.setText("Start");
            } else {
                    frameAnimation.start();
                    b.setText("Stop");
            }
    }

public boolean onTouchEvent(MotionEvent event) {

        int eventaction = event.getAction();

        switch (eventaction)
        {
          case MotionEvent.ACTION_DOWN:              // finger touches the screen
                 this.setupButton();

                 break;
          case MotionEvent.ACTION_MOVE:             // finger moves on the screen
                  //this.setupButton();
                  break;
          case MotionEvent.ACTION_UP:                    // finger leaves the screen

              break;
       }

       return true;

    }

}
```

Android Bluetooth test

Android sample Bluetooth Test

(Android supported modules) Android program, when this application runs on the android device it invokes the bluetooth interface.

Test files

This sample program contains several files, the /src folder contains the source files.

Bluetooth test sample application

The Android Animation Test sample application can be found here: BlueToothTest.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of MainActivity.java file:

```
package com.digi.bluetoothtest;

import java.io.IOException;
import java.io.OutputStream;
import java.lang.reflect.Method;
import java.util.Set;
import java.util.UUID;

import android.os.Bundle;
import android.os.Message;
import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothClass;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
        private static final int REQUEST_ENABLE_BT = 1;
        private Button _scanBlueToothButton;
        private EditText _logEditText;

        private BluetoothAdapter _bluetoothAdapter = null;


        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
```

```
                setContentView(R.layout.activity_main);

                connectUi();

                _bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

                if (_bluetoothAdapter == null) {
                        Toast.makeText(this, "No BT adapter", Toast.LENGTH_
LONG).show();
                        return;
                }

                if (!_bluetoothAdapter.isEnabled()) {
                        Intent enableBt = new Intent(BluetoothAdapter.ACTION_
REQUEST_ENABLE);
                        startActivityForResult(enableBt, REQUEST_ENABLE_BT);
                }
                else {}
        }


        private void scanBlueToothButton_OnClick() {
                ProgressDialog dialog = ProgressDialog.show(this, "",
                        "Loading. Please wait...", true);

                _bluetoothAdapter.disable();
                _bluetoothAdapter.enable();

                _bluetoothAdapter.startDiscovery();

                dialog.dismiss();

                Set<BluetoothDevice> devices = _bluetoothAdapter.getBondedDevices
();
                StringBuilder sb = new StringBuilder();

                if (devices.size() > 0) {
                        for (BluetoothDevice device : devices) {
                                sb.append(device.getName());
                                sb.append("\n");
                                // 1e0ca4ea-299d-4335-93eb-27fcfe7fa848

                                try {
                                        Method m = device.getClass().getMethod(
                                                "createRfcommSocket", new Class[]
{ int.class });

                                BluetoothSocket sock = (BluetoothSocket) m.invoke
(device, 1);

                                sock.connect();

                                OutputStream stream = sock.getOutputStream();
                                stream.write("Hello bt world!".getBytes());
                                stream.close();
                        } catch (Exception e) {
                                e.printStackTrace();

                        }
```

```
                }
        }
        _logEditText.setText(sb.toString());
        }


        @Override
        protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
                super.onActivityResult(requestCode, resultCode, data);

                switch (requestCode) {
                        case REQUEST_ENABLE_BT:
                                break;

                }

        }

        @Override
        public boolean onCreateOptionsMenu(Menu menu) {
                getMenuInflater().inflate(R.menu.activity_main, menu);
                return true;
         }


         private void connectUi() {
                _scanBlueToothButton = (Button) findViewById
(R.id.scanBluetoothButton);
                _scanBlueToothButton.setOnClickListener(new View.OnClickListener
() {
                public void onClick(View v) {
                scanBlueToothButton_OnClick();
                }
                });

                _logEditText = (EditText) findViewById(R.id.logEditText);
        }
}
```

Android HelloBox2D

Android sample HelloBox2D test

(Android supported modules) Android program, This application is a porting of the Box2D into android environment.

Test files

This sample program contains several files, the /src folder contains the source files.

HelloBox2D test sample application

The Android HelloBox2D Test sample application can be found here: HelloBox2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of HelloBox2DActivity.java file:

```
package com.digi.box2d;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;

public class HelloBox2DActivity extends Activity {
         private PhysicsWorld mWorld;
        private Handler mHandler;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mWorld = new PhysicsWorld();
        mWorld.create();

        mHandler = new Handler();
        mHandler.post(update);

    }

    @Override
    protected void onPause() {
        super.onPause();

        mHandler.removeCallbacks(update);
    }

    private Runnable update = new Runnable() {
        public void run() {
            mWorld.update();
            if (mWorld.FallingBox.isAwake()){
                 mHandler.postDelayed(update, (long) (mWorld.timeStep*1000));
            } else {
                mHandler.removeCallbacks(update);
```

```
                }
            }
        };
}
```

Android RenderAmovingSprite

Android sample render moving sprite tests

(Android supported modules) Android program, It renders a sprite on the screen and moves it.

Test files

This sample program contains several files, the /src folder contains the source files.

Render Moving Sprite Test Sample Application

The Android Render a Moving Sprite Test sample application can be found here: RenderAMovingSprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderAMovingSpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.view.Window;

public class RenderAMovingSpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                requestWindowFeature(Window.FEATURE_NO_TITLE);

                mGLSurfaceView = new GLSurfaceView(this);

                MyRenderer spriteRenderer = new MyRenderer(this);

                GLSprite sprite = new GLSprite(R.drawable.digi);

                DisplayMetrics dm = new DisplayMetrics();

                getWindowManager().getDefaultDisplay().getMetrics(dm);

                Grid spriteGrid = null;

                // Setup a quad for the sprite to use. All sprites will use the
                // same sprite grid instance.
                spriteGrid = new Grid(2, 2, false);
                spriteGrid.set(0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, null);
                spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
                spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
                spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);
```

```
                sprite.x = 100;
                sprite.y = 150;
                sprite.width = 64;
                sprite.height = 64;

                sprite.setGrid(spriteGrid);

                Runtime r = Runtime.getRuntime();
                r.gc();

                spriteRenderer.sprite = sprite;
                spriteRenderer.setVertMode(true, true);
                mGLSurfaceView.setRenderer(spriteRenderer);

                setContentView(mGLSurfaceView);

                Thread gameThread = new Thread(new Game(sprite, this));

                gameThread.start();
        }
}
```

Android RenderAsprite

Android sample Render a sprite test

(Android supported modules) Android program, Renders a stationary sprite on the screen.

Test files

This sample program contains several files, the /src folder contains the source files.

Render a sprite test sample application

The Android Render a Sprite Test sample application can be found here: RenderASprite.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of RenderASpriteActivity.java file:

```
package com.digi;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.util.DisplayMetrics;

public class RenderASpriteActivity extends Activity {
        private GLSurfaceView mGLSurfaceView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mGLSurfaceView = new GLSurfaceView(this);

        MyRenderer spriteRenderer = new MyRenderer(this);

        GLSprite sprite = new GLSprite(R.drawable.digi);

        DisplayMetrics dm = new DisplayMetrics();

        getWindowManager().getDefaultDisplay().getMetrics(dm);

        Grid spriteGrid = null;

        // Setup a quad for the sprite to use.  All sprites will use the
        // same sprite grid instance.
        spriteGrid = new Grid(2, 2, false);
        spriteGrid.set(0, 0,  0.0f, 0.0f, 0.0f, 0.0f , 1.0f, null);
        spriteGrid.set(1, 0, 64, 0.0f, 0.0f, 1.0f, 1.0f, null);
        spriteGrid.set(0, 1, 0.0f, 64, 0.0f, 0.0f, 0.0f, null);
        spriteGrid.set(1, 1, 64, 64, 0.0f, 1.0f, 0.0f, null);

        sprite.x = 100;
        sprite.y = 100;
        sprite.width = 64;
        sprite.height = 64;
```

```
        sprite.setGrid(spriteGrid);

        Runtime r = Runtime.getRuntime();
        r.gc();

        spriteRenderer.sprite = sprite;
        spriteRenderer.setVertMode(true,true);

        mGLSurfaceView.setRenderer(spriteRenderer);

        setContentView(mGLSurfaceView);


    }
}
```

Android Test2D

Android sample Test2D test

(Android supported modules) Android program, a 2D rendering test using canvas.draw.

Test files

This sample program contains several files, the /src folder contains the source files.

Test2D test sample application

The Android Render a Moving Sprite Test sample application can be found here: Test2D.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of Test2DActivity.java file:

```
package com.digi.test2d;

import android.app.Activity;
import android.os.Bundle;
import android.view.Display;
import android.view.Window;
import android.view.WindowManager;

public class Test2DActivity extends Activity{
        private drawView view;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                        WindowManager.LayoutParams.FLAG_FULLSCREEN);

        Art.loadBitmaps(getResources());

        Display display = getWindowManager().getDefaultDisplay();

                view = new drawView(this, display.getWidth(), display.getHeight
());

        setContentView(view);
    }
}
```

Android UDP client

Android sample for UDP client test

(Android modules i.MX51 and i.MX53) Android program, when this application runs on the android device, it will show "temp" and "humi" buttons on the android UI, and as we click on those buttons it will communicate with the UDPserver.

Test files

This sample program contains several files and the /src folder contains the source files.

UDP Claient Test Sample Application

The Android UDP Client Test sample application can be found here: AndroidUDPClient.zip.

Basic usage

Sample of ChatServerActivity.java file:

```
package test.chat.serv;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Iterator;





import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class ChatServerActivity extends Activity {
    private static final String host = null;
        private int port;
        String str=null;
        /** Called when the activity is first created. */
        TextView txt5,txt1;
        byte[] send_data = new byte[1024];
        byte[] receiveData = new byte[1024];
```

```
        String modifiedSentence;
        Button bt1,bt2,bt3,bt4;


    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txt1   = (TextView)findViewById(R.id.textView1);
        txt5   = (TextView)findViewById(R.id.textView5);

        bt1 = (Button) findViewById(R.id.button1);
        bt2 = (Button) findViewById(R.id.button2);
        bt3 = (Button) findViewById(R.id.button3);
        bt4 = (Button) findViewById(R.id.button4);
        //textIn.setText("oncreate");




        bt1.setOnClickListener(new View.OnClickListener(){
                public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);
                str="temp";
                 try {
                                client();
                                //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                }
        }



    });

bt2.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);

                str="test";
               try {
                                 client();
                                 //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                 }
    });
```

```
    bt3.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
                // Perform action on click
                //textIn.setText("test");
                //txt2.setText("text2");
                //task.execute(null);

                str="humi";
                try {
                                client();
                                //txt1.setText(modifiedSentence);
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

        }
    });

        bt4.setOnClickListener(new View.OnClickListener(){
                public void onClick(View v) {
                        // Perform action on click
                        //textIn.setText("test");
                        //txt2.setText("text2");
                        //task.execute(null);
                txt1.setText("null");
                txt5.setText("null");

                }

        });





    }


    public void client() throws IOException{


        DatagramSocket client_socket = new DatagramSocket(2362);
        InetAddress IPAddress =  InetAddress.getByName("10.80.1.95");

        //while (true)
        //
{
                send_data = str.getBytes();
                //System.out.println("Type Something (q or Q to quit): ");

            DatagramPacket send_packet = new DatagramPacket(send_data,str.length
(), IPAddress, 2362);
            client_socket.send(send_packet);
          //chandra
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
                client_socket.receive(receivePacket);
```

```
                modifiedSentence = new String(receivePacket.getData());
                //System.out.println("FROM SERVER:" + modifiedSentence);
                if(modifiedSentence.charAt(2)=='%')
                 txt5.setText(modifiedSentence.substring(0, 3));
                else
                        txt1.setText(modifiedSentence);
                modifiedSentence=null;
                client_socket.close();

          // }

        }

 }
```

Android WatchDogDemo

Android sample WatchDogDemo test

(Android module CCWMX53) Android program, Demonstrates how to access Watch Dog in Android on Digi modules.

Test files

This sample program contains several files, the /src folder contains the source files.

WatchDogDemo test sample application

The Android Watch Dog DemoTest sample application can be found here: WatchDogDemoHome.zip.

Basic usage

Compile, load and run program using Android environment.

Sample of WatchDogDemoHome.java file:

```
package com.digi.wddemo;

import com.digi.wdandrolib.WDLib;

import android.app.Activity;
import android.os.Bundle;
import android.os.CountDownTimer;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.TextView;

public class WatchDogDemoHome extends Activity {

        private Button btnWDtest;
        private Spinner timeOutSpinner;
        private Spinner keepAliveSpinner;
        private int timeOut;
        private int keepAliveValue;
        private int wdHandler;
        private TextView counterTextField;
        private TextView welcomeTextField;
        private int initialized = 0;

        WDLib wdObject = new WDLib();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.home);

        btnWDtest = (Button) findViewById(R.id.startWDbutton);
        timeOutSpinner = (Spinner) findViewById(R.id.timeOutSpinner);
```

```
        keepAliveSpinner = (Spinner) findViewById(R.id.KeepalivesSpinner);
        counterTextField = (TextView) findViewById(R.id.timeRemaining);
        welcomeTextField = (TextView) findViewById(R.id.wText);

        configureGUI();
        configureWD();
    }
    void configureGUI() {
        timeOutSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {
            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                timeOut = Integer.parseInt(timeOutSpinner.getSelectedItem
().toString());
                Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > timeout is
:" + timeOut);
            }
            public void onNothingSelected(AdapterView<?> parent) {
                timeOut = 15;
            }
        });

        keepAliveSpinner.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {

            public void onItemSelected(AdapterView<?> parent, View view, int pos,
long id) {
                keepAliveValue = Integer.parseInt
(keepAliveSpinner.getSelectedItem().toString());
               Log.d("WDAndro", ".WatchDogDemoHome.configureGUI() > keepalive is
:" + keepAliveValue);
        }
        public void onNothingSelected(AdapterView<?> parent) {
            keepAliveValue = 15;
        }
    });
}

public void configureWD() {
    btnWDtest.setOnClickListener(
      new OnClickListener() {
              public void onClick(View view) {

                      if (initialized == 0){

                             /*
                              * Opening watchdog
                              * */
                              wdHandler = wdObject.open();
                              initialized = 1;
                              Log.d("WDAndro", ".WatchDogDemoHome.configureWD() >
fd is :" + wdHandler);
                              if(wdHandler > 0){
                                      /*
                                       * Setting timeout for watchdog
                                       * */
                                       if(wdObject.setTimeOut(wdHandler, timeOut)
== 0)
                                               Log.d("WDAndro",
```

```
".WatchDogDemoHome.configureWD() > timeout is :" + timeOut  + " seconds");
                                        else
                                                Log.d("WDAndro", "Failed to set 
Timeout");
                              }

                              /*
                               * A countdown timer to show remaining time for 
watchdog to reboot
                               * */
                               new CountDownTimer(keepAliveValue * 1000, 1000) {
                                 public void onTick(long millisUntilFinished) {
                                        counterTextField.setText("Keepalive for: " 
+ millisUntilFinished / 1000 +" seconds");
                                        Log.d("WDAndro", 
".WatchDogDemoHome.configureWD() > keepalive for :" + millisUntilFinished/1000  + 
" seconds");
                               }

                                 public void onFinish() {
                                        counterTextField.setText("Rebooting in "+ 
timeOut +" seconds");
                                 }
                               }.start();

                               /*
                                * Spawning a seperate timer for ticking watchdog 
so that this process wont hang UI
                                * */
                                new CountDownTimer(keepAliveValue * 1000, 1000) {

                                   public void onTick(long millisUntilFinished) {
                                           /*
                                            * Ticking watchdog for a second
                                            * */
                                         wdObject.keepAliveFor(wdHandler, 1);
                                   }

                                   public void onFinish() {
                                   }
                                }.start();

                         }

                 }
         });

    }


}
```

ConnectPort x

Python program for ConnectPort X products

ConnectPort X Test (Python program) This example program sets hex value to KY parameter.

Test files

This sample program contains two files. File name "ReadMe.doc" and "Set_ddo_param_KY_with_hexvalues.py".

ConnectPort hex value test sample application

The Set_ddo_param_KY_with_hexvalues.py Python Test sample application can be found here: Set_ddo_param_KY_with_hexvalues.zip.

Basic Usage

Provide input in values below;

1. DESTINATION = “Provide the Extended address or OUI of the node to which KY(encryption) needs to be set”
2. Value = Hex value for the KY parameter

```
Sample code;
# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'
```

Sample of Set_ddo_param_KY_with_hexvalues.py file:

```
''' This program accomplish setting hex value to KY(encryption) parameter
    without any error.
        KY(Link Key) - Set the 128-bit AES link key.
        KY parameter take either int or string as values,
        but we cannot provide hex values, this application helps in providing
        hex values using a user defined function to the KY parameter.
'''

import sys
import os
import zigbee
import xbee
from _zigbee import *
import time
import traceback

# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'

# Converts a character string of hex digits into a byte string
def hex_str_to_bin_str(hex_string):
```

```
    start_index = 0
    if hex_string[0:2] == "0x":
        start_index = 2
    result = ""
    for index in range(start_index, len(hex_string), 2):
        byte = int(hex_string[index:index+2], 16)
        print byte
        result += chr(byte)
        print result
return result

try:
        # ddo_set_param - set a Digi Device Objects parameter value.
        # KY(Link Key) - Set the 128-bit AES link key.
        zigbee.ddo_set_param(DESTINATION, "KY", hex_str_to_bin_str(value))
        # EE(Encryption Enable)-this parameter set the encryption enable setting.
      zigbee.ddo_set_param(DESTINATION, "EE", 1)
      # WR - Write parameter values to non-volatile memory so that parameter
modifications
      # persist through subsequent resets.
          zigbee.ddo_set_param(None, "WR", 1)
      print "Writing parameters, waiting five seconds..."
      time.sleep(5)

except Exception, e:
    print "Exception %s" %e
    straceback.print_exc()

print "end of the program"
```

Create your own display mesh command

Create your own 'disp mesh' command

A realistic Python application using the get_node_list() function.

Users of the Digi CLI (telnet/command line) are familiar with the **disp mesh** command. It shows the same basic information as the web interface mesh networking page. However, suppose you wish to see other information listed? Suppose you wish to see it sorted in extended MAC address or node identifier order? Suppose you wish to see a list containing only end-devices which are temperature sensors and include the current temperature?

You can use the getnodelist() function to create your own custom tool, which creates your own dream-list. The fully functional Python program linked below runs on any Digi ConnectPort X gateway and shows a list of nodes sorted by either node identifier, extended HW MAC addres, or product type. This sample application was created because the author tests large Zigbee systems with up to 50 devices and needs to quickly confirm if all nodes are active - and the various sort orders rapidly allows detecting if any nodes are missing.

Routines used; things you can learn

The program my_disp_mesh.py does the following:

- Uses **sys.argv[]** to specify the desired sort order from the command line.
- Uses **zigbee.getnodelist()** to obtain a list of associated nodes.
- The device_type information as part of the getnodelist response is converted to a string such as "XB24-ZB:XBee232" for a Digi Xbee RS-232 adapter running on a ZB network.
- A list of dictionary items are sorted in various ways based on dictionary keys using an inline lamda function - (for example **nodes.sort(key=lambda x: x['name']**) to sort the list of dictionaries on the key 'name'.
- The results are printed in a table form.

Sample output

```
#> Python my_disp_mesh.py help
These arguments can be used with my_disp_mesh.py:
 name    = sorts node list on Node Id
 mac     = sorts node list on HW MAC Address
 short   = sorts node list on 16 bit Address
 type    = sorts node list on Product Type
 refresh = do a mesh refresh before creating your list

#> Python my_disp_mesh.py mac

My DISP MESH - sorted on HW MAC Address
GW: [00:13:a2:00:40:3e:1c:80]! [0000]! n:TankMom XB24-ZB:X4

01: [00:13:a2:00:40:34:16:14]! [a8d2]! n:DEBI_4 t:XB24-ZB:XBee232
02: [00:13:a2:00:40:3e:15:18]! [d756]! n:ANNA_1 t:XB24-ZB:XBee232
03: [00:13:a2:00:40:3e:15:2d]! [a865]! n:BELA_2 t:XB24-ZB:XBee232
04: [00:13:a2:00:40:4a:70:7e]! [6789]! n:CALI_3 t:XB24-ZB:XBee232
```

```
05: [00:13:a2:00:40:52:29:d7]! [458a]! n:FANI_6 t:XB24-ZB:XBee232PH
06: [00:13:a2:00:40:52:29:f9]! [1234]! n:ELSA_5 t:XB24-ZB:XBee232PH

#> Python my_disp_mesh.py name

My_Disp_Mesh - sorted on Node Name + HW MAC Address
GW: [00:13:a2:00:40:3e:1c:80]! [0000]! n:TankMom XB24-ZB:X4

01: [00:13:a2:00:40:3e:15:18]! [d756]! n:ANNA_1 t:XB24-ZB:XBee232
02: [00:13:a2:00:40:3e:15:2d]! [a865]! n:BELA_2 t:XB24-ZB:XBee232
03: [00:13:a2:00:40:4a:70:7e]! [6789]! n:CALI_3 t:XB24-ZB:XBee232
04: [00:13:a2:00:40:34:16:14]! [a8d2]! n:DEBI_4 t:XB24-ZB:XBee232
05: [00:13:a2:00:40:52:29:f9]! [1234]! n:ELSA_5 t:XB24-ZB:XBee232PH
06: [00:13:a2:00:40:52:29:d7]! [458a]! n:FANI_6 t:XB24-ZB:XBee232PH
```

Download the Python code

This code only runs on a Digi ConnectPort X gateway:Python program "My_disp_mesh.zip" in ZIP form.

Device Cloud easy demo

Purpose

This page is supposed to introduce you into a sample Device Cloud application. These are the special points that are demonstrated in this demo:

- The demos shows the advantages of Device Cloud and our Drop in Networking products:

```
  - Build local intelligence with the Python programming engine that works even
without Cellular Connectivity
  - Connect to the Devices easily via Device Cloud without knowing the IP address
```

- The local intelligence toggles power control adapter either based on sensor info or I/O toggle (Push button)
- Remote connection to the demo is done via a hosted website that includes the access via webservices (XML Messages)

```
  This website shows as well the Energy information provided by the smartplug
(load, work, Current, Voltage..)
```

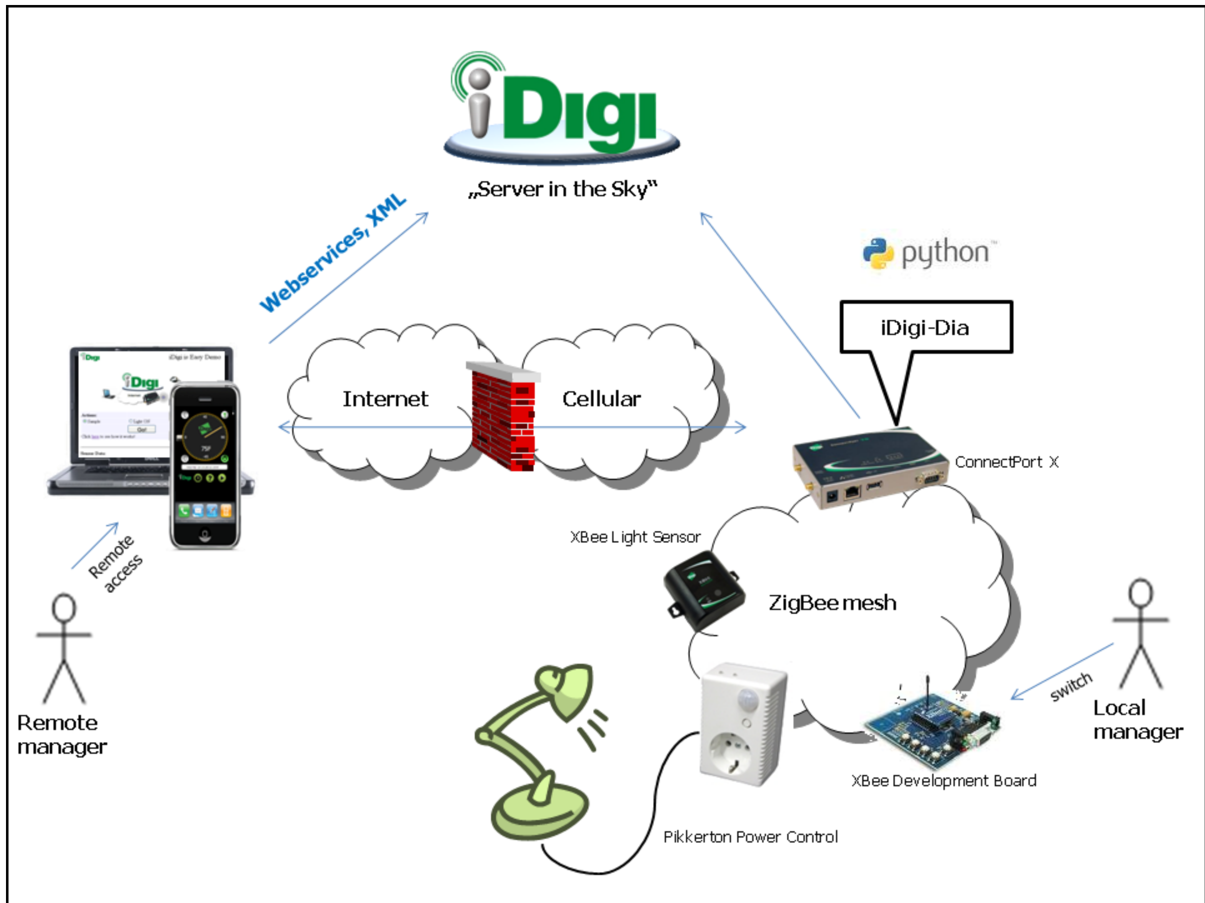
- It is also possible to connect to the device via a smartphone, showing this website.

Requirements

- ConnectPort X gateways (CPX) with cellular connectivity
- Power control device (here Third Party device: Pikkerton router)
- XBEE sensors
- XBee Development Board
- Sample power plugged device (e.g. lamp)

Introduction

The Demo configuration is shown as below:



Files

The ConnectPort X gateways (CPX) has to be equipped with these Python files:

- dia.zip (generated with make command)
- dia.py
- DigiXBeeDrivers.zip
- Python.zip
- zigbee.py

To generate the dia.zip shown above, you also need the complete DIA demo folder which can be downloaded in the last section of this page.

Getting started

1. first step...
2. second step...

How it works / How it looks

1. Local intelligence

    - Switch 2 on the XBee Development Board turns on the power control adapter. => Light on.
    - Switch 3 turns it off
    - Switch 4 activates the "auto mode": If the /L/T/H Sensor Adapter measures a light value below a specific level, the power control is turned on, otherwise it is turned off.
    - LED 1 represents the state of the power control adapter.
    - LED 2 represents the state of the auto mode.

Please visit IDigi Easy Demo Details - Support for iDigi Easy Demo for details.

2. Remote Access

    - You can access to the power control information via the Device Cloud server and using any browser-enabled device.
    - A dedicated web site allows you to remotely change the state of the demo (switching on/off manually or (de-)activating the auto mode.

Here is a screenshot of the iDigi Easy Demo web interface:

Interesting code

We will give you a short introduction in the key source code, soon with more detail.

Beside the standard DIA library, you need three new pieces:

```
 - device driver for the smartplug
 - presentation that contains the local intelligence
 - the .yml file, here is an example
```

Source

IDigiEnergyDemo_Extrafiles.zip

Device Cloud RPM demo

This topic is under construction

Purpose

This page is supposed to introduce you into a further sample Device Cloud application. These are the special points that are demonstrated in this demo:

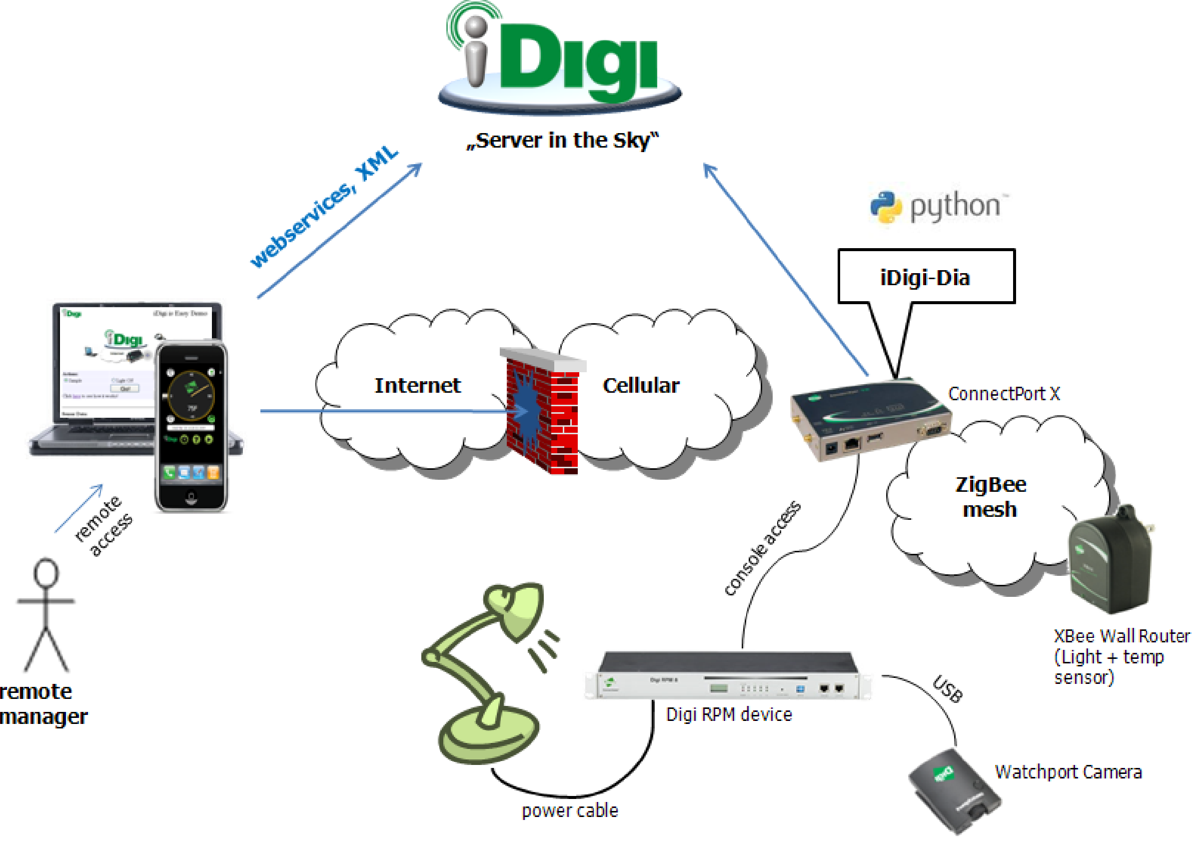
- The application is available via Device Cloud service although the ConnectPort X has a **private IP** in the cellular network.
- There is local intelligence that remotely toggles a Digi RPM device if some sensor values are dropping below a specific level.
- It is also possible to configure the power control via a website.

Requirements

- ConnectPort X gateways (CPX) with cellular connectivity
- Digi RPM device
- XBee Wall Router
- Module: camera
- Sample power plugged device (e.g. lamp)

Introduction

The Demo configuration is shown as below:



Files

The ConnectPort X gateways must be equipped with these Python files:

- dia.zip (generated with make command)
- dia.py
- DigiXBeeDrivers.zip
- Python.zip
- zigbee.py

To generate the dia.zip shown above, you also need the complete DIA demo folder which can be downloaded in the last section of this page.

Getting started

1. first step...

Please don't forget to make sure that the Serial port is free for use by Python. Otherwise you cannot open the COM1 in the Python code.

In the CLI the setting should be:

```
set term port=1 state=off
```

In the CPX website the Serial Config should show:

```
Port 1 None Custom 9600  8N1 or
Port 1 None Unassigned Custom 9600  8N1
```

Don't forget to reboot after changing the setting!

2. second step...

How it works / How it looks

1. Local intelligence

If the XBee Wall Router measures a light value below a specific level, the RPM power outlet is turned on, otherwise it is turned off.

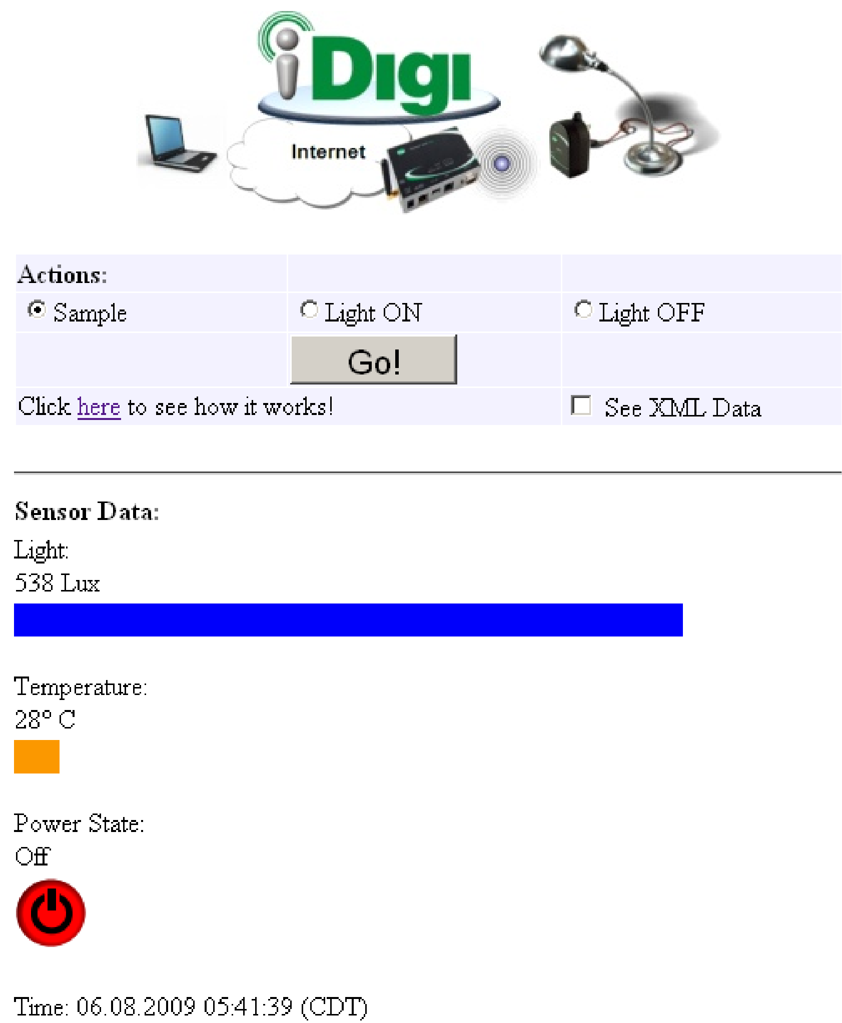
Please refer to Remote Power Management Demo for details on how to configure an Digi RPM device via Python.

2. Remote Access

You can access to the power control information via the Device Cloud server and using any browser enabled device.

A dedicated web site allows you to remotely change the state of the demo (switching on/off manually or (de-)activating the auto mode.

Here is a screenshot of the iDigi Energy Demo web interface:



Interesting code

We will give you a short introduction in the key source code, soon.

Source

There is no source, yet.

DogFighter

DogFighter sample test

(**For java supported modules**) Java program; An image rendering sample.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java DogFighter test sample application

The DogFighter Test sample application can be found here: DogFighter.zip.

Basic usage

Compile, load and run program using java environment.

Sample of DogFighter.java file:

```
package com.digi.DogFighter;

import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.image.BufferStrategy;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferInt;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class DogFighter extends Canvas implements Runnable {
        private static final long serialVersionUID = 1L;

        private static final int WIDTH = 320;
        private static final int HEIGHT = 240;
        private static final int SCALE = 2;

        private boolean running;
        private Thread thread;

        private Game game;
        private Screen screen;
        private BufferedImage img;
        private int[] pixels;
//      private InputHandler inputHandler;
//      private Cursor emptyCursor, defaultCursor;
//      private boolean hadFocus = false;

        // handles the FPS counter
        private long elapsed;
```

```
        private long start_time;

        Random random = new Random();

        public DogFighter() {
                Dimension size = new Dimension(WIDTH * SCALE, HEIGHT * SCALE);
                setSize(size);
                setPreferredSize(size);
                setMinimumSize(size);
                setMaximumSize(size);

                game = new Game();
                screen = new Screen(WIDTH, HEIGHT);
                img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_
RGB);
                pixels = ((DataBufferInt) img.getRaster().getDataBuffer
()).getData();
                screen.pixels = pixels;

                // inputHandler = new InputHandler();

               // addKeyListener(inputHandler);
               // addFocusListener(inputHandler);
               // addMouseListener(inputHandler);
               // addMouseMotionListener(inputHandler);
//             emptyCursor = Toolkit.getDefaultToolkit().createCustomCursor(
//                             new BufferedImage(16, 16, BufferedImage.TYPE_INT_ARGB),
//                             new Point(0, 0), "empty");
//             defaultCursor = getCursor();
        }

        public synchronized void start() {
               if (running)
                      return;
               running = true;
               thread = new Thread(this);
               thread.start();
       }

        public synchronized void stop() {
                if (!running)
                         return;
                running = false;
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        @Override
        public void run() {
                int count = 0;
                elapsed = 0;
                start_time = System.nanoTime();
                game.init();

                //runs the game for one tick at a time
                while (running) {
```

```
                // used to estimate FPS
                if (count >= 200) {
                        elapsed = (System.nanoTime() - start_time) / 200;
                        start_time = System.nanoTime();
                        count = 0;
                }

                // run the game
                game.doTick();

                // render it
                render();

                count++;
        }

}

private void render() {
        // get the next frame buffer (we use triple buffering?)
        BufferStrategy bs = getBufferStrategy();
        if (bs == null) {
                createBufferStrategy(3);
                return;
        }

        // have the screen object render the game
        screen.render(game);

        // draw in the FPS counter
        screen.draw("FPS:" + (int) (1 / (elapsed / 1000000000.0)), 0, 0);

        // draw the screen onto the BufferedImage
        //for (int i = 0; i < WIDTH * HEIGHT; i++)
{
        //pixels[i] = screen.pixels[i];
        //}

        // get the next frame buffer
        Graphics g = bs.getDrawGraphics();
        // draw the screen to it.
        g.fillRect(0, 0, getWidth(), getHeight());
        g.drawImage(img, 0, 0, WIDTH * SCALE, HEIGHT * SCALE, null);
        g.dispose();
        // flip the buffer to the front
        bs.show();
}


// creates and configures the window
public static void main(String[] args) {
        DogFighter df = new DogFighter();

        JFrame frame = new JFrame("DogFighter Sample");

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(df, BorderLayout.CENTER);

        frame.setContentPane(panel);
```

```
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setResizable(false);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);

                df.start();
         }
}
```

Dry contact monitoring using serial signal lines

Purpose

This demo shows how to monitor two dry contacts using the serial port of a ConnectPort X8 (CPX)
Note: You could use any Digi device that has a serial port and Python.

A sample application could be a door reed switch, which short-circuits a pair of serial signal lines.

We recommend the use of the following signal lines:

Dry Contact 1: DCD, DTR (DB9 Pin 1 and 4)

Dry Contact 2: RTS, CTS (DB9 Pin 7 and 8)

Getting started

- Please upload all files listed at the end of this page on the CPX.
- For initial testing, open a telnet connection to the CPX.
- Make sure that the serial port is active for Digi CLI access: Type in "set term". The term state should be "on". If it´s not, type "set term state=on" and reboot the device.
- Run the Python application, just type "Python serial.py".
- If the application should start up automatically, please use an auto-start slot at the CPX configuration page (**Applications->Python->Auto-start Settings**).
- You can integrate this code into your application environment.

How it works

Please have a look at the source code.

```
import os, sys
import digicli
import time

oldstateCTS = digicli.digicli('display serial')[1][5][16:19]
oldstateDCD = digicli.digicli('display serial')[1][5][28:31]

while True:
    time.sleep(0.1)
    stateCTS = digicli.digicli('display serial')[1][5][16:19]
    stateDCD = digicli.digicli('display serial')[1][5][28:31]
    if not stateCTS == oldstateCTS:
        print "Signal CTS changed to %s!" % stateCTS
    if not stateDCD == oldstateDCD:
        print "Signal DCD changed to %s!" % stateDCD
    oldstateCTS = stateCTS
    oldstateDCD = stateDCD
```

Source

Please unpack the following ZIP archive and upload the .py files on the CPX.

Dry_Contact_Monitor.zip

Fleet management demo

This demos the using a LVDVD-S AutoTap Streamer, GPS and a Digi Gateway to monitor a vehicle.

Requirements

- LVDVD-S AutoTap Streamer
- XBee RS-232 serial adapter
- ConnectPort X8 or ConnectPort X4
- NEMA 0183 GPS support on gateway
- Digi Connectware Manager server (v 3.5 or greater)

Set up

The Gateway must be associated with the XBee RS-232 serial adapter, as well as be able to make a TCP connection to the Connectware server. To verify association with the XBee RS-232 serial adapter, perform a discovery on the mesh network. The AutoTap Streamer must be connected to the XBee RS-232 adapter. Under Remote Management on the gateways web interface configure it so it will connect to the Connectware server.

Code overview

AutoTap_Gateway.py provides the driver for the XBee AutoTap Streamer. It sends and parses messages through the socket to the AutoTap Streamer. There is a high level class, AutoTapStreamer, which is intended for external use. The constructor basically takes in the extended address of a node, and then allows you to perform high level interaction with the AutoTap Streamer.

```
class AutoTapStreamer:
    """Provides high level access to the LVDVD-S AutoTap Streamer.
    It provides simplified methods for ensuring that the device is capable
    of communicating with the vehicle, and then retrieving the desired
    information once communication has been established.
    """
    PID_NAME_MAP = {0x00:"Vehicle Speed", 0x01:"Engine Speed",
                  0x02:"Throttle Position", 0x03:"Odometer",
                  0x04:"Fuel Level", 0x05:"Fuel Level Remaining",
                  0x06:"Transmission Gear", 0x08:"Ignition Status",
                  0x09:"MIL Status", 0x0A:"Airbag Dash Indicator",
                  0x0B:"ABS Dash Indicator", 0x0C:"Fuel Rate",
                  0x0D:"Battery Voltage", 0x0E:"PTO Status",
                  0x0F:"Seatbelt Fastened", 0x10:"Misfire Monitor",
                  0x11:"Fuel System Monitor", 0x12:"Comprehensive Component
Monitor",
                  0x13:"Catalyst Monitor", 0x14:"Heated Catalyst Monitor",
                  0x15:"Evaporative System Monitor", 0x16:"Secondary Air System
Monitor",
                  0x17:"A/C System Refrigerant Monitor", 0x18:"Oxygen Sensor
Monitor",
                  0x19:"Oxygen Sensor Heater Monitor", 0x1A:"EGR System
Monitor",0x1B:"Brake Switch Status", 0x1D:"Cruise Control Status",
                  0x1E:"Turn Signal Status", 0x1F:"Oil Pressure Lamp",
```

```
                  0x20:"Brake Indicator Light", 0x21:"Coolant Hot Lamp",
                  0x22:"Trip Odometer", 0x23:"Trip Fuel Consumption"}

    PID_UNIT_MAP = {0x00:"MPH", 0x01:"RPM",
                  0x02:"%", 0x03:"Miles",
                  0x04:"%", 0x05:"Gallons",
                  0x06:"", 0x08:"",
                  0x09:"", 0x0A:"",
                  0x0B:"", 0x0C:"Gallons per Hour",
                  0x0D:"Volts", 0x0E:"",
                  0x0F:"", 0x10:"",
                  0x11:"", 0x12:"",
                  0x13:"", 0x14:"",
                  0x15:"", 0x16:"",
                  0x17:"", 0x18:"",
                  0x19:"", 0x1A:"",
                  0x1B:"", 0x1D:"",
                  0x1E:"", 0x1F:"",
                  0x20:"", 0x21:"",
                  0x22:"Miles", 0x23:"Gallons"}

    def __init__(self, addr_extended):
        """Create an instance of the AutoTapStreamer using COM1 for
communication"""
        self.parameterCallbacks = set()
        self.frameManager = FrameManager(addr_extended,
self.handleTimeBasedParameterUpdate)

    def close(self):
        self.frameManager.close()

    def forceRedetect(self):
        """Force the AutoTap Streamer to retrieve vehicle information.

        The AutoTap Streamer saves information about the vehicle it is connected
to
        in order to reduce the time it takes to go from power up to being ready
for
        communication.  This command needs to be called when moving the AutoTap
Streamer
        to a new vehicle, so it will retrieve the necessary information from the
vehicle

        Returns True on a successful send, False otherwise.
        """


        return bool(self.frameManager.forceRedetect())

    def readyForCommunication(self):
        """Return communication state of the AutoTap Streamer.

        It may take up to a minute for the AutoTap Streamer to perform
        startup procedures and be ready to communicate with the vehicle.


        This method returns True if the device is ready for communication
        with the vehicle, False if it is still pending, or None if we were
        unable to detect the state of the device.
```

```
        """

        return self.frameManager.getDeviceIsReady()

    def getVIN(self):
        """Return VIN of the vehicle.

        Returns the VIN of the vehicle, else False on a failure to read
        """

        returnValue = self.frameManager.getVIN()
        if returnValue == None:
            return False
        else:
            return returnValue

    def getDiagnosticTroubleCodes(self):
        """Return list of diagnostic trouble codes.
        Returns a list of 5 character diagnostic codes, or False on
        failure to retrieve.
        """

        returnValue = self.frameManager.getDTCs()
        if returnValue == None:
            return False
        else:
            return returnValue

    def getSupportedParameters(self):
        """Return list of supported parameters for the vehicle
        Returns a list of Parameter IDs that are supported by this vehicle, or
False if we
        were unable to retrieve them.
        """

        returnValue = self.frameManager.getSupportedParameters()
        if returnValue == None:
            return False
        else:
            return returnValue


    def getParameterValues(self, parameters):
        """Retrieve current vehicle parameters
        Return a diction mapping the requested parameters to their current value,
or False
        on a failure during retrieval.  Up to 11 parameters can be requested per
month.
        """
        values = self.frameManager.getParameters(parameters)
        if values == None:
            return False

        returnData = {}

        for pid,val in zip(parameters, values):
            returnData[pid] = val

        return returnData
```

```
    def enableTimeBasedRetrieval(self, parameter, enable, interval):
        """Configure automatic reporting of given parameter.
        For the given parameter, enable=True will turn on automatic
        updates for the parameter, where enable=False will disable this
functionality.
        interval is given as a multiple of 50 ms, ranging from 1 (50 ms update)
to 65535
        for an interval of 54.6 minutes.
        Parameter updates will be returned asynchrously through the callback
specified
        in the addParameterCallback function.

        Returns True on successful configuration, False otherwise.
        """
        return self.frameManager.enableTimeBasedRetrieval(parameter, enable,
interval) != None

    def enableTimeBasedMode(self, enable):
        """Configure automatic parameter update mode
        This method turns on or off the entire time based
        updating mode.
        Returns True on successful configuration, False otherwise.
        """

        return self.frameManager.enableTimeBasedMode(enable) != None

    def addParameterCallback(self, callback):
        """Register a callback for parameter updates.
        The function should take two parameters, the first being the
        parameter ID, and the second being the value of the function
        """
        self.parameterCallbacks.add(callback)

    def removeParameterCallback(self, callback):
        """Remove a registered parameter callback function.
        Returns True if a parameter was found and removed,
        or False if the function was not registered.
        """

        try:
            self.parameterCallbacks.remove(callback)
        except KeyError:
            return False

        return True

    def handleTimeBasedParameterUpdate(self, paramMap):
        for callback in self.parameterCallbacks:
            callback(paramMap)

    def convertValueToReadableFormat(self, pid, incomingValue):
        value = math.floor(incomingValue)
        readableValue = "Invalid Input"

        if pid == 0x06: #Transmission
            if value == 0:
                readableValue = "Unknown"
            elif value == 1:
```

```
                readableValue = "Park"
            elif value == 2:
                readableValue = "Neutral"
            elif value == 3:
                readableValue = "Drive"
            elif value == 4:
                readableValue = "Reverse"
        elif pid == 0x08 or pid == 0x09 or pid == 0x0A or pid == 0x0B or pid ==
0x0E or pid == 0x1D or pid == 0x1F or pid == 0x20 or pid == 0x21:
            if value == 0:
                readableValue = "On"
            elif value == 1:
                readableValue = "Off"
        elif pid == 0x0F:
            if value == 0:
                readableValue = "Yes"
            elif value == 1:
                readableValue = "No"
        elif pid >= 0x10 and pid <= 0x1A:
            if value == 0:
                readableValue = "Complete"
            elif value == 1:
                readableValue = "Not Complete"
        elif pid == 0x1B:
            if value == 0:
                readableValue = "Pressed"
            elif value == 1:
                readableValue = "Not Pressed"
        elif pid == 0x1E:
            if value == 0:
                readableValue = "Left"
            elif value == 1:
                readableValue = "Right"
            elif value == 2:
                readableValue = "Off"
        else:
            readableValue = str(round(incomingValue, 2))

         return readableValue
```

AutoTap_Gateway_Demo.py is the main logic for the application. It opens up a serial port to grab an NMEA stream, passes it to nmea.py for parsing:

```
threading.Thread(target=gpsListener).start()
# Listener for the NMEA stream, we read and feed it to the
# NMEA parser
def gpsListener():
        print "running..."
        # select vars
        rlist = []
        wlist = []
        xlist = []
        # Create serial connection
        rlist.append(serialfd)
        while True:
            ready = select(rlist, wlist, xlist)
            if serialfd in ready[0]:
                gps.feed(os.read(serialfd, 16384))
```

Then combines that data along with an instantiation of the AutoTapStreamer class to collect all of the data for the demo.

```
# Grabs the latest data from the GPS and AutoTap Streamer, and then
# formats it into an XML string
def getAutoTapXML():
    pos = gps.position()
    print "Got Pos: " + str(pos)
    time = gps.time()

    print "Got Time: " + str(time)
    if time[1] == "000000" or time[1] == "":
        return (False, "<error>Failed to get time from GPS</error>")
    troubleCodeString = ""
    vin = autoTap.getVIN()
    if vin == False:
        vin = "Not Available"

    troubleCodes = autoTap.getDiagnosticTroubleCodes()
    if troubleCodes == False:
        troubleCodes = ["Not Available"]
    elif len(troubleCodes) == 0:
        troubleCodes = ["None"]

    supportedParameters = autoTap.getSupportedParameters()
    if supportedParameters == False:
        supportedParameters = []

    fail_count = 0

    allParamValMap = {}
    params = []
    for pid in supportedParameters:
        vals = autoTap.getParameterValues([pid])
        if vals != False:
            for pidReturned in vals.keys():
                allParamValMap[pidReturned] = vals[pidReturned]
        else:
            fail_count += 1
            if fail_count > 2:
                return (False, "<error>Failed to retrieve at least 3 automotive
parameters</error>")

    # Creating XML
    doc = Document()
    deviceSampleElement = doc.createElement("device_vehicle_sample")
    deviceSampleElement.setAttribute("sec", str(time[0])[4:6])
    deviceSampleElement.setAttribute("min", str(time[0])[2:4])
    deviceSampleElement.setAttribute("hour", str(time[0])[0:2])
    deviceSampleElement.setAttribute("day", str(time[1])[0:2])
    deviceSampleElement.setAttribute("month", str(time[1])[2:4])
    deviceSampleElement.setAttribute("year", str(time[1])[4:6])
    doc.appendChild(deviceSampleElement)
    deviceElement = doc.createElement("device")
    deviceElement.setAttribute("id", VEHICLE_NAME)
    deviceSampleElement.appendChild(deviceElement)
    positionElement = doc.createElement("position")
    deviceElement.appendChild(positionElement)
    latElement = doc.createElement("lat")
```

```
    latElement.setAttribute("value", str(pos[0]))
    positionElement.appendChild(latElement)
    lonElement = doc.createElement("lon")
    lonElement.setAttribute("value", str(pos[1]))
    positionElement.appendChild(lonElement)
    automotiveElement = doc.createElement("automotive")
    deviceElement.appendChild(automotiveElement)
    dtcElement = doc.createElement("diagnostic_trouble_codes")
    dtcElement.setAttribute("display", "Diagnostic Trouble Codes")
    dtcElement.setAttribute("value", troubleCodeString)
    dtcElement.setAttribute("units", "")
    automotiveElement.appendChild(dtcElement)

    for pid in allParamValMap:
        pidElement = doc.createElement(XML_PID_NAME_MAP[pid])
        pidElement.setAttribute("display", AutoTapStreamer.PID_NAME_MAP[pid])
        pidElement.setAttribute("value", autoTap.convertValueToReadableFormat
(pid, allParamValMap[pid]))
        pidElement.setAttribute("units", AutoTapStreamer.PID_UNIT_MAP[pid])
        automotiveElement.appendChild(pidElement)

    print doc.toprettyxml(indent="  ")

    return (True, doc.toprettyxml(indent="  "))
```

At a certain interval it uses these two pieces to push up automotive and location data to a Connectware server.

```
# Grab the XML and then upload it to the Connectware data service
def uploader():
    count = 0
    while True:
        print "Getting XML"
        (result, xml) = getAutoTapXML()
        if result == False:
            cwm_data.send_cwm_data(xml, "error.xml", secure=False)
            print "Encountered Error: %s" % xml
        else:
            filename = "fleetmanagementsample_%i.xml" % (count)

            try:
                (success, code, msg) = cwm_data.send_cwm_data(xml, filename,
secure=False)
            except:
                success = False
            if success:
                print "Succeeded in pushing sample %i to Connectware" % (count)
                count += 1
                count %= WRAP_NUMBER
            else:
                print "Failed to send sample to Connectware"

        print "Sleeping"
        time.sleep(SLEEP_TIME)
```

Notes

This document does not go into detail on how to send or retrieve data with the Connectware Manager Server.

Source

Fleetmanagement.zip

GPS Data UDP Forwarder

Summary

This demo application runs forever, reading in NMEA sentences (output data from a GPS device) and forwards a subset of the data to one or more remote hosts via UDP/IP. The hypothetical system used for the demo is a Digi CP device with GPS within the vehicle, which is to forward ALL GPS data via Ethernet to a computer mounted within the vehicle, plus only send the RMC (Recommended Minimum Navigation Information) via cellular UDP/IP to a central Vehicle-Location-System (VLS).

To save data costs, the RMC message for the central VLS is only sent once per minute when the vehicle is moving, and once per 15 minutes if the vehicle is idle or the GPS signal is in error. This is important since the GPS device used in this demo creates about 200MB of data per month - and this model is a fairly QUIET one! Other GPS devices with more features can create 5 or 10 times more traffic.

Tested with the following equipment

- The USGlobalStat BU-353 USB GPS Receiver ($35 to $40 online - available everywhere; search for the model and you'll find offers galore)
- The scripts running on a Windows XP computer, with the GPS auto installed as COM6 (your's might vary)
- The scripts running on a Digi ConnectPort X4, with the GPS auto installed as /gps/0

Note that the method used for serial data allows this script to run on EITHER a Windows computer with the GPS installed as a USB-based serial port, or on Digi ConnectPort products supporting the GPS service (X4,X8,CP-WAN and so on) Check Virtual GPS NMEA Access to make sure your model and firmware includes this service.

Files required to run

Upload all of these to your Digi product's Python directory:

- digi_gps.py – main file, run this file
- digi_gps_config.py – edit this file to set the DESTINATION info, such as IP address, forward rates etc.
- digi_nmea.py – parses the GPS information stream

- digi_serial.py – the hardware-independent “serial port” object; works on the CPX4 and also under Windows
- Python_ext.zip - several standard Python modules needed – including the copy module

Other files

- pyserial-2.2.win32.exe pyserial module from sourceforge is required to run this under Windows. The Linux version of pyserial would allow this script to run under LINUX, however the serial port notation may change since I don't know where the GPS module will install itself.
- udp_sink_2101.py – simple dummy UDP host/server; listens on UDP port 2101 and prints (with timestamp) any ASCII messages seen
- udp_sink_2102.py – same as udp_sink_2101.py, but listens on UDP port 2102

When it runs, you will see SOMETHING like this (results may vary :-)

```
central does not want <GSA> sentences
local sending <$GPGSA,A,3,10,24,30,21,,,,,,,,,5.0,2.8,4.2*3C
>
>>SPEED: 0.98 Kts - GPS says we are IDLE - not moving
local sending
<$GPRMC,183037.000,A,4453.9342,N,09324.9895,W,0.98,30.37,090209,,*20
>
central does not want <GGA> sentences
local sending <$GPGGA,183038.000,4453.9342,N,09324.9898,W,1,04,2.8,307.4,M,-
31.7,M,,0000*6F
>
central does not want <GSA> sentences
local sending <$GPGSA,A,3,10,24,30,21,,,,,,,,,5.0,2.8,4.2*3C
>
>>SPEED: 1.19 Kts - GPS says we are IDLE - not moving
local sending
<$GPRMC,183038.000,A,4453.9342,N,09324.9898,W,1.19,27.91,090209,,*20
>
```

Hopefully the comments in the file digi_gps_config.py explain what is required. Just note that you CANNOT use Windows NotePad.exe to edit these files because it mixes DOS and UNIX formats. Instead, WordPad.exe (also included in Windows) does NOT cause this problem

ZIP of the files

Digi_gps.zip

Extensions

This design could be extended (by you) to do the following:

- Use TCP/IP instead of UDP/IP. Since each GPS destination maintains its own socket, that socket could be changed to TCP or even SSL. However, TCP will increase your cost by 60% to 95%, and my own tests show UDP/IP is very reliable over cellular - a few dropped packets per 10,000.

- This demo has only 2 speeds for sending packets: 1 for idle and 1 for moving, yet a truck traveling at 60 miles-per-hour might benefit from 3 times more GPS data forwards than one traveling at 20 miles-per-hour. Thus make the rate at which packets are sent vary based on the change in position would help maximize central accuracy while minimizing data costs.
- Encrypt the payload with a simple AES algorithm - of course your host would need to use the same keys to decrypt.
- Manually monitor the time in the Digi product, and update if it drifts. Just be aware that GPS time IS NOT true time. Unlike UTC and time you get from a cell tower or via the Internet, GPS time is not adjusted for the wobble in the earth's path and so on. Thus there is a fudge adjustment you need to add, which today is about 15 seconds ... read online about GPS to learn how.
- Support a simple web page showing GPS info.

GetConnectTankAttributes

GetConnectTankAttributes sample test

(**For java supported modules**) This application gets the connect tank attributes from the device cloud and displays it to the user.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java GetConnectTankAttributes test sample application

The GetConnectTankAttributes Test sample application can be found here: GetConnectTankAttributes.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.DCT;
//imports
import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JT
xtArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

/* Brief description of the terms used in the program.
- JPanel - JPanel is a generic lightweight container
- JLabel - A JLabel object can display either text, an image, or both
- JTextField -  JTextField is a lightweight component that allows the editing of
a single line of text
- JButton - An implementation of a "push" button when it is clicked an event
should takes place
- JTable - The JTable is used to display and edit regular two-dimensional tables
of cells
- Font - Setting Font for the Components
*/

public class Main extends JFrame {


	/**
	 *
	 */
	 private static final long serialVersionUID = 1L;

	 private JTextField txtUsername;
	 private JTextField txtPassword;
	 private JTable tblDevices;
	 private JTable tblAttributeList;
	 private DeviceListTableModel devicesTableModel;
	 private AttributeTableModel AttributeListTableModel;

	 private String username;
	 private String password;
	 private String highlightedDevice;
	 public String indicator = null;
	 public String login_indicator = null;
	 final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
	 final JLabel credentialsInforLabel = new JLabel("Enter Device Cloud
credentials, click Connect and please wait for few seconds!!");
	 private static final Color cl_black = new Color(21, 45, 60);
	 private static final Color cl_dkgray = new Color(110, 110, 110);
	 private static final Color cl_grn = new Color(12, 130, 68);
	 public Font label = new Font("Times New Roman", Font.BOLD, 15);
	 final JLabel wrongDeviceInfo;
	 JPanel devicesScrollPanePanel = new JPanel();
	 JPanel attributeScrollPanePanel = new JPanel();
	 JPanel motherPanel = new JPanel();

	 JPanel basePanel = new JPanel();
	 JPanel extraPanel = new JPanel();
```

```
        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public Font fontDevice = new Font("Times New Roman", Font.PLAIN, 13);
        public Font fontAttribute = new Font("Times New Roman", Font.PLAIN, 15);
        public final JLabel waitInfo = new JLabel("Please select Connect Tank
device and wait!!");


        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }
        //constructor of the Main class which is responsible for the GUI
        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 950);

                setTitle("Digi Connect Tank");

                //adding main panel to the main window
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));
                credentialsInforLabel.setSize(2, 2);
                credentialsInforLabel.setFont(font_bold);
                credentialsInforLabel.setForeground(cl_grn);

                // panel which holds the user info
                JPanel credentialsPanel = new JPanel();
                credentialsPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
5));
                credentialsPanel.add(credentialsInforLabel);
                // adding Panels to the mother panel
                motherPanel.add(credentialsPanel);
                // adding basePanel which holds other panels to main panel i.e.,
mother panel
                motherPanel.add(basePanel);
```

```
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                //userPanel which holds username and username textfield
                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
                // declaring a Label
                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);
                userLabel.setFont(label);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);

                //passPanel which holds password label and password textfield
                JPanel passPanel = new JPanel();
                //adding passPanel to basePanel
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
                passLabel.setFont(label);

                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);

                //Panel which holds URL combo box
                JPanel dropListConnectPanel = new JPanel();
                //adding panel to basic panel
                basePanel.add(dropListConnectPanel);
                final JLabel correctDeviceInfo = new JLabel("Latest values sent
by the Connect Tank to Device Cloud!");
                correctDeviceInfo.setSize(5, 5);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(cl_grn);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select Connect Tank and wait for few seconds!");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(cl_grn);

                waitInfo.setVisible(false);
                waitInfo.setSize(5, 5);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(cl_grn);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(cl_grn);
                wrongCredentialsInfo.setFont(font_bold)        ;
```

```
                    //combo box which has list of URL's
                    JComboBox dropDownURLList = new JComboBox();
                    dropDownURLList.addItem(listURLItems[0]);
                    dropDownURLList.addItem(listURLItems[1]);
                    //adding URL's list to panel
                    dropListConnectPanel.add(dropDownURLList);
                    dropDownURLList.setSize(50,25);
                    dropDownURLList.setFont(label);

                    // declaring a button
                        JButton btnConnect = new JButton("Connect");
                        //adding button to the panel
                        dropListConnectPanel.add(btnConnect);
                        btnConnect.setOpaque(true);
                        btnConnect.setSize(new Dimension(50, 25));
                        //set font to button
                        btnConnect.setFont(label);

                        //adding scroll panel to main motherPanel
                        motherPanel.add(devicesScrollPanePanel
                        devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));
                        //adding user info to panel
                        devicesScrollPanePanel.add(wrongCredentialsInfo);
                        devicesScrollPanePanel.add(waitInfo);
                        //declaring a scroll pane
                        JScrollPane devicesScrollPane = new JScrollPane();
                        //adding scroll pane to panel
                        devicesScrollPanePanel.add(devicesScrollPane);
                        //table which displays all the devices in teh user account
                        tblDevices = new JTable();
                        tblDevices.setFillsViewportHeight(true);
                        tblDevices.setPreferredScrollableViewportSize(new Dimension
(800, 150));
                        devicesScrollPane.setViewportView(tblDevices);
                        tblDevices.setColumnSelectionAllowed(true);
                        // object of deviceLidtTableModel
                        devicesTableModel = new DeviceListTableModel();
                        tblDevices.setModel(devicesTableModel);
                        tblDevices.setRowHeight(22);
                        //set font for the table
                        tblDevices.setFont(fontDevice);
                        // Setting column width
                        tblDevices.getColumnModel().getColumn(0).setPreferredWidth
(140);
                        tblDevices.getColumnModel().getColumn(1).setPreferredWidth
(300);
                        tblDevices.getColumnModel().getColumn(2).setPreferredWidth
(125);
                        tblDevices.getColumnModel().getColumn(3).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(4).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(5).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(6).setPreferredWidth
(124);
```

```
                    //adding terminalOuter to main motherPanel
                    motherPanel.add(terminalOuter);

                    terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                    terminalOuter.add(cmdInputPanel);
                    cmdInputPanel.add(correctDeviceInfo);
                    cmdInputPanel.add(wrongDeviceInfo);

                    terminalOuter.add(attributeScrollPanePanel);

                    JScrollPane attributeListScrollPane = new JScrollPane();

                    attributeScrollPanePanel.add(attributeListScrollPane);

                   //table which displays attributes of a selected device
                   tblAttributeList = new JTable();
                   tblAttributeList.setFillsViewportHeight(true);
                   tblAttributeList.setPreferredScrollableViewportSize(new
Dimension(600, 300));
                   attributeListScrollPane.setViewportView(tblAttributeList);
                   tblAttributeList.setColumnSelectionAllowed(true);
                   AttributeListTableModel = new AttributeTableModel();
                   tblAttributeList.setModel(AttributeListTableModel);
                   tblAttributeList.getColumnModel().getColumn
(0).setPreferredWidth(0);
                   tblAttributeList.setRowHeight(40);
                   tblAttributeList.setFont(fontAttribute);
                   tblAttributeList.getColumnModel().getColumn
(1).setPreferredWidth(130);
                   tblAttributeList.getColumnModel().getColumn
(2).setPreferredWidth(60);


                   terminalOuter.setVisible(true);

                   //listener for Connect button
                   btnConnect.addActionListener(new ActionListener() {
                           public void actionPerformed(ActionEvent arg0) {
                                   //method to call after clicking Connect button
                                   btnConnect_onClick();
                           }
                   });

                   //listener for dropdownlist combo box
                dropDownURLList.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                            chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                    }
                  });

                //listener for devices table
                    tblDevices.addMouseListener(new java.awt.event.MouseAdapter()
{
                            public void mouseClicked(java.awt.event.MouseEvent e)
{

                                    int row=tblDevices.rowAtPoint(e.getPoint());
```

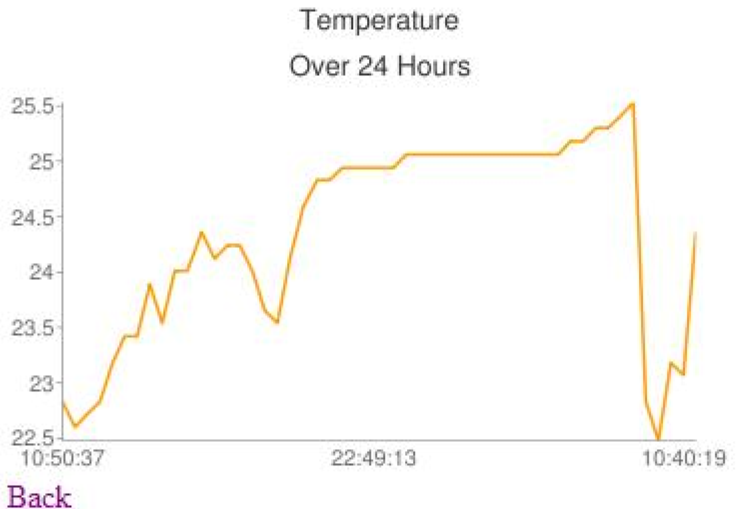
```
                                    correctDeviceInfo.setVisible(false);
                                    wrongDeviceInfo.setVisible(false);

                                    highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                    try {
                                            // calls connect_cloud()
                                            indicator =
AttributeListTableModel.connect_cloud(username, password,chosenServer,
highlightedDevice);
                                    if(indicator.compareTo("correct device") !=
0){

                                            waitInfo.setVisible(false);
                                            correctDeviceInfo.setVisible(false);
                                            wrongDeviceInfo.setVisible(true);
                                            tblDevices.removeAll();
                                            tblAttributeList.removeAll();

                                            }
                                    else
                                    {
                                            correctDeviceInfo.setVisible(true);
                                    }
                                    }
                                    catch(Exception e1){
                                            System.out.println(e1);
                                    }
                                    System.out.println(indicator);

                            }
                    });

                    this.pack();

                    this.setVisible(true);

        }
        // this method is called when you click Connect button
        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                wrongCredentialsInfo.setVisible(false);
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        waitInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }
```

```
}
```

Google App Engine Device Cloud Client

Purpose

The Google App Engine Device Cloud Client is one example of a client application that uses web service (XML) communication to a Digi gateway using Device Cloud. It is deployed to a Google Appspot account and runs on the servers provided by Google. The advantage of this hosted service is that there is no server infrastructure needed at the customer and applications can run 24/7 on the Google servers, making it an ideal solution if continuous sensor readings are required, e.g. for daily/weekly statistics of sensor data, displayed to the users in form of graphs accessible over the web:



Requirements

Hardware

Digi ConnectPortX ZB Gateway, Digi ZB Wall Router and one embedded XBee ZB module on development board. Recommended kits that include these pieces are

- Device Cloud Professional Development Kit ZB
- Device Cloud X4 Starter Kit ZB

Software

- Digi Python Development Environment (Digi ESP to be able to create and build DIA projects and execute them on the gateway
- Python 2.5
- Google App Engine SDK for Python (requires a Google Appspot account)

Usage

Installation

1. Create DIA project that includes drivers at least for the Digi Wall Router and the XBee ZB module on development board. Device names should be *xbr0* for the wall router and xbib0 for the development board. In addition to the drivers it is mandatory that the DIA project includes the RCI Handler presentation to enable the gateway to talk to the Device Cloud. Optional the *Embedded web* presentation can be included for local tests.

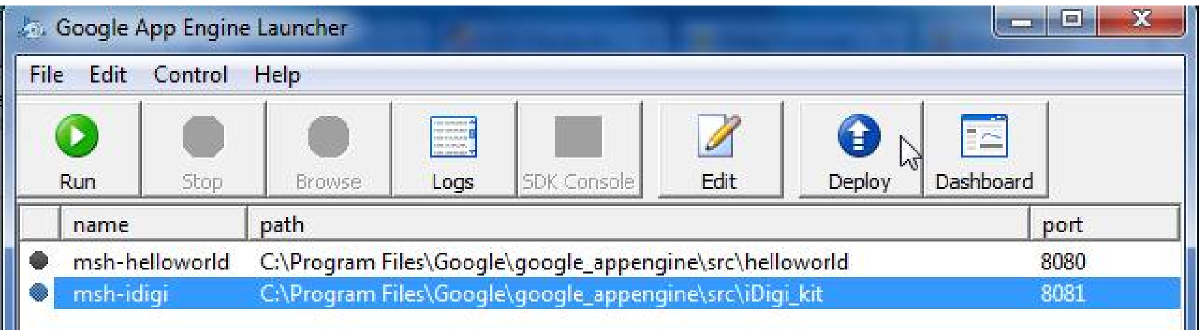
   Recommended is to start with the **iDigi Professional Development Kit** sample that can be found in the ESP by choosing File | New | iDigi DIA sample project. This sample comprises the drivers with correct device names as well as the necessary presentations mentioned above. For more information, consult the Device Cloud Professional Development Kit ZB Getting Started Guide (see steps 6 and 7)

- Run DIA project on the Gateway as explained in the Getting Started Guide.
- If available, test with embedded web page if wall router and development board are working as expected.
- Add Digi gateway to your developer.idigi.com account as explained in the Getting Started Guide (steps 1 and 2) and make sure it shows up as Connected.
- Install Python 2.5 and the Google App Engine SDK for Python.
- Extract the necessary source files to a subfolder of your Google App Engine SDK installation, by default \Program Files\Google\google_appengine.
- Use a text editor to open Config.py in the iDigi_kit folder. Change the username and password fields with the data of your Device Cloud developer account and modify the Device ID field with the Device ID of your gateway. Change the other fields if necessary and if you know what you are doing.

- Create a new application in your Google Appspot account, e.g. "<your name>-idigi" and choose any title for the application, e.g. "iDigiGoogleApp"

Deployment

1. Open the Google App Engine Launcher and
2. Add the Device Cloud client with **File | Add Existing Application** and choosing the iDigi_kit subfolder extracted previously, change the name of the application to match the application that you created on your Google Appspot account by selecting the application in the list, clicking Edit and modifying the application: entry in the file.
3. Deploy the application by selecting it in the list and clicking **Deploy**.



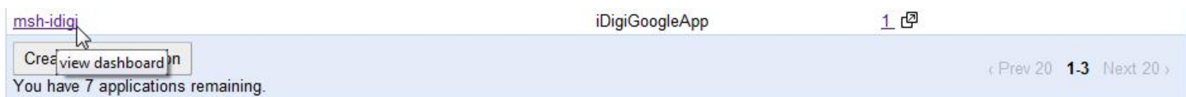
4. close the deployment window when instructed to do so.

These steps can be repeated as often as you change the application source code.

To check the status and output of the application.

1. Use the Current Version link in your Google Appspot application list to see the output of the application, click on the gauges to see the corresponding graphs.

2. After a few moments of operation, check the **Logs, Cron Jobs** and **Datastore Viewer** views in your application Dashboard view (accessible by clicking on the application name in the Google Appspot application list) to ensure the application operates as expected. **Cron Jobs** should list a job */update (sensor data update) running every 2 minutes (UTC)*, the **Datastore Viewer** should list sensor values collected from the wall router and development boards.



Current limitations

- Only Celsius temperature values are supported
- Device names need to be xbr0 and xbib0

Sources

Feel free to edit sources to your needs:

Google_appengine_iDigi_kit.zip

How to send email via smtp

Purpose

This page is to show how to send email from Digi Python enabled devices using the SMTP protocol.

Requirements

- Python enabled Digi device
- SMTP server

Introduction

The standard Python smtplib.py is compatible with our device with the exception of one function call 'getfqdn()'. In the Digi compliant version, this call is replaced by a custom function call 'get_host()'. The 'get_host()' call returns the hostname of the device using the digicli [link needed] library.

Special Code You Need to Add

```
  def get_host():
    (flag, response) = digicli.digicli('set host')

    if flag:
      for line in response:
        line = line.strip()
        line = line.lower()

        if line.startswith('name'):
          hostname = line.split(':', 1)[1]
          hostname = hostname.strip()

          ##Special string indicating hostname was not set
          if hostname == '(not set)' or len(hostname) == 0:
            return None

          return hostname
      else:
        raise ValueError("Unable to locate the hostname field in the returned
response")
    else:
      raise ValueError("Received error when querying device for hostname")
```

The 'get_host()' call interacts with the device using the 'set host' command. The 'set host' returns the hostname of the device. There are two conditions of this command returning None, if the hostname field has been reverted using the command 'revert host', which results in the field having a value of '(Not set)' or if the returned hostname has a length of zero.

Example

Make sure the hostname of the device is set (*use set host name="HOSTNAME"* command and also consider the code above). Also be sure that you are using an accessible SMTP server in the local

network. Upload smtlib_digi.py and the modified Python.zip. Use this example script (change the <...> entries at the beginning to appropriate values):

```
import smtplib_digi

fromaddr = "<SENDING_EMAILADDRESS>"
toaddrs  = "<RECEIVING_EMAILADDRESS>"
subject = "<Email for you>"
text = "<YOUR MESSAGE>"
smtp_server = '<SMTP_SERVER>'

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\nSubject: %s\r\n"
       % (fromaddr, toaddrs, subject))
msg = msg + "\r\n" + text

print "Message length is " + repr(len(msg))

server = smtplib_digi.SMTP(smtp_server)
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Tip You can get a simple SMTP Server for Win32 that is free for non-commercial use at [1].

Source

- Smtplib_digi.zip
- Smtplib_Python.zip

IOT Demo TradeShow

IOT_Demo_TradeShow sample test

(**For java supported modules**) This application lists attributes and provides a way to turn the FAN ON and OFF.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java IOT_Demo_TradeShow Test Sample Application

The IOT_Demo_TradeShow Test sample application can be found here: IOT_Demo_TradeShow.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.etherios;

import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import com.digi.etherios.DataTableModel;
import com.digi.etherios.DeviceListTableModel;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

public class Main extends JFrame {

       /**
        *
        */
        private static final long serialVersionUID = 1L;

        private JTextField txtUsername;
        private JTextField txtPassword;
        private JTextField txtRefreshInterval;
        private JTable tblDevices;
        private JTable tblAttributeList;

        private DeviceListTableModel devicesTableModel;
        private DataTableModel dataTableModel;

        private String username;
        private String password;
        private String interval;
        private String highlightedDevice;
        public String indicator = null;
        public String login_indicator = null;
        final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
        private static final Color cl_black = new Color(21, 45, 60);
        private static final Color cl_btn_grn = new Color(10, 148, 54);
        private static final Color cl_dkgray = new Color(110, 110, 110);
        private static final Color cl_ltgray = new Color(153, 153, 153);
        private static final Color cl_white = new Color(255, 255, 255);
        final JLabel wrongDeviceInfo;
        JPanel devicesScrollPanePanel = new JPanel();
        JPanel cmdOutputScrollPanePanel = new JPanel();
        JPanel motherPanel = new JPanel();

        JPanel basePanel = new JPanel();
        JPanel extraPanel = new JPanel();

        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        private static String[][] data = null;
        private static int rowCount = 0;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public final JLabel waitInfo = new JLabel("Please select device on which
IOT_demo program is running and wait for few seconds!!");
        boolean clickedonce = false;
```

```
        private static String[] ValueList = null;

        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }

        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 750);

                setTitle("Digi Connect Tank");
                //getContentPane().setLayout(new GridLayout(0, 1, 0, 0));
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));

                motherPanel.add(basePanel);
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);


                JPanel passPanel = new JPanel();
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
```

```
                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);
               JPanel dropListConnectPanel = new JPanel();
                basePanel.add(dropListConnectPanel);
                JPanel intervalPanel = new JPanel();
                basePanel.add(intervalPanel);

                final JLabel correctDeviceInfo = new JLabel("Latest values of
selected device in Device Cloud and will be refreshed based on interval!!");
                correctDeviceInfo.setSize(4, 4);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(Color.BLACK);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select device on which IOT_demo program is running");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(Color.BLACK);


                waitInfo.setVisible(false);
                waitInfo.setSize(7,7);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(Color.BLACK);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(Color.black);

                JComboBox dropDownURLList = new JComboBox();
                dropDownURLList.addItem(listURLItems[0]);
            dropDownURLList.addItem(listURLItems[1]);
            dropListConnectPanel.add(dropDownURLList);
            dropDownURLList.setBackground(cl_dkgray);
            dropDownURLList.setForeground(cl_black);
            dropDownURLList.setSize(50,25);

            JLabel intervalLabel = new JLabel("  Interval in min");
                intervalLabel.setForeground(cl_black);
                intervalLabel.setBackground(cl_dkgray);
                intervalLabel.setSize(60,25);
                dropListConnectPanel.add(intervalLabel);

                txtRefreshInterval = new JTextField();
                dropListConnectPanel.add(txtRefreshInterval);
                txtRefreshInterval.setColumns(5);

                JButton btnConnect = new JButton("Connect");
                dropListConnectPanel.add(btnConnect);

                Font f = new Font("Arial", Font.BOLD, 13);
                btnConnect.setOpaque(true);
                btnConnect.setBackground(cl_dkgray);
                btnConnect.setForeground(cl_black);
                btnConnect.setSize(new Dimension(50, 25));
```

```
                btnConnect.setFont(f);

                motherPanel.add(devicesScrollPanePanel);
                devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));

                devicesScrollPanePanel.add(wrongCredentialsInfo);
                devicesScrollPanePanel.add(waitInfo);
                JScrollPane devicesScrollPane = new JScrollPane();
                devicesScrollPanePanel.add(devicesScrollPane);


                tblDevices = new JTable();
                tblDevices.setFillsViewportHeight(true);

                tblDevices.setPreferredScrollableViewportSize(new Dimension(800,
150));

                devicesScrollPane.setViewportView(tblDevices);
                tblDevices.setColumnSelectionAllowed(true);
                devicesTableModel = new DeviceListTableModel();
                tblDevices.setModel(devicesTableModel);
                tblDevices.setRowHeight(22);
                tblDevices.getColumnModel().getColumn(0).setPreferredWidth(140);
                tblDevices.getColumnModel().getColumn(1).setPreferredWidth(300);
                tblDevices.getColumnModel().getColumn(2).setPreferredWidth(125);
                tblDevices.getColumnModel().getColumn(3).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(4).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(5).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(6).setPreferredWidth(124);
                motherPanel.add(terminalOuter);

                terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                terminalOuter.add(cmdInputPanel);
                cmdInputPanel.add(correctDeviceInfo);
                cmdInputPanel.add(wrongDeviceInfo);
                terminalOuter.add(cmdOutputScrollPanePanel);

                JScrollPane cmdOutputsListScrollPane = new JScrollPane();

                cmdOutputScrollPanePanel.add(cmdOutputsListScrollPane);

                tblAttributeList = new JTable();
                tblAttributeList.setFillsViewportHeight(true);

                tblAttributeList.setPreferredScrollableViewportSize(new Dimension
(600, 350));


                cmdOutputsListScrollPane.setViewportView(tblAttributeList);
                tblAttributeList.setColumnSelectionAllowed(true);


                dataTableModel = new DataTableModel();
                tblAttributeList.setModel(dataTableModel);
                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(0);
                tblAttributeList.setRowHeight(40);
```

```
                tblAttributeList.setFont(font);

                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(60);
                tblAttributeList.getColumnModel().getColumn(1).setPreferredWidth
(130);
                tblAttributeList.getColumnModel().getColumn(2).setPreferredWidth
(60);


                terminalOuter.setVisible(true);

                btnConnect.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                btnConnect_onClick();
                        }
                });


            dropDownURLList.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                }
            });

                tblDevices.addMouseListener(new java.awt.event.MouseAdapter(){
                        public void mouseClicked(java.awt.event.MouseEvent e){


                                if (clickedonce  == false){
                                        clickedonce = true;

                                        int row=tblDevices.rowAtPoint(e.getPoint
());
                                        correctDeviceInfo.setVisible(false);
                                        wrongDeviceInfo.setVisible(false);


                                        highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                        try {
                                                indicator =
dataTableModel.connect_cloud(username, password, chosenServer,
highlightedDevice);
                                                BasicThread refreshThread = new
BasicThread(username, password, chosenServer, highlightedDevice,
interval,dataTableModel);
                                                refreshThread.start();

                                        if(indicator.compareTo("correct device")
!= 0){
                                                System.out.println("indicator :
"+indicator);
                                                correctDeviceInfo.setVisible
(false);
                                                wrongDeviceInfo.setVisible(true);
                                                tblDevices.removeAll();
                                                tblAttributeList.removeAll();
```

```
                                                }
                                        else
                                        {
                                                correctDeviceInfo.setVisible
(true);

                                        }
                                }
                                        catch(Exception e1){
                                                System.out.println(e1);
                                        }
                                        System.out.println(indicator);

                                }
                                else {
                                        System.out.println("Clicked already");
                                }
                        }


                });
                this.pack();

                this.setVisible(true);
        }

        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                interval = txtRefreshInterval.getText();
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }

}
```

LibDeviceCloud

LibDeviceCloud sample test

(**For java supported modules**) This program is a library for communicating to the device cloud in java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibDeviceCloud Test Sample Application

The LibDeviceCloud Test sample application can be found here: LibDeviceCloud.zip.

Basic usage

Compile, load and run program using java environment.

Sample of MonitorTest.java file:

```
package com.digi.devicecloud.test;

import java.io.IOException;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

import com.digi.devicecloud.DeviceCloud;
import com.digi.devicecloud.MonitorConfig;
import com.digi.devicecloud.Result;
import com.digi.devicecloud.WsMonitor;
import com.digi.devicecloud.monitor.MonConnectionRequestPacket;
import com.digi.devicecloud.monitor.MonConnectionResponsePacket;
import com.digi.devicecloud.monitor.MonPacket;
import com.digi.devicecloud.monitor.MonPublishMessagePacket;
import com.digi.devicecloud.monitor.TcpMonitor;
import com.digi.devicecloud.monitor.TcpMonitorListener;
import com.digi.json.JsonArray;
import com.digi.json.JsonObject;

public class MonitorTest implements TcpMonitorListener {
        private static String username = "";
        private static String password = "";
        private static String hostname = "login.etherios.com";
        private static final int MONITOR_ID = 109537;

        private DeviceCloud cloud = new DeviceCloud(hostname, username,
password);


        public static void main(String[] args) {
                MonitorTest test = new MonitorTest();

                try {
                        test.listenMonitor();
```

```
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

        while (true) {
                try {
                        Thread.sleep(10000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

}


private void start() {
        try {

                WsMonitor monitorWs = new WsMonitor(cloud);

                List<Result> results = monitorWs.list();

                for (int i = 0; i < results.size(); i++) {
                        Result result = results.get(i);

                        JsonArray items = result.getItems();

                        for (int j = 0; j < items.size(); j++) {
                                JsonObject obj = items.getJsonObject(j);

                                System.out.println(obj.display());
                        }
                }

                // create a monitor
                // createMonitor();
                // listen to monitor
                // listenMonitor();

                // while (true) {
                // Thread.sleep(1000);
                // }

        } catch (Exception ex) {
                ex.printStackTrace();
        }
}


private void createMonitor() throws IOException, ParseException {
        WsMonitor monitorWs = new WsMonitor(cloud);

        MonitorConfig config = new MonitorConfig();

        // using compresion
        config.setCompression(MonitorConfig.COMPRESSION_ZLIB);
        // using JSON
        config.setFormatType(MonitorConfig.FORMAT_JSON);
```

```
                // listening to Device Core changes
                config.setTopic(MonitorConfig.TOPIC_DEVICE_CORE);
                // using TCP/IP
                config.setTransportType(MonitorConfig.TRANSPORT_TYPE_TCP);

                // create the monitor
                Result result = monitorWs.create(config);

                // print the monitor ID
                System.out.println(result.getDataAsString());
        }


        private void listenMonitor() throws InterruptedException {
                // create a new monitor with a high timeout
                TcpMonitor monitor = new TcpMonitor(900000);

                // add myself as a listener
                monitor.addListener(this);

                // start the monitor
                monitor.start("login.etherios.com", 3200, false);

                // create the request packet
                MonConnectionRequestPacket request = new
MonConnectionRequestPacket(
                username, password, MONITOR_ID);

                // send the packet
                monitor.sendPacket(request);

        }


        @Override
        public void tcpMonitorIncommingPacket(TcpMonitor tcpMonitor, MonPacket
packet) {
                System.out.println("Recieved: " + packet.getType());

                if (packet.getType() == MonPacket.TYPE_PUBLISH_MESSAGE) {
                        MonPublishMessagePacket publish =
(MonPublishMessagePacket) packet;

                        try {
                                System.out.println(new JsonObject(new String(
                                                publish.getPayload())).display
());
                        } catch (ParseException e) {
                                e.printStackTrace();
                        }
                }

                if (packet.getType() == MonPacket.TYPE_CONNECTION_RESPONSE) {
                        MonConnectionResponsePacket response =
(MonConnectionResponsePacket) packet;

                        System.out.println("Connection Response: " +
response.getStatus());
                }
```

```
        }


        @Override
        public void tcpMonitorConnected(TcpMonitor tcpMonitor) {
                // TODO Auto-generated method stub

        }
}
```

LibFastDb

LibFastDb sample test

(**For java supported modules**) This program is an interface for easily accessing MySQL from java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibFastDb Test Sample Application

The LibFastDb Test sample application can be found here: LibFastDb.zip.

Basic usage

```
Compile, load and run program using java environment.
```

Sample of FastDb.java file:

```
package com.digi.fastdb;

import java.lang.reflect.Field;
import java.sql.PreparedStatement;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.digi.fastdb.utils.utils;

public class FastDb {
	private static FastDb _instance = null;

	private Set<String> _tables = new HashSet<String>();


	public static FastDb getInstance() {
		if (_instance == null) {
			_instance = new FastDb();
		}
		return _instance;
	}


	private FastDb() {

	}


	public boolean objectExists(Object object) throws Exception {
		checkTables(object.getClass());
		return utils.objectExists(object);
	}


	public void writeNewObject(Object object) throws Exception {
```

```
                checkTables(object.getClass());
                utils.writeNewObject(object);
        }


       /**
        * Attempts to push an update to the store based on the object's primary
key
        *
        * @param object
        * @throws Exception
        */
        public void pushObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.pushObject(object);
        }

       /**
        * If the object exists.. then the object is updated. Otherwise a new one
is
        * created.
        *
        * @param object
        * @throws Exception
        */
        public void saveObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.saveObject(object);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement) throws Exception {
                checkTables(klass);
                return getObjects(klass, statement, true);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement, boolean includeChildren) throws Exception {
                checkTables(klass);
                return utils.getObjects(klass, statement, includeChildren);
        }


        public void deleteObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.deleteObject(object);
        }


        public void removeLink(Object parent, String fieldName, Object
fieldValue) throws Exception {
                checkTables(parent.getClass());

                Field field = parent.getClass().getDeclaredField(fieldName);

                utils.deleteLink(parent, field, fieldValue);
        }
```

```
        private void checkTables(Class<?> klass) throws Exception {
                String tableName = utils.getTableName(klass);

                if (_tables.contains(tableName)) {
                        return;
                }

                if (utils.tableExists(tableName)) {
                        _tables.add(tableName);
                        return;
                }

                List<String> schemas = utils.generateTableSchema(klass);
                for (int i = 0; i < schemas.size(); i++) {
                        utils.createTable(schemas.get(i));
                }


        }
}
```

LibJil

LibJil sample test

(**For java supported modules**) This program is an interpreter for the JIL language.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJil Test Sample Application

The LibJil Test sample application can be found here: LibJil.zip.

Basic usage

Compile, load and run program using java environment.

Sample of JilProgram.java file:

```
package com.digi.jil;

import java.util.Hashtable;

import com.digi.jil.lang.JilStatement;
import com.digi.json.JsonObject;

public class JilProgram {
        private JilStatement _rootNode;
        private Hashtable<String, JilStatement> _labels;


        public JilProgram(JsonObject source) throws Exception {
                _labels = new Hashtable<String, JilStatement>();
                _rootNode = JilStatementFactory.newStatement(this, source);
                _rootNode.setOid("r");
        }


        public void step(JilContext jilContext) throws Exception {
                String oid = jilContext.getStatementOid();

                if (oid == null) {
                        jilContext.setStatementOid("r");
                        oid = "r";
                }

                JilStatement statement = find(oid);

                JilResult result = statement.execute(jilContext);

                switch (result.state) {
                // there was a horrible error
                        case JilResult.RESULT_ERROR:
                                handleError(result, jilContext);
                                break;
                        case JilResult.RESULT_GOTO:
```

```
                                handleGoto(result, jilContext);
                                break;
                        case JilResult.RESULT_NEXT_SIBLING:
                                handleNextSibling(result, jilContext);
                                break;
                        case JilResult.RESULT_COMPLETE:
                                handleComplete(result, jilContext);
                                break;
                }
        }

        public void registerLabel(JilStatement label) {
                _labels.put(label.getLabelName(), label);
        }


        public JilStatement lookupLabel(String labelName) {
                if (_labels.containsKey(labelName)) {
                        return _labels.get(labelName);
                }

                return null;
        }


        private void handleComplete(JilResult result, JilContext jilContext)
throws Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_COMPLETE);
        }


        private void handleError(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_FAILED);
        }


        private void handleGoto(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setStatementOid(result.nextOid);
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
                // TODO, set other stuff !?
        }


        private void handleNextSibling(JilResult result, JilContext jilContext)
throws Exception {
                // find next command
                String oid = result.nextOid;

                if (oid == null || "r".equals(oid)) {
                        // we are done!
                        jilContext.setStatementOid(oid);
                        jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_
COMPLETE);
                        return;
                }

                oid = JilUtils.nextSibling(oid);
```

```
                JilStatement statement = find(oid);
                while (statement == null) {
                        oid = JilUtils.parseParent(oid);

                        if (oid == null || "r".equals(oid)) {
                                // we are done!
                                jilContext.setStatementOid(oid);
                                jilContext.setExecutionStatus(JilStatement.EXEC_
STATUS_COMPLETE);
                                return;
                        }

                        oid = JilUtils.nextSibling(oid);

                        statement = find(oid);
                }

                jilContext.setStatementOid(oid);
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
        }


        public JilStatement find(String oid) {

                return _rootNode.find(oid);
        }

}
```

LibJson

LibJson sample test

(**For java supported modules**) This program is a JSON parser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJson Test Sample Application

The libJson Test sample application can be found here: LibJson.zip

Basic usage

Compile, load and run program using java environment.

Sample of JsonArray.java file:

```
package com.digi.json;

import java.text.ParseException;
import java.util.LinkedList;
import java.util.List;

public class JsonArray {
        private List<Object> _objects = new LinkedList<Object>();


        public JsonArray() {

        }


        public JsonArray(String string) throws ParseException {
                JsonTokenizer tokenizer = new JsonTokenizer(string);
                fromTokenizer(tokenizer);

        }


        public JsonArray(JsonTokenizer tokenizer) throws ParseException {
                fromTokenizer(tokenizer);
        }


        private void fromTokenizer(JsonTokenizer tokenizer) throws ParseException
{
                if (!tokenizer.isNextTokenStartArray()) {
                        throw new ParseException("Not an array!", -1);
                }
                // pop the open brace
                tokenizer.nextToken();

                // check for empty array
                if (tokenizer.isNextTokenFinishArray()) {
```

```
                tokenizer.nextToken();
                return;
        }

        while (true) {
                _objects.add(tokenizer.parseValue());

                // if not comma, then it should've been a ']'
                if (!tokenizer.nextToken().equals(",")) {
                        break;
                }
        }

}


public int size() {
        return _objects.size();
}


public Object get(int index) {
        return _objects.get(index);
}


public Object get(String path) {
        List<String> dir = JsonTokenizer.parsePath(path);

        return get(dir);
}


protected Object get(List<String> directions) {
        if (directions.size() == 0)
                return this;

        String value = directions.remove(0);

        value = value.replace("[", "");
        value = value.replace("]", "");

        int item = Integer.parseInt(value);

        Object obj = _objects.get(item);
        if (obj instanceof JsonObject) {
                return ((JsonObject) obj).get(directions);
        }
        else if (obj instanceof JsonArray) {
                return ((JsonArray) obj).get(directions);
        }

        return obj;
}

public String getString(int index) {
        return (String) get(index);
}
```

```
        public JsonObject getJsonObject(int index) {
                return (JsonObject) get(index);
        }


        public JsonArray getJsonArray(int index) {
                return (JsonArray) get(index);
        }


        public int getInt(int index) {
                return Integer.parseInt(getString(index));
        }


        public long getLong(int index) {
                return Long.parseLong(getString(index));
        }


        public float getFloat(int index) {
                return Float.parseFloat(getString(index));
        }


        public double getDouble(int index) {
                return Double.parseDouble(getString(index));
        }


        public Object set(int index, Object value) {
                while (index >= _objects.size())
                        _objects.add(null);

                return _objects.set(index, value);
        }


        protected void set(List<String> directions, Object value) throws
ParseException {
                if (directions.size() == 0)
                        throw new ParseException("Invalid path", 0);

                String item = directions.remove(0);

                item = item.replaceAll("\\[", "");
                item = item.replaceAll("\\]", "");

                int index = Integer.parseInt(item);

                if (directions.size() == 0) {
                        set(index, value);
                }
                else {
                        // already exists
                        if (_objects.size() > index) {
                                Object obj = _objects.get(index);
```

```
				if (obj instanceof JsonObject) {
					JsonObject jo = (JsonObject) obj;

					jo.put(directions, value);
					return;
				}
				else if (obj instanceof JsonArray) {
					JsonArray ja = (JsonArray) obj;
					String child = directions.get(0);
					if (child.equals("[]"))
						ja.add(directions, value);
					else
						ja.set(directions, value);
					return;
				}
			}
			else {

				String child = directions.get(0);
				if (child.startsWith("[") && child.endsWith("]"))
{
					JsonArray ja = new JsonArray();

					set(index, ja);

					if (child.equals("[]"))
						ja.add(directions, value);
					else
						ja.set(directions, value);

					return;
				}
				else {
					JsonObject jo = new JsonObject();

					set(index, jo);

					jo.put(directions, value);
					return;
				}
			}

		}

	}


	public boolean add(Object value) {
		return _objects.add(value);
	}


	public void add(int index, Object value) {
		_objects.add(index, value);
	}


	protected void add(List<String> directions, Object value) throws
ParseException {
```

```
        if (directions.size() == 0)
                throw new ParseException("Invalid path", 0);

        String item = directions.remove(0);

        if (!item.equals("[]"))
                throw new ParseException("This should never happen!", 0);

        if (directions.size() == 0) {
                add(value);
        }
        else {
                String child = directions.get(0);
                if (child.startsWith("[") && child.endsWith("]")) {
                        JsonArray ja = new JsonArray();

                        _objects.add(ja);

                        if (child.equals("[]"))
                                ja.add(directions, value);
                        else
                                ja.set(directions, value);

                        return;
                }
                else {
                        JsonObject jo = new JsonObject();

                        _objects.add(jo);

                        jo.put(directions, value);
                        return;

                }

        }

}


public Object remove(int index) {
        return _objects.remove(index);
}


protected Object remove(List<String> directions) throws ParseException {
        if (directions.size() == 0)
                throw new ParseException("Invalid path!", -1);
        String item = directions.remove(0);

        item = item.replaceAll("\\[", "");
        item = item.replaceAll("\\]", "");

        int index = Integer.parseInt(item);

        if (directions.size() == 0) {
                return remove(index);
        }
        else {
```

```
                        Object obj = get(index);

                        if (obj instanceof JsonObject) {
                                JsonObject jo = (JsonObject) obj;

                                return jo.remove(directions);
                        }
                        else if (obj instanceof JsonArray) {
                                JsonArray jo = (JsonArray) obj;

                                return jo.remove(directions);
                        }

                }
                return null;

        }


        public void clear() {
                _objects.clear();
        }


        public String display() {
                return display(0).toString();
        }


        protected StringBuilder display(int depth) {
                StringBuilder sb = new StringBuilder();

                sb.append("[\n");
                for (int i = 0; i < _objects.size(); i++) {
                        Object obj = _objects.get(i);

                        sb.append(JsonTokenizer.repeat("  ", depth + 1));
                        if (obj == null) {
                                sb.append("null");
                        }
                        else if (obj instanceof JsonObject) {
                                sb.append(((JsonObject) obj).display(depth + 1));
                        }
                        else if (obj instanceof JsonArray) {
                                sb.append(((JsonArray) obj).display(depth + 1));
                        }
                        else if (obj instanceof String) {
                                sb.append(JsonTokenizer.escapeString((String)
obj));
                        }
                        else {
                                sb.append(obj + "");
                        }

                        if (i < _objects.size() - 1) {
                                sb.append(",");
                        }

                        sb.append("\n");
```

```
            }
            sb.append(JsonTokenizer.repeat("  ", depth));
            sb.append("]");

            return sb;
    }


    public StringBuilder toStringBuilder() {
            StringBuilder sb = new StringBuilder();

            sb.append("[");
            for (int i = 0; i < _objects.size(); i++) {
                    Object obj = _objects.get(i);

                    if (obj == null) {
                            sb.append("null");
                    }
                    else if (obj instanceof JsonObject) {
                            sb.append(((JsonObject) obj).toStringBuilder());
                    }
                    else if (obj instanceof JsonArray) {
                            sb.append(((JsonArray) obj).toStringBuilder());
                    }
                    else if (obj instanceof String) {
                            sb.append(JsonTokenizer.escapeString((String)
obj));
                    }
                    else {
                            sb.append(obj + "");
                    }
                    if (i < _objects.size() - 1) {
                            sb.append(",");
                    }
            }
            sb.append("]");

            return sb;
    }

    public boolean has(int index) {
            if (index < _objects.size())
                    return true;

            return false;
    }


    protected boolean has(List<String> directions) throws ParseException {
            if (directions.size() == 0)
                    throw new ParseException("Invalid path!", -1);

            String item = directions.remove(0);

            item = item.replaceAll("\\[", "");
            item = item.replaceAll("\\]", "");

            int index = Integer.parseInt(item);
```

```
		if (!has(index))
			return false;
		else if (directions.size() == 0)
			return true;

		Object obj = get(index);

		if (obj instanceof JsonObject) {
			JsonObject jo = (JsonObject) obj;

			return jo.has(directions);
		}
		else if (obj instanceof JsonArray) {
			JsonArray ja = (JsonArray) obj;

			return ja.has(directions);
		}

		return false;
	}


	@Override
	public String toString() {
		return toStringBuilder().toString();

	}

}
```

LibUtils

LibUtils sample test

(**For java supported modules**) This program is a collection of utility classes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibUtils Test Sample Application

The libUtils Test sample application can be found here: LibUtils.zip.

Basic usage

Compile, load and run program using java environment.

Sample of ConfigFile.java file:

```
package com.digi.utils;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;

/**
* Loads a simple config file into memory. Config files are of the following
* form
*
* ### Comments start with a #
* key = value
*
* @author mcarver
*
*/
public class ConfigFile {
        private static Hashtable<String, ConfigFile> _configFiles = new
Hashtable<String, ConfigFile>();

        private Hashtable<String, String> _parameters;
        private String _filename;


       /**
        * Loads the config file at filename and caches the data. If the config
file
        * is already loaded, then the cache is returned.
        *
        * @param filename
        * @return
        * @throws IOException
        */
        public static ConfigFile getInstance(String filename) throws IOException
```

```
{
                if (_configFiles.containsKey(filename)) {
                        return _configFiles.get(filename);
                }

                ConfigFile config = new ConfigFile(filename);

                _configFiles.put(filename, config);

                return config;
        }


        private ConfigFile(String file) throws IOException {
                _parameters = new Hashtable<String, String>();
                _filename = file;

                loadFile();
        }


        private void loadFile() throws IOException {

                File file = new File(_filename);

                BufferedReader stream = new BufferedReader(new InputStreamReader(
                                new FileInputStream(file)));

                String line = null;
                while ((line = stream.readLine()) != null) {

                        line = line.trim();

                        if (line.startsWith("#")) {
                                continue;
                        }

                        if (!line.contains("=")) {
                                continue;
                        }

                        String name = line.substring(0, line.indexOf("=")).trim
().toLowerCase();
                        String value = line.substring(line.indexOf("=") + 1).trim
();

                        _parameters.put(name, value);
                }

                stream.close();

        }


        /**
         * Returns the value of the parameter named name. Names are case
         * insensitive.
         *
         * @param name
```

```
	 * @return
	 */
	public String getString(String name) {
		if (!_parameters.containsKey(name.toLowerCase())) {
			return null;
		}

		return _parameters.get(name.toLowerCase());
	}
}
```

LibZkConfigProtocol

LibZkConfigProtocol sample test

(**For java supported modules**) This program is a protocol library ZooKeeper (zk.digi.com) which uses to communicate with its slave nodes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibZkConfigProtocol Test Sample Application

The libZkConfigProtocol Test sample application can be found here: LibZkConfigProtocol.zip.

Basic usage

Compile, load and run program using java environment.

Sample of FieldField.java file:

```
package com.digi.configurepackets;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class FileField extends Field {
        public StringField fileName;
        public File file;


        FileField(DataInputStream dataIn) throws IOException {
                fileName = new StringField(dataIn);
                int fileLength = dataIn.readInt();
                file = File.createTempFile("FileField_", ".tmp");

                OutputStream out = new BufferedOutputStream(new FileOutputStream
(file));
                try {
                        int pos = 0;
                        int read = 0;
                        int packet_size = 2048;
                        if (packet_size > fileLength) {
                                packet_size = fileLength;
                        }

                        byte[] data = new byte[packet_size];
                        while (pos < fileLength) {
                                read = dataIn.read(data);
```

```
                                if (read > 0) {
                                        out.write(data, 0, read);
                                        pos += read;
                                        if (fileLength - pos < packet_size) {
                                                packet_size = fileLength - pos;
                                                data = new byte[packet_size];
                                        }
                                }
                                else if (read == 0) {
                                        try {
                                                Thread.sleep(100);
                                        } catch (InterruptedException e) {}
                                }
                                else if (read < 0) {
                                        throw new IOException(
                                                        "Could not read entire file from
stream!");
                                        }
                                }

                                if (pos != fileLength) {
                                        throw new IOException(
                                                        "There was an error reading the
file from the data stream!");
                                }
                        } catch (IOException ex) {
                                out.close();
                                file.delete();
                                out = null;
                        } finally {
                                if (out != null) {
                                        out.close();
                                }
                        }

        }


        public FileField(File file) {
                fileName = new StringField(file.getName());
                this.file = file;
        }


        public FileField(String name, File file) {
                fileName = new StringField(name);
                this.file = file;
        }


        public void delete() {
                file.delete();
        }


        @Override
        public int length() {
                return fileName.length() + 4 + (int) file.length();
        }
```

```
		@Override
	public void toDataStream(DataOutputStream dataOut) throws IOException {
		fileName.toDataStream(dataOut);
		dataOut.writeInt((int) file.length());

		InputStream in = new BufferedInputStream(new FileInputStream
(file));

		try {
			int read = 0;
			byte[] packet = new byte[2048];
			while ((read = in.read(packet)) != -1) {
				if (read > 0) {
					dataOut.write(packet, 0, read);
				}
				else if (read == 0) {
					try {
						Thread.sleep(100);
					} catch (InterruptedException e) {}
				}
			}
		} finally {
			in.close();
		}
	}


}
```

Module: camera

Introduction

Built-in Python module provided by Digi for interacting with a Watchport Camera/V2 camera.

Functions

get_image()

Purpose

Retrieve the most recent camera image as a JPEG.

Syntax

```
get_image() . (image, timestamp)
```

- *image* is buffer containing a JPEG of the most recent camera image
- *timestamp* is the timestamp the image was acquired by the device, in milliseconds since device power on.

Description

Returns the most recent image from the attached Watchport camera. The camera parameters affecting the resolution, image quality, frame rate, etc are configured via the device web interface or command line interface.

The image is a standard JPEG.

The timestamp returned with the image is the time when the image was acquired by the device. The value is the same type as the uptime of the system (milliseconds since last boot time). The timestamp can be used to find the elapsed time between two images. It can also be used to determine if two calls to get_image() are the same image. Note, the timestamp of the image will be different than the current uptime of the device. This is because camera images are acquired continuously by the device based on the Frame Delay setting. The most recent image acquired by the device is what is returned by get_image() which can be as old as the value in Frame Delay.

If an image is not available (no camera attached, camera disabled, etc) (None, None) is returned.

Example

```
##Internal module
import camera

##Destination of the image
fh = open('WEB/Python/demo_image.jpg', 'wb')

##image, and timestamp
image, timestamp = camera.get_image()

##write buffer and close
fh.write(image)
fh.close()
```

The above code saves the current camera image to a file on the device file system.

Note the file handle must be opened in binary mode. Writing the image buffer without specifying 'b' will result in a corrupted image.

Availability

Products which support this module

This feature is available only on Digi products with built-in USB host ports.

Products which DO NOT support this module

This feature is not currently supported on:

- Digi ConnectPort X2/WiX2
- Digi ConnectPort X3

Module: iridium

The Iridium Module

Introduction

Built-in Python module provided by Digi for utilizing the Iridium satellite interface.

Functions

Power Control

```
iridium_power_get()
```

- In digipowercontrol module
- Returns a truth value

```
iridium_power_set(value)
```

- Accepts a truth value
- Errors result in exception
- No return value

Status

```
digi_iridium.state()
```

- No parameters
- Returns a dictionary
- Values including:

power (a truth value) serial_number (a Python string)

- --network_availability (a truth value)
- --signal_strength (0-5)
- If power is off, only power state returned

Send

```
digi_iridium.send(msg)
```

- No return value
- Error results in exception

- Blocks until message sent or rejected
- Parameter is payload of message
- Expected to be a Python string
- Always transferred as a “binary” blob
- No destination address!

Receive

```
digi_iridium.Callback(fn)
```

- Returns a callback handle
- Callback remains registered while handle exists “fn” is a one parameter function
- Parameter is message as a Python string
- Use a lambda wrapper to pass context if needed (see example)
- Incoming messages delivered to all registered callbacks
- Receive “Gotchas”
- --Controlling Latency
- --Source Address Management
- --“Hidden” Behaviors

Python Example

```
            # Roughly once per second, print updated
# statistics.  Whenever we receive a message,
# attempt to retransmit with location added

import digipowercontrol as dpc
import digi_iridium as di
import digihw as dhw
import Queue

dpc.iridium_power_set(True) # Power on the Iridium modem

data_queue= Queue.Queue() # Prepare receive path

def handle_rx_msg(input_queue, payload):
    input_queue.put(payload)

cbhandle = di.Callback(lambda msg: \
    handle_rx_msg(data_queue, msg))

txcnt = 0
rxcnt = 0
while True:
    print "RxCnt: %-10d  TxCnt: %-10d" % \
        (rxcnt, txcnt)

    try:
```

```
        msg = data_queue.get(True, 1.0)
        rxcnt = rxcnt + 1
        try:
            msg += ' ' + str(dhw.gps_location())
        except:
            msg += ' (location unknown)'
        try:
            print “Trying to transmit a message"
            di.send(msg)
            txcnt = txcnt + 1
        except:
            print "Currently unable to transmit”
        except Queue.Empty: pass
```

Code Analysis

```
            # Roughly once per second, print updated
# statistics.  Whenever we receive a message,
# attempt to retransmit with location added
import digipowercontrol as dpc
import digi_iridium as di
import digihw as dhw
import Queue
```

- Not representative of a “finished” Python app
- Libraries required for the application
- Python trick for space savings (not necessary)

```
# Power on the Iridium modem
dpc.iridium_power_set(True)
```

- Ensure that the Iridium is powered on
- Default power state of modem is “off”

```
            # Prepare receive path
data_queue= Queue.Queue()
def handle_rx_msg(input_queue, payload):
    input_queue.put(payload)
cbhandle = di.Callback(lambda msg: handle_rx_msg(data_queue, msg))
```

- Queue to pass data from callback to main code
- Lambda wrapper example, demonstrating how to pass two parameters to a callback

```
while True:
    print "RxCnt: %-10d  TxCnt: %-10d" % rxcnt, txcnt)
    try:
        msg = data_queue.get(True, 1.0)
        rxcnt = rxcnt + 1
          :
    except Queue.Empty: pass
```

- Simple sample main loop with Queue “polling”

```
try:
    msg += ' ' + str(dhw.gps_location())
except:
    msg += ' (location unknown)‘
```

- Try/except block in case the GPS location function throws an exception
- Compose response based on the received message

```
try:
    print “Trying to transmit a message“
    di.send(msg)
    txcnt = txcnt + 1
except:
    print "Currently unable to transmit”
```

- Simple transmission sample
- Transmission will fail if the satellite network is not available at transmission time
- Successful transfer to the Iridium gateway servers will “almost always” result in a successful end delivery
- No feedback to device if gateway end fails

Availability

Products which support this module

This feature is available only on the Digi Connect X5/

XBee sensor

Python program for XBee sensor

XBee sensor (Python program) This program gives a user information regaridng battery status of sensor.

Test files

This sample program contains one file. File name "check_battery_life_sensor.py".

XBee battery sensor test sample application

The check_battery_life_sensor.py Python Test sample application can be found here: Check_battery_life_sensor.zip.

Basic usage

Make sure that the sensors are in the network coordinator. Provide the inputs where neccesary.

Sample of check_battery_life_sensor.py file:

```
import sys
import os
import struct
import zigbee
import xbee
from _zigbee import *
import time
import traceback
from struct import *

''' Provide the extended address(OUI) of the xbee sensor'''
###################################################
DESTINATION="00:13:a2:00:40:86:cd:11!"
###################################################

def calculate_battery(bat_vol):
    mv = float(bat_vol)
    mv = ((mv * 1200)/1024)
    v = mv/1000
    v = round(v, 2)
    return v

try:
    print "reading parameters, waiting five seconds..."
    # %V- Supply Voltage. Reads the voltage on the Vcc pin. Scale by 1200/1024 to
    # convert to mV units.
    param_value = zigbee.ddo_get_param(DESTINATION, "%V")
    param_value = struct.unpack("h", param_value)
    str_param_value = str(param_value)        # converting into a string
    s_index = str_param_value.find("(")
    e_index = str_param_value.find(",)")
    bat_voltage = str_param_value[s_index+1:e_index]
    battery_mv = calculate_battery(bat_voltage)
    if (battery_mv) >= 2.8 and (battery_mv) <= 3.4:
        print "battery is in the range of 2.8 and 3.4 and the battery value" + \
        + "is %s" %str(battery_mv)
    else:
        print "low battery"

except Exception, e:
    print "Exception %s" %e
    traceback.print_exc()

print "end of the program"
```

Motion Detection with XBee

How to cub-class a DIA driver.

This page covers how to leverage an existing DIA driver without modifying the original file.

Motion detector example

In this example, I wish to connect a common Motion-Detector/Glass-Break device with relay contacts ('dry-contacts') via ZigBee to Device Cloud and DIA.

The motion-detector sensor requires 12vdc to run, so I used an older XBee DIO Adapter which runs on 9-30vdc. With a 12vdc supply, the entire setup can be powered including directly driving 12vdc relays and 12vdc low-voltage garden-style lighting.

Technically, the existing DIA DIO driver could be used to read the sensor contacts and drive the lamp output, however the motion detection event is very short-lived. It may be true for only a few seconds, and turning on a lamp for a few seconds is not what is required. Instead, a timer is required to turn the lamp on for a few minutes. In theory one could use the DIA transforms to do this, but at present they are stateless, so cannot function as timers.

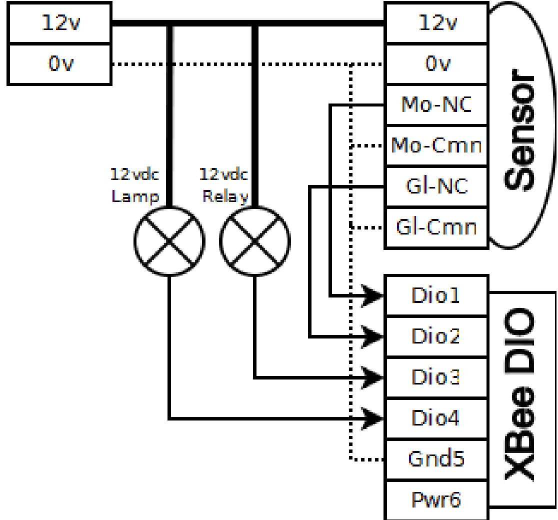
So one has two choices:

- Clone and rename the existing DIA DIO driver, and modify it directly.
- Create a new subclass of the existing DIA DIO driver which adds the new functionality.

In this demo example I chose to subclass. The new driver allows the existing DIO driver to manage incoming data and update the standard DIO data channels. Then it reads the input channels and adjusts the output channels as required.

Ideally, the time the lamp is held on should be a setting, which complicates the subclass slightly. That is left for a future TODO - including the support for the glass-break input and the 12vdc relay to drive brighter AC lights. If all goes well, my final design will have a programmable XBee to manage the light and relay control locally within the XBee adapter itself.

Hardware example



Wiring the system is quite strait forward. The items required:

- XBee Digital I/O Adapter preferably with 9-30vdc supply, but if you have a 3-6 vdc model, buy a PCB to step-down 12 vdc to 5 vdc. For example, ebay seller electronics-salon sells a nice assembled model for us$10 which I have used on several projects.
- Any 12 vdc supply large enough to power everything - anything larger than 1.5 Amp should be okay. As is, the 10-watt halogen garden lamp (at about 0.8 A) is the most power-hungry device.
- Motion Sensor, such as Bravo BV-500GB by DSC, sold by smarthome.com
- 12 vdc Low-Voltage Lamp, such as the Malibu brand in the image, purchased from Home Depo. Once the halogen lamp fails, they are easy to replace with LED versions from superbrightleds.com
- 12 vdc Power Relay with AC-rated contacts, such as Omron LY1F-DC12 from digikey.com which is rated at 15 Amp @ 110 vac. It requires 75m A @ 12 vdc to operate.

Notes

- The 12 vdc power ground link to the XBee DIO Adapter is optional if the XBee DIO Adapter is powered by the 12 vdc supply, but it is required if the XBee DIO Adapter is powered by (for example) the 5vdc wall-wart Digi will supply if you buy the 3-6 vdc XBee DIO Adapter
- The four DIO terminals of the XBee DIO Adapter when used as outputs can only sink current, so pull the 12 vdc low. That is why both the lamp and relay are tied to 12 vdc, with the XBee DIO Adapter acting as ground.
- The power-output (t erminal #6) of the XBee DIO Adapter can only supply 50 mA @ 12 vdc, which is not enough for either the landscape lamp or the power relay. However, it would be enough for direct LED lights or a smaller reed-relay which could power a larger lamp or relay. Moving an 'output' to this terminal would free up one of the XBee adapter IO for use as another input, such as a tamper contact, or perhaps even a sensor to detect if the lamp is really on and emitting light.

- The reason for having both a dim 12 vdc lamp and bright 120 vac lamp (via 12 vdc relay) is I plan to use the dim 12vdc lamp more as a night-light then as security. So based on time of day, sunlight level, and whether the security level is 'we are home' or 'we are out', either or neither lamp may be lit due to motion.
- XBee DIO Adapter DIP Switch settings: 2, 3, and 4 are on.
- For inputs, only terminals 1 & 2 have internal pull-up. To use terminals 3 & 4 as input, you may need to use resisters (for example 10K ohm) to pull the inputs up to 12vdc when floating/open.

YML code

This is just the fragment required for the custom DIO driver. We desire the DIO adapter to fresh the channel status every 30 seconds, plus the channels will be updated any time they change. The four IO are assigned like this:

- Channel 1 input is the Motion-Detect NC/normally-closed contact - it opens when motion is seen.
- Channel 2 input is the Glass-Break NC/normally-closed contact - it opens when the distinctive sound of break glass is heard. This input is for future use, and is ignored now.
- Channel 3 output is reserved for a 12 vdc relay which can drive 120 vac lamps, which would be appropriate for yard or full-room lights.
- Channel 4 output directly drives a 12 vdc garden-style lamp, which is used for demo purposes and night-lighting.

```
  - name: motion
    driver: devices.experimental.xbee_motion:XBeeMotion
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:33:4e:a9!"
        sample_rate_ms: 30000
        sleep: False
        power: Off
        channel1_dir: "In"
        channel2_dir: "In"
        channel3_dir: "Out"
        channel4_dir: "Out"
```

Source code

Here is the entire custom file. Since it uses the normal XBee DIO driver, there is little function required. Basically, this driver allows the XBee DIO driver to process the incoming data, then compares the Motion-Detect input to the current light state.

```
import traceback
import time

from devices.device_base import DeviceBase
from devices.xbee.xbee_devices.xbee_base import XBeeBase
from settings.settings_base import SettingsBase, Setting

from devices.xbee.xbee_devices.xbee_dio import *

class XBeeMotion(XBeeDIO):
```

```
    OFF_DELAY = 30

    def __init__(self, name, core_services):

        ## Initialize the base class
        XBeeDIO.__init__(self, name, core_services)

        # zero means off, else holds time.time turned on
        self.__light_on = 0

        return

    ## Locally defined functions:
    def sample_indication(self, buf, addr, force=False):

        ## allow base class to process data message
        XBeeDIO.sample_indication(self, buf, addr, force)

        # then do our reaction to the status
        now = time.time()

        motion = self.property_get( 'channel1_input').value
        if motion:
            if self.__light_on == 0:
                print 'Motion Seen, turning light on'
                self.set_output(Sample(now, True, "On"), 3)

            # else: print 'Motion Seen, light already on'

            # in all cases, bump light_on time to now
            self.__light_on = now

      else:
            # else no motion
            if self.__light_on != 0:
                # then light is on
                if (now - self.__light_on) > self.OFF_DELAY:
                    print 'No Motion, turning light off'
                    self.set_output(Sample(now, False, "Off"), 3)
                    self.__light_on = 0

                # else: print 'No Motion Seen, light is on, should stay on'

      return
```

NET OS 9P9360 external RTC

Program for 9P9360 external RTC (DS1337) in NET+OS

NET+OS 9P9360 External RTC (For NET+OS 7.4.2 - 7.5.2 modules) This example adds support for the onboard DS1337 RTC on the 9P9360 in NET+OS.

Test files

This sample program contains six files. The main function is in root.c.

9P9360 External RTC (DS1337) Test Sample Application

The 9P9360 External RTC (DS1337) Test sample application can be found here: 9P9360_External_RTC_(DS1337).zip.

Basic usage

```
Copy rtc_drv.custom over
\netos\src\bsp\devices\ns9xxx\ns9360\rtc\rtc_drv.c
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include "rtc.h"
#include <time.h>

void applicationTcpDown (void)
{
        static int ticksPassed = 0;

        ticksPassed++;
}

static NaRtcTime_t rtc_time;

static char* const WEEKDAYS[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday","Friday", "Saturday"};

static char* const MONTHS[12] = {"January", "Febuary", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};

void applicationStart (void)
{
        NaStatus rc;
        char* data;
        time_t t;

        while(1)
        {
                t = time(NULL);
                data = ctime(&t);
```

```
                printf("time(): %s", data);
                rc = naRtcGetTime(0, &rtc_time);
                switch (rc)
                {
                        case NASTATUS_SUCCESS:
                                // Wednesday, December 16, 2009 at 12:00:00
                                // day of the week/month index starts at 1, our
arrays start at 0
                                printf("Date/Time: ");
                                printf("%s, ",WEEKDAYS[rtc_time.dayOfWeek-1]);
                                printf("%s %i, %i at", MONTHS[rtc_time.month-1],
rtc_time.date, rtc_time.year);
                                printf(" %i:%i:%i\n", rtc_time.hours, rtc_
time.minutes, rtc_time.seconds);
                                break;
                        case NASTATUS_RTC_NOT_INITIALIZED:
                                printf("The real time clock is not
initialized.\n");
                                break;
                        case NASTATUS_RTC_FAIL:
                                printf("The real time clock was unsuccesffully
updated with the new time. It's given invalid rtc_time.\n");
                                break;
                        case NASTATUS_RTC_NOT_AVAILABLE:
                                printf("The real time clock doesn't support this
action.\n");
                                break;
                }
                tx_thread_sleep(500);
        }

        printf ("Hello World!\n");
        tx_thread_suspend(tx_thread_identify());
}
```

NET OS Appkit Rio

Program for controlling the Rabbit RIO chip in NET+OS

Appkit RIO Test (For NET+OS 7.4.2 - 7.5.2 modules) A driver for controlling the Rabbit RIO chip.

Test files

This sample program contains eight files. The main function is in root.c.

CPU test sample application

The Appkit Rio Driver Test sample application can be found here: RIO_Appkit_Driver.zip.

Basic usage

```
Test Hardware setup on RIO APPKIT board:
--With an LED (or Scope). Connect ground (neg) to P4.9(GND). Connect VCC(pos)
     to P4.8(RC0P0). This LED is where the majority of all the tests will be
     performed
--With a STDP(single throw, double poll) switch. Connect the common pin to
     P4.4(RC1P0). Connect one poll to VCC and the other poll to GND. This pin
     will be used for all the input tests.
BSP SETUP

Digi Connect ME 9210
Change the following in gpio.h:
#define BSP_GPIO_MUX_SERIAL_A            BSP_GPIO_MUX_SERIAL_2_WIRE_UART
#define BSP_GPIO_MUX_IRQ_1               BSP_GPIO_MUX_USE_PRIMARY_PATH
#define BSP_GPIO_MUX_IRQ_1_CONFIG                    BSP_GPIO_MUX_IRQ_FALLING_EDGE

Right Click on your project. Go to properties. Select Net+OS from the list.
Under bsp_sys.h change 'Dialog Port' and 'STDIO Port' to 'Serial Port C'

ConnectCore 9P 9215
Change the following in gpio.h:
#define BSP_GPIO_MUX_IRQ_3               BSP_GPIO_MUX_USE_3RD_ALTERNATE_PATH
#define BSP_GPIO_MUX_IRQ_3_CONFIG                    BSP_GPIO_MUX_IRQ_FALLING_EDGE
```

Known issues

The Input/Output pins on the RIO APPKIT board are all floating. This means that any pin not pulled up or down and set for an input will report random state changes due to cross talk and other interference.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

#include "rio_appkit.h"


void applicationTcpDown (void)
```

```
{
        static int ticksPassed = 0;
        ticksPassed++;
}

void TestGPIO();
void TestPWM();
void TestGPIOInput();
void TestHardwareReset();
void TestSoftReset();
void TestPPM();
void TestInterrupt();

void applicationStart (void)
{
#if (PROCESSOR == ns9215)
       /*
        * ConnectCore 9P9215 Connections
        * GPIO67 = X21.C9 - Reset Line
        * GPIO101(IRQ3) = X21.D16 - Interrupt Line (IRQ3)
        */
        naRIOInit(5,67,EXTERNAL3_INTERRUPT);
#elif (PROCESSOR == ns9210)
       /*
        * Connect ME 9210 Setup
        * GPIO13 = P3.20 - Reset Line
        * GPIO2 = P3.12 - Interrupt Line (IRQ1)
        */
        naRIOInit(5,13,EXTERNAL1_INTERRUPT);
#endif

        printf("Starting RIO Test\n");

        TestHardwareReset();
        TestSoftReset();
        TestGPIO();
        TestPWM();
        TestGPIOInput();
        TestPPM();
        TestInterrupt();

    tx_thread_suspend(tx_thread_identify());
}

void TestGPIO()
{
        int port,pin;
        int x;
        int retval;

        printf ("Starting GPIO Test...\n");
        for (x=0; x < 20; x++)
        {
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, TRUE);

                printf("All are high\n");
                tx_thread_sleep(10);
```

```
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, FALSE);

                printf("All are low\n");
                tx_thread_sleep(10);
        }
        printf ("GPIO Test complete...\n");
}

void TestPWM()
{
        int retval;
        int x,i;
       /*
        * 16666666 = 60hz
        *  20000000 = 50hz
        *  10000000 = 100hz
        */
        long period = 10000000;

        printf ("Starting PWM Test.\n");
        retval = naRIOReset(TRUE);
        retval = naRIOSetPrescaler(period);

        retval = naRIOSetPWM(0,0,period,0);
        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 100; x++)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
                for (x = 100; x >=1 ; x--)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
        }
        printf("PWM Test Complete...\n");
}

void TestHardwareReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Hardware Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should 
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(TRUE);
        printf("RIO has been reset!\n");
```

```
        printf("Hardware Reset Test complete...\n");
        tx_thread_sleep(500);
}

void TestSoftReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Software Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(FALSE);
        printf("RIO has been reset!\n");
        printf("Software Reset Test Compelte...\n");
        tx_thread_sleep(500);
}

void TestGPIOInput()
{
        BOOL val = FALSE;
        BOOL last = FALSE;
        int retval;
        int x;

        retval = naRIOReset(TRUE);

        printf("Starting GPIO Input Test Port 1, Pin 0...\n");

        retval = naRIOSetInput(1,0);
        for (x=0; x<20; x++)
        {
                // wait for a state change
                while ( retval == last)
                        val = naRIOReadInput(1,0);

                printf("Pin is now %d\n", retval);
                last = retval;
        }

        printf("GPIO Input Test Complete...\n");
}

void TestPPM()
{
        int retval;
        long period = 10000000;
        int x,i;

        naRIOReset(TRUE);

        printf("Starting PPM Test. Port 0, Pin 0...\n");
        retval = naRIOSetPrescaler(period);
```

```
        /* set pins 2,3 on port0 to a PWM */
        retval = naRIOSetPWM(0,2,period,period/3);
        retval = naRIOSetPWM(0,3,period,period/3);

        /* set pin0 to a PPM with a phase shift */
        retval = naRIOSetPPM(0,0,period,0,period/3);

       /*
        * set pin1 (the dead pin) to a GPIO.
        * pin1's match register is being used,
        * so it can only be used for GPIO
        */
        retval = naRIOSetOutput(0,1,FALSE);

        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 360; x++)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
                for (x = 360; x >=1 ; x--)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
        }

        printf("PPM Test Complete...\n");
}

void InterruptCallBack(int port, int pin)
{
        int retval;
        printf("Callback called on port: %d, pin: %d\n", port, pin);
        retval = naRIOResetGPIOInterrupt(1,0);
}

void TestInterrupt()
{
        int retval;
        printf("Starting Interrupt Test. Port 1, Pin 0...\n");
        retval = naRIOReset(TRUE);
        retval = naRIOGPIOInterrupt(1,0, InterruptCallBack);
}
```

NET OS CPU

Program to read/write CPU registers in NET+OS

CPU TEST (For NET+OS 7.4.2 - 7.5.2 modules) Shows how to read/write directly from CPU registers in NET+OS.

Test files

This sample program contains two files, root.c and readme.txt. The main function is in root.c.

CPU test sample application

The CPU Test sample application can be found here: Read-Write_CPU_Registers.zip.

Basic usage

Compile, load and run program using NET+OS.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

volatile unsigned long *reg_ptr = (unsigned long *) 0xA0902008;

void applicationStart (void)
{
        printf ("Register Value: 0x%X\n", *reg_ptr);
        *reg_ptr = (*reg_ptr) | 0x6C;
        printf ("Register Value: 0x%X\n", *reg_ptr);

    tx_thread_suspend(tx_thread_identify());
}

void applicationTcpDown (void){    static int ticksPassed = 0;    ticksPassed++;}
```

NET OS Ping

Program to generate a ping from a NET+OS module

PING TEST (For NET+OS 7.4.2 - 7.5.2 modules) Developed to generate a ping from the NET+OS development module.

How does it work?

PING TEST is a client based application set up to run on a NET+OS development module. It sends ping requests to a specific IP address entered in the root.c file.

Test files

This sample program contains three files, ping.c, ping.h and root.c. The main function in ping.c pings a target IP address. In the file ping.h you can specify the amount of system ticks per second.

Ping test sample application

A simple Ping Test sample application can be found here: Ping_test.zip.

Basic usage

First, open the root.c file and enter the ipaddress you would like to ping and save file

```
retval = ping("10.4.110.1");
```

Second, Build application, load the application into the embedded module and run it.

Sample of root.c file:

```
{
        int retval;

        while(1)

        {
                retval = ping("10.4.110.1");

                if(retval == -1)
                {
                        printf("Error\n");
                }
                else if(retval == 0)
                {
                        printf("No response\n");
                }
                else
                {
                        printf("Response heard\n");
                }
        }
```

```
        return;
}
```

NET OS Telnet Session

Program for Telnet_Customized_SessionID in NET+OS

NET+OS Telnet Session ID (For NET+OS 7.4.2 - 7.5.2 modules) Showcase how to customize telnet session ID using NETOS APIs. Sample application showing working of a Telnet Server.

Test files

This sample program contains six files. The main function is in root.c.

Telnet Session ID Test Sample Application

The Telnet Session ID Test sample application can be found here: Telnet_Customized_SessionID.zip.

Basic usage

Before Building and debugging please check the following in bsp_cli.h

```
1. #define BSP_CLI_TELNET_ENABLE   FALSE
2. #define BSP_CLI_ENABLE  FALSE

Debug/Run :

Open command prompt and
$telnet (ipaddress) 5000
login :
password :


Enter any characters and hit enter.
You can start more than one telnet session.
|Open another command prompt and :
$telnet (ipaddress) 5000

This application will display the below details to serial port for each session:

Session ID
Bytes Received for the particular Session
Username of the session.

Charlie & Bob :) :)
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include <sysAccess.h>


#include "telnet.h"
#include <bsp_api.h>


/*
```

```
* Set this to 1 to run the manufacturing burn in tests.
*/
int APP_BURN_IN_TEST = 0;


void applicationStart (void)

{
        int ret_telnet;
        unsigned long port_count = 0;
        iconfig_ptr.max_entries = TELNET_MAX_SERVERS;

    /*  Add Username digi password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "digi", "sysadm", NASYSACC_LEVEL_RW, NULL);

    /*  Add Username sysadm password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "sysadm", "sysadm", NASYSACC_LEVEL_R, NULL);

    /*  Add Username debug password debug to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "debug", "debug", NASYSACC_LEVEL_R, NULL);

    /*
         *Initializing Telnet Server .
        */
    if( (ret_telnet = TSInitServer(&iconfig_ptr)) != SUCCESS){
    printf("Return value : %d\n",ret_telnet);
            if(ret_telnet == TS_NO_MEMORY)
                    printf("Unable to allocate memory.\n");
            else if(ret_telnet == TS_INVALID_PARAMETER)
                    printf("The value in parameter is invalid.\n");
            else if(ret_telnet == TS_SYSTEM_ERROR)
                    printf("An internal error occurred.\n");
            else
                    printf("Can't open without calling
TSInitServer.\nExiting........\n");
            }
    /*
        *Configuring Telnet Server
        *Here only one server is configured.
        * If you want you can configure one more.
        * passing port_count+1
        */

        if(setup_telnet_server(port_count) != 0){
                printf("Telnet Server Setup Failed\n");
                return;
        }

        TS_t *list;
        while(1)
        {
                tx_thread_sleep(5*NABspTicksPerSecond);

                if(tx_semaphore_get(&semaphore_ptr,TX_WAIT_FOREVER ) != TX_
SUCCESS){
                        printf("Semaphore Get Failed\n");
                        return;
                }
                list = head;
```

```
                while(list != NULL)
                {
                        printf("\nSession ID : %d\n",list->ts_id);
                        printf("Bytes Received in this Session : %d\n",list-
>byte_recv);
                        printf("Username : %s\n",list->username);
                        list = list->ts_nxt;
                }
                if(tx_semaphore_put(&semaphore_ptr) != TX_SUCCESS){
                        printf("Semaphore Put Failed\n");
                        return;}
                }

        }

}

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;
/*
* Code to handle error condition if the stack doesn't come up goes here.
*/
}
```

Reboot gateway at a specific time

Reboot Gateway - Time of Day Reboot Application

Introduction

This application will reboot the gateway (Digi Connectport X2, X4, or X8) during a time of day or day of week that the user specifies.

The reasons for this application vary. If you wanted to keep the system uptime down, for whatever reason, this application would be a good fit. Occasionally, for troubleshooting purposes, it's desireable to reset the device.

The user can configure this application to reboot the gateway under one of two conditions:

1. Reboot everyday at a particular time.
2. Reboot once during the day of the week at a particular time.

Setup Directions

Edit the configuration file (ntpreboot_config.txt) with the desired settings. Specific directions can be found below under CONFIGURATION FILE DIRECTIONS.

Load the files (ntpreboot.py, ntpreboot_config.txt) onto the gateway from it's web interface (**homepage -> applications -> Python**). Then, configure the gateway to automatically launch the program when it starts up (**homepage -> applications -> Auto-start settings**, enter '**ntpreboot.py**' and check '**enable**' box).

Configuration File Directions

The settings for the ntpreboot application are located in the ntpreboot_config.txt file.

There are 3 possible items to set:

1. **NTP_SERVER** - the server that ntpreboot talks to in order to determine what time it is. Please note: this is usually UTC (Universal, also known as Greenwhich) time. You will need to factor this in according to the timezone you are in.
2. **REBOOT_DAY** - day of the week the user wants the gateway to reboot upon (optional). valid formats: 'Sunday', 'sunday', 'Sun', 'sun'. If left blank the script will reboot everyday at the specified time.
3. **REBOOT_TIME_OF_DAY** - (time of the day the user wants the gateway to rebot. Military time must be used (example, 6pm = 18:00, 6am = 06:00).

Source

Ntpreboot.zip

Remote Power Management Demo

Purpose

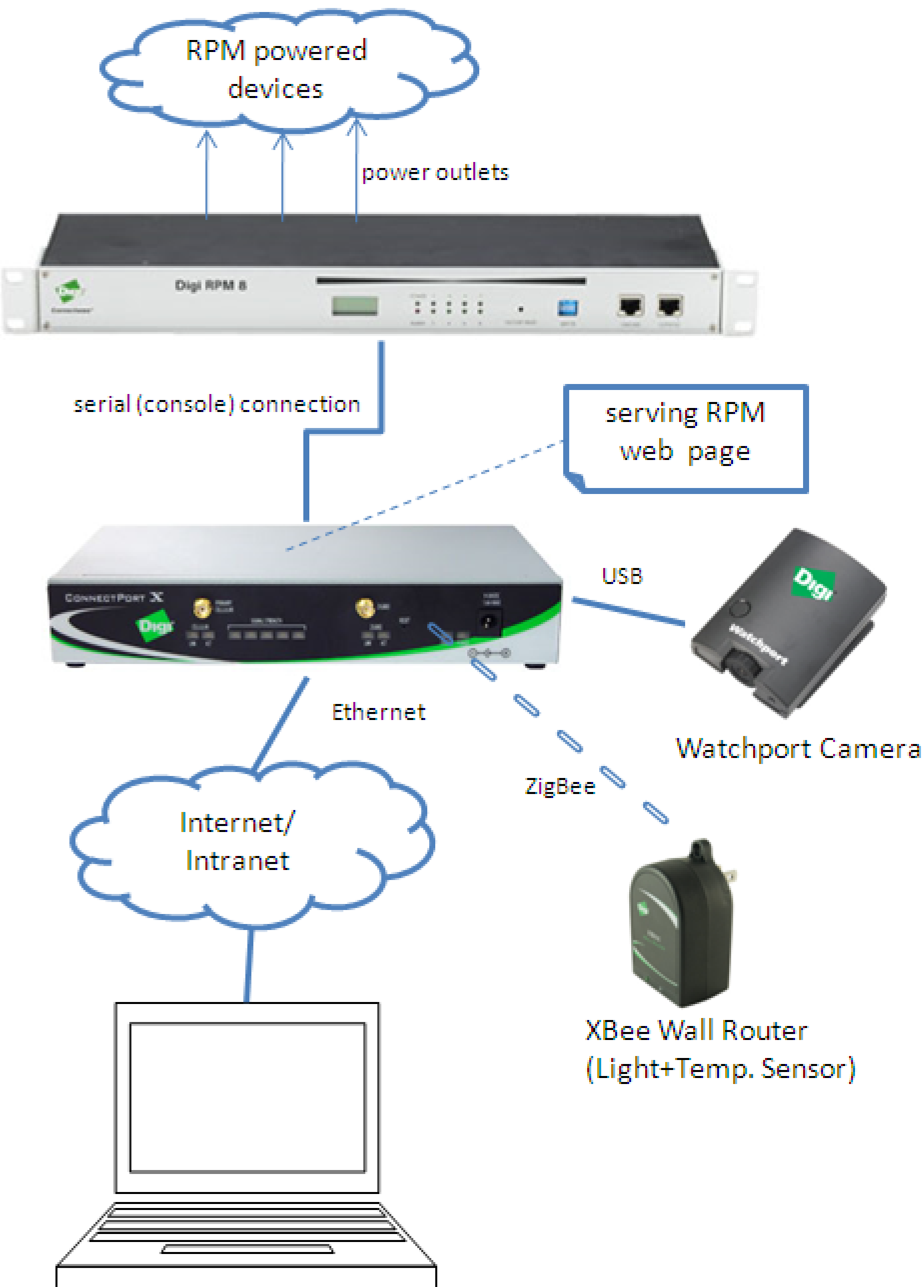
This page is to introduce you in the Demo of the Remote Power Management Interface using a Python generated web page on a Connect Port X.

Requirements

- Connect Port X (CPX)
- Digi RPM device (RPM 8 and 10 supported)
- SMTP server
- Wall Router
- Watchport Camera

Introduction

The Demo configuration is shown below:



The standard Python smtplib.py is compatible with CPX devices with the exception of one function call 'getfqdn()'. In the Digi compliant version, this call is replaced by a custom function call 'get_host()'. The 'get_host()' call returns the hostname of the device using the digicli [link needed] library.

Files

We are using a couple of techniques in this demo, so it´s clear that we need a couple of source. You have to add the following Python files using the ConnectPort X web interface:

- config.xml
- DigiXBeeDrivers.zip
- HttpDrivers.zip
- PythonExt.zip
- ngg_main.py
- sharing.py
- NGGAux.zip
- Python.zip
- xml.zip
- zigbee.py

Getting started

1. At first you have to adjust the config file settings according to your needs.

    a. **WALL_ROUTER_ENABLED**: Set to 1 if the website should check sensor data from a wall router (specified in WallRouterAddress).

    b. **PAGE_REFRESH**: Adjust the period of the web site auto refresh (in seconds)

    c. **ALERT_EMAIL_FROM**: Set the "from" field of the alert email

    d. **ALERT_EMAIL_TO**: Set the destination email address

    e. **SMTP_SERVER**: Set the SMTP server host name

    f. **TEMPERATURE_THRESHOLD_HIGH**: An alert email is sent if the wall router temperature sensor value is above this threshold.

    g. **TEMPERATURE_THRESHOLD_LOW**: An alert email is sent if the wall router temperature sensor value is below this threshold.

    h. **LIGHT_THRESHOLD_HIGH**: An alert email is sent if the wall router light sensor value is above this threshold.

    i. **LIGHT_THRESHOLD_LOW**: An alert email is sent if the wall router light sensor value is below this threshold.

    j. **DiagnosticsPagePort**: Set the port on which the web page is accessible.

    k. **WallRouterAddress**: Set the 64 bit destination address of the wall router (format: [XX:XX:XX:XX:XX:XX:XX:XX]!)

    l. **User:** Set the username for access to the web page

    m. **Password**: Set the password

2. The Wall Router destination address is the same as its serial number. You can find that value on the device´s cover.

3. Check if your CPX device can connect to the Wall Router. Using the web interface, **Configuration->XBee Network** should contain an according entry. If not, press the reset button of the wall router quickly 4 times. It then should re-associate with the CPX.
4. Then, upload all files specified above on the CPX device.
5. Add an auto-start slot at the CPX configuration page (**Applications->Python->Auto-start Settings**) for "sharing.py".
6. Reboot the device.
7. If you wish to name your RPM outlets, open a serial connection from the CPX to the RPM device by typing "**connect 1**" on the command line interface. If everything worked fine, you will see the prompt "RPM>:". Type "**name 0 X,Y**" where X is the outlet id and Y is the name you whish to specify.
8. If you are experiencing problems on connecting to the RPM, type "**set term state=off**" and reboot the CPX.
9. Telnet and run "**py ngg_main.py**" to start the web server manually.
10. If the web server should start up automatically, please use a second auto-start slot (**Applications->Python->Auto-start Settings**). But be aware: You get no debug information from the console.
11. Wait about a minute and type in your browser "**http://<CPX-IP>:8080**".
12. To authenticate, add "**?user=admin&password=secret**" to the URL and enter (for example, **http://<CPX-IP>:8080/?user=admin&password=secret**).
13. To logout, add "**?logout=.**" to the URL and press **Enter**.

Please note: There is no session timeout, so don´t forget to logout!

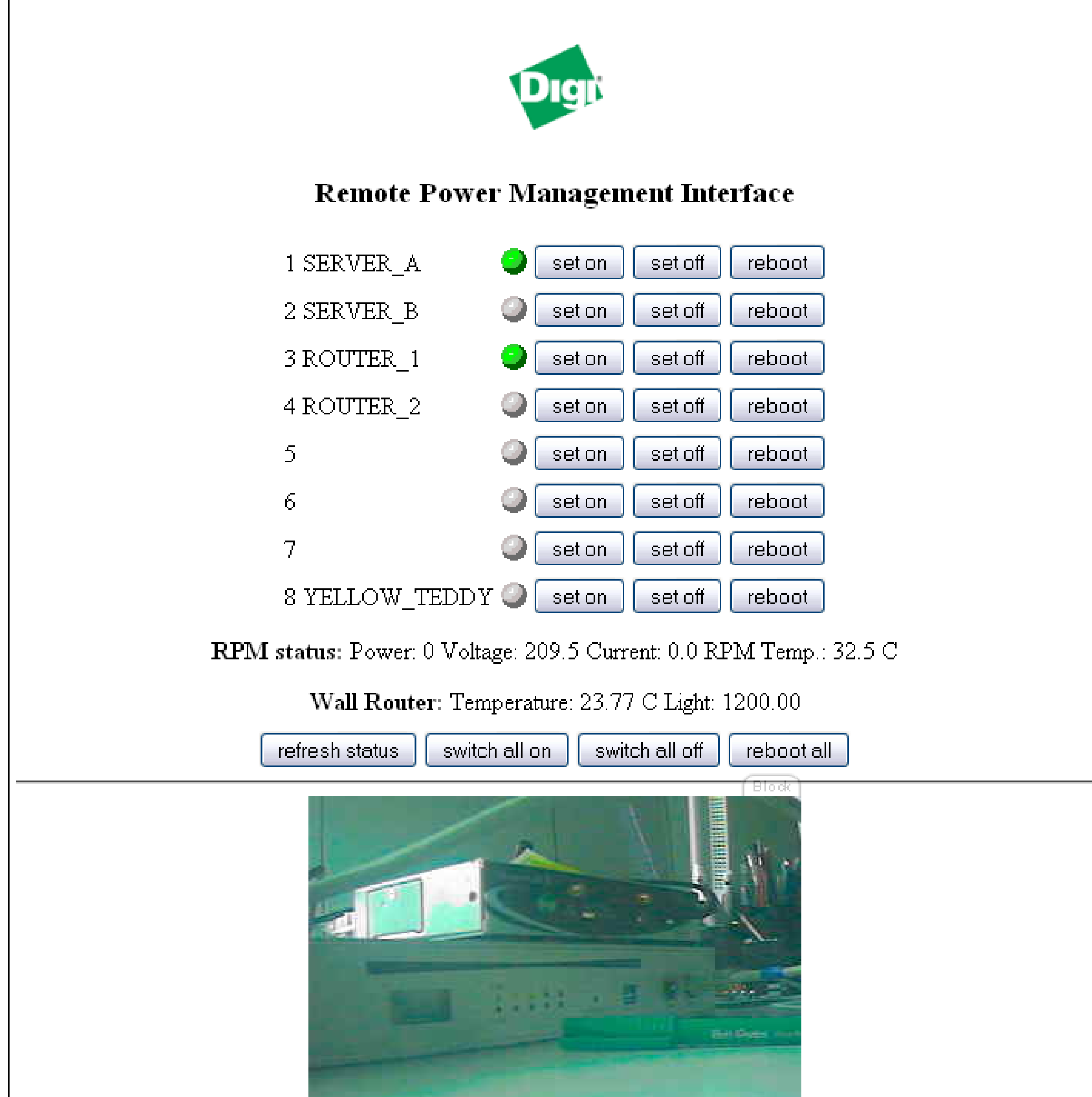
How it works / how it looks

ngg_main.py serves a web page that gives you the opportunity to manage the connected RPM. You are able to

- Monitor the state of each outlet
- Set on/off a specific outlet
- Reboot a specific outlet (automatically reset the outlet within 10 seconds)
- Set on/off all outlets
- Reboot all outlets
- Monitor the Wall Router sensor values
- Have a look at the camera video
- Use basic authentication to limit access to the web page (see hints on the "Getting started" section).

As a big feature, you are getting an alert E-Mail if the Wall Router sensor values are out of the range specified in the config file.

Here is a screenshot of the Remote Power Management Interface:



Interesting code

Below, we will give a short introduction in the key source code.

Sending an E-Mail is quite simple. Here is a snippet from email.py:

```
def sendEmail(fromaddr, toaddrs, subject, msg):

    ## Define Header
    header = ("From: %s\r\nTo: %s\r\nSubject: %s\r\n"
      % (fromaddr, toaddrs, subject))
```

```
    msg = header + msg
    server = smtplib_digi.SMTP(smtp_server)
    server.set_debuglevel(1)
    server.sendmail(fromaddr, toaddrs, msg)
    server.quit()
```

Tip You can get a simple SMTP Server for Win32 that is free for non-commercial use at [1].

Retrieving the Wall Router sensor values is easy, too. Have a look at wallrouter.py:

```
def getDataSample():
    ## Create a sensor object
    wallrouter = xbeewr.XBeeWRN(DESTINATION)

    ## Retrieve a sample reading from the wallrouter
    sample = wallrouter.sample()

    return (sample['temperature'], sample['light'])
```

Communicating with the RPM is gained using the serial port. Here is an example from rpm.py:

```
def getStatus(rpmId):
    serialfd = openConsole()
    os.write(serialfd, "status %d\x0D" % rpmId)
    time.sleep(2)
    data = os.read(serialfd, 2048)
    os.close(serialfd)
    return data

def openConsole():
    # Open serial port
    serialfd = os.open(serialport, os.O_RDWR)
    os.write(serialfd, "\x0D")
    time.sleep(0.1)
    data = os.read(serialfd, 1024)
    return serialfd
```

Starting the web server is a bit more difficult. Please look at this snippet of ngg_main.py:

```
WebServerThread = Thread(group=None, target=WebServerThreadFunction, name="Web
Server Thread", args=(), kwargs={})
WebServerThread.start()

def WebServerThreadFunction():
    lastpage = "Error."
    server = SimpleHTTPServer.BaseHTTPServer.HTTPServer(
                ('', HTTP_PORT), MyHTTPRequestHandler)
    print "Serving HTTP Requests at port ", HTTP_PORT
    server.serve_forever()

class MyHTTPRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        global lastpage
        parameters = {}
        try:
            parameters = dict(
                [ map(lambda s: urllib.unquote(s), pair.split('=')) \
```

```
                    for pair in urlparse.urlparse(self.path)[4].split('&')])
        except:
            pass

        s = WEB_do_GET(parameters)
        if (s!="Error"):
            lastpage = s
            self.wfile.write(s)
        else:
            self.wfile.write(lastpage)
```

WEB_do_GET is responsible for generating the HTML code and handling actions. At first it checks if some buttons have been pressed and according actions have to be done:

```
def WEB_do_GET(parameters):

    # Perform action
    if 'action' in parameters:
        if parameters['action'].startswith('setOn'):
            outletId = parameters['action'].split('setOn')[1]
            setOutlet(0, outletId, 'on')
        elif # and so on...
```

After that, the RPM status and Wall Router values are being retrieved using the modules described above.

Having that values in mind, we are able to generate a string template and fill it properly. The template is stored in NGGAux.zip.

```
s = Template(ngg_main_webpage)
s = s.safe_substitute(REFRESH=PAGE_REFRESH,
                IP_ADDRESS=ipaddress,
                PORT=HTTP_PORT,
                # [...]
                )
return s
```

Monitoring the Wall Router sensor values periodically is done by the main loop:

```
while True:
        # [...]
        time.sleep(10)
        # check if th Wall Router is enabled (as defined in config.xml)
        if (WALL_ROUTER_ENABLED==1):
            try:
                # get Wall Router sensor data
                sample = getDataSample()
            except:
                print "Error getting data sample from Wall router."
        continue
            try:
                # check sensor data and send an Email if neccessary
                if (not temperature_alerted) and (not TEMPERATURE_THRESHOLD_
LOW<=sample[0]<=TEMPERATURE_THRESHOLD_HIGH):
                    print "Temperature alert! "
                    alertcounter = alertcounter + 1
                    msg = "\r\n" + "Temperature is bad:\r\n\r\n" + str(sample[0])
+ "(C)\r\n\r\n" \
                         "\r\n\r\n\r\n" + "This is alarm number " + str
```

```
(alertcounter) + " since start of the monitoring script."
                    sendEmail(ALERT_EMAIL_FROM, ALERT_EMAIL_TO, "Digi Sensor
Alert!", msg)
                    temperature_alerted = True
                elif (TEMPERATURE_THRESHOLD_LOW<=sample[0]<=TEMPERATURE_
THRESHOLD_HIGH):
                    temperature_alerted = False
                #
                # [...] doing the same for the light sensor value...
                #
            except:
                print "Error sending alert E-Mail."
```

Future Versions

There are no recent changes scheduled so far.

Troubleshooting

In some installations it can occur, that the connection between the CPX and the RPM device cannot be established. In these cases, just remove the power plug of the RPM for some seconds and replug it. Maybe a reboot of the CPX should necessary, too.

Warning:

There is no protocol for the communication between the CPX and the RPM device. This demo uses a parsing algorithm to retrieve the information in the RPM status message. However, if the status message syntax changes due to product/firmware updates in the future, the parsing algorithm has to be adapted acordingly.

Source

- DigiXBeeDrivers.zip
- Python.zip
- PythonExt.zip
- Xml.zip
- NGGAux.zip
- HttpDrivers.zip

The next archive contains some files that have to be uploaded unpacked on the ConnectPort X.

- RPMfiles.zip

SMTP Email

Python program for SMTP Email

SMTP Email Notification (Python program) This example application sends an email notification to the specified email-id's.

Test files

This sample program contains one file. File name "SMTP_email_notofication.py".

SMTP Email Notification Test Sample Application

The SMTP_email_notofication.py Python Test sample application can be found here: SMTP_email_notofication.zip.

Basic usage

Provide inputs where neccesary.

Sample of SMTP_email_notofication.py file:

```
#!/usr/bin/Python

import os
import sys
import smtplib
import traceback

##########################################################################
##########################################################################
'''###PROVIDE INPUTS HERE###'''
EMAIL_HOST = "mail.<your_domian>.com"
sender = 'M2M'   #ANY NAME
# Receivers email id
receivers = ['first_name.last_name@digi.com', 'second_name.last_name@digi.com']
message = """ From: Digi Sample <M2M@digi.com>
To: <first_name.lastname@digi.com>,<second_name.last_name@digi.com>
CC: <first_name.lastname@digi.com>
Subject: SMTP e-mail test

This is a test e-mail
"""
##########################################################################
##########################################################################
try:
      smtpObj = smtplib.SMTP('EMAIL_HOST', 25)
      smtpObj.sendmail(sender, receivers, message)
      print "Successfully sent email"
except Exception, e:
      print "Exception %s" %e
      traceback.print_exc()

print "End of the program"
```

Simple RCI by HTTP

Reading DIA channels via web commands

Digi provides many detailed documents explaining web services and remote RCI calls, but most provide too much detail or partial examples. They assume you already know how to move the requests, so just want the core syntax.

Simple Python for use on a PC

Below is a simple script which allows using XML to query data in real-time from a gateway running DIA.

Sample YML

What this DIA system does isn't important. The RCI calls below will read or write the properties from the device named 'level', which are named **level.alert** and **level.config** respectively. The **RCIHandler** must be enabled, plus the setting **target_name** must match what you place into your RCI calls.

```
devices:
  - name: count
    driver: devices.template_device:TemplateDevice
    settings:
        update_rate: 10

  - name: level
    driver: devices.experimental.alert_output:AlertOutput
    settings:
        source: 'count.adder_total'
        rising: 12.0
        falling: 11.5

presentations:
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
        target_name: idigi_dia
```

Writing the RCI request

Detailed information on the RCI commands accepted by the DIA RCI presentation can be found in the DIA user documentation.

Those RCI commands, such as <channel_dump />, <channel_set name="..." value="..." />, and <logger_set name="..." /> are wrapped in the following syntax. Notice that the string **target="idigi_dia"** matches the target name in the YML file.

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
      </do_command>
</rci_request>
```

In DIA versions 2.2.0.1 and below, the RCI handler can only accept a single tag in the <do_command> block. A workaround is to wrap multiple commands in an arbitrary wrapper tag like the following:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
`                <channel_get name="level.alert"/>
                <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

The only thing to remember with this is that the <blob> tag will wrap the response code in a similar fashion.

For DIA versions **newer than 2.2.0.1**, the RCI handler does not need a wrapper tag, so the below syntax would work:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
                 <channel_get name="level.alert"/>
                 <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

However, this will generate an XML parse error with DIA versions 2.2.0.1 and below.

Source Code

Below is an actual script written and used under Python 2.4.3 on a Windows 7 PC.

```
            # Simple PC example to query the DIA device

import httplib, urllib

msg_dump = \
"""<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
       </do_command>
   </rci_request>"""

msg_get = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_get name="level.alert"/>
           </do_command>
           <do_command target="idigi_dia">
               <channel_get name="level.config"/>
           </do_command>
       </rci_request>"""

# notice that the VALUE this channel takes is complex - a list of 3 values.
# That is defined by the DRIVER involved.  Most channels will take a
# simple True/False, integer, floating point or string value.
# examine the channel_get response to discover what to return channel_set
msg_set = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_set name="level.config" value="[10,9,True]"/>
```

```
            </do_command>
        </rci_request>"""

 if __name__ == '__main__':

    msg = msg_get

    conn = httplib.HTTPConnection("192.168.196.204:80")
    conn.request("POST", "/UE/rci", msg)

    response = conn.getresponse()
    print response.status, response.reason
    data = response.read()
    print data
    conn.close()
```

Sample Output

```
C:\py\dia\work>Python test_rci.py
200 OK
<rci_reply version="1.1"><do_command target="idigi_dia"><channel_get
name="level.alert"
value="False" units="alert" timestamp="Mon Jul 12 11:08:46 2010"></channel_
get></do_command>
<do_command target="idigi_dia"><channel_get name="level.config"
value="[12.000000,11.500000,False,'count.adder_total']"
units="" timestamp="Mon Jul 12 11:08:46 2010"></channel_get></do_command></rci_
reply>
```

See Also

RCI request` : This is geared more to using RCI to read/write values in the Digi hardware and not in DIA.

Look up the RCI handler module in the DIA HTML documentation, or see the doc strings directly in the DIA source file "src\presentations\rci\rci_handler.py". It supports the commands including, but not limited to:

- <channel_dump/>
- <channel_get name="..."/>
- <channel_refresh name="..."/>
- <channel_set name="..." value="..."/>
- <channel_info name="..."/>
- Plus there is a series of logger commands

Simple traffic generator

Send a single UDP packet per time period

This simple script sends a UDP packet to a remote IP at fixed time-periods, and can be used to hold open a VPN tunnel which auto-closes when idle.

```
            # Simple UDP client to push some nonsense data

from socket import *
import time

# put the IP address to UDP to here: safer NOT to use DNS
HOST = '192.168.196.6'

# put a valid port number here - 7 is echo server, usually disabled these days
# but we won't be expecting an answer anyway
PORT = 7

# put something to send here
DATA = "Hi"

# put time to delay here - is in seconds, so 5 * 60 = once per 5 minutes
# the expression will have Python calc the seconds for us
SLEEP_TIME = 5 * 60

while 1:
# we recreate, close and free up socket every time
# this is more robust if the time delay is large
udpSock = socket(AF_INET, SOCK_DGRAM)
print "Sending data <%s>" % DATA
udpSock.sendto(DATA,(HOST,PORT))
udpSock.close()

time.sleep(SLEEP_TIME)
```

Smart Plug Interactive Demo

Below is a simple interactive demo that makes use of the functionality for the XBee Smart Plug. It is designed to run on a Digi ConnectPort gateway such as an X2 or X4.

```
from zigbee import getnodelist, ddo_get_param, ddo_set_param
import struct, sys

def parseIS(data):
    if len(data) % 2 == 0:
        sets, datamask, analogmask = struct.unpack("!BHB", data[:4])
        data = data[4:]

    else:
        sets, mask = struct.unpack("!BH", data[:3])
        data = data[3:]
        datamask = mask % 512 # Move the first 9 bits into a separate mask
        analogmask  = mask >> 9 #Move the last 7 bits into a separate mask

    retdir = {}

    if datamask:
        datavals = struct.unpack("!H", data[:2])[0]
        data = data[2:]

        currentDI = 0
        while datamask:
            if datamask & 1:
                retdir["DIO%d" % currentDI] = datavals & 1
            datamask >>= 1
            datavals >>= 1
            currentDI += 1

        currentAI = 0
            while analogmask:
                if analogmask & 1:
                    aval = struct.unpack("!H", data[:2])[0]
                    data = data[2:]

                    retdir["AI%d" % currentAI] = aval
                analogmask >>= 1
                currentAI += 1

            return retdir

class sp:
  def __init__(self, addr):
    self.addr = addr
    self.status = 'Unknown'

  def get_status(self):
    return self.status

  def enable_sensors(self):
    try:
      ddo_set_param(self.addr, 'D1', 2)
      ddo_set_param(self.addr, 'D3', 2)
      ddo_set_param(self.addr, 'WR')
```

```
      ddo_set_param(self.addr, 'AC')
    except Exception, e:
      raise Exception("Failed to enable sensors on device: %s" %self.addr)

  def get_sensor(self):
    try:
      raw_sample = ddo_get_param(self.addr, 'IS')
    except Exception, e:
      print "Failed to get sample from device: %s" %self.addr
      return (0, 0)

    parsed_sample = parseIS(raw_sample)

    lux = (float(parsed_sample["AI1"]) / 1023.0) * 1200.0

    mV = (float(parsed_sample["AI3"]) /1023.0) * 1200.0
    current = (mV*(156.0/47.0) - 520.0) /180.0 * .7071

    return (lux, current)

  def turn_on(self):
    try:
      ddo_set_param(self.addr, 'D4', 5)
      ddo_set_param(self.addr, 'WR')
      ddo_set_param(self.addr, 'AC')
      self.status = 'on'
    except Exception, e:
      print "Error with turning on device %s" %self.addr

  def turn_off(self):
    try:
      ddo_set_param(self.addr, 'D4', 4)
      ddo_set_param(self.addr, 'WR')
      ddo_set_param(self.addr, 'AC')
     elf.status = 'off'
    except Exception, e:
      print "Error with turning off device %s" %self.addr

  def main():
    print "Searching for Smart Plugs on XBee network..."
    node_list = getnodelist()
    smart_plugs = []
    for node in node_list:
      if node.device_type & 0x000F == 0x000F:
        t = sp(node.addr_extended)
        smart_plugs.append(t)

    assert len(smart_plugs) != 0, 'No Smart plugs detected'

    print "Enabling sensors on device"
    for node in smart_plugs:
      node.enable_sensors()

    status_str = "\nIndex: %d Addr: %s \n\tstatus = %s Light = %f Current = %f"
    help_str = "Type 'quit' to exit\n\t'on 1' to turn on node 1\n\t'off 1' to
turn off node 1\n"

    print help_str
    while 1:
```

```
      for node in smart_plugs:
        i = smart_plugs.index(node)
        a = node.addr
        s = node.get_status()
        l, c = node.get_sensor()
        print status_str %(i, a, s, l, c)

      input = raw_input("\n=>")
      input = input.lower().strip()

      if input == 'quit':
        print "Quitting"
        sys.exit(0)
      if input in ['help', '?']:
        print help_str
        continue
      else:
        parsed = input.split()
        if len(parsed) == 2:
          action = parsed[0]
          try:
            index = int(parsed[1])
            assert len(smart_plugs) > index >= 0, '%d > %d >= %d' %(len(smart_
plugs), index, 0)
          except Exception, e:
            print "Index error: %s" %parsed[1]
            continue

          if action == 'on':
            print "Turning smart plug %s on" %smart_plugs[index].addr
            smart_plugs[index].turn_on()

          elif action == 'off':
            print "Turning smart plug %s off" %smart_plugs[index].addr
            smart_plugs[index].turn_off()

          else:
            print "Unknown action: %s" %action
            continue

if __name__ == '__main__':
  main()
```

Smart_plug.zip

UDP to XBee network

Introduction

This page describes a simple application that forwards UDP traffic to a Gateway to a XBee destination while forwarding XBee traffic from that destination to the UDP network.

Requirements

- Gateway product that supports the Python interpreter.
- XBee device associated with the Gateway
- UDP access to the Gateway

Script

```
""" This script is intended for demonstration purposes.  It meets the minimum
requirements needed to move the data from the UDP network to the XBee network.

A more robust application may have additional error checking, command line
argument support or another features beyond this.

The basics of this application are to create a number of UDP sockets that
are logically connected to a like number of XBee nodes connected to the
Gateway product.  Traffic received on the UDP socket will be sent over the
XBee network to the paired XBee node, and likewise traffic received from the
XBee node will be sent to the paired socket.

The UDP sockets use the 'last known address'(LKA) to know the destination to
send the XBee data to.  The LKA is the last received UDP packet on that socket
source address, and the script will blindly send data back to that address,
regardless of whether or not it is listening.

Initially, the UDP socket does not have a LKA, and any received data that would
normally be forwarded to the LKA is dropped.  To set the LKA of a UDP socket to
stop forwarding data from the XBee network, send it a UDP packet of length 0.

NOTE:  There is no limit to how much the application will attempt to queue up
to send to the XBee network or the UDP network.  If too much is queued up,
unexpected errors may occur, resulting in strange Python behavior, device
panicing, or loss of data.
"""

import socket
import zigbee
import select
import bind_table

###############################################################################
# We need to map several pieces of data from the UDP socket.
# These dictionaries will provide a inexpensive look up table for that.
# Declarations
###############################################################################
```

```
udp_port_dict  = {} ##Udp socket to port it was bound to
udp_lka_dict   = {} ##Udp socket to its LKA it received data from
udp_queue_dict = {} ##Udp socket to its queued up data
udp_socks      = [] ##List of all Udp sockets

zig_port_dict  = bind_table.node_list ## list of XBee address to port numbers
zig_data_queue = []                   ## Data/address queued for the XBee socket

MAX_UDP_SIZE = 8192 ## Maximum UDP packet size to read/write
MAX_ZIG_SIZE = 84   ## Maximum XBee packet size to read/write

end_point  = 0xe8   ## Endpoint to bind the XBee socket to
profile_id = 0xc105 ## Profile ID to bind the XBee socket to
cluster_id = 0x11   ## Cluster ID to bind the XBee socket to

################################################################################
# To populate the above dictionaries with relevent data
# All data is based off the address to port dictionary in bind_table
################################################################################

for item,port in zig_port_dict.items():
  zig_port_dict[port] = item # provide the reverse lookup from port to address

  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  sock.bind(("", port))  #Create and bind a UDP socket

  udp_port_dict[sock] = port  #provide a udp socket to port and vise versa
  udp_port_dict[port] = sock

  udp_lka_dict[sock] = None   #Provide a udp socket to LKA
  udp_queue_dict[sock] = []   #Create a list object to act as a data queue

  udp_socks.append(sock)      #Append the socket to the UDP socket list


#Create and bind the XBee socket
zig_sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
zig_sock.bind(("", end_point, profile_id, cluster_id))

#Create a list of the udp_sockets plus XBee socket to monitor
sock_list = udp_socks + [zig_sock]

################################################################################
# For the main portion of the script, we perform a select call on the list of
# sockets, and must handle 4 distinct results from the select call.
#  - XBee data to read
#  - XBee data to write
#  - UDP data to read
#  - UDP data to write
################################################################################

print "Starting main"
while True:
  rl, wl, el = select.select(sock_list, sock_list, [])

  ##############################################################################
  # XBee data to read
```

```
  # Read the data, if the address read from is mapped to a UDP socket,
  # Queue the data up for that socket's data queue
  ###############################################################################

  if zig_sock in rl:
    data, addr = zig_sock.recvfrom(255)
    print "Received %d bytes from address: " %len(data), addr
    if addr[0] in zig_port_dict:
      port = zig_port_dict[addr[0]]
      udp_queue_dict[udp_port_dict[port]].append(data)
    else:
      print "Unknown zigbee node contacted us!"

  ##############################################################################
  # XBee data to write
  # If we have data to write, send data to the address specified in the queue's
  # tuple.  Save any data that wasn't sent, and if empty, pop the element off
  ##############################################################################

  if zig_sock in wl and len(zig_data_queue) != 0:
    data, addr = zig_data_queue[0]
    segment = ((len(data) > MAX_ZIG_SIZE) and MAX_ZIG_SIZE) or len(data)
    sent = zig_sock.sendto(data[:segment], 0, addr)
    print "Wrote %d bytes to address: " %sent, addr
    data = data[sent:]
    if len(data) == 0:
      zig_data_queue.pop(0)
    else:
      zig_data_queue[0] = (data, addr)

  ##############################################################################
  # UDP data to read
  # Get the data and address, if the data is of length 0, remove the LKA and
  # move on.  If the LKA address doesn't match the source address of this
  # packet, the LKA address becomes the source address of this packet.
  # Find the UDP socket's associated XBee address, and queue it up in the
  # XBee data queue
  ##############################################################################

  for sock in udp_socks:
    if sock in rl:
      data, addr = sock.recvfrom(MAX_UDP_SIZE)
      print "Read %d bytes from address: " %len(data), addr
      if len(data) == 0:
        udp_lka_dict[sock] = None
        continue

      if udp_lka_dict[sock] != addr:
        udp_lka_dict[sock] = addr
        udp_queue_dict[sock] = []

      port = udp_port_dict[sock]
      node_addr = zig_port_dict[udp_port_dict[sock]]
      zig_data_queue.append((data, (node_addr, end_point, profile_id, cluster_
id)))

  ##############################################################################
  # UDP data to write
  # If we have data to write and have a LKA address, send that LKA address
```

```
  # the data we have queued for it.  If we have sent it all, pop off the data
  # or write back the remainder to the queue
  #############################################################################

  for sock in udp_socks:
    if sock in wl and len(udp_queue_dict[sock]) != 0 and udp_lka_dict[sock] is
not None:
      data = udp_queue_dict[sock][0]
      segment = ((len(data) > MAX_UDP_SIZE) and MAX_UDP_SIZE) or len(data)

      sent = sock.sendto(data[:segment], 0, udp_lka_dict[sock])
      print "Wrote %d bytes to address: " %sent, udp_lka_dict[sock]
      data = data[sent:]

      if len(data) == 0:
        udp_queue_dict[sock].pop(0)
      else:
        udp_queue_dict[sock][0] = data
```

Files

This script makes use of a configuration generator called "table_generator.py" to create the initial table located in bind_table.node_list. See the TCP to Zigbee page for more details. The below archive contains the source of the UDP_zig_tunneling script and the "table_generator.py" file.

Udp_zig_tunneling.zip

Using Digi Realport with Python

Digi Realport is a set of operating system drivers which make remote IP-based serial ports appear as local physical ports. Traditionally Digi Realport uses TCP/IP only and talks to a very special low-level driver in Digi products. Unfortunately, at present these low-level drivers in products such as the X4 and X8 gateways literally expects to talk to the hardware serial ports. Thus there is no way to connect standard Digi Realport to a Python script.

However the latest versions of Digi Realport for Windows has added a UDP mode which ONLY moves serial data, emulating a 3-wire RS-232 cable. The data it sends is well packed into a single UDP packet and works very well with protocols such as Modbus or Rockwell DF1. Since none of the Digi Realport protocol is included, any Python script can wait on the UDP socket and interact with a remote Windows-based host application.

Supported products

Digi Realport for Windows (such as P/N 40002549_xx.zip) Older versions or versions for a different OS may not have UDP support.

Digi products which support Python Zmatrix.

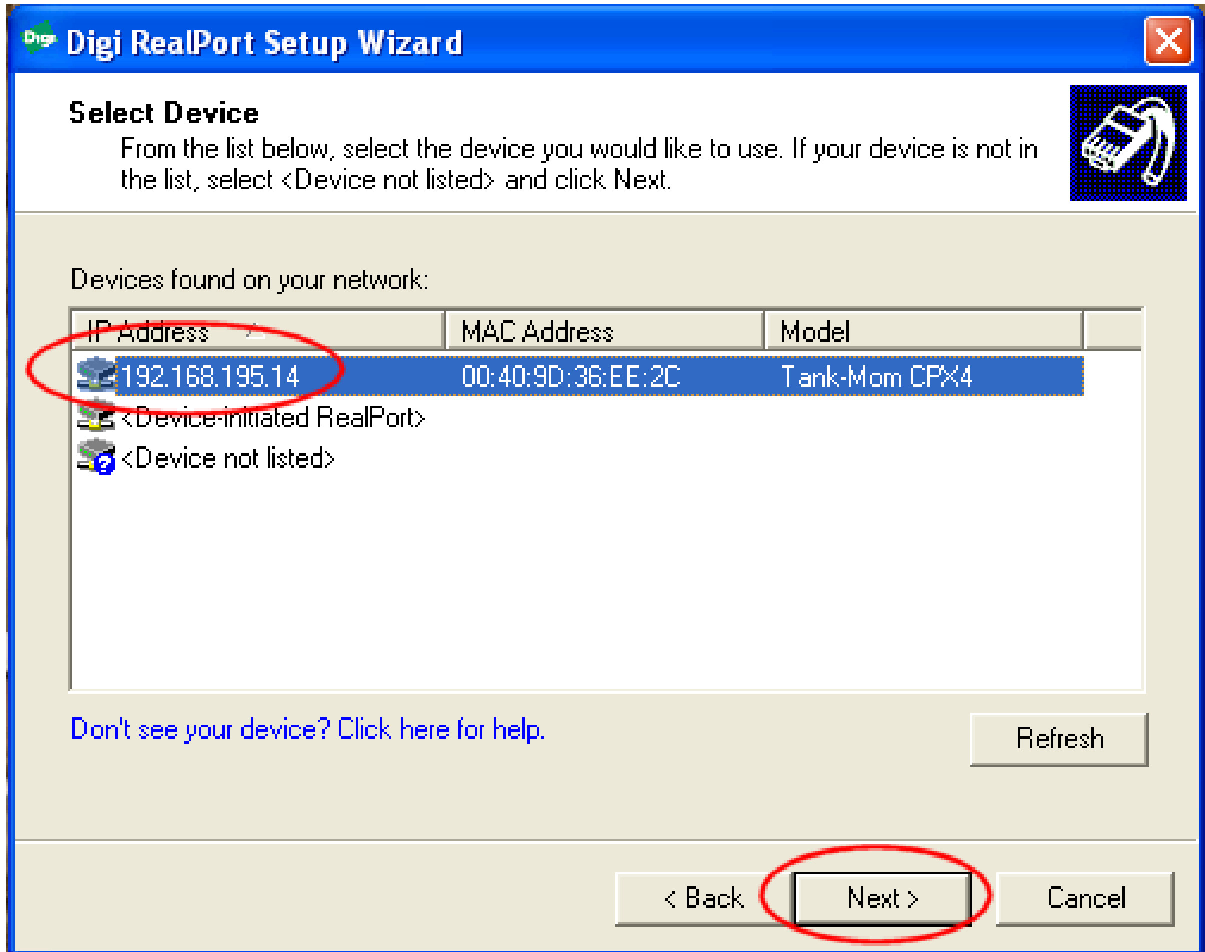
Setting up Digi Realport under Windows

Download the latest version of Digi Realport. This March_2010_ver.4.4.365.0_Realportdriver.zip is for Microsoft Windows XP/2003/Vista/2008 (both 32/64 bit arch).

CLICK HERE to find Realport for another Operating System or verify you have the latest Microsoft Windows version.

Installation

Unpack (unzip) the files in a suitable directory. Run the SETUP.EXE and you should see this this display:

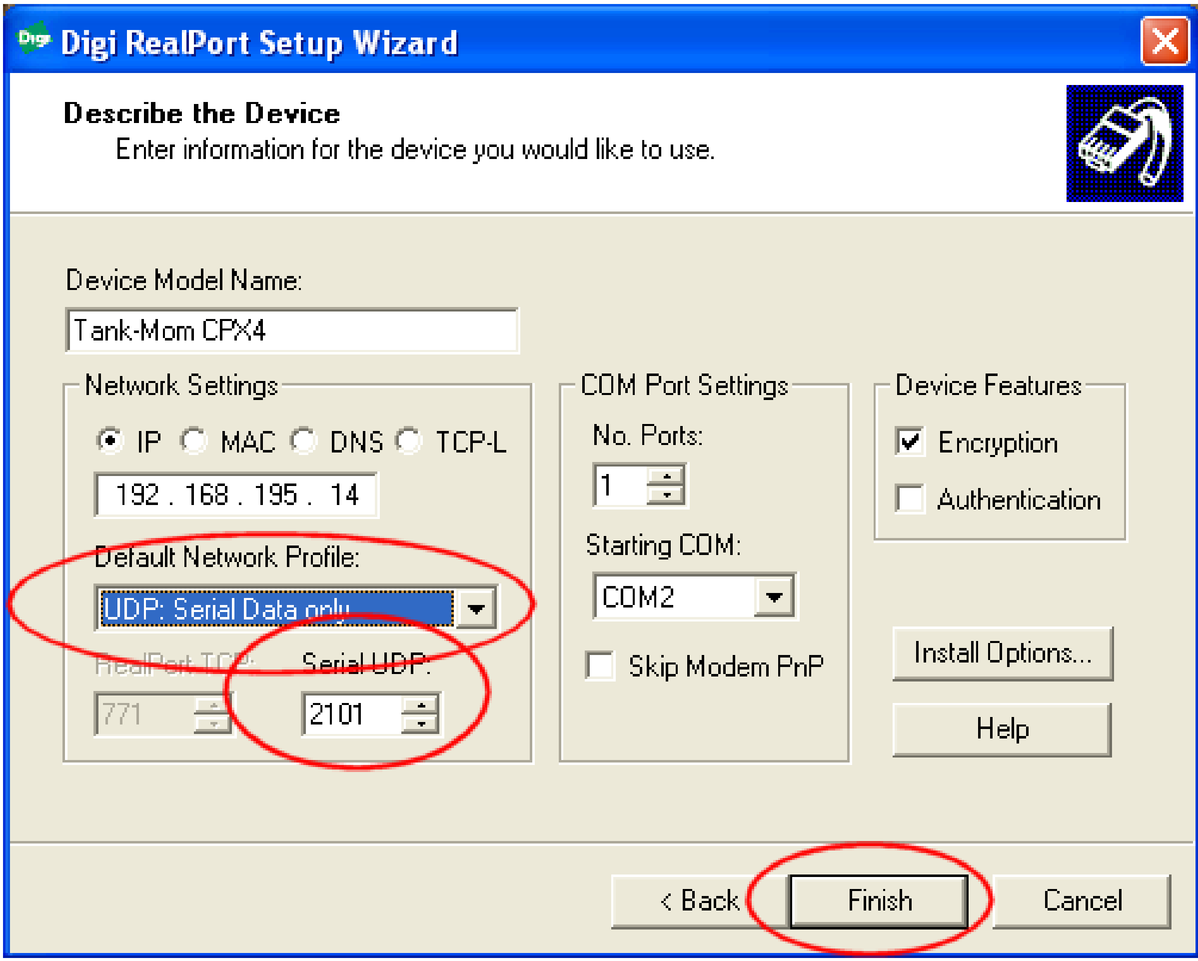


It is easiest to install Digi Realport with your device (or one of the same model) sitting next to you on your local Ethernet. This way the setup program should find it by browsing, plus learn all of the correct capabilities automatically - once installed, it is easy to change the IP address if your device is remotely located over wide-area-network such as cellular. In the example above, we'll be enabling Realport to a Digi ConnectPort X4 gateway. If you don't see your product listed, it might not have a proper IP address configured or your Windows firewall is blocking the UDP multi-cast being used. Always temporarily disable your Windows firewall when looking for LOST Digi devices, since they will reply to the Wizard with unreachable or even NULL (0.0.0.0) IP addresses and firewalls will always discard such 'mal-formed' UDP packets.

When you see this display, then:

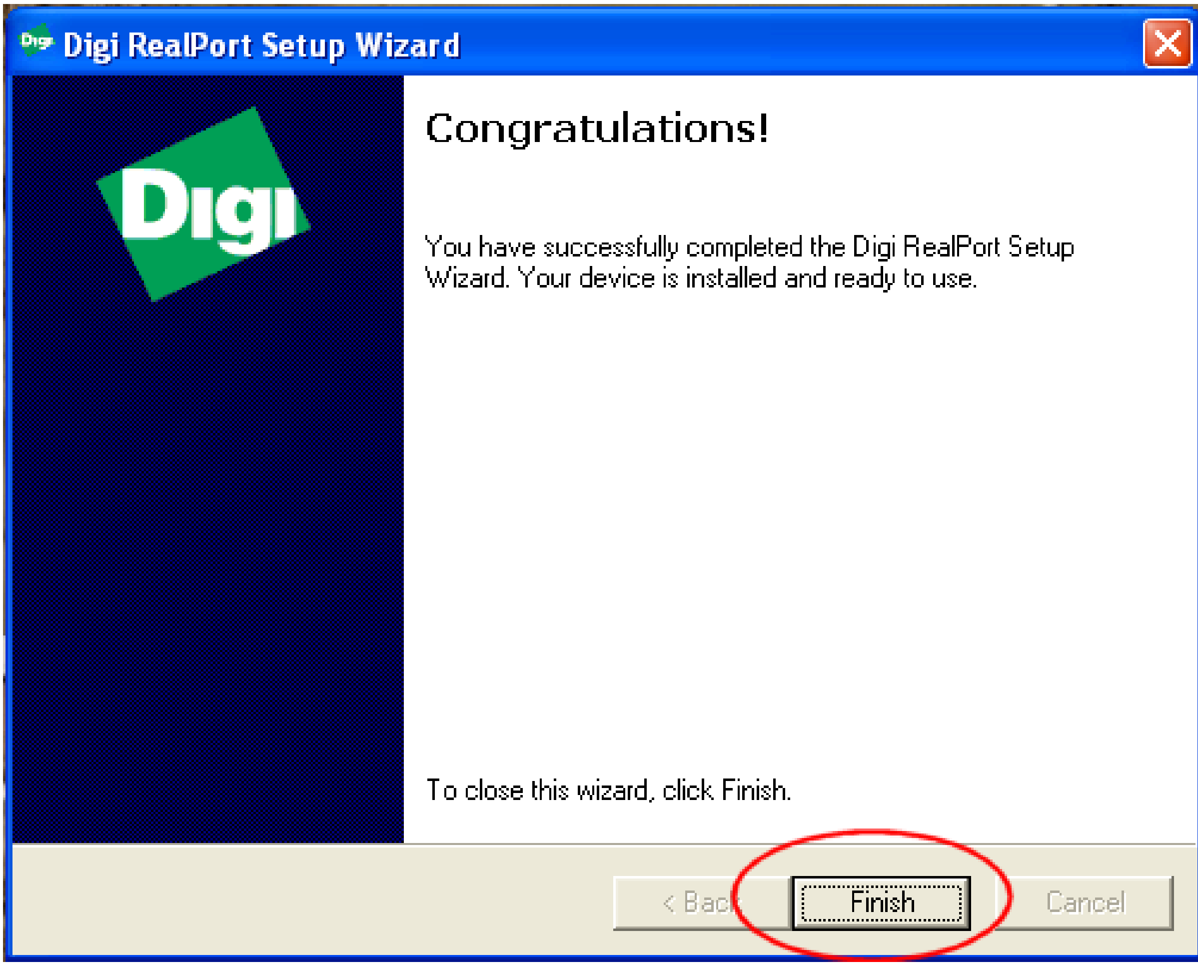
1. Select the device to configure.
2. Click **Next**.

   You should the see this display, where you can configure the basic features to use:



3. Change the Default Network Profile to **UDP: Serial Data only**. This changes Digi Realport to use UDP/IP instead of TCP/IP, plus the Serial Data only warning means all control signals, the ability to change baud rate (etc) is lost in this mode. Digi Realport in UDP mode literally mimics a 3-wire RS-232 line with only Txd, Rxd and signal ground lines.
4. Set the appropriate UDP port number to target as destination - the default of 2101 is likely okay.
5. Tweak other settings as desired. In this display we are configuring COM2 - you could move this to COM6 or other values. Also, with Realport in UDP mode things like Encryption and Authentication are NOT usable, even if the device supports it.
6. Click **Finish**.

   You should see the setup now do some work, run through a few progress displays and finally show this display:

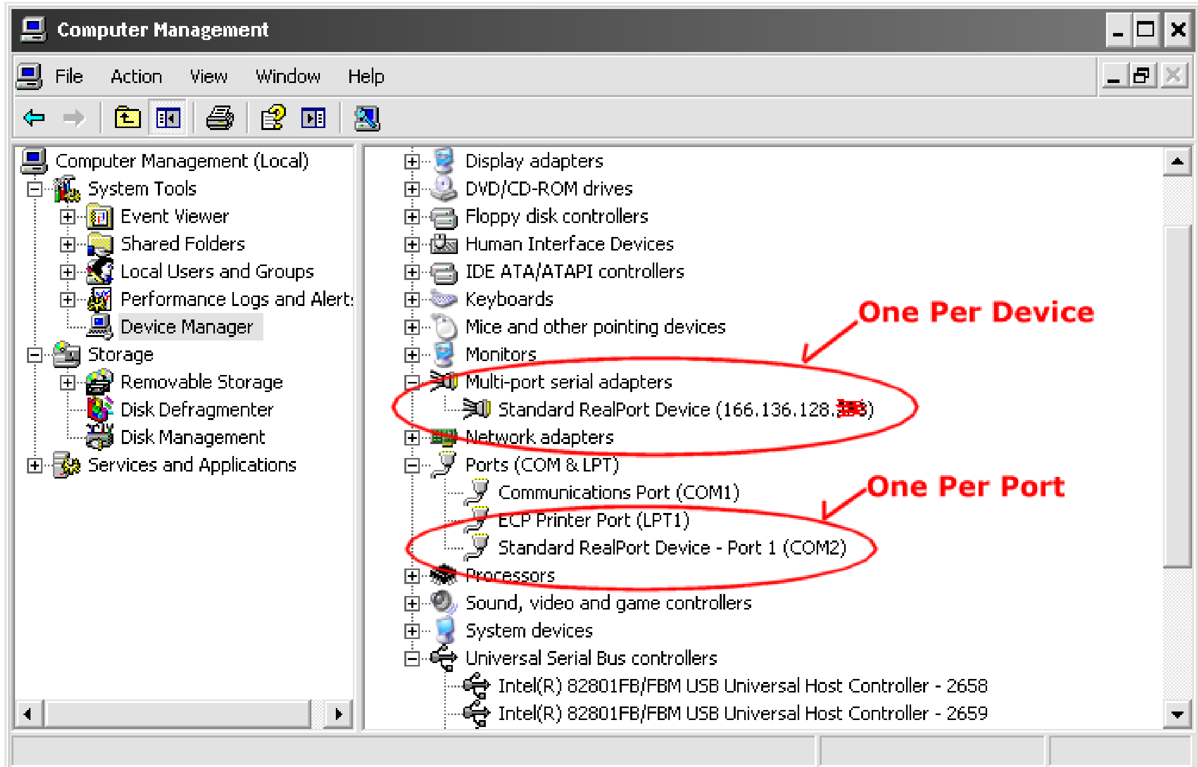


At this point, the computer will send any serial data written by a Windows application to COM2 to the IP address 192.168.195.14 in UDP packets to port 2101.

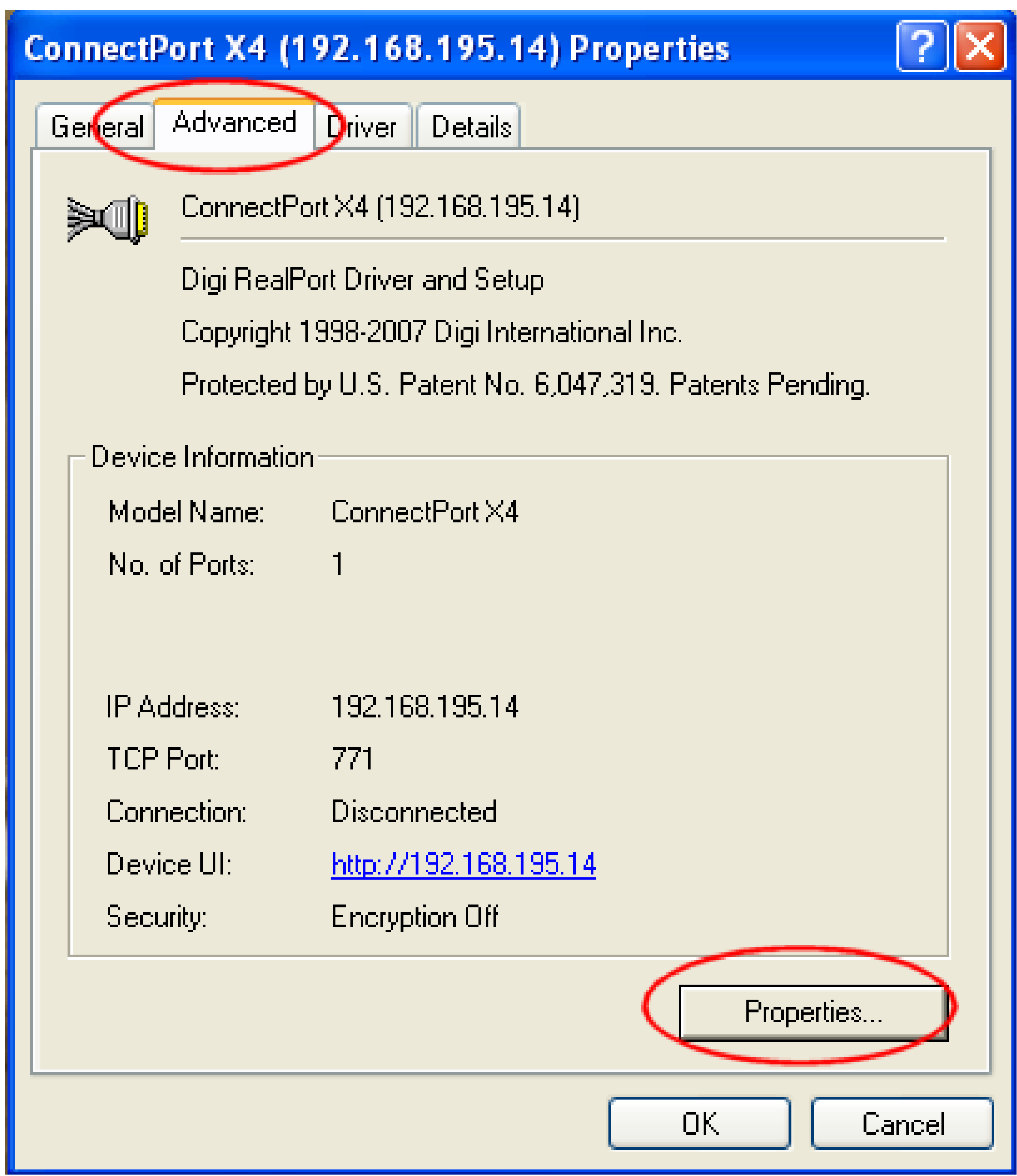
Changing settings within Realport in UDP mode

After you've installed Digi Realport, you can change setting through the Device Manager. You can open it several ways:

1. Right click My Computer, click Manage, Click Device Manager
2. Open System Properties, click the Hardware Tab, click the Device Manager



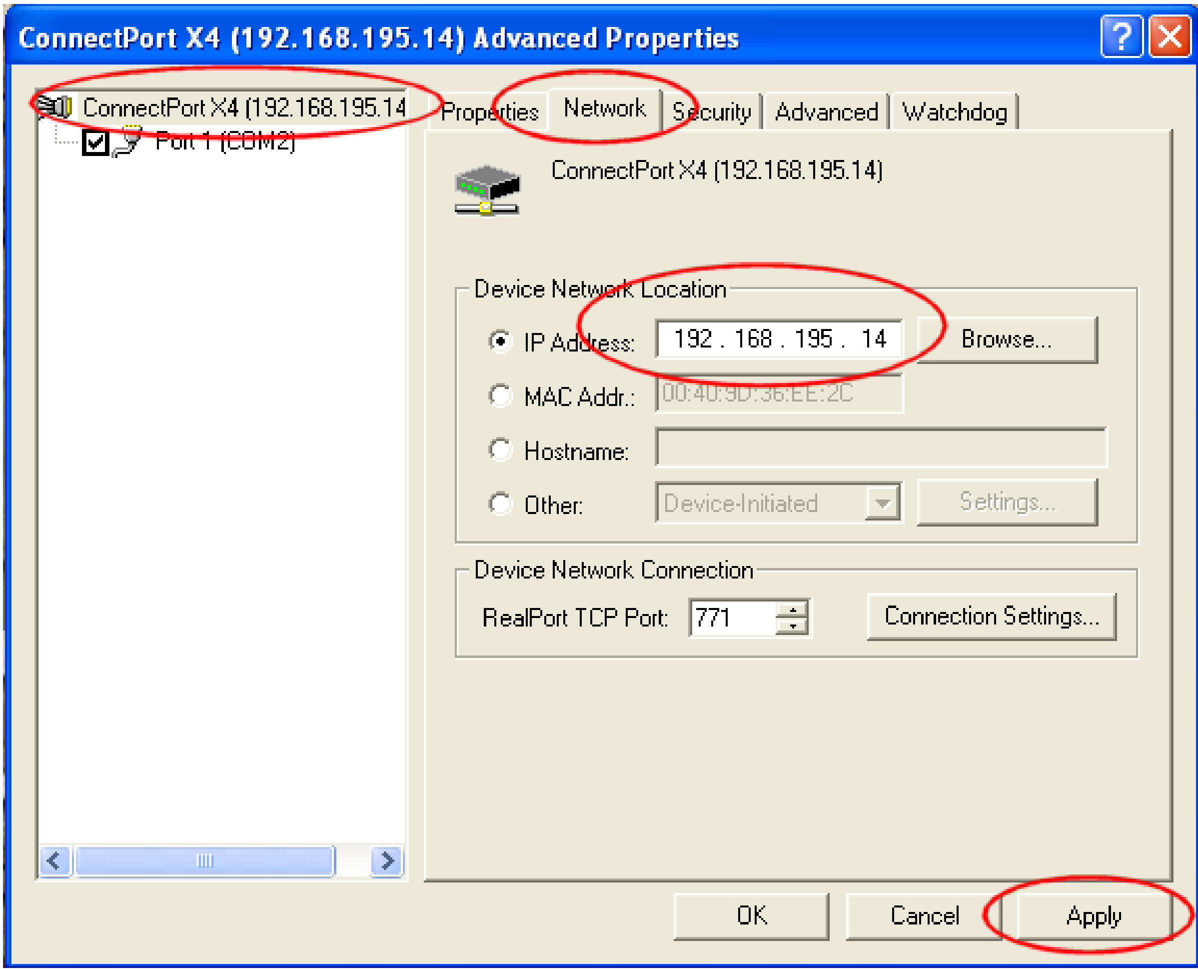
The Digi Realport device installs itself as a Multi-Port Adapter, right click it and select Properties. You will see this display:



Select the Advanced Tab, and select the Properties button. You will notice the Connection status is Disconnected - this is true since this is UDP/IP, not TCP/IP.

To Change the IP address

Select your Device (in this case ConnectPort X4) in the left panel and not the Port, then select the Network tab. Here you can select to use and change an IP address, or you can change to use a DNS Hostname. So while in this case the device was installed locally with a non-routing IP of 192.168.195.14, you could change it here to be the actual public IP such as 166.x.x.x. Remember to hit Apply when done.

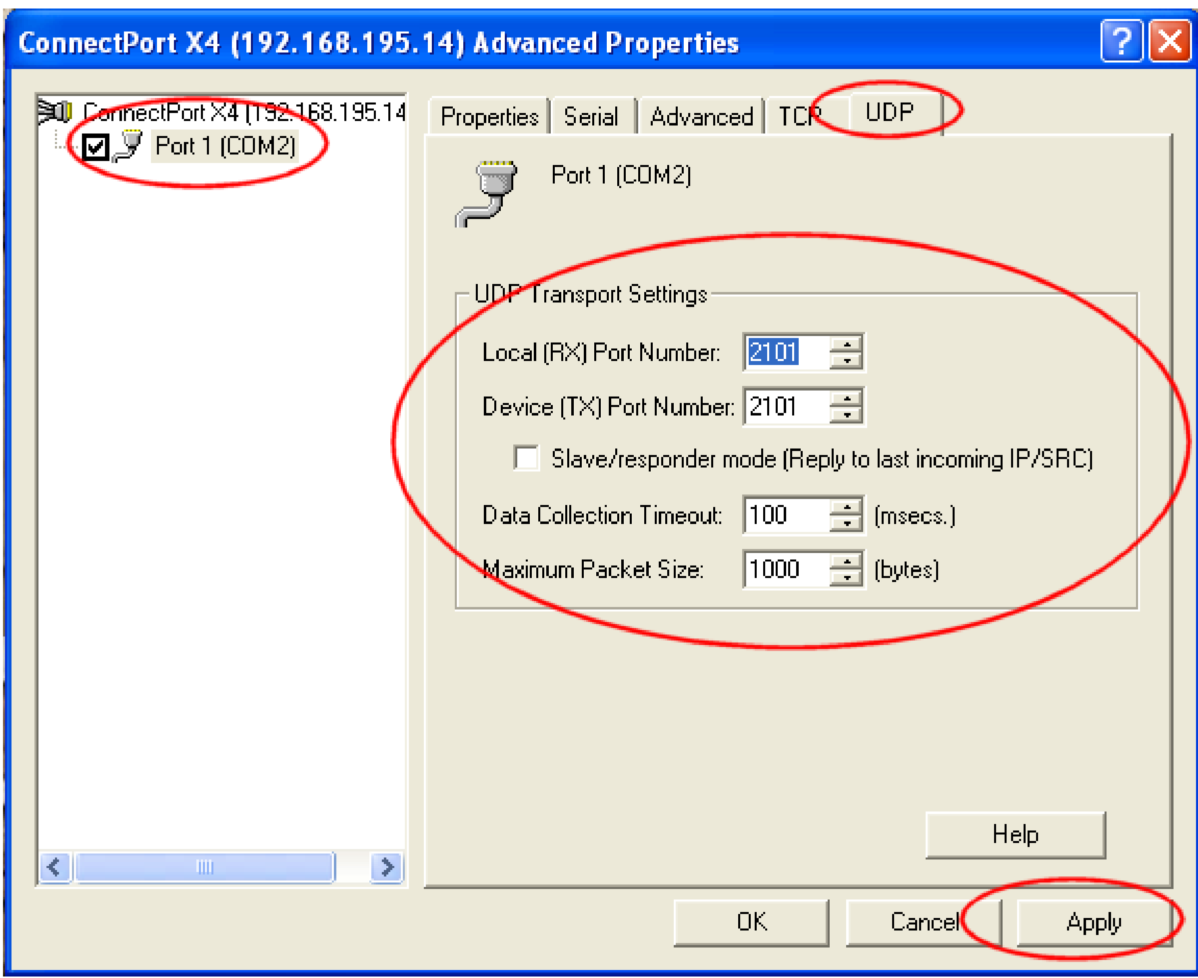


To Change the UDP settings

1. Select the **Port 1**, then the tab. The COM2 shown here will match the port you installed Realport on. Pressing the HELP button will - surprise - show you some fairly complete help information.

   The Local (RX) Port is the UDP source port within UDP packets, while the Device (TX) Port is the UDP destination port. In this example, the two numbers of 2101 match, but they can be any valid port numbers. The default is ideal for Windows-based clients polling the remote Digi product configured in UDP Sockets serial port profile. It sends data from the application to the Digi, and it must return it to the Local (RX) Port number.

   If the Digi product is a WANIA, CPX4 or DOIAP with the IA Engine active sending requests to the Windows host, then check the box labeled as Slave/responder mode, which disables the Device (TX) Port. In this mode, Digi realport remembers the UDP source/destination information to send the responses back to the Digi product.



2. Click **Apply** when you are done.

Utility to set dest addr in all associated nodes

Utility to set dest_addr (DH/DL) in all associated nodes

A realistic Python application using the ddo_get_param(), ddo_set_param(), and get_node_list() functions.

Users who send serially encapsulated data to RS-232/485 adpaters running AT-mode firmware need to ensure the DH/DL (destination address) registers of each associated node are correct. This is tedious to do by hand - especially if testing requires the same collection of remote RS-232/485 to be routinely moved between different gateways.

The fully functional Python program linked below runs on any Digi ConnectPort X gateway. It obtains a list of associated nodes and makes sure the DH/DL registers of each are set to the gateway running the program. The application can also be MODIFIED to be a more general-purpose configuration refresh tool - for example the code could be updated to ensure the baud rate and RS-232 control signals are properly configured.

Routines used; Things you can learn

The program **dest_addr_to_me.py** does the following:

- Uses **zigbee.getnodelist()** to obtain a list of associated nodes.
- Uses **zigbee.ddo_get_param()** to read parameters from each node.
- Uses **zigbee.ddo_set_param()** to write parameter to each node.
- Detects the Xbee RS-232 PH (Power Harvesting) adapters and limits the ddo_get_param ()/ddo_set_param() calls to a slow enough rate to NOT deplete the charge of the super-cap, which causes the adapter to go offline
- After running, all DH/DL parameters will be as desired. Users can manually set an alternative address in the Python program if they do not wish to use the gateway's address.

Sample output

```
#> Python dest_addr_to_me.py

Will confirm that Gateway/Coordinator address is every node's dest_addr
  FYI: Coordinator is ZB_2007, with address=[00:13:a2:00:40:3e:1c:80]!

Checking node BELA_2=[00:13:a2:00:40:3e:15:2d]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:3e:15:2d]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:3e:15:2d]!,WR) returned True

Checking node FANI_6=[00:13:a2:00:40:52:29:d7]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:52:29:d7]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:52:29:d7]!,WR) returned True

Checking node ANNA_1=[00:13:a2:00:40:3e:15:18]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:3e:15:18]!,DL,1077812352) returned True
```

```
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:3e:15:18]!,WR) returned True

Checking node DEBI_4=[00:13:a2:00:40:34:16:14]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:34:16:14]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:34:16:14]!,WR) returned True

Checking node ELSA_5=[00:13:a2:00:40:52:29:f9]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:52:29:f9]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:52:29:f9]!,WR) returned True

Checking node CALI_3=[00:13:a2:00:40:4a:70:7e]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:4a:70:7e]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:4a:70:7e]!,WR) returned True

#> Python dest_addr_to_me.py

Will confirm that Gateway/Coordinator address is every node's dest_addr
    FYI: Coordinator is ZB_2007, with address=[00:13:a2:00:40:3e:1c:80]!

Checking node BELA_2=[00:13:a2:00:40:3e:15:2d]! settings
- Node has desired settings already

Checking node FANI_6=[00:13:a2:00:40:52:29:d7]! settings
- Node has desired settings already

Checking node ANNA_1=[00:13:a2:00:40:3e:15:18]! settings
- Node has desired settings already

Checking node DEBI_4=[00:13:a2:00:40:34:16:14]! settings
- Node has desired settings already

Checking node ELSA_5=[00:13:a2:00:40:52:29:f9]! settings
- Node has desired settings already

Checking node CALI_3=[00:13:a2:00:40:4a:70:7e]! settings
- Node has desired settings already

#>
```

Download the Python code

This code only runs on a Digi ConnectPort X gateway: Python program "Dest_addr_to_me.zip.py" in ZIP form

WSBrowser

WSBrowser sample test

(**For java supported modules**) This program is a webservice browser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java WSBrowser Test Sample Application

The WSBrowser Test sample application can be found here: WSBrowser.zip.

Basic usage

Compile, load and run program using java environment.

Sample of WSB.java file:

```
package com.digi.wsb;

import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.text.ParseException;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import com.digi.json.JsonObject;
import com.digi.utils.Base64;
import com.digi.utils.misc;

public class WSB extends WsbUi {
        private static final long serialVersionUID = 1L;


       /**
        * @param args
        */
        public static void main(String[] args) {
                new WSB().setVisible(true);
        }


        public WSB() {
                super();

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
```

```
                                System.exit(0);
                        }
                });
        }


        @Override
        public void goButton_onClick(ActionEvent e) {
                try {
                        String method = ((String) methodComboBox.getSelectedItem
()).toUpperCase();
                        String url = urlTextField.getText();
                        String username = usernameTextField.getText();
                        String password = new String
(passwordTextField.getPassword());
                        String mime = (String) mimeContentType.getSelectedItem();
                        String payload = postDataTextArea.getText();

                        if (username.equals("")) {
                                username = null;
                        }

                        if ("GET".equals(method) || "DELETE".equals(method)) {
                                getData(method, url, username, password);
                        }
                        else if ("POST".equals(method) || "PUT".equals(method)) {
                                pushData(method, url, payload, mime, username,
password);
                        }
                } catch (Exception ex) {
                        log(ex);
                }

        }


        public void log(String logging) {
                try {
                        JsonObject jobj = new JsonObject(logging);
                        logTextArea.append(jobj.display() + "\n");
                } catch (Exception e) {
                        logTextArea.append(logging + "\n");
                }

                logTextArea.setCaretPosition(logTextArea.getText().length());
        }


        public void log(Exception e) {
                log(misc.getStackTrace(e));
        }


        public WebResult getData(String method, String path, String username,
String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
```

```
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");

                        // configure the authorization
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);

                        // get the steam from the server
                        InputStream str = conn.getInputStream();
                        int size = conn.getContentLength();
                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public WebResult pushData(String method, String path, String payload,
String contentType, String username, String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", contentType);
                        conn.setRequestProperty("Content-Length", payload.length
() + "");
                        conn.setDoOutput(true);

                        // configure the authorization
```

```
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);
                        log(payload);

                        // write the data
                        OutputStream out = conn.getOutputStream();
                        out.write(payload.getBytes());
                        out.close();

                        // get the steam from the server
                        InputStream str = conn.getInputStream();

                        int size = conn.getContentLength();

                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public void dumpConnectionRequest(HttpURLConnection conn) {
                Map<String, List<String>> map = conn.getRequestProperties();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
```

```
                        }

                        log(header.trim());
                }
        }


        public void dumpConnectionResponse(HttpURLConnection conn) throws
IOException {
                Map<String, List<String>> map = conn.getHeaderFields();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
                        }

                        log(header.trim());
                }
        }
}
```

XBIB display LEDs

Gateways (ConnectPort X series Devices) talk to the Device Cloud Server Platform through RCI Request and RCI Response. The RCI Request is sent to the Gateway in the xml string format.

To send continuously different RCI Request commands to the Gateway two types of **Python Code** are required :

- Python Code run on Gateway and talks to Device Cloud Server Platform through RCI
- Python Code run on PC and talks to Device Cloud Server Platform to send RCI Request and get RCI Response

Python code runs on Gateway

The Python Script "led_gateway.py", runs on Gateway to display LEDs on XBIB-U-Dev Board or XBIB-R-Dev Board in following manner.

1. Turn ON/OFF LEDs as switch pressed along with RCI Request command as "LED" or "SWITCH" .e.g. Switch 1 is pressed then LED1 will turn ON.
2. LED Randomly blinks with RCI Request command as "LED_DANCE".

Detail description of LED display code

The Python script below, runs in following different way.

1. If RCI request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">SWITCH</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

Then if for example **Switch 3** is pressed and on Device Cloud Send is pressed together, the LED 3 turns ON and in *RCI Response Window* Switch 3 is shown ON and other three switches are shown OFF. and if again example **Switch 3** is pressed and on Device Cloud Send is pressed together, the LED 3 turns OFF.

The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">SWITCH1 : OFF ; SWITCH2 : OFF
; SWITCH3 : ON ; SWITCH4 : OFF </do_command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

2. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">LED</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

   Then if for example **Switch 2** is pressed and on Device Cloud Send is pressed together, the *LED 2* turns ON and in *RCI Response Window* LED 2 is shown ON and other three LEDs are shown OFF.

   The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">LED1 : OFF ; LED2 : ON ; LED3
: OFF ; LED4 : OFF </do_command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

3. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">LED_DANCE</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

   Here by sending RCI Request command as **LED_DANCE**, LEDs on XBIB Dev Board turns ON/OFF in a predefined manner.
   The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">LED Dance Finished</do_
command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

4. If RCI Request on Device Cloud Server Platform under Web Service Console is written like this:

```
<sci_request version="1.0">
  <send_message>
    <targets>
      <device id="00000000-00000000-xxxxxxxx-xxxxxxxx"/>
    </targets>
    <rci_request version="1.1">
      <do_command target="LED_SWITCH">Swi</do_command>
    </rci_request>
  </send_message>
</sci_request>
```

Then if for example switch 3 is pressed and on Device Cloud Server Platform **Send** is pressed the LED 3 turns ON but in RCI response Window *Wrong Choice Entered* will be shown. As if either **SWITCH** or **LED** is typed in RCI request then only the result will be displayed on RCI Response.

The RCI Response will be displayed like this:

```
<sci_reply version="1.0">
  <send_message>
    <device id="00000000-00000000-xxxxxxxx-xxxxxxxx">
      <rci_reply version="1.1">
        <do_command target="LED_SWITCH">Wrong Choice Entered</do_
command>
      </rci_reply>
    </device>
  </send_message>
</sci_reply>
```

ZIP LED_GATEWAY

Here is the Python code as a ZIP file: Led_gateway.zip.

Here is the DigiXBeeDrivers.zip : DigiXBeeDrivers.zip.

Full Code

If you wish to browse the file for ideas, it is included inline below

```
            # led_gateway.py - RCI Callback functionality run on the gateway to
make the LEDs
#                  turn on and off based on the complimentary code and XBIB board
#                  switch presses

import sys
import zigbee
import rci
sys.path.append("WEB/Python/DigiXBeeDrivers.zip")
from sensor_io import parseIS

# User Entered Extended Address of the remote XBee
DEVICE_ID = "xx:xx:xx:xx:xx:xx:xx:xx"

#RCI Call Back Function
def rci_callback(xml):
    SW = zigbee.ddo_get_param(DEVICE_ID, 'IS')    #Read IO Samples
    SWDIR = parseIS(SW)
```

```
    #Check for Switch and LED Status
    cnt=0
    SA = ['DIO0','DIO1','DIO2','DIO3']
    for port in SA:
        SB = "%s" % port
        cnt=cnt+1

        if SWDIR[SB]==0:
            if cnt==1:
                if SWDIR["DIO12"]==0:
                    zigbee.ddo_set_param(DEVICE_ID,'P2',5)
                else:
                    zigbee.ddo_set_param(DEVICE_ID,'P2',4)
                elif cnt==2:
                    if SWDIR["DIO11"]==0:
                        zigbee.ddo_set_param(DEVICE_ID,'P1',5)
                    else:
                        zigbee.ddo_set_param(DEVICE_ID,'P1',4)
                elif cnt==3:
                    if SWDIR["DIO4"]==0:
                        zigbee.ddo_set_param(DEVICE_ID,'D4',5)
                    else:
                        zigbee.ddo_set_param(DEVICE_ID,'D4',4)
                elif cnt==4:
                    if SWDIR["DIO5"]==0:
                        zigbee.ddo_set_param(DEVICE_ID,'D5',5)
                    else:
                        zigbee.ddo_set_param(DEVICE_ID,'D5',4)

    SW = zigbee.ddo_get_param(DEVICE_ID, 'IS')
    SWDIR = parseIS(SW)

    #Check for valid xml string
    if (xml=='LED'):
        cnt=0
        LED_INFO = ['DIO12','DIO11','DIO4','DIO5']
        for port in LED_INFO:
            L_NO = "%s" % port
            cnt=cnt+1
            if SWDIR[L_NO]==0:
                SWDIR[L_NO]='ON'
            else:
                SWDIR[L_NO]='OFF'
            return "LED1 : %s ; LED2 : %s ; LED3 : %s ; LED4 : %s" %\
               (SWDIR['DIO12'],SWDIR['DIO11'],SWDIR['DIO4'],SWDIR['DIO5'])

    elif (xml=='SWITCH'):
        cnt=0
        SWITCH_INFO = ['DIO0','DIO1','DIO2','DIO3']
        for port in SWITCH_INFO:
            S_NO = "%s" % port
            cnt=cnt+1
            if SWDIR[S_NO]==0:
                SWDIR[S_NO]='ON'
            else:
                SWDIR[S_NO]='OFF'
        return "SWITCH1 : %s ; SWITCH2 : %s ; SWITCH3 : %s ; SWITCH4 : %s " %\
                (SWDIR['DIO0'],SWDIR['DIO1'],SWDIR['DIO2'],SWDIR['DIO3'])
```

```
    elif (xml=='LED_DANCE'):
        LED_D = ['P2','P1','D4','D5']
        for port in LED_D:
            zigbee.ddo_set_param(DEVICE_ID,port,4)
            LED_C = ['D4','P1','D5','P2']
            for port in LED_C:
                zigbee.ddo_set_param(DEVICE_ID,port,5)
                LED_B = ['D5','P2','D4','P1']
                for port in LED_B:
                    zigbee.ddo_set_param(DEVICE_ID,port,4)
                    LED_A = ['P1','D5','D4','P2']
                    for port in LED_A:
                        zigbee.ddo_set_param(DEVICE_ID,port,5)
        return "LED Dance Finished"

    else:
        return "Wrong Choice Entered"


# valid Extended Address
def valid_Extended_Address(DEVICE_ID):
    split_addr = DEVICE_ID.split(':')

    if len(split_addr) != 8:
        return False

    for char in split_addr:
        try:
            char = int(char, 16)
        except:
            return False


### Entry Point ###
if valid_Extended_Address(DEVICE_ID) == False: # Verify Length of Extended
Address
    print "Invalid Extended address given: %s" % DEVICE_ID
else:
    DEVICE_ID = '[%s]!' % DEVICE_ID

#Enable LEDs to DO High Configuration
led = ['P2', 'P1', 'D4', 'D5']
for at in led:
    try:
        zigbee.ddo_set_param(DEVICE_ID,at,5)
    except:
        print "parameter: %s can not be executed at address: %s" %(at, DEVICE_ID)
        print "command failed!"

#Enable Switches to DI Configuration
Switch = ['D0', 'D1', 'D2', 'D3']
for at in Switch:
    try:
        zigbee.ddo_set_param(DEVICE_ID,at,3)
    except:
        print "parameter: %s can not be executed at address: %s" %(at, DEVICE_ID)
        print "command failed!"

#Listening RCI message
```

```
rci.add_rci_callback("LED_SWITCH", rci_callback)
```

Python code runs on PC

Python scripts can be written to send standard HTTP requests to the server. These scripts use Python libraries to handle connecting to the server, sending the request, and getting the reply.

The Python Script “led_pc.py” runs on PC to send RCI callback requests through Device Cloud Server Platform to the Gateway and to show SCI Response on Console Window in Digi ESP for Python. Here HTTP POST of an SCI request written in Python.

ZIP LED_PC

Here is the Python code as a ZIP file: Led_pc.zip.

Full Code

If you wish to browse the file for ideas, it is included inline below

```
            # led_gateway.py - RCI Request and Response through Digi ESP for
Python

import httplib
import base64
import time



IDIGI_USERNAME  = "username"     #Username for developer.iDigi.com
IDIGI_PASSWORD  = "password"     #Password for developer.iDigi.com
IDIGI_ADDR      = "developer.idigi.com"
GATEWAY_ID      = "xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx"     #Target Gateway
Device ID


def rci_post_cmd(cmd):
    auth = base64.b64encode("%s:%s" % (IDIGI_USERNAME, IDIGI_PASSWORD))

    scimsg = '<sci_request version="1.0"><send_message><targets><device id=\"' +
GATEWAY_ID + \
             '\"/></targets><rci_request version="1.1"><do_command target="LED_
SWITCH">' + \
             cmd + '</do_command></rci_request></send_message></sci_request>'

    webservice = httplib.HTTP(IDIGI_ADDR,80)
    webservice.putrequest("POST", "/ws/sci")
    webservice.putheader("Authorization", "Basic %s"%auth)

    webservice.putheader("Content-type", 'text/xml; charset="UTF-8"')
    webservice.putheader("Content-length", "%d" % len(scimsg))
    webservice.endheaders()
    webservice.send(scimsg)

    webservice.getreply()       #Get SCI Response
    print webservice.getfile().read()


### Entry Point ###
```

```
while 1:
    rci_post_cmd("SWITCH")
    time.sleep(60)   #60 sec Delay

    rci_post_cmd("LED")
    time.sleep(60)   #60 sec Delay

    rci_post_cmd("LED_DANCE")
    time.sleep(60)   #60 sec Delay
```

XBee 868 distance/link quality demo

Goals

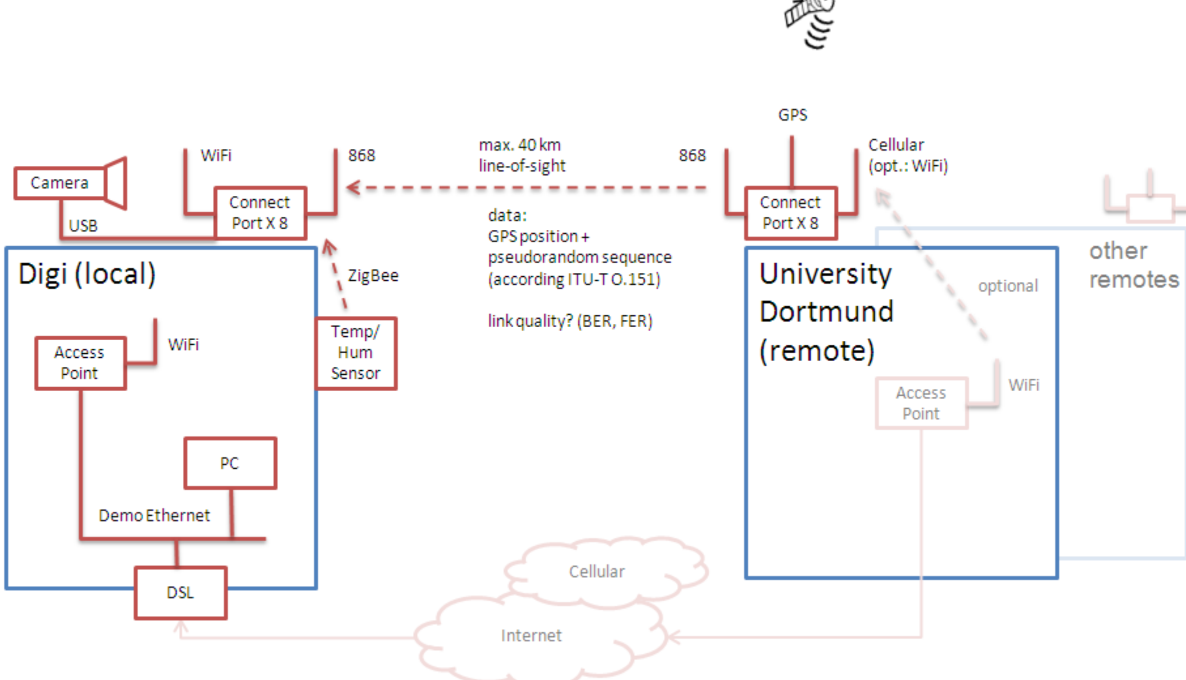
- Demonstrating high-range connection functionality live
- Integration of a sample Drop-In-Network application

Tools

- Test driver implementing pseudorandom sequence (according to ITU-T O.151 recommendation)
- Temperature/humidity sensor connected via ZigBee Protocol retrieves influencing environment data
- GPS demo running on remote device, which also periodically sends the position to local device

(tentative) Setup

- Two installations in Dortmund, Germany:
  - Local: Digi rooftop (Joseph-von-Fraunhofer-Str. 23)
  - Remote: rooftop of the University of Applied Sciences Dortmund (Emil-Figge-Str. 44)
- Maintenance access to the local devices via test ethernet
- Maintenance access to remote devices via cellular or WiFi/VPN access



How it works

Local setup

The local Connect Port X (CPX) device provides the following functionality:

Distance measurement:

- Receiving GPS position data from remote CPX via proprietary 868 protocol
- Calculating distance using remote GPS data
- Providing a public web server using Google Maps API (see screenshot below)

Link quality measurement:

- Receiving the pseudorandom sequence (PRS) from remote CPX
- Calculating link quality indicator values using remote RPS data
  - BER (Bit Error Rate)
  - FER (Frame Error Rate)

Remote setup

According to that, the remote CPX device provides this functionality:

Distance measurement:

- Obtaining GPS position data via built-in GPS module
- Forwarding position data to the local CPX via proprietary 868 protocol

Link quality measurement:

- Generating PRS data (polynomial x^15+x^14+1, according to ITU-T O.151 recommendation)
- Forwarding PRS data to the local CPX via 868

Miscellaneous

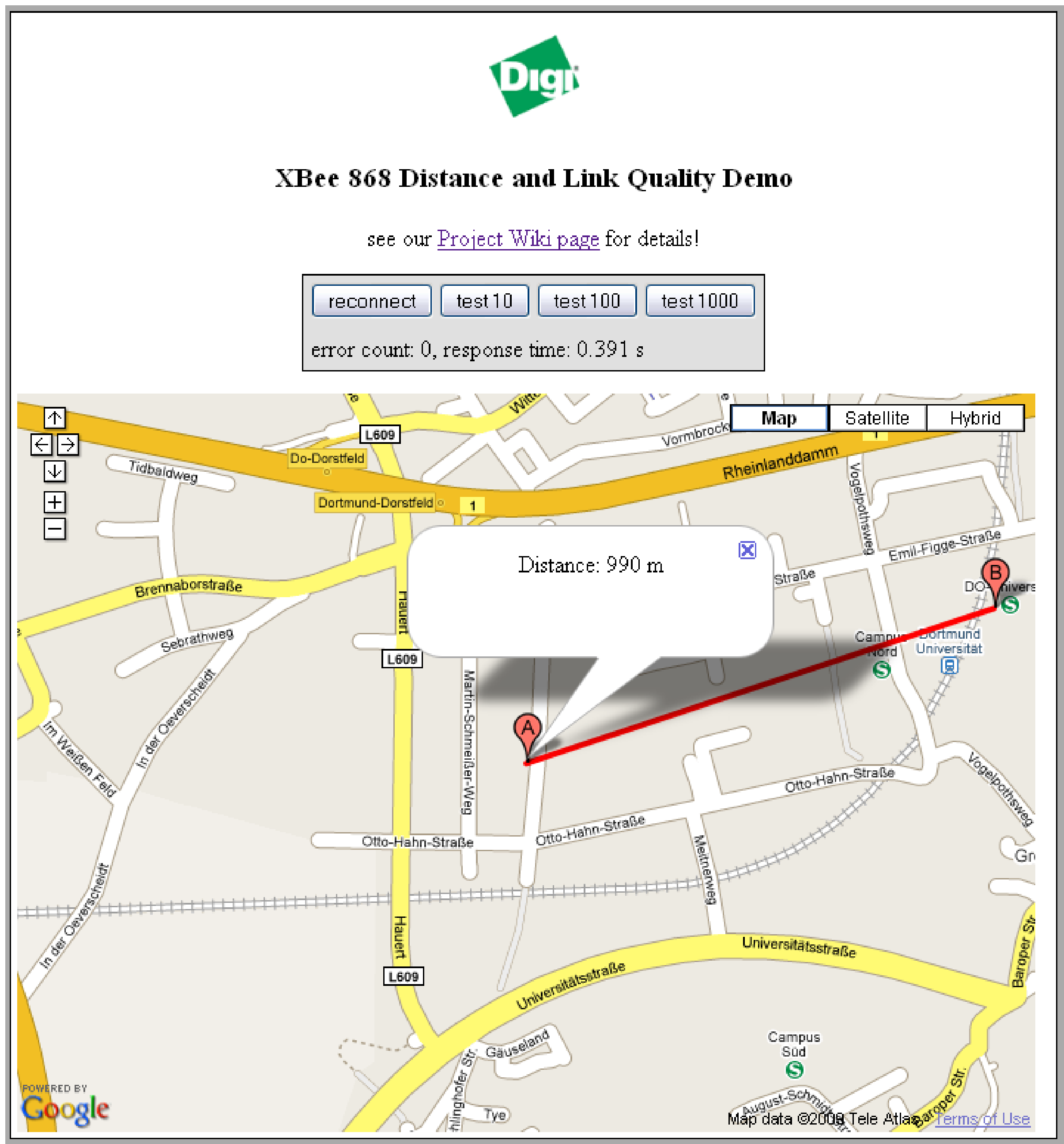
Providing remote maintenance access via cellular connection

Both

Each CPX device can be accessed via Digi ConnectWare Manager.

How it looks

Please take a look at a first version of the website that runs on the local CPX:



You can see the Google Maps API with the distance information and some buttons:

- Reconnect: initializes the connection between local CPX and remote CPX. Necessary if the local and remote pseudo random sequence generators are not longer synchronized.
- Test 10: send 10 packets with PRS data.
- Test 100: send 100 packets with PRS data.
- Test 1000: send 1000 packets with PRS data.

After performing a link quality test, a message with error counts and response time is generated below the buttons.

XBEE API packets

Program to design API packets

(**For ZigBee modules**) Program "Tx_Req 0x10" API packets generator for Zigbee modules(Xbee RF Modules (S1 {only DigiMesh}, S2, S3 and S8).

Test files

This sample program contains one file, "Tx_Req 0x10 packet generator.py".

XBee API Packet generator Test Sample Application

The XBee API Packet sample application can be found here: Tx_Req_0x10_packet_generator.zip.

Basic usage

Provide input where necessary.

Sample of Tx_Req 0x10 packet generator.py file:

```
########################################################################
#                   IMPORTS                                            #
########################################################################

import serial
import sys
import binascii


## Known Issue:
## 1. ONLY 1 byte length packet not transmitting



######  User Data #######
com_port = 'COM1'
dest_64bit = '0013A20040762859'
dest_16bit = 'FFFE'
rf_data = """!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`"""
#########################

def b_u(st):
    ## function to convert big-endian binary string into bytes[0-1] as int
    if len(st) == 1:
        return ord(st)
    else:
        length = len(st)
        sft = 8*(length-1)
                return (b_u(st[0])<<sft) + b_u(st[1:])



## create serial socket connection to talk with module
ser = serial.Serial(com_port, 9600, timeout=0.1, rtscts=True)

## convert RF data to hex
rf_hex = binascii.hexlify(rf_data)
```

```
##print "rf_hex=", rf_hex


                        ## calculate packet length
hex_len = hex(14 + (len(rf_hex)/2))
hex_len = hex_len.replace('x','0')
##print "hex_len=", hex_len

## calculate checksum
## 0x17 is the sum of all parameters minus 64bit & 16bit dest addr & payload
    checksum = 17


for i in range(0,len(dest_64bit),2):
    checksum = checksum + int(dest_64bit[i:i+2],16)

for i in range(0,len(dest_16bit),2):
    checksum = checksum + int(dest_16bit[i:i+2],16)

for i in range(0,len(rf_hex),2):
    checksum = checksum + int(rf_hex[i:i+2],16)

## checksum = 0xFF - 8-bit sum of bytes between the length and checksum
checksum = checksum%256
checksum = 255 - checksum
checksum = hex(checksum)
checksum = checksum[-2:]

## designing packet
tx_req = ("7E" + hex_len + "10" + "01" + dest_64bit
          + dest_16bit + "00" + "00" + rf_hex + checksum)
print "Tx packet = ", tx_req

## convert packet from hex to binary
data = binascii.unhexlify(tx_req)

## send data on serial line to module
ser.write(data)

## listin COM port for response
resp = ser.readline()

## convert response from binary to int
resp = b_u(resp)

## convert response from int to hex
resp = '%x' % resp
hex_data = resp.upper()
print "Response (in hex) = ", hex_data

## close connection
ser.close()
```

XBee active RFID

Active RFID with XBee 232 serial

Everyone should be familiar with the concept of RFID (or Radio-Frequency Identification):

- Major retail chains use RFID to detect theft (aka: those door alarms which go off too often)
- Many companies use an RFID badge or key-fob to manage door access.
- Automated highway tolls use an RFID signal from the car to charge tolls.

The basic concept is simple and consists of three distinct parts.

- **The RFID tag** sends out a code at a fixed time interval - generally every few seconds. Most RFID tags are transmit-only devices which have no concern if any reader received the 'message'.
  - **Passive RFID tags** are normally without power, thus normally quiet. They physically *harvest power* from a field generated by the reader, so can only send out codes when within a few inches of a suitable reader. Most door access and retail theft devices use passive tags. The tags are cheap and there are no batteries to fuss with.
  - In contrast, **Active RFID tags** use a small battery to constantly transmit a code, giving them a much larger range - commonly dozens or even hundreds of feet. Active RFID is often used in vehicle or out-of-door systems where close-proximity with a reader is impractical.

- **The RFID reader**, which acts as a bridge between radio-frequency and 'the application'. Most readers do little more than announce to the central application when tags are seen.
- **The Application** is responsible to match up the actual presence or absence of tags with required actions. For example, the application might signal a door to open when a correctly coded tag is seen, or it might notify security if tags disappear from a controlled location.

Example projects

In this project we'll use ACTIVE RFID tags because we want to detect the presence or movement of people - specifically allowing a 'work-space' to know when you leave, then turn off unrequired accessories like computer monitors, lamps and so on. A passive tag with the range of a few inches would force a change in user behavior, which would get tiresome and thus the system would not be used for long. Also, an active RFID tag with a range of 20-plus feet means the equipment starts powering up before you actually sit down in your chair.

In addition, a larger active RFID tag placed in your car would allow your house to know when you are out. It could for example text-page your cell phone if the garage door is still open 5 minutes after you leave, or text-page you if the garage door opens when you are not home. You can also have the outside lights turned on to greet you when you come home late at night.

A note on security/privacy

Remember that an active RFID tag is broadcasting to your environment "I - unit XYZ - am here". Literally, someone who knows which brand of tag you use can place readers to track your movement. A modern update of the old story where the mice wished to place a bell on the cat would have them placing an RFID tag on the cat instead, allowing the mice to locate the cat even when it slept!

In fact, most product brands allow the creation of DUPLICATE TAGS, so do not use this simple design to automatically open your home door! ***These products are not suitable for home security - they are for home automation and hobbyist fun!***

Obtaining Hardware

You will find very few RFID products on sale for consumers - most large manufacturers only sell directly to OEM companies which customize the products to be sold as a total solution including the large central application. You will also find several source for hobbyist-style RFID components - raw boards which you can place in your own enclosure. Most brands would work. What you desire is a simple reader which sends the tag codes out as they are received.

The RFID devices being sold by http://cliste.sailwhatcom.com/ were selected for this project. You will require at least:

Active RFID Receiver (the RS-232 version - NOT the USB version)

Active RFID Transmitter (this model has a range of about 25 feet - there is another model with a range of 120 feet)

Digi XBee ZB RS-232 adapter with the appropriate power supply

Again, most brands of RFID equipment will work - you need an RS-232 reader and a tag to read.

Embedding the XBee 232 adapter into the reader

The first hurdle you will discover with this particular RS-232 RFID reader is that it expects to draw 12vdc power from the RS-232 port of a traditional IBM PC-class computer - this is a trait shared by many 'self-powered' RS-232 devices.

However, the EIA/RS-232 standard permits operation with any voltage between +3vdc and +15vdc. A Digi gateway such as the ConnectPort X4 and the Digi Xbee 232 Adapter both put out roughly 5 to

6vdc only, so while you'll find the RS-232 RFID reader works fine on your desktop, it won't function on either of those Digi products.

Fortunately, the XBee 232 Adapter offers an (almost) perfect solution - it will supply up to 50mA of 12vdc power on the DB9 pin #9 (this would be perfect if the power was the pin 4/DTR output instead of pin 9!). However, since we'll be hiding the XBee adapter inside of the RFID reader box, we will need to create a short custom cable anyway. So we'll run the pin #9 of the XBee 232 Adapter to the DCE DTR 'input', which provides the RFID reader with far more reliable power than any standard RS-232 port can offer. Another nice thing about this output power is it remains 12vdc even if the XBee 232 Adapter is being powered by 4vdc or 30vdc.

Make a 2 or 3 inch long, 4-wire cable with these pin-outs:

XBee 232 DTE DB9 male, so cable is female	RFID Reader DCE DB9 female, so cable is male
Pin 2 - Rxd input	Pin 2 - Rxd output
Pin 3 - Txd output	Pin 2 - Txd input
Pin 5 - Signal Ground	Pin 5 - Signal Ground
Pin 9 - Auxilliary power output	Pin 4 - DTR input (this is DCE port!)

You need to keep the custom cable small and flexible, so do not use 'hoods' or 'shells', plus use individual wires, not a multi-core cable. The entire cable will be hidden within the RFID reader housing. You could loop the pin #9 back to the XBee 232 Adapter's pin #6 (DSR Input) to create a

Configuring the XBee 232 adapter

Review the full, up-to-date information on the page Digi XBee RS-232 adapter.

In summary, configure the XBee 232 Adapter to forward all data to another node. Most likely this is to your gateway, but you could also send it to another XBee 232 Adapter. You need to configure the following settings:

- Set the PAN Id and other operational settings
- Set DH/DL to the MAC address of the destination for the RFID data
- Set the baud rate to 9600,8,N,1 (is the default for most XBee Adapters)
- Set D2 to 5 to assert (turns on auxiliary power)
- Although none of the control signals are required, you should probably set them as digital outputs and inputs as indicated on Digi XBee RS-232 Adapter
- Setting IR to non-zero will cause the status of the digital I/O to be automatically sent on a time cycle to the address in DH/DL (aka: send you the RS-232 control signal status)

Parsing the incoming data

Here is an actual packet from an RFID tag read - the data is the five bytes "31 61 4b 6f 20".

```
7e 00 17 91 00 13 a2 00 40 52 18 99 19 14 e8 e8 00 11 c1 05 01 31 61 4b 6f 20 35
```

Review the standard PDF manual for your XBee flavor to fully decipher this message, which is a frame type 0x91 or Explicit Rx Indicator. Without going into detail, is can be parsed as:

XBee 232 DTE DB9 male, so cable is female	RFID Reader DCE DB9 female, so cable is male
7e 00 17	Start and length of the API Frame
91	Frame Type
00 13 a2 00 40 52 18 99	MAC address of the XBee+Reader (you can have multiple)
14 e8 e8 00 11 c1 05 01	}
31 61 4b 6f 20	Serial Payload - 5 bytes of data
35	Checksum of the frame

Additional discussions

The fuzzy nature of RFID range

Unless you have metal-lined walls, you won't be able to use RFID to turn lights on or off on a room by room basis. An RFID tag with a 25-foot range means you can at best know if someone is a certain part of a house - for example if someone is in 'the garage area' or the 'bedroom area'.

You can play with the antenna wire of the reader or tag in an attempt to reduce range, yet then you risk creating dead spots where the user is in the room, but the signal is lost and the lights turn off. You'd also experience all the room lights needlessly turning on then off as you walk along a hallway.

I have used this Active RFID tag in my 'cube' at work for over a half-year. At times my equipment powers up when I am a good 50 feet way. At other times it powers up only when I am about to sit at my desk. If I walk past my cube on the way to someplace else - or I fill the bird-feeders outside of my office window - then my equipment powers up. So ideally, the application should use multiple sensors to detect my presence or absence.

If the RFID tag is in my pocket, then my keys or cell phones can block the signal, causing my computer displays to power off. In addition, some day the battery in the RFID tag will die. Make sure you have an easy way to manually override the powering off of your displays! In my design I have manual push-buttons which allow me to turn the equipment on or off without any intelligent XBee control

Noisy Data

Since the RFID tags send out a faint signal, low-cost units may return bad values (tags which do not exist). If two tags transmit at the same instant, the reader might merge them into one bad message. So if you expect 8 bytes of data, be prepared to see more or less than 8 bytes. Your neighbor might also buy the s

ke and powered - the RFID tags blindly 'chirp' out their code so the reader has to ready at every second. This makes them nice **general-purpose routers** in your mesh.

If you buy the 3-6vdc version of the XBee 232 Adapter, then you can couple it with an XBee LTH Sensor (the light, temperature, humidity device). Although by design it operates on 3 'AA' batteries (so 4-5.5v) you can easily modify it to run on 5vdc by soldering a linking wire to the XBee 232 Adapter. Then you have not only the RFID reader, but a light and temperature input at that site.

Making the RFID detection more directional

Mounting the RFID reader on a metal plate is an easy way blind the reader in one direction. Mounting the RFID reader inside a metal 'wall outlet box' or mounting it on a metal plate on a closet wall can easily make it more directional - to for example only detect tags in one of the three bedrooms of your home, but not the rest of the house.

Since RF is RF, this would interfere with the use of the XBee attached as a general purpose router, but you could mount the XBee adapter externally in this case.

XBee Digital I/O Adapter Relay Demo

IO-Adapter Relay Demo (aka DIN-Mesh-Light)

Introduction

This demo shows the basic functionality of the Digi XBee Digital IO (DIO) Adapter by using the digital output mode to drive a Relay which switches a light bulb.

A benefit of the XBee DIO Adapter is the support of the Open Collector (aka Open Drain) technology for digital output ports. Using that technology, you are able to operate with different voltages on different ports (e.g. 12V Relay on port 1, 24V device on port 2)

Requirements

This demo consists of the following components:

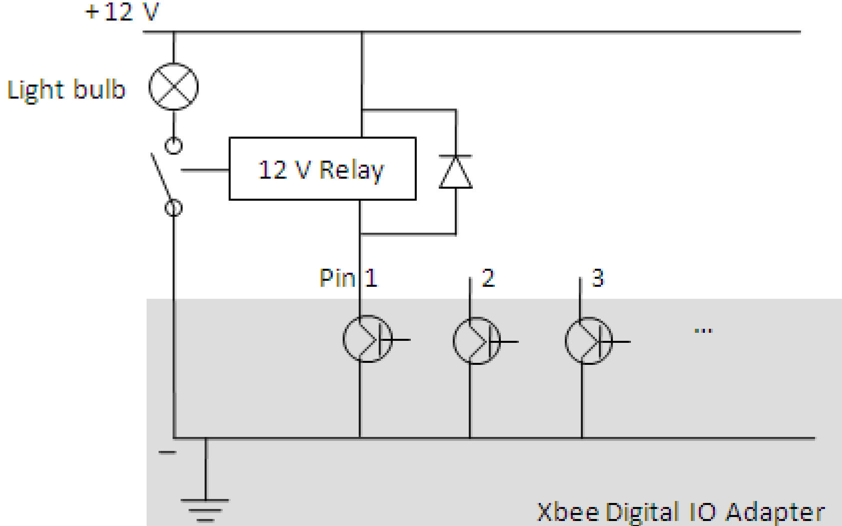
- Digi DIO Adapter
- Digi ConnectPort X
- Digi Development Board
- 12V Relay
- Light bulb
- According power supplys

Setup

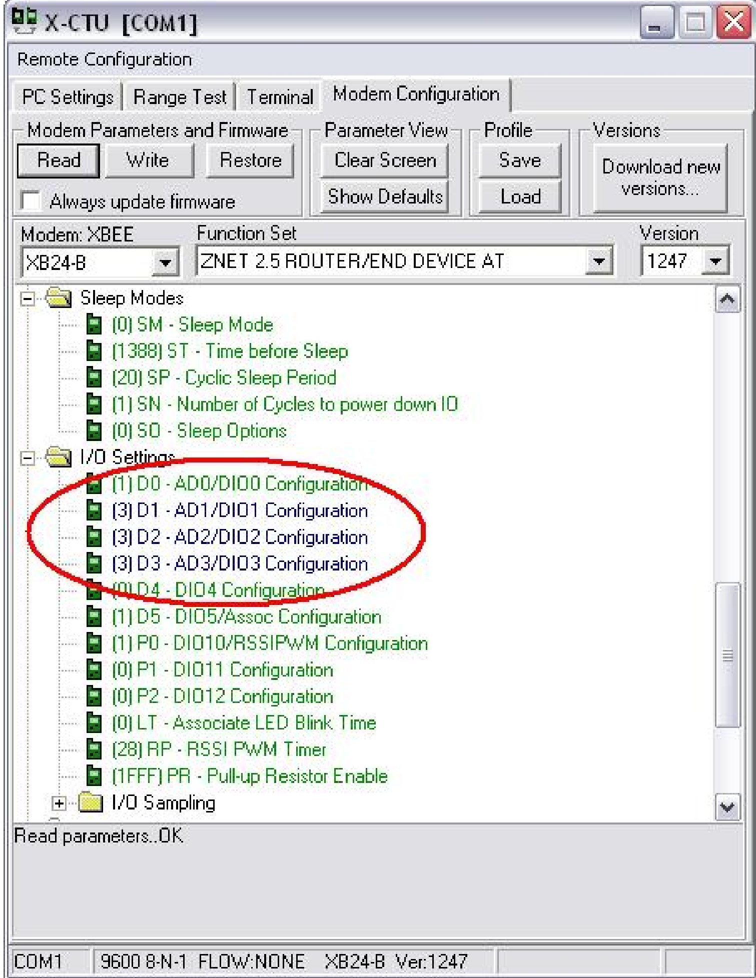
To better understand the responsibility for each device in this demo:

- The ConnectPortX device runs the Python scripts. It monitors the switches on the Digi Development Board and sends a command corresponding to the pressed Switch to the DIO Adapter in order to drive the Relay that switch on/off the light bulb.
- The Digi Development Board is set up to supports 3 switches, which have different behaviours as described in the following pages.
- The DIO Adapter waits to receive commands that change the state of PIN1, where the relay is connected to.

To gain an overview of the demo composition, please take a look at the following picture:



To set up the development board, start X-CTU and make sure that the parameters D1, D2 and D3 are all set to the value "3" (see below).



To set up the ConnectPort X, please follow these instructions:

- Upload all neccessary Python modules:
  - Python.zip
  - DigiXBeeDrivers.zip
  - Security_monitor.zip
  - zigbee.py
  - xbee_sensor.py
  - light_d.py
- Check if all three modules are visible in the "XBee network" menu (the module of the ConnectPort X itself, the Development Board and the Digital I/O Adapter).
- Open a telnet session to the ConnectPort X device and run "Python light_d.py" or start the Python script via autostart.

Starting up

You can **test your setup** by switching the light manually. To do this, please follow these steps:

1. Open the web interface of your ConnectPort X device.
2. Under the category Configuration, select **XBee Network**.
3. Select the device entry of your Digital IO Adapter in the Network View of the XBee Devices.
4. Expand the section Advanced Settings.
5. Look for the entry AD4/DIO4 configuration (D4) (which is responsible for switching pin 1) and change the value to **5**".
6. Click **Apply** and verify that the light will switch on.
7. Change the value back to **4** and the light will turn off.

If you want to **use the Python application** (light_d.py), you just have to ...

1. Open a telnet session to your ConnectPort X device and enter **Python light_d.py**.
2. Now you should be able to switch the light by using the switches on the development board:

- SW4: light on
- SW3: light off
- SW2: light on for 30 seconds

How it works

main

```
security_monitor = SecurityMonitor()
security_monitor.start()

security_monitor.add_monitor(light_off_timer, SCAN_DELAY)

# ensure that distance sensor is powered properly from adapter:
```

```
if (watchportD_exists == True):
    print "activate watchportD sensor"
    zigbee.ddo_set_param(watchportD_addr, 'D2', 5)
    zigbee.ddo_set_param(watchportD_addr, 'AC', '')
    security_monitor.add_monitor(check_range, SCAN_DELAY)

print "Application ready"
while 1:
    if (devboard_exists == True):
        check_keys()
        if (sw2_pressed == True):
            print "light on for 30 sec"
            light_on()
            lightCnt = 30
        elif (sw4_pressed == True):
            light_on()
        elif (sw3_pressed == True):
            light_off()
    else:
      sleep (1)
```

Limitations

Source

- Python.zip
- DigiXBeeDrivers.zip
- Security_monitor.zip

The next archive contains Python files that have to be uploaded unpacked on the Connect Port X.

- XBeeDIOAdapterRelay.zip

XBee bootloader menu

Program to bypass bootloader menu

XBee bypass bootloader menu (Xbee program) This sample application skips and bypasses freescale MCU & directly access radio.

Test files

This sample program contains several files. Program is in file "main.c".

Bypass bootloader menu Test Sample Application

The bootloader menu Test sample application can be found here: Bypass_bootloader_menu.zip.

Basic usage

```
  APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250.
```

Sample of bootloader bypass menu:

```
#include <xbee_config.h>
/* #include <types.h>
#include <xbee/platform.h>*/


void main(void)
{
/*       sys_hw_init();
        sys_xbee_init();
        sys_app_banner(); */
        //APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250
        sys_reset(APP_CAUSE_BYPASS_MODE);
}
```

XBee sensor

Python program for XBee sensor

XBee sensor (Python program) This program gives a user information regaridng battery status of sensor.

Test files

This sample program contains one file. File name "check_battery_life_sensor.py".

XBee battery sensor test sample application

The check_battery_life_sensor.py Python Test sample application can be found here: Check_battery_life_sensor.zip.

Basic usage

Make sure that the sensors are in the network coordinator. Provide the inputs where neccesary.

Sample of check_battery_life_sensor.py file:

```
import sys
import os
import struct
import zigbee
import xbee
from _zigbee import *
import time
import traceback
from struct import *

''' Provide the extended address(OUI) of the xbee sensor'''
###################################################
DESTINATION="00:13:a2:00:40:86:cd:11!"
###################################################

def calculate_battery(bat_vol):
    mv = float(bat_vol)
    mv = ((mv * 1200)/1024)
    v = mv/1000
    v = round(v, 2)
    return v

try:
```

```
    print "reading parameters, waiting five seconds..."
    # %V- Supply Voltage. Reads the voltage on the Vcc pin. Scale by 1200/1024 to
    # convert to mV units.
    param_value = zigbee.ddo_get_param(DESTINATION, "%V")
    param_value = struct.unpack("h", param_value)
    str_param_value = str(param_value)        # converting into a string
    s_index = str_param_value.find("(")
    e_index = str_param_value.find(",)")
    bat_voltage = str_param_value[s_index+1:e_index]
    battery_mv = calculate_battery(bat_voltage)
    if (battery_mv) >= 2.8 and (battery_mv) <= 3.4:
        print "battery is in the range of 2.8 and 3.4 and the battery value" + \
        + "is %s" %str(battery_mv)
    else:
        print "low battery"

except Exception, e:
    print "Exception %s" %e
    traceback.print_exc()

print "end of the program"
```

XBeeComm

Program to communicate XBeeComm through Com port with Windows 7

XBeeComm Sample (For Windows 7 PC) This application can communicate with the Xbee module from the PC through the COM port. Using the different buttons on the form we can get the different Xbee parameters like pan.

How does it work?

Using this C sharp application , we can communicate with the Xbee module from the PC. Using the different buttons on the form we can get the different Xbee parameters like pan id, version of the software.

Test files

This sample program contains nine files.

XBeeComm Test Sample Application

The XBeeComm Test sample application can be found here: XbeeComm.zip.

Basic usage

Compile, load and run program using Windows Tools.

Sample Program.cs file:

```
#using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace app2
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Zmatrix

Zigbee/802.15.4 to TCP/UDP Transfer Application: Zmatrix

This application moves data back and forth between Ethernet and Zigbee/802.15.4. The Ethernet sockets can be either TCP server or TCP Port. They can also be a variety of UDP sockets: UDP nocast, unicast, multicast, broadcast, and special cast.

This application is configured through a user specified configuration file. It is marked with examples to give you an idea how to configure it. You will however need to customize it to get it working in your environment. Specifically, you'll need to enter the hardware addresses of your Zigbee/802.15.4 modules into this configuration file.

The application will bind a particular Zigbee/802.15.4 end node to a particular behavior. For example, you may want your Zigbee/802.15.4 end device to talk to a TCP server on the network through the gateway. You will then need to setup the configuration file to have the application make a network connection to the TCP server when it starts.

Once this application is running, it can be terminated by making a tcp connection to the gateway on it's port number 47285. This is easily done with the telnet command in windows. For example: telnet 192.168.1.100 47285

... will accomplish this.

Here are the 7 possible type of connections to the zigbee side: TCP client TCP server UDP1 UDP2 UDP3 UDP4 UDP5

Connection Type Description

TCP client When the script executes, it will attempt to make an outbound TCP connection on the network side (as opposed to Zigbee/802.15.4 side). Once that connection is established, it will then pass data back and forth between the Zigbee/802.15.4 device and the device on the network that the TCP connection is connected to. When the program terminates (see above), this socket will be closed.

TCP Server: The script will setup a TCP socket and listen for an incoming TCP connection. Once the connection comes in the TCP connection is setup. Data will then be passed and forth between the Zigbee/802.15.4 device and the incoming TCP connection. When the program terminates (see above), this socket will be closed.

UDP1: (no cast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. However, any data received from the Zigbee/802.15.4 side is dropped

UDP2: (unicast) The script will listen on a user specified port for incoming UDP data. IF it receives any, it sends it to the Zigbee/802.15.4 side. If it receives any data from the Zigbee/802.15.4 side it will forward that to a specified IP address and port number via UDP.

UDP3: (special cast - sticky UDP) The script will listen on a user specified port for incoming UDP data. IF it receives any, it sends it to the Zigbee/802.15.4 side. It will also remember the last place it received network data (IP address and port number). When it receives Zigbee/802.15.4 data it will forward it to IP address and port number of last device that sent it UDP data on that port.

UDP4: (broadcast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. Data coming from the Zigbee/802.15.4 side will be sent over the network as a UDP broadcast.

UDP5: (multicast) The script will listen on a user specified port for incoming UDP data. If it receives any, it sends it to the Zigbee/802.15.4 side. Data coming from the Zigbee/802.15.4 side will be sent over the network as a UDP multicast.

Downloading the source

Zmatrix.zip

Digi Hardware Access

ConnectPort serial port access

Accessing the Digi Connect/ConnectPort serial ports

See also: termios

Enabling serial port a ccess

By default many Digi products assign the direct serial ports for use by terminal login (meaning they send out the "Login:" prompt). Since only 1 task can control a serial port at one time, this means attempts to open the serial port in Python will always fail.

Therefore, to enable Python access to the serial port you must open the Digi product's web interface and select the **Configuration | Serial Port** option on the left, then set the **Serial Port Profile** to RealPort, which frees the port up for Python access - as long as no host activates RealPort. You do not want the profile to be <unassigned>, plus you will need to reboot after setting this new profile.

Opening the serial port

Access the serial port of the Digi Connect/Connect Port family products using the Python "os" module (import os). While this provides an interface quite unlike most serial port libraries on Windows or other embedded system, it has the benefit of allowing the serial port to be treated as a normal TCP socket within a select() statement. Thus a Python program implementing a TCP server function to offer access to serial data can handle socket I/O and serial I/O within a single thread context.

The serial ports are named '/com/0', '/com/1', '/com/2' and so on. Digi products with the GPS virtual service have a device named '/gps/0', which acts like a serial port returning NMEA formated data; see Virtual GPS NMEA Access.

This code opens a serial port for read-write access, plus enables non-blocking behavior:

```
try:
  serfd = os.open( '/com/0', os.O_RDWR | os.O_NONBLOCK)
except:
  ( ... handle port in use or invalid name ... )
```

Reading from the Serial Port

As mentioned, if your application already receives TCP socket data via the select() statement, then add the serial port handle to the list of sockets for select to wait upon. This handle is the 'serfd' variable in the open() example above.

However, you can also read the port directly. In this example we handle non-blocking behavior to enable response timeouts. This example requires importing both 'os' and 'errno':

```
try:
  data = os.read( serfd, max_bytes)

except OSError, e:
    if( e.errno == errno.EAGAIN):  # EAGAIN just means NO data ready, try again
      ( ... handle NO data received or response timeout here ... )
```

```
    else: # other errors
      ( ... handle port faults - likely fatal ... )
      traceback.print_exc() # make sure you show the user what error was
```

Be forewarned: some of the 'OSError' text messages may be misleading since they are used for many purposes. Don't Panic.

Write to the serial port

Writing data is straight-forward. A 'try-except' is shown below, however unlike the non-blocking read() we do not expect to see errors here:

```
  try:
    count = os.write( serfd, data)
  except:
    ( ... handle port faults - likely fatal ... )
```

Closing the serial port

Closing is also straight-forward, and we ignore errors since any port faults might have invalidated the 'serfd' handle already:

```
  try:
    os.close( serfd)
  except:
    pass # ignore errors here
```

Setting serial port characteristics like Baud rate

See termios for complete details, but specific details are explained here.

Use 'import termios' to access the tcgetattr/tcsetattr functions. These are available on Digi Python products and Unix/Linux. They are not standard in Python for Windows computers.

Reading settings

```
print 'termios =', termios.tcgetattr( self.fd)
```

tcgetattr returns a list, with the above code printing this example from a Digi Connect WAN IA:

termios = [5120, 0, 48, 0, 9600, 9600, ['\x11', '\x13', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00', '\x00']]

The tcgetattr list is defined as: [iflag, oflag, cflag, lflag, ispeed, ospeed, cc]

The termios explains the constants required to decode these fields. The most interesting are:

- cflag defines the parity, start bits, stop bits and so on (shown as 48 or 8,N,1 here)
- ispeed is baud rate in (shown as 9600 above)
- ospeed is baud rate out (shown as 9600 above)

Writing settings

The easiest way to write the serial port setting is to first read them with termios.tcgetattr(), modify the returned list, and then resubmit the list to termios.tcsetattr(). You should not change settings you don't understand.

```
old = termios.tcgetattr(fd)
old[4] = 19200 # ispeed
old[5] = 19200 # ospeed
try:
  termios.tcsetattr(fd, termios.TCSADRAIN, old)
except:
  ( ... handle a port not accepting your settings ... )
```

Reading and Setting Control Signals

The termios gives nice examples of the special Digi-specific routines to use.

termios.tcgetstats() returns the status of flow, any incoming signals such as DSR, CTS, DCD and RI, and also the current status of the outgoing signals DTR and RTS.

termios.tcsetsignals() can be used to set the outgoing signals DTR and RTS - *as long as they are not already controlled by flow control*.

How to retrieve available free memory on a device

Retrieving the amount of Free memory on a device

This script is designed to retrieve memory on a Python enabled Digi product via the custom module Module: digicli.

Requirements

A Python enabled Digi product that natively contains the Module: digicli module.

Code

```
import digicli

def get_mem():
(flag, response) = digicli.digicli("display memory")

  if flag:

    for line in response:
      line = line.strip()
      line = line.lower()

      if line.startswith("free memory"):
        meminfo = line.split(":", 1)

        if len(meminfo) != 2:
          raise MemoryError("Unable to retrieve memory information from the
device")

        else:
          try:
              return int(meminfo[1])
        except:
            return meminfo[1]

     raise MemoryError("Unable to retrieve memory information from the device")
```

The Module: digicli is critical to our script. It allows us to interact with the product similar to how a user could, but in a programmatic way.

In this case, we use the string 'display memory' to give the output of the device's available memory. The return tuple from digicli is a flag to determine the success of the command, and the standard out of the command. If the flag is true, the response is a list of strings, one for each carriage return. If the flag is false, we raise a MemoryError exception, indicating we failed to retrieve the memory information.

We perform the strip() and lower() functions to massage the line into the form we want. We then use the startswith() function to find which line the free memory is on, split the line by ':' and validate it was split correctly. Then return the second element, as that should contain the amount of free memory on the device.

Notes and source

Note This function is dependent on the format of the memory information. At the time of writing this was performed on a ConnectPort X8 running EOS 7/01/08 daily firmware. The format may change at some time in the future, so retest this code anytime you attempt a new firmware or platform.

```
if __name__ == '__main__':
  print get_mem()
```

For easier testing, the above code can be added so when calling the script directly, it will call the function and display the results, whatever they may be.

Get_mem.zip

How to use a USB flash drive in Python

How to access an attached USB flash drive in Python

This article is about using the Attached USB Flash drive in Python, methods of reading and writing to it, and the current limitations of such.

See this Wiki page for information on FLASH filesystem statistics: Estimating free flash file space.

Hardware requirements

- A Digi USB/Python Enabled device. Typically a ConnectPort X4 or ConnectPort X8 device.
- A Compatible USB flash drive. Many flash drives are compatible, however any which include hardware (or Windows-specific) encryption or compression will not work. Also, the flash drive must be 4GB or smaller due to FAT filesystem limitations. To check, see 'Detecting your USB flash drive' section below.
- Command line interface access to the Digi device.

Digi Connect transport comments

In the examples below, the USB Flash drive is indicated by 'A/'. The Transport line (such as WR21 or WR44) use 'u:' instead. So instead of "A/tmp.txt", you would use "u:tmp.txt" - plus are restricted to the smaller 8.3 DOS-like names. Instead of duplicating the example, just use your imagination to see 'u:' instead of 'A\' in the examples below.

Detecting your USB flash drive

To detect your USB flash drive, attach the USB drive to an external USB port of the Digi device. Connect to the command line interface of the device, and type the command flashdrv. If a USB drive is detected, it will display volume information such as: The name of the USB flash drive, total space, used space, available space, and which volume it is mounted as. Make note of which volume the device is mounted as, typically A. Drive volume letters are assigned starting with A and progressing sequentially through the alphabet. It is not recommended that more than one drive be used, without an additional mechanism to identify the volume label, because the enumeration process may assign different letters on boot in this case.

Here is an example of the output:

```
#> flashdrv
Volume           Bytes        Used    Available         Bad    Use%   Mounted on
DISK_IMG     127.63 MB   124.09 MB      3.54 MB     0.00 MB    97.4   A/
```

Writing and reading to your USB flash drive

To write and read from the USB flash drive, the volume the drive is mounted at is needed. For the example below, we will be assuming your drive is mounted as A.

```
fh = open("A/tmp.file", 'w')
fh.write("Foobar")
fh.close()
```

```
fh = open("A/tmp.file", 'r')
print (fh.read())
```

The above code opens a file on the volume mounted at A, writes to it, closes it, then reopens it and reads back the data within. If the file does not exist, the file will be created. The write method is destructive and will overwrite existing data. Use the append method below if needed.

```
fh = open("A/tmp.file", 'a')
fh.write("Foobar 2")
fh.close()
fh = open("A/tmp.file", 'r')
print (fh.read())
```

The above code opens a file on the volume mounted at A, appends it, closes it, then reopens it and reads back the data within. Data by default appended sequentially. If a carriage return is desired use `"\n"`.

Limitations of the USB flash drive

Similar to built in flash of the Digi product, the supported Python os commands are listed in the Digi Python Programming guide. In addition to that, there is no way currently in the web user interface and command line interface to list the contents either.

To test whether or not a file exists, you must attempt to open it. The exception that is generated will show you whether or not the file exists. If the file does not exist, an "IOError: [Errno 13] Permission Denied" exception will be generated.

Locking Power Connecter

Buying Locking Power Plugs for Digi Products

Most Digi products which come with a barrel-jack power now support a twist locking connector. Nice! No more power plugs falling out under abuse ... but where do you buy such connectors? Your local big-box or spare-parts store won't have them.

Digi supplied

Digi sells a four (4) foot pigtail with the correct connector attached, it is Part Number 76000732 and can be found on this web page: http://www.digi.com/products/model?mid=2441

Digi also sells a power "Y-cable" with one socket split into two plugs. It is 63000287-01 (it is shown on the Digi DialServ data sheet here: https://www.digi.com/products/models/760007www.digi.com/products/models/76000732.

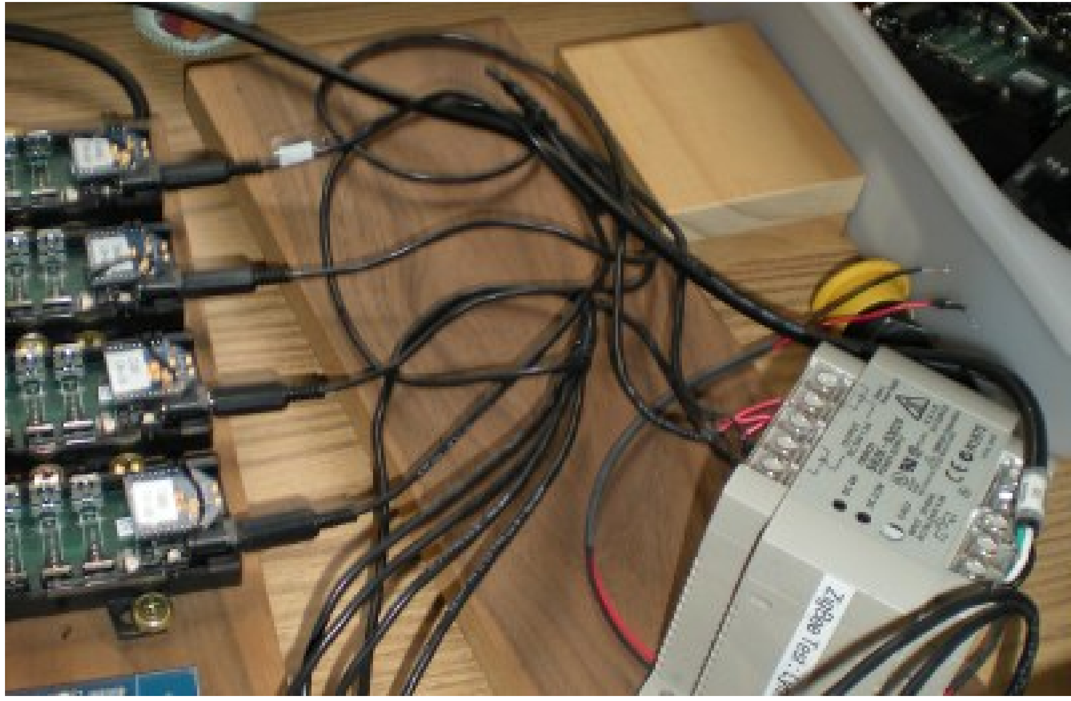
Self-installable

If you desire the loose connector which you can attach to a battery pack or other wire, one online source is http://www.mouser.com, which sells both the plug and sockets.

- Plug (male part) is 171-0725-EX and in July 2008 cost about $1.35 each. (The PDF spec is KC-300897)
- Jack (female part) is 163-1100-EX, but is for plate or PCB mount. (The PDF spec is KC-300840)

Making your life easier

As you develop for Digi Zigbee/Mesh products, you may find plugging in a dozen small power supplies (aka: wall-warts) a painful use of power strips. One simple solution is to buy some Digi supplied pigtails or the loose plugs, then retrofit the Digi supplied power packs to support multiple devices. For example you may have a 1A 12vdc or 3A 5vdc supply, yet none of the XBee adapters require that much power. You can splice in 2 or 3 extra plugs and use a single power supply to power multiple devices.



Another option is to buy a larger 5A or 25A off-the-shelf power supply and connect a dozen Digi supplied pigtails to it. The image above shows an industrial AC to 24vdc supply which enables a single power cord to powers a test setup consisting of eight 9-30vdc XBee adapters and one Digi ConnectPort X4 gateway.

Of course use your engineering common-sense; estimate the required current draw of your multiple products and make sure you are NOT overloading the power supply! These multi-headed wall-warts probably are a BAD IDEA for actual site installations, but wonderful for your own development.

Module: xbee sensor

The *xbee_sensor* module provides a convenient object interface for parsing sensor data returned from Digi XBee™ Sensor Adapters. Sensor data is obtained from an XBee Sensor adapter by using the **1S** command of the DDO (“Digi Data Objects”) interface. See the ddo_get_param() function for details.

XBeeWatchportA

XBeeWatchportA – interpret 1S sample for Watchport®/A accelerometer.

Attributes

xout

Acceleration in force units g along the x axis.

yout

Acceleration in force units g along the y axis.

pitch

Degrees of tilt along x axis (-90° = x = 90°).

roll

Degrees of tilt along y axis (-90° = y = 90°).

Methods

XBeeWatchportA()

Purpose

Construct a new XBeeWatchportA object.

Syntax

```
XBeeWatchportA()
```

Description

Create an *XBeeWatchportA* object in order to parse DDO 1S samples into real-world units.

set_0g_calibration()

Purpose

Calibrate the accelerometer to 0 at rest.

Syntax

```
set_0g_calibration(sample) . None
```

Description

This method is used to calibrate the object for use with samples from a Watchport/A accelerometer. Place the sensor flat, aligning both the X and Y axis at 0°, acquire a 1S sample from the XBee Adapter via DDO, and call this method with the sample. The object will then be calibrated for this accelerometer.

parse_sample()

Purpose

Parse a *1S* sample into real-world units.

Syntax

```
parse_sample(sample) . self
```

Description

This method parses a given DDO 1S sample into real world units. After calling this function, the sensor data will be available by accessing the object's attributes.

XBeeWatchportD

XBeeWatchportD – interpret 1S sample for Watchport/D distance sensor.

Attributes

distance

Distance from sensor in cm (20 cm = x = 150 cm).

Methods

XBeeWatchportD()

Purpose

Construct a new *XBeeWatchportD* object.

Syntax

```
XBeeWatchportD()
```

Description

Create an *XBeeWatchportD* object for parsing DDO 1S samples into real-world units.

parse_sample()

Purpose

Parse a 1S sample into real–world units.

Syntax

```
parse_sample(sample) . self
```

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

XBeeWatchportH

XBeeWatchportH – interpret 1S sample for Watchport/H humidity sensor

Attributes

sensor_rh

Relative humidity without considering temperature component.

true_rh

True relative humidity derived from considering ambient temperature.

temperature

Temperature in degrees Celsius.

Methods

XBeeWatchportH()

Purpose

Construct a new *XBeeWatchportH* object.

Syntax

```
XBeeWatchportH()
```

Description

Create an *XBeeWatchportH* object in order to parse DDO 1S samples into real-world units.

parse_sample()

Purpose

Parse a 1S sample into real-world units.

Syntax

```
parse_sample(sample) . self
```

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

XBeeWatchportT

XBeeWatchportT – interpret 1S sample for Watchport/T temperature sensor

Attributes

temperature

Temperature in degrees Celsius.

Methods

XBeeWatchportT()

Purpose

Construct a new XBeeWatchportT object.

Syntax

```
XBeeWatchportT()
```

Description

Create an *XBeeWatchportT* object in order to parse DDO 1S samples into real-world units.

parse_sample()

Purpose

Parse a 1S sample into real-world units.

Syntax

```
parse_sample(sample) . self
```

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

Power Cord Reminders

AC Power Cord Reminders

The information on this page is not strictly related to Digi products, but mis-wired AC cords can be a very expensive mistake.

In general, Digi products shipped to the US follow the traditional Black/White/Green colors, while international products should follow the IEC colors of Brown/Blue/Green+Yellow. However, users will (from time to time) be required to modify a US product for international use, or modify an international product for US use.

Wire Color Codes

AC Function	USA Color	IEC Color	USA Socket Remarks
Live / Hot	Black	Brown	Brass Screw, Narrower blade in polarized socket
Neutral	White	Blue	Silver Screw, Wider blade in polarized socket
Safety Earth	Green	Green + Yellow Stripe	Green-Tinted Screw

RS485 DB9 on Connect Products

Using RS-485 on the DB9 port of Digi Connect / ConnectPort Products

Not all Digi products support RS-422/485; some support RS-232 only. Those which do support selectable RS-422/485 refer to this as Multi-Electrical Interface or MEI.

Digi Support has an official document that covers EIA-232/422/485 in general, which is document 90000253 and titled *Cable Guide Including all PortServer TS, Digi Connect, and Digi One Products*. The PDF is here: Document 90000253_E as-of Sept-2009

This Wiki page directly covers products which use a standard DB9 serial port (a DTE or PC-like port).

Relevant Digi Products

This discussion covers at least the following Digi Products:

- Digi Connect SP
- Digi Connect Wi-SP
- Digi Connect WAN, WAN IA, WAN VPN
- Digi Connect WAN 3G, WAN 3G IA

Pinouts

Pin	RS-232	RS-422/485 4-wire	RS-485 2-Wire
1	DCD = Input	CTS-- = Input	Not Used
2	RXD = Input	RXD+ = Input	RXD+ or Not Used
3	TXD = Output	TXD+ = Output	DAT+ (often B)
4	DTR = Output	RTS-- = Output	Not Used
5	GND, Signal	GND, Signal	GND, Signal
6	DSR = Input	RXD-- = Input	RXD-- or Not Used
7	RTS = Output	RTS+ = Output	Not Used
8	CTS = Input	CTS+ = Input	Not Used
9	Not Used	TXD-- = Output	DAT-- (often A)

Note: "A" is often "-" and "B" is often "+". For historical reasons, some vendors reverse label "A" and "B", so it is a less predictable naming convention than +/- (which a few vendors also reverse label!) So make sure you try swapping the +/- (A/B) if you cannot communicate. EIA-422/485 compliant devices **cannot** be damaged by reverse wiring the +/- (A/B) signals.

Cable: RS-422/485 4-wire Full

- Pins 3 and 9 are your Transmit pair, and they will connect to the RX/Receive pins of your remote or slave device(s).
- Pins 2 and 6 are your Receive pair, and they will connect to the TX/Transmit pins of your remote or slave device(s).
- Pins 7 should short to pin 8, this is RTS->CTS (+)
- Pins 4 should short to pin 1, this is RTS->CTS (-)

Note Do you require the RTS/CTS loopback? Technically, the Digi product does not require this. However, this loopback is required if a user enables hardware flow control on the port, or if a remote application using Digi RealPort were to assert RTS and then expect CTS to be viewed as also asserted.

Cable: RS-422/485 4-wire Minimum

- Pins 3 and 9 are your Transmit pair, and they will connect to the RX/Receive pins of your remote or slave device(s).
- Pins 2 and 6 are your Receive pair, and they will connect to the TX/Transmit pins of your remote or slave device(s).
- The RTS/CTS pins are ignored - See the RTS/CTS note under RS-422/485 4-wire Full cable.

Cable: RS-485 2-wire Full

- Pins 3 and 9 are your Data+/Data- lines respectively
- Pins 2 should short to pin 3, this shorts the Txd+ and Rxd+ to form Dat+
- Pins 6 should short to pin 9, this shorts the Txd- and Rxd- to form Dat-
- The RTS/CTS pins do not apply

Note Do you require the Txd/Rxd shorting? Technically, most Digi products do not require this. However, some future product might require it.

Cable: RS-485 2-wire Minimum

- Pins 3 and 9 are your Data+/Data- lines respectively
- Pins 2 and 6 are ignored - *See the Txd/Rxd note under RS-485 2-wire Full cable*.
- The RTS/CTS pins do not apply

Products tested to work fine with either the RS-485 2-wire Full or Minimum 3-wire cable

- Digi Connect WAN IA with 16MB memory (which implies it works on WAN and WAN VPN with 16MB memory since they share main boards)
- Digi Connect WAN 3G/3G IA

Enabling RS-422/485

Software Select - Web UI

You will find the setting under the: **Configuration > Serial Ports > Serial Port Configuration >** The Multiple Electrical Interface (MEI) Serial Settings web page.

Software Select - Telnet/CLI

The set switches command is used. (See the appropriate Family Command Reference such as 90000566 for more information). RS-422 is considered RS-485 4-wire.

```
set switches [port=range]
    [mode={232|485}]
    [wires={two|four}]
    [termination={on|off}]
```

DIP Switches

Older Digi Connect products included 4 DIP switches to change between RS-232 and RS-485. Since most of these products have been converted to software select, these settings won't be covered here. Such products will have a label attached showing the settings.

About Termination and Bias

Although "termination" rarely has noticeable impact on RS-485, the line bias used to quiet idle floating lines is general critical. Failing to apply adequate line bias either causes complete RS-485 communications failure, or else causes intermittent noise problems which can be very hard to diagnose. You should always enable the termination on a Digi RS-485 port by default and then only disable it as a last resort when trying to solve a communication problem.

If the device(s) you are connecting already apply a line bias, then line bias within the Digi might be redundant or even disruptive.

Pin 5 Connection (Ground/Signal Reference) is Critical

Although RS-485 products **FROM A SINGLE VENDOR** can often run as pure 2 or 4-wire and no explicit ground, this rarely works reliably when mixing product from multiple vendors. This is because RS-485 2-wire is NOT a loop - a transmitting driver is sourcing a tiny current on both "+" and "-", while a receiving driver is sinking a tiny current on both "+" and "-". Thus a common ground reference must exist to complete the circuit.

So while a specific Vendor X can say "Our products work together without a 3rd (or 5th) Ground wire", they CANNOT say "Therefore other unknown products won't need a 3rd (or 5th) Ground wire to talk to us." ***This is a fallacy***.

Using RS-422/485 to connect a Digi product to a third-party product without an explicit ground reference MIGHT work - but it might not. Or it might generally work and intermittently NOT work. Therefore assume that you need to properly handle the ground/signal return.

If the third party product lacks a signal reference for the RS-422/485, try to find any signal which that product treats as a logic or signal ground. Worstcase, connect the Digi's signal ground via a 1-watt, 1000-ohm resister to the 0-volt/negative return of that product's DC power supply.

Symptoms of Missing Ground Wires

The most common symptom is that the communications works for a few hours, days, or weeks, then stops working for hours at a time. Rebooting the devices may restores communications (or may not). Refreshing the Digi MEI settings may restore communications (or may not). Disconnecting and reconnecting the cable may restore communications (or may not). But in all cases the failure will repeat again in a few hours, days or weeks.

Read and Write Flash USB

Digi products supporting Python allow you to read or write files from FLASH - both the main system flash and (if available) USB flash drives with standard FAT formatting.

Big Warning

We start with the warnings, because a bad (stupid) program design on your part will destroy your product in a few months. Flash memory is NOT a hard drive, so do not plan on using it like one.

Do not plan to write to flash every second!

Many programmers start with the idea "I'll save my status (or rewrite a web page) every second." You cannot do this, so instead:

- Dynamic Web pages should be handled using the Module: digiweb, which allows the product's main web server to pass unknown URL to your Python program. This allows you to have dynamic web pages which are always up-to-date without writing to flash at all.
- For data logs - be reasonable. Assuming your product isn't crashing and rebooting every few seconds, then the primary reason for RAM-based data to be lost is when the product loses power. So stop and think - if the power is out for 30 minutes or 7 hours hours, how earth-shattering is losing 35 minutes or 7 hours & 5 minutes worth of data instead of 30 minutes and 7 hours respectively?
- If you really require constant update, do so ONLY in external USB flash drives because they are much faster to update and not critical to the survival of the product.

Do not plan to maintain a complex database system!

While there are no seek times under Flash, the slowness of the updates means a power glitch or reboot will corrupt your data. The best designs are either flat files which are only appended to until they need to be trimmed to reduce size, or flat files with simple fixed sized records. Any indexing or sort orders should be maintained ONLY in RAM and be manually rebuilt anytime the product restarts.

Do not use write() to build large data blocks!

While it is temping to use a for-loop to write 20 strings to a file in 20 write() calls, the proper solution is to assemble all 20 strings into one larger string in RAM, then use a single write() to update Flash. This will cause a noticeable improvement in both performance and Flash wear.

Always use Try/Except around file operation

Complex operating systems (Windows/Linux/MAC) do complex things to enable graceful multi-user access of the shared filesystems; The Digi embedded OS is not a complex multi-user system. Reading/Writing a USB drive will fail if the user pulls out the drive while you are updating. Reading/Writing the main flash might fail if the user deletes or replaces the files through the Web interface. Multiple threads affecting the filesystem might impact each other, and certainly two threads CANNOT share the same file unless the file is protected by a semaphore.

In general do NOT expect files to remain open forever

Assuming you are (correctly) updating your file only once per few minutes, then you should close and later reopen the file.

Code Examples

Now that the warnings have been issued, here are code fragments used in a successful data logging application. In general, the file I/O routines like open/read/write/close work. The functions like os.access() do NOT work.

Testing File Size

This routine (which we assume is part of a log-manager class) allows a thread to test the file size. Note the blocking semaphore which protects the flash file if multiple threads share the log, which in this example is true because one thread adds to the file, and another reads data from it for serving out to remote TCP clients.

```
    def get_bytes_in_file(self):
        """Return Byte count in FLASH file"""

        self.disklog_acquire_blocking()
        try:
            fileHan = file(self.get_disklog_filename(),'rb')
            fileHan.seek(0,2)
            self.bytes_in_file = fileHan.tell()
            # print 'Discover %d bytes in file' % self.bytes_in_file
            fileHan.close()

        except:
            # traceback.print_exc() - in case this should never happen
            # print "File not found"
            self.bytes_in_file = -1

        self.disklog_release()
        return self.bytes_in_file
```

Append Data to the File

This routine (which we assume is part of a log-manager class) takes a LIST of text or binary strings and appends them to a Flash file. In this application the data is being appended to a list of strings held only in RAM, then every five (5) minutes an event causes the disklog manager to dump the accumulated data to the Flash file.

A number of the functions here are not explicitly shown. For example, before appending the data the code needs to confirm if the file size has grown to the point it should be trimmed (cut in half for example). It also assumes a simple file "header" exists, which might be little more than some text showing the time/date the file was started (or was last compacted).

One simple method of "compaction" with low Flash overhead is to maintain two files, each one-half the maximum desired size. When the active one fills up, the older file (older half) is deleted and a new file with time-stamp in the name is used to start a new half. The actually application being shown in this example has both data-item and time-stamp info in fixed size records, thus its compaction is rather complex since it wants to delete the OLDEST records, yet to maintain some minimum number of records for each data-item.

```
    def disklog_append_rec_list(self,rec):
        """Store/Append LIST of bin-string to log file"""

        # pack the list into a single bin-string, we do NOT want to
        # issue any more than one (1) write() call
        rec = "".join(rec)
```

```
        # psuedo-code here, you'll need to consider this action
        # - test if the existing file will be TOO large if we add this
        # - if it is, then we need to COMPACT the file first

        bResult = False
        self.disklog_acquire_blocking()
        try:
            fileHan = file( self.get_disklog_filename(),'ab') # append
            if(fileHan.tell()==0): # then is NEW file
                print "Starting new log file %s" % self.get_disklog_filename()
                fileHan.write(self.make_diskfile_header())
            fileHan.write(rec)
            self.bytes_in_file = fileHan.tell()
            fileHan.close()
            bResult = True

        except:
            # else just make as EMPTY
            traceback.print_exc()
            self.bytes_in_file = -1

        self.disklog_release()
        return bResult
```

Delete the File

This routine (which we assume is part of a log-manager class) deletes a file by resetting it to zero bytes.

```
    def disklog_delete(self):
        """Delete FLASH file"""

        bResult = False
        self.disklog_acquire_blocking()

        try:
            fileHan = file( self.get_disklog_filename(),'wb') # clobber, don't
append
            fileHan.write(self.make_diskfile_header())
            fileHan.close()
            bResult = True

        except:
            traceback.print_exc()

        self.disklog_release()
        return bResult
```

Using Main Flash

In general you only want to use the internal Main Flash for configuration files, or for data records which are NOT updated any faster than once per five (5) minutes. This is because:

- The internal Flash is fairly slow to update, which impacts your Python program
- Digi's main flash is NOT wear-leveled; certain critical sectors are used extensively and will fail fastest!

- Updating this flash too often will render the product inoperable (meaning it breaks and you must BUY A NEW Digi device)
- The size is limited, and future firmware upgrades may unexpectedly squeeze the size even smaller.
- Storing data here consumes space desired for Python code

How to Access

The main FLASH appears as the directory "WEB/Python". Creating a file as "WEB/Python/data.csv" puts the file into Main Flash. The really nice thing about this main flash is the files show up within the Web Interface as the Python files for upload. For example, if you upload a file named "run.py" via the Web Interface, it can be accessed as "WEB/Python/run.py"

Using USB Flash Drives

In general using external USB Flash for Log and Event files is safest because:

- It has no impact on the critical internal Flash
- The FAT file system supported is faster, more standard, and of course allows you to move the files to a normal computer
- Larger sizes are possible - as of Oct-2008 once can easily buy 2GB to 4GB for from us$9 to $19.

How to Access

Any supported USB Flash drives appear as mounted drives, so the first USB port is "A/", the second is "B/" and so on. There is no drive ':', so it is NOT A:/. Creating a file as "A/data.csv" puts the file out in the USB Flash Device.

Questions: Does mkdir work? How to create/delete subdirectories? Questions: How to see USB files from Web UI or CLI/Telnet Questions: How to read space usage - how much free space exists

Products known NOT to work

- In general, many of the U3-style Flash sticks which support PASSWORD or ENCRYPTION do not work. To a Windows computer they look NOT like a FAT file system, but like a small CD-ROM (read-only) drive, which runs a Windows application, which asks the user for password and only mounts the normal FAT filesystem once the password has been accepted.
- Sony MicroVault Tiny (example: USM2GH, tested Jun-2008) does not work because it has built-in hardware compression which assumes use in Windows computers, so is NOT a simple FAT file system. Si nce these USB drives are literally about the size of cell-phone SIM, this is a big disappointment!

Read and write the Realtime clock

Python-enabled Digi Products with a hardware RTC

- Digi ConnectPort X4
- Digi ConnectPort X8
- Digi Connect WAN 3G
- Digi ConnectPort WAN VPN

The Realtime Clock is supported by a supercap designed to retain the time for 1 week without power. The hardware RTC only records time in seconds, so no sense of milliseconds are maintained. It also can drift up to 2 seconds per day or 1 minute per month.

Note that the Digi ConnectPort X2 does **NOT** have a hardware RTC. Although you can set the time and date, it will drift several minutes PER DAY due to the use of spread-spectrum clock signals which constantly vary the CPU speed.

*So any application expecting to use real time-of-day values on the **Digi ConnectPort X2** will require external update, such as NTP or manual rewrite at least once per day.*

Reading the RTC

From the telnet/command line

The **show time** command returns the date and time in the following format. Note that if you have never set the realtime clock, you'll see a warning message instead.

```
#> show time
Current Date and Time: Tue Jun 24 16:27:12 2008
```

To set the clock, use the set time command. Enter "**set time ?**" to see the help, then just format the new date and time as required.

```
#> set time ?
Sets real time clock time/date.

syntax: set time [options...]

options:

    time=hh:mm:ss -or- hh:mm
    date=mm.dd.yy

#> set time date=03.31.09 time=09:23:17
```

From Python

- import the *time* module.
- **time.time()** returns the UNIX-style time since 1-Jan-1970 as a float, assume the milliseconds are meaningless. This routine might involve serial shifting from the hardware RTC, so might take 5 to 7 msec to complete - or it might take longer. Literally, any call to time.time() might cause blocking hardware access and the return delay will be unpredictable. Moral; don't use this to just calculate durations - how many seconds some function required to run. If the system clock on the device has not been set, this will instead return seconds since the device was powered on. For consistent behavior, ensure that you set the time in the system. See below.
- **time.clock()** instead just returns a number derived from the system clock and has no basis in "real time". However it is very fast and the milliseconds are valid, so this is the ideal function to use for determining durations - as a stop-watch or timeout.

Writing the RTC

From the telnet/command line

```
#> set time
Sets real time clock time/date.

syntax: set time [options...]
options:
    time=hh:mm:ss -or- hh:mm
    date=mm.dd.yy
```

From Python

- Import the *time* module.
- Import the *digicli* module. This module acts much like a telnet session, so to set the RTC we need to format a text string which looks like: s**et time date=02.29.08 time=15:23:56**. The routine below creates such a string.

```
def MakeDigiSetTimeString(secsSinceEpoch):
    """Given secsSinceEpoch, create string required for DigiCli SET TIME
    such as "set time date=02.29.08 time=15:23:56"
    """
    try:
        timtup = time.localtime(secsSinceEpoch)
        # yr4, mon, day, hr, min, sec, wekdy
        st = 'set time date=%02d.%02d.%02d ' % (timtup[1],timtup[2],(timtup
[0]%100))
        st += 'time=%02d:%02d:%02d' % (timtup[3],timtup[4],timtup[5])
        return st
    except:
        return ""
```

Then this routine feeds the string into the digicli module:

```
def SetTimeUnix(secsSinceEpoch):
        """\
    Like time.time(), but returns int (not float) plus allows spoofing time
    """
    try:
        st = MakeDigiSetTimeString(secsSinceEpoch)
        if sys.platform[:4] == 'digi':
            success, response = digicli.digicli(st)
            # print 'digicli.success = ', success
            # print 'digicli.response = ', response
        else:
            print st
        return True
    except:
         return False
```

Here is a routine which sends the time from the computer running the script

```
            # cpx_settime.py
#
# use telnet to set CPX8 time/date

import sys
import time
import telnetlib

if __name__ == '__main__':

    bLive = True
    settings = { 'cpxip':"192.168.1.20",'tcpport':23,'username':"",'password':""}

    if len(sys.argv) > 1:
        # see if we have any arguments on command line
        settings.update( { 'cpxip' : sys.argv[1] } )
    else:
        print 'Copies local time from your PC to a Digi CPX with real-time clock'
        print 'usage" %s {ip of CPX}' % sys.argv[0]
        sys.exit( 0)

    now = time.localtime()
    # time = HH:MM:SS date=mm.dd.yy
    setim = "set time time=%02d:%02d:%02d" % (now[3],now[4],now[5])
    setim += " date=%02d.%02d.%02d" % (now[1],now[2],(now[0]%100))


    if not bLive:
        print "Would send RTC to <%s>" % setim

    else:

        cpxip = settings.get("cpxip","192.168.1.20")
        tcpport = settings.get("tcpport",23)
        username = settings.get("username","root")
        password = settings.get("password","dbps")

        print "Attempting to set RTC time <%s> on CPX at %s:%d" %
(setim,cpxip,tcpport)

        tn = telnetlib.Telnet(cpxip,tcpport)
```

```
if len(username) > 0:
    tn.read_until( "login: ")
    tn.write( username + "\n")
    tn.read_until( "password:")
    tn.write( password + "\n")

print tn.read_until( "#> ", 10)
tn.write( "show time\n")
print tn.read_until( "#> ", 10)
tn.write( setim + "\n")
print tn.read_until( "#> ", 10)
tn.write( "show time\n")
print tn.read_until( "#> ", 10)

time.sleep( 3.0)
tn.close()
```

Transport Python programmer's guide

Purpose of this guide

This guide introduces the Python programming language by showing how to create and run a simple Python program. It describes how to load and run Python programs onto Digi Transport devices, either through the command-line or Web user interfaces. It reviews Python modules, particularly those modules with Digi-specific behavior. This guide describes how to run the executable programs and describes program files.

What Is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

For more information on the Python Programming Language, go to http://www.Python.org/ and click the Documentation link.

The Transports use Python version 2.6.1.

Running Python

How to run a Python program on a Digi Transport router. Firstly check your firmware version on the router, we recommend using firmware version 5090 or later. Start by checking your router has Python in its firmware by following these simple steps:

Step 1: Using either a telnet or serial connection (default login/password = username/password) to the router issue the following commands

```
pycfg 0 stderr2stdout on
Python
```

Step 2: At the prompt now type the following command:

```
help()
```

The following should then be displayed:

```
Welcome to Python 2.6!  This is the online help utility.
```

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.Python.org/tutorial/.

Step 3: Type the following to exit the Python interpreter exit() Now you can upload your Python script to the router via FTP. For testing purposes you can simply run the script by using the following command:

```
Python filename.py
```

For the router to start the script automatically on Powerup / Reboot issue the following commands:

```
cmd 0 autocmd "Python filename.py"
config 0 save
```

First Python script

Lets start with a hello world program. Firstly create a text file with the following text inside:

```
print "Hello World!"
```

Save this file with a file name of myfirst.py Now FTP the myfirst.py text file onto the router and issue the following command: Python myfirst.py

The router should produce the following output:

```
OK
Hello World!
```

Miscellaneous items

Your Python code can detect when it is running on a Transport product by importing the SYS module, then testing the sys.platform variable like this:

```
if sys.platform == 'digiSarOS':
   print "Running on Digi Transport"
```

File names on the Transport are limited to 8 characters or less. There is also a 3 character limit on file type extensions. Example:

```
  myfile.py  - is valid (6 characters and 2 character file type extension)
  myLongFileName.py - is not valid
  myfile.superlongextension - is not valid (6 characters are ok but we have more
than 3 characters on the filetype extension)
```

Also, please note that the Digi ESP, which is used as a code development tool, is not aware of this limitation. That means the programmer must make sure to use short filenames on their projects in the ESP.

Python examples can be generated on the Digi ESP code development tool (Digi ESP -> file -> new -> digi Python application sample project). This tool can be downloaded from the Digi website.

RCI is a remote protocol used for configuring the Transport from an application. RCI is defined on the Digi website in the RCI command reference Guide.

Device ID can be determined via the 'ati5' command.

Other example scripts

WR44 - bus demo, Python script

Below is the complete script we are using in our Bus Demo, this script collects the status of the two Digital Inputs and the current GPS co-ordinates. The script will only send data when there is a change in either the GPS co-ordinates or one of the Digital Inputs alters state. The file has been separated into individual modules for ease of explanation however you can download the complete script here bus-demo.py The required libraries import sarcli import time import socket Modules The first module converts the NMEA GPS info to lat / long co-ordinates:

```
def Lat_Long(raw_gps):
        gps_array = []
        gps_array = (raw_gps.split(','))

        if gps_array[1] == '':
                gps_array[1] = "4791.75429"
                gps_array[2] = "N"
        if gps_array[3] == '':
                gps_array[3] = "01208.62562"
                gps_array[4] = "E"

        lat_raw = gps_array[1]
        long_raw = gps_array[3]

        lat_dd = lat_raw[0:2]
        long_dd = long_raw[0:3]
        lat_mm = lat_raw[2:]
        long_mm = long_raw[3:]
        lat_dd = int(lat_dd)
        lat_mm = float(lat_mm)
        lat = (lat_mm / 60) + lat_dd
        lat = round(lat,3)

        long_dd = int(long_dd)
        long_mm = float(long_mm)
        long = (long_mm / 60) + long_dd
        long = round(long,3)

        if gps_array[2] == "S":
                lat = '-' + str(lat)
        else:
                lat = str(lat)

        if gps_array[4] == "W":
                long = '-' + str(long)
        else:
                long = str(long)
        return [lat,long]
```

This module creates a TCP Socket to the WR44 itself and collect the GPS info:

```
def getSocket_Data(gpsHOST, gpsPORT):

  data = 'None'

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((gpsHOST, gpsPORT))
        print 'Getting current GPS co-ordinates'
        data = s.recv(50)
        return data

  except:

        print 'Error - Cannot Connect to ', gpsHOST
        return data
```

This module create a TCP Socket to the web server and sends the Data:

```
def sendSocket_Data(HOST, PORT, data):

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
        print 'Successfully sent new status and GPS to', HOST
        s.send (data)

  except:
        print 'Error - sending new data to', HOST
```

This module issues a gpio command to the Transport command line interface to collect the status of the Digital Inputs:

```
def Get_Cli():
        clidata = ""
        cli = sarcli.open()
        cli.write("gpio")
        while True:
                tmpdata = cli.read(-1)
                if not tmpdata:
                        break
                clidata += tmpdata
        cli.close()
        return clidata
```

Checks the individual status of the gpio line:

```
def gpio_check(str, pin):
        if (pin + " : OFF") in str:
                return "off"

        return "on"
```

Writes the individual status of the 3 gpio lines into the variables:

```
def gpio(gpio_str):

        idx = gpio_str.find("Output:")

        input  = gpio_str[:idx-1]
        output = gpio_str[idx:]

        in_status     = gpio_check(input, "in")
        io_in_status  = gpio_check(input, "inout")
        io_out_status = gpio_check(output, "inout")

        return in_status, io_in_status, io_out_status
```

Define constants and Arrays

```
gpsHOST = '127.0.0.1'     # The remote host
gpsPORT = 2000            # The same port as used by the server
latlong = []              # Lat Long array
oldlatlong = []           # the old lat and long array
iostatus = []             # io status array
panic_status = "off"      # set the initial panic status to off
ign_status = "off"        # set the initial alarm status to off
```

```
HOST = 'gromit.mobilemonkey.co.uk'    # The remote web server hostname
PORT = 9999              # This is the port the web server is listening on


oldlatlong.append(50.000000003)
oldlatlong.append(-1.990900000)
```

The main program loop

```
            # Main
print "To stop, type \"Python kill\""

while True:
        clidata = Get_Cli()
        iostatus = gpio(clidata)
        changed = False
        if iostatus[0] == "off":
                if panic_status == "off":
                        panic_status = "on"
                        changed = True
        else:
                if panic_status == "on":
                        panic_status = "off"
                        changed = True

        if iostatus [1] == "on":
                if ign_status == "off":
                        ign_status = "on"
                        changed = True
        else:
                if ign_status == "on":
                        ign_status = "off"
                        changed = True



        rawgps = getSocket_Data(gpsHOST, gpsPORT)
#       print 'new data line' , rawgps
        latlong = Lat_Long(rawgps)
        if latlong[0] != oldlatlong[0]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[0] = latlong[0]
        if latlong[1] != oldlatlong[1]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[1] = latlong[1]
        if changed == True:
                status = panic_status + ',' + ign_status
                data = status + ','+ str(latlong[0]) + ',' + str(latlong[1])
                print data
                sendSocket_Data(HOST, PORT, data)
        time.sleep(1)
```

Timed event, Python script

This script waits until a specific time of the day and then completes a task. The script has been separated into sections for ease of explanation however you can download the complete script here time-evn.py

```
import sarcli
import time

#Modules
#The module issues commands to the command line:

def cli(command):
           cli =  sarcli.open()
           cli.write(command)
           cli.close()


#Define constants and Arrays

event_time = "13:22"
running = True


#The main program loop

while running:

        # gets the current time ie "13:01"
        current_time = time.strftime ("%H:%M", time.localtime())

        if current_time == event_time:

                #  It is time to do something..

                print " I'm Doing something "
                command = 'setevent "time_evn.py: Its time to do something"'
                cli(command)

                time.sleep(61) # Sleep for 61 seconds so that we do not do the
event again.

        else:
                print current_time
                time.sleep(50) # Its not time go to sleep for 50 seconds.
```

Below is a telemetry card control Python script

Below is the complete script for controlling the telemetry card, this script waits for an SMS message which contains either a "Camera on" or "Camera off". After receiving the command it will process it changing the state of the relay on the Telemetry board and the reply back to the sender with a "camera now on /off" message. Download the complete script here

Note This script is offered as an example and its reliability can not be guaranteed, the router must have Firmware version 5100 or later to run. smsctrl.py

```
            #The required libraries

import threading
import sarcli
import os
import sys
import time
```

```
'''
Threads
The first thread runs continuously and checks the eventlog for 'SMS Received:'
Messages:
'''

class eventlog (threading.Thread):
    def run (self):

        # Constants
        running = True
          # This is what we are looking for in the eventlog.txt
        string_match = "SMS Received:"
        filename = "eventlog.txt"

        while running:
          # Try to open the eventlog.txt file
          try:
            file = open(filename, 'r')

          except:
           # Output to the eventlog if there is a problem opening the
eventlog.txt
           cli =  sarcli.open()
           cli.write('setevent "smsctrl:error opening file"')
           cli.close()

          # Check the eventlog.txt file for the string_match value.
          for line in file:
                if string_match in line:
                  line = line.strip('\r\n')
                  command_str = "basic 0 nv " + '"' + line + '"'
                  cli = sarcli.open()
                  cli.write(command_str)
                  cli.close()
                  break

        file.close()
        time.sleep(10)

# This is thread that turns on the Relay waits 30secs then turns it off:

class DelayOnOff (threading.Thread):
    def run (self):

        answer = SetRelay("on")
        ReplySms(answer, cmd_array[1])
        # Wait 30 seconds
        time.sleep(30)
        answer = SetRelay("off")
        ReplySms(answer, cmd_array[1])

#This module takes the eventlog item and separates the command and phone number:

def GetEvent(output_string):
  event_array = []
  command_array = []
```

```
  event_array = (output_string.split(','))
  command= str(event_array[2])
  command_array=(command.split(':'))
  command_array[1] = str(command_array[1]).lstrip()
  command_array[2] = str(command_array[2]).lstrip()
  return command_array

#This module alters the State of the Relay on the Telemetry board and returns the
relays status:

def SetRelay(state):
  clir = sarcli.open()
  if state == "on":
        try:
           clir.write("anaconda -y 1")
           clir.write('setevent "Smsctrl:Camera now on"')
           answer= "on"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"

  elif state == "off":
        try:
           clir.write("anaconda -y 0")
           clir.write('setevent "Smsctrl:Camera now off"')
           answer= "off"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"
  clir.close()
  return answer


#This module sends back an SMS reply to the originators phone number with the
status:

def ReplySms(answer, phonenum):
  clir = sarcli.open()
  if answer == "on":
            sms = 'sendsms ' + phonenum + ' "Camera now on"'

  elif answer == "off":
            sms = 'sendsms ' + phonenum + ' "Camera now off"'
  else:
            sms = 'sendsms ' + phonenum + ' " Error setting relay"'

  try:
    clir.write(sms)
  except:
    clir.write('setevent "smsctrl:error sending sms"')
    clir.close()


#Define constants and Arrays

# Constants
running = True

#Variables
```

```
completed_command = "null"
cmd_array = []

#The main program loop


# Main
print "To stop, type \"Python kill\""

# Start eventlog Thread
eventlog().start()

while running:

        cli = sarcli.open()
        try:
           cli.write("basic 0 nv")
           tmpdata = cli.read(-1)
        except:
          print "error writing command: ", tmpdata
          errormsg = 'setevent ' + '"' + tmpdata + '"'
          cli.write("errormsg")
        cli.close()
        if completed_command != tmpdata:
                print "new event: ", tmpdata
                cmd_array = GetEvent(tmpdata)
                if cmd_array[2] == "Camera on":
                        answer = SetRelay("on")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera off":
                        answer = SetRelay("off")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera on 30":
                        DelayOnOff().start()
                        completed_command = tmpdata
                else:
                        time.sleep(5)
        else:
                time.sleep(5)
```

Wake on Lan, Python script

This script sends out a Wake on Lan packet to a specific host. The script has been separated into sections for ease of explanation however you can download the complete script here wol.py.

```
            #The required libraries

import socket
import struct

#Modules
#The module sends out Wake On LAN packets

def wake_on_lan(macaddress):

    """ Switches on remote computers using WOL. """
```

```
    # Check macaddress format and try to compensate.

    if len(macaddress) == 12:
        pass

    elif len(macaddress) == 12 + 5:

        sep = macaddress[2]
        macaddress = macaddress.replace(sep, '')

   else:
        raise ValueError('Incorrect MAC address format')

    # Pad the synchronization stream.
    data = ''.join(['FFFFFFFFFFFF', macaddress * 20])
    send_data = ''

    # Split up the hex values and pack.

    for i in range(0, len(data), 2):

        send_data = ''.join([send_data,

                             struct.pack('B', int(data[i: i + 2], 16))])

    # Broadcast it to the LAN.

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    sock.sendto(send_data, ('192.168.1.255', 9))

#The main program loop

if __name__ == '__main__':

    # Use macaddresses with any seperators.
    wake_on_lan('00:40:63:FC:0C:75')
    wake_on_lan('00-40-63-FC-0C-75')

    # or without any seperators.
    wake_on_lan('004063FC0C75')
```

Supported Python modules

```
    A
        abc
        aifc
        anydbm
        array
        ast
        asynchat
        asyncore
        atexit
        audiodev
    B
        BaseHTTPServer
        Bastion
```

```
    base64
    bdb
    binascii
    binhex
    bisect
C
    CGIHTTPServer
    ConfigParser
    Cookie
    cPickle
    cProfile
    cStringIO
    calendar
    cgi
    cgitb
    chunk
    cmath
    cmd
    code
    codecs
    codeop
    collections
    colorsys
    commands
    compileall
    contextlib
    cookielib
    copy
    copy_reg
    csv
D
    DocXMLRPCServer
    datetime
    dbhash
    decimal
    difflib
    digihw
    digiwdog
    digiweb
    dircache
    dis
    doctest
    dumbdbm
    dummy_thread
    dummy_threading
E
    email
    encodings
    errno
    exceptions
F
    filecmp
    fileinput
    fnmatch
    formatter
    fpformat
    fractions
    ftplib
    functools
```

```
    future_builtins
G
    gc
    genericpath
    getopt
    getpass
    gettext
    glob
    gzip
H
    HTMLParser
    hashlib
    heapq
    hmac
    htmlentitydefs
    htmllib
    httplib
I
    ihooks
    imaplib
    imghdr
    imp
    imputil
    inspect
    io
    itertools
K
    keyword
L
    linecache
    locale
    logging
M
    MimeWriter
    macpath
    macurl2path
    mailbox
    mailcap
    markupbase
    marshal
    math
    md5
    mhlib
    mime
    mimetools
    mimetypes
    mimify
    modulefinder
    multifile
    mutex
N
    netrc
    new
    nntplib
    ntpath
    nturl2path
    numbers
O
    opcode
```

```
    operator
    optparse
    os
    os2emxpath
P
    parser
    pdb
    pickle
    pickletools
    pipes
    pkgutil
    platform
    plistlib
    popen2
    poplib
    posix
    posixfile
    posixpath
    pprint
    profile
    pstats
    pty
    py_compile
    pyclbr
    pydoc
    pydoc_topics
    pyexpat
Q
    Queue
    quopri
R
    random
    re
    repr
    rexec
    rfc822
    rlcompleter
    robotparser
    runpy
S
    SimpleHTTPServer
    SimpleXMLRPCServer
    SocketServer
    StringIO
    sarcli
    sarutils
    sched
    select
    sets
    sgmllib
    sha
    shelve
    shlex
    shutil
    site
    smtpd
    smtplib
    sndhdr
    socket
```

```
      sre
      sre_compile
      sre_constants
      sre_parse
      ssl
      stat
      statvfs
      string
      stringold
      stringprep
      strop
      struct
      subprocess
      sunau
      sunaudio
      symbol
      symtable
      sys
 T
      tabnanny
      tarfile
      telnetlib
      tempfile
      termios
      textwrap
      this
      thread
      threading
      time
      timeit
      toaiff
      token
      tokenize
      trace
      traceback
      tty
      types
 U
      UserDict
      UserList
      UserString
      unittest
      urllib
      urllib2
      urlparse
      user
      uu
      uuid
  W
      warnings
      wave
      weakref
      webbrowser
      whichdb
      X
      xdrlib
      xmllib
      xmlrpclib
  Z
```

```
        zipfile
        zipimport
        zlib
```

References

Portions of this document are reproduced from documents by Matt jameson & Jon Lyons: extranet.jrp2.com/~jpowell/gromit.mobilemonkey.co.uk/trans-Python.html

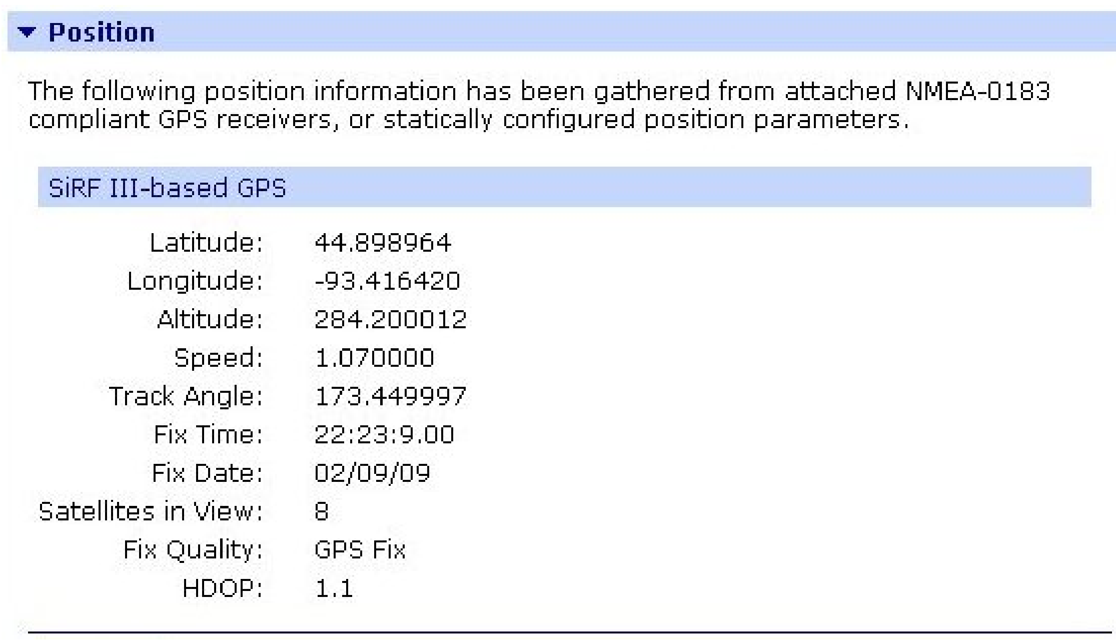
Virtual GPS NMEA Access

Virtual GPS Access

Many Digi products supporting Python include a GPS service which you open as '/gps/0', treating it like a read-only serial port. You will NOT need to set port settings like baud rate because you are not really talking to the hardware. Only one Python script can open the port at a time - it cannot be shared. If there are no valid GPS hardware(s) present, then port '/gps/0' still exists but no data will be returned. If port '/gps/0' does not exist, then your firmware and/or hardware does not have the GPS service available.

The NMEA sentence data returned from '/gps/0' might be from any valid GPS source - including one built into a cellular modem or an external USB GPS such as the USGlobalSat BU-353. Be warned that the cellular GPS is NOT free - you will need at least an external GPS antenna to obtain meaningful data, plus some cellular systems require a minimum level of cell service to exist before the GPS data can be accessed. The external USB GPS model BU-353 sells online for between $35 and $40 in quantity one.

If a GPS data source exists, then the Digi product automatically shows you the GPS Position information on the product's web pages under **Administration -> System Information -> Position**. Here is a screen shot from a Digi ConnectPort X4 with the BU-353 connected (and no, my desk is NOT moving at 1.07 knots - that is normal analog 'noise' in GPS systems):



Digi Product List

These products have the 'gps/0' port support for cell modem, serial or USB:

Model	Firmware	Cellular	USB	Serial	Comments

Model	Firmware	Cellular	USB	Serial	Comments
ConnectPort X4	82001536D or newer	No	Yes	Yes	
ConnectPort X8	82001115E or newer	Possible	Yes	Yes	For internal cell modem support, unit must have GPS antenna connected to secondary port
ConnectPort WAN VPN	82001276J or newer	Possible	Yes	Yes	For internal cell modem support, unit must have GPS antenna connected to secondary port
ConnectPort GPS	All Versions	Possible	Yes	Yes	Has separate GPS hardware built in with dedicated SMC connector for GPS antenna
Connect WAN - 16MB Memory	82001660_B or newer	No	No	Yes	
Connect WAN IA - 16MB Memory	82001661_B or newer	No	No	Yes	
Connect WAN VPN - 16MB Memory	82001662_B or newer	No	No	Yes	
Connect WAN 3G	82001532_D or newer	No	Yes	Yes	
Connect WAN 3G IA	82001912_A or newer	No	Yes	Yes	

Cellular Notes: 'Possible' means some (but not all) cellular modules have GPS option and separate antenna is possible; 'No' means even if cellular module has GPS, lack of second antenna connector means signal quality will be poor and unreliable.

USB Notes: requires a compatible USB product which the Digi product can install as a virtual serial port; not all USB GPS will work

Serial Notes: most RS-232 products following NMEA guidelines should work; you MUST manually set the appropriate Serial Port Profile to the "GPS" setting and make sure the basic settings are 4800 baud, 8 data bits, no parity, 1 stop bit, and no flow control

These products do NOT support GPS via the '/gps/0' port, however Python could support directly by RS-232 serial.

Digi Connect TS1 to TS16 (serial only, requires full Python management; USB units NOT supported)

External GPS Products Known to Work

Note Digi is not affiliated with any of these suppliers, but these products are known to work.

USGlobalSat BU-353 USB GPS Navigation Receiver

Using NMEA Sentences

If you want a simple way to understand "Where am I? What time is it? How fast am I moving?", then you only need to understand the single RMC sentence, which stand for "Recommended Minimum Navigation Information" and looks like this "$GPRMC,154516.000,A,4453.8294,N,09324.9550,W,0.35,189.45,070209,,*1C".

- "154516" and "070209" is the time and date, so it is "07-Feb-2009 at 15:46:16" (note that GPS time was synchronized with UTC back in 1980's and is now about 15 seconds 'slow')
- "4453.8294,N" is the latitude and "09324.9550,W" is the longitude.
- "0.35" is the speed in knots, where 1 knot = 1.15077945 mile/hour or 1.85200 km/hour

Up to a dozen "sentences" or lines of data are dumped out per second, so **overall traffic could be well above 200MB per month**. You do not want to just forward all traffic over a cost-sensitive network like cellular.

While the NMEA 0183 standard document is sold by the US-based National Marine Electronics Association, many web sites world wide summarize the "sentences".

See Also

- GPS Data UDP Forwarder
- http://www.tronico.fi/OH6NT/docs/NMEA0183.pdf

Voltage readings in XBee module

Understanding the Voltage Setting within the XBee Modules

Supply Voltage (%V)

This measures the Vcc/power pin to the XBee, which equates the voltage being output from your hardware voltage regulator. This is not directly related to any external battery or supply voltage.

Reading the Supply Voltage Directly

Read the Vcc/power pin level directly with the AT command V%. The actual interpretation of the 16-bit value returned may vary based upon the technology, but for example in the XB24-ZB a reading of 0x0900 (2304) represents 2700mV. The two formulas used in the gateway are:

- set_value = ((1024 * user_mv_desired) + 600) / 1200
- read_mv = ((1200 * get_value) + 512) / 1024

(The +600/+512 handles the truncation due to integer division.)

Including the Supply Voltage in periodic I/O sampling

To avoid the need to manually poll the V% value, use the AT command V+ to selectively have the Supply Voltage included in the periodic I/O Sampling (the IS packet). Bit 7 (0x80) of the IS Packet's Analog Channel Mask is set to 1 when the Supply Voltage has been appended. It is appended with the following logic:

- Setting V+ to a big number (like 9999) forces the Supply Voltage to always be appended.
- Setting V+ to zero prevents the Supply Voltage from ever being appended.
- Setting V+ to a reasonable value near V% (Vcc) will append the Supply Voltage only when V% is less than V+, meaning you do not consume two extra bytes worth of transmitter power until the Supply Voltage starts to fall below normal.

Battery Voltage

Nothing prevents a user-designed XBee sensor product from including an actual battery voltage level in a protocol response. It could be tied to one of the XBee module's four analog input signals (AD0-AD3). However, this will increase battery drain during sleeping.

Most Digi Adapter products do not provide access to the battery voltage.

Low Battery Signal (Digi Adapter Specific)

Certain Digi Adapter products include a digital low-battery signal read directly from the low-loss battery-friendly voltage regulator used. These include (but may not be limited to):

- XBee Sensor Adapter (the L/T/ and L/T/H/ adapters)
- XBee 232 and 485 Adapters
- XBee AIO and DIO Adapters

Note An important limitation: this bit only shows low-battery on internal batteries. Externally suppling 9-30vdc or 3-6vdc causes this bit to be low (in alarm) or undefined.

In these products, and ERROR FLAG output is tied to the DIO11 (P1) input (or XBee external module pin 7). Configure this XBee pin to be Digital Input to enable, and it is returned within the I/O Sampling digital channel mask as bit 11 or 0x0800. It is 1 (high) when the voltage regulator is able to maintain the specified output voltage. When the output voltage drops below spec by approximately 5% (due to low input voltage or excessive current draw) then this bit is 0 (low).

The psudeo-code handling this bit could be viewed as:

```
if digital_channel_mask AND 0x0800
  then print 'power supply is within spec'
  else print 'power supply is too low, is under-voltage condition'
```

XBee Hardware Codes

What XBee Firmware to Flash into the X2/X4?

Open the gateway's web interface, then look on the page Xbee Network > Gateway Firmware Update. Towards the bottom you will see the Radio Type enumerated. Assuming that you plan to load Zigbee API Coordinator firmware numbered 2xA7, then:

- **Radio Type: XBee-PRO S2B ZB / SE** means upgrade with file **XBP24-ZB_21A7_S2B.EBL**
- **Radio Type: XBee-PRO ZNet 2.5 / ZB / SE** means upgrade with file **XBP24-ZB_21A7.EBL**
- **Radio Type: XBee ZNet 2.5 / ZB / SE** means upgrade with file **XB24-ZB_21A7.EBL**

XBee Hardware (HV) Codes

There is no official public list of XBee Hardware (HV) codes, and the system used to assign the codes makes it impossible to parse an unknown HV code into the hardware type. Therefore, this list enumerates some of the known codes.

Note: The antenna type is not included in the HV code, and is listed for information only.

HV	Product Code	Date Code	Familiar Name	Antenna	XBee Firmware Prefix
1742	XB24-AWI-001-revB	2009-07-22	XBEE 802.15.4 S1	Whip	XB24-15_4_1xxx
1941	XB24-BWIT-004	2007-11-08-07	ZNet/ZB S2	Whip	XB24-ZB_2xxx
1943	XB24-BWIT-508-revC1	2009-09-25	ZigBee S2	Whip	XB24-ZB_2xxx
1945	XB24-Z7WIT-507-revE1	2009-02-25	ZigBee S2	Whip	XB24-ZB_2xxx
1947	XB24-Z7CIT-004-revG	093010-20:41	ZigBee S2	Chip	XB24-ZB_2xxx
194A	XB24-Z7CIT-004-revJ	121211-05:57	ZigBee S2	Chip	XB24-ZB_2xxx
1A43	XBP24-Z7WIT-505-revC	2009-06-16	ZigBee S2 PRO	Whip	XBP24-ZB_2xxx
1E41	XBP24BZ7SIT-500-revA	080510-05:38	ZigBee S2B PRO	RPSMA	XBP24-ZB_2xxx_S2B
1E43	XBP24BZ7SIT-004-revC	081711-14:09	ZigBee S2B PRO	RPSMA	XBP24-ZB_2xxx_S2B

XBee L-T-H sensor adapter

This information has been moved. See XBEE sensors.

XBee Product Codes

Digi Product Types

Adapter or boxed Xbee products purchased from Digi will come correctly configured and be displayed within the web interface and telnet node lists with a type string - such as "RS-232 Adapter" or "X4 gateway". The XBee module's "DD" parameter encodes this knowledge as a 32-bit value.

Yet if you replace the supplied Xbee module with a new one - for example to swap a higher power XBP24-B module for the supplied XB24-B module, then the device may show up as "Unspecified Device" instead. Some XBee firmware - such as the special purpose Digital or Analog I/O versions - force the "DD" value to the correct codes, while others - such as the generic AT or API firmware usable in RS-232 or RS-485 adapters will leave set DD set to zero and be shown as "Unspecified".

It is important that the "DD" value be correct, as many gateway and Python functions rely upon the "DD" value to understand how to use the device.

Python access to DD Parameter

Your Python program can read or write the DD or Digi Device Type with the ddo_get_param() and ddo_set_param() call. If the ddo_get_param() is successful, then 'result' will be a 4-byte binary string in form AA BB CC DD, where Module Device Type is 0xAABB and Product Type is 0xCCDD:

```
import zigbee

result = zigbee.ddo_get_param( "[00:13:a2:00:40:0a:07:8d]!", 'DD')
```

Forcing New DD from a Digi Gateway

You can preload the DD value correctly from XCTU when you load the XBee module firmware

You can also use telnet (or the command-line) access to a Digi X2/X4/X8 gateway to write new DD values to an active device. In the example below, the four "Unspecified" devices are RS-232 Adapters which had new XBee modules installed. The set xbee command is shown used to set new DD values, plus the WR command must be added to save the new DD value to NVRAM. The 0x03 in the upper word defines these correctly as Zigbee 2007, and the 0x0005 in the lower word defines them as RS-232 adapters.

Since the gateway does NOT expect the DD value of devices to change dynamically, you may need to reboot either the affected XBee nodes or the gateway to have the new device information show up. **Of course, setting the incorrect DD values will confuse and perhaps disable mesh applications which base functional and I/O expectations on the DD value!**

```
#> disp xbee

XBee network device list

PAN ID:            0x3261
Channel:           0x0d (2415 MHz)
Gateway address:   00:13:a2:00:40:4b:87:c7!

Node ID      Network Extended address         Product type
------------ ------- ------------------------ ------------
COORDINATOR
[0000]! 00:13:a2:00:40:4b:87:c7! X4 Gateway

ROUTERS
```

```
AN05         [5aea]! 00:13:a2:00:40:52:94:8b! Analog IO Adapter
AN06         [0c06]! 00:13:a2:00:40:52:94:b8! Analog IO Adapter
AN17         [7ad6]! 00:13:a2:00:40:52:94:ac! Analog IO Adapter
AN26         [0480]! 00:13:a2:00:40:52:94:ad! Analog IO Adapter
ANNA_ZN      [05e8]! 00:13:a2:00:40:52:94:a0! Unspecified
BELA_ZN      [794a]! 00:13:a2:00:40:34:16:20! Unspecified
CALI_ZN      [3c39]! 00:13:a2:00:40:34:16:4e! Unspecified
DEBI_ZN      [3921]! 00:13:a2:00:40:52:94:b2! Unspecified

#> set xbee address=00:13:a2:00:40:52:94:b2! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:34:16:20! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:52:94:a0! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:34:16:4e! DD=0x30005 WR

(Then after some time delay ... the types will refresh and show correctly)

#> disp xbee
...
ANNA_ZN      [05e8]! 00:13:a2:00:40:52:94:a0! RS-232 Adapter
DEBI_ZN      [3921]! 00:13:a2:00:40:52:94:b2! RS-232 Adapter
CALI_ZN      [3c39]! 00:13:a2:00:40:34:16:4e! RS-232 Adapter
BELA_ZN      [794a]! 00:13:a2:00:40:34:16:20! RS-232 Adapter
...
```

Note that the telnet "DD=" command assumes decimal unless the "0x" prefix is added. So setting "DD=30005" will instead set the DD value to 0x00007535. You need to use "DD=0x30005"

DD upper word : Module Type

16-bit	Description	FW / HW Mnemonics
0x0000	Unspecified	
0x0001	XBee 802.15.4 (Series 1)	XB24 (or XB24-A)
0x0002	XBee ZNet 2.5	XB24-B
0x0003	XBee ZB (Zigbee 2007)	XB24-ZB (or XB24-Z7)
0x0004	XBee DigiMesh 900 (900MHz)	XB09-DM
0x0005	XBee DigiMesh 2.4 (2.4GHz)	XB24-DM
0x0006	XBee 868 point to multi-point (868MHz for EU market)	XB08-DP
0x0007	XBee Point to Multi-point 900Mhz	XB09-DP
0x0008	XTend DigiMesh 900Mhz	
0x0009	XBee 802.11 Wifi	
0x000A	XBee ZB on S2C (surface mount)	
0x000B	XBee DigiMesh 900 S3B	
0x000C	XBee DigiMesh 868	

The PRO or higher wattage XBee modules use XBP instead of XB in firmware and hardware mnemonics. However the upper DD word is used to learn the general class of API command functionality within the firmware in a module, *thus do NOT attempt to assign hardware meaning to these values!*

Example code to convert these values to strings:

```
dd_upper_names = ['Bad_Code','ZB24-A','ZB24-B','XB24-ZB','XB09-DM','XB24-
DM','XB08-DP','XB09-DP']
def get_dd_upper_code_string( code):
    """Given upper/module-type word of the DD response as 16-bit integer, return
string"""
    try:
        return( dd_upper_names[code])
    except:
        return( dd_upper_names[0])
```

Important feature/limitation in the DD upper word

While some XBee technology force this value to be as expected, others allow the user to redefine the meaning. Thus you can never fully trust the DD value returned. For example, an OEM who produced products using ZNet 2.5 might load a value such as 0x00021234 into their product. After they start creating Zigbee 2007 or DigiMesh 2.4Mhz models, the SHOULD change the DD value to be 0x00031234 and 0x00051234 respectively - but they might not. ***Therefore it is safest to have your Python code use the upper-word of the gateway XBee to determine 'mesh/xbee' type, and only read the DD lower word of attached devices.***

DD lower word : Digi Product Type

0x0000	Unspecified
0x0001	ConnectPort X8 Gateway
0x0002	ConnectPort X4 Gateway
0x0003	ConnectPort X2 Gateway
0x0004	XBee Commissioning Tool
0x0005	XBee RS-232 Adapter
0x0006	XBee RS-485 Adapter
0x0007	XBee Sensor (1-wire) Adapter
0x0008	XBee Wall Router
0x0009	XBee RS-232PH (Power Harvesting) Adapter
0x000A	XBee Digital IO Adapter
0x000B	XBee Analog IO Adapter
0x000C	X-Stick
0x000D	XBee Sensor /L/T/H Adapter
0x000E	XBee Sensor /L/T Adapter

0x000F	Smart Plug
0x0010	USB Dongle
0x0011	LCD Display
0x0013	ConnectPort X5 Gateway
0x0014	Embedded Gateway
0x0015	ConnectPort X3 Gateway
0x0016	Net OS Device
0x0017	XG3 Gateway
0x0018	LTS Gateway
0x0019	CC3G Gateway
0x001A	X2 ULC Gateway
0xFF00-0xFFFF	Available for private customer use

Example code to convert these values to strings:

```
dd_lower_names = ['Unspecified','X8','X4','X2','XBee Commissioning
Tool','XBee232','XBee485',
    'XBee1W','XBee Wall Router', 'XBee232PH','XBeeDIO','XBeeAIO','X-Stick',
    'XBee /L/T/H','XBee /L/T','SmartPlug','USB Dongle','LCD Display','Undefined',
    'X5','Embbeded GWay','X3','NetOS',
          ]
def get_dd_lower_code_string( code):
    """Given lower/product word of the DD response as 16-bit integer, return
string"""
    try:
        return( dd_lower_names[code])
    except:
        return( dd_lower_names[0])
```

Currently assigned 'Private Customer Use' code

Note Although these are official, Digi is not rigidly enforcing use. Thus you may encounter other XBee users reusing these codes for other products.

Rabbit-Brand Products

0x0100	Generic Rabbit-Brand Product
0x0101	RCM4510W
0x0102	BL4S1xx
0x0103	BL4S230
0x01F0-0x01FF	Rabbit End-Customer Use

Non-Digi End Products

0x0201	Massa M3
0x0210	B&B Electronics LDVDS-XB
0x0220	SSI Embedded Systems PSU Sensor M
0x0221	SSI Embedded Systems PSU Sensor O
0x0231	PointSix Temperature Sensor
0x0240	Henny Penny 485 Adapter with PXBee
0x02B0-BF	Bejouled Solar Inverters
0x02C0-CF	Windpower
0x02D0-DF	RobustMesh, RS-232/485

If you have a device utilizing an XBee radio and desire a registered public DD value, please contact Digi support.

XBee RS-232 adapter

Product description

The XBee RS-232 Adapter provides short range wireless connectivity to any RS-232 serial device.

Note that this Wiki page documents the packaged box product, and not the RS-232 development bare board! It also covers operation when powered full-time, not when batteries are used and the product sleeps.

Assuming the serial end device is a traditional 'slave/server' which answers remote polls and is unaware of the mesh, then load the standard "AT" firmware and not the API firmware. Then use a Digi ConnectPort X gateway or another XBee module with the API firmware loaded to act as the 'master/client' and issue requests directly to end devices via MAC address, and the 'slave/server' XBee will always return responses to the MAC address of the 'master/client'.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

XBee Module Support

All Xbee modules should work in the RS-232 adapter, and they would use the standard AT or API versions. There are not (at this time) any special builds for this adapter. However, X-CTU cannot reflash firmware in this RS-232 module - you will need to move the XBee module to a USB or RS-232 development board to reflash firmware.

Module	Description	Tested	Comments	Output Defaults
XB24-A	802.15.4 on 2.4Ghz	Yes	No access to DTR or DSR	DTR and RTS asserted/high
XB24-B	ZNet 2.5 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB09-DM	DigiMesh on 900Mhz	Pending	No access to DSR	DTR is disabled/low, RTS asserted/high

Module	Description	Tested	Comments	Output Defaults
XB24-DM	DigiMesh on 2.4Ghz	Pending	No access to DTR or DSR	
XB08-DP	Point-to-Multipoint on 868Mhz	Pending		DTR is disabled/low, RTS asserted/high

Configuration to consider

These values can be set from X-CTU - or use the CLI or Web UI of a Digi ConnectPort X gateway:

- Set dest_addr (DH/DL) to the MAC of your CPX gateway or the XBee module which asts as master/client. This prevents broadcast responses.
- Set the baud rate and other port characteristics (requires X-CTU?).
- Set dio12_config (P2) to either 4 or 5; 4 (DO Low) causes DTR to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio7_config (D7) to either 4 or 5; 4 (DO Low) causes RTS to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio6_config (D6) to 3 to enable CTS as an input.
- Set dio3_config (D3) to 3 to enable CD as an input.
- Set dio1_config (D1) to 3 to enable RI as an input.

Note DSR cannot be read; there is no configuration required for it.

Pinouts

The RS-232 connector is an industry-standard DB9 male connector with a DTE configuration, similar to a PC serial port. However, there are limitations in the support for DTR/DSR. Pinouts for the connector are:

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
1	CD	Input	17	DIO3	D3	0x0008		
2	RXD	Input	3					
3	TXD	Output	2					
4	DTR	Output	4	DIO12	P2 (See Note)	0x1000	'P2\x04'	'P2\x05'
5	GND	---						
6	DSR	Input	9	DI8	Not Readable	Not Readable		
7	RTS	Output	12	DIO7	D7	0x0080	'D7\x04'	'D7\x05'

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
8	CTS	Input	16	DIO6	D6	0x0040		
9	RI	Input	19	DIO1	D1	0x0002		
9-alt	12vdc Switched Power	Output	18	DIO2	D2 (See Note)	0x0004	'D2\x05'	'D2\x04'

Note on Pin 4/DTR. Some XBee modules do not offer access to raise or lower DTR; the "P2" command is not available.

- XBee modules known **NOT** to support P2/DTR control: **XB24-A** (802.15.4), **XB24-DM**
- XBee modules known to **support** P2/DTR control: **XB24-B** (Znet2.5), **XB24-ZB**, **XB09-DM**

Note on Pin 6/DSR. None of the XBee module allow reading the level of DSR via the "IS" command.

Note on Pin 9. By default it is an input, however setting DIO2 high turns the 12vdc 50mA switched power output on and reading RI/DIO1 returns TRUE if power is on. Setting DIO2 low sets the switched power to tri-state, thus reading RI/DIO1 returns the RI-like status of pin 9. So do NOT connect RI to any device which might be damaged by a +12vdc signal - while it will NOT damage any true EIA/RS-232 compliant device, it can't be good for any device attempting to drive RI low.

Powering 'green' or 'port-powered' RS-232 devices

Some external devices (such as RFID readers or short-haul modems) attempt to draw power from the RS-232 driver circuit. The voltage output from the XBee 232 Adapter may be too low to power such external devices.

However, cross-wiring the RS-232 cable so that the 12vdc Aux-Power (pin 9) from the XBee 232 Adapter connects to the external DTE DSR input (pin 6 of DB9) or the DCE DTR input (pin 4 of DB9) provide a solid solution. A +12vdc signal is well within the RS-232 voltage signal specification, plus the approximately 12vdc 50mA supplied is more power than the 'Port Powered' device expects to tap.

Python programming examples

Polls or Requests sent to field devices: The 'master/client' XBee module should send serial data via addressed unicast with one of the API Transmit Request frames, such as 0x00, 0x01 and 0x10.

Responses or unsolicited data from field devices: If the dest_addr (DH/DL) registers have been set properly, then any serial data received from the field devices will be forwarded to the central 'master/client' XBee module. API Receive Packet frames will be received by the 'master/client' XBee module.

Driving RS-232 control signals

DTR/RTS signals are raised or lowered by sending a 3-byte command with the API Remote AT (command 0x17) - the examples below are coded as Python expects:

- To assert (or raise) the outgoing DTE signal DTR, send the AT command 'P2\x04'
- To deassert (or drop) the outgoing DTE signal DTR, send the AT command 'P2\x05'
- To assert (or raise) the outgoing DTE signal RTS, send the AT command 'D7\x04'
- To deassert (or drop) the outgoing DTE signal RTS, send the AT command 'D7\x05'

Note that these commands affect the pin within one second, yet do not save the state in FLASH. Thus a reboot of the XBee adapter puts the DTR or RTS signal back into the configured default; the factory default is low/not asserted. If it is desired to have DTR or RTS asserted upon power-up, manually set the DIO12/DIO7 parameters to 4.

Why does output DO = Low assert the RS-232 signal and DO = High drop the RS-232 signal? This is how historically TTL communications systems worked. A 5v line was assumed idle, thus pulled weakly up to 5v and representing OFF or binary zero (0). A 0v line was being actively shunted to ground by a powered transistor, thus represented ON or binary one (1). Even today, most RS-232/485 chips assume 0v = 1/on and 5v = 0/off.

Reading RS-232 control signals

To read the signal status, issue the 'IS' command. **You must DELAY at least one (1) second after issuing any of the D2/D7/P2 commands before issuing the 'IS' command or the output status won't be correctly returned in the response**. The IS command returns 6 bytes by direct API, and 5 bytes with ddo_get_param() function, so the last 5 bytes can be decoded as:

- Response[0] should equal '\x01' (is Sample Set Count)
- Digital_mask = (ord(response[1]) * 256) + ord(response[2]); shows which bits of following data are valid
- Response[3] should equal '\x00' (is Analog Data Mask - unless adapter voltage is being read)
- Digital_data = (ord(response[4]) * 256) + ord(response[5]); shows the actual I/O states

The RS-232 signals show as inverted for historical reasons, so a '0' means the signal is HIGH/ASSERTED and '1' means LOW/DROPPED

However, the Auxiliary Power Output (pin 9) shows up per digital logic levels. Therefore, a '0' means power is off and '1' means power is on. The Auxiliary Power Output can drive 12vdc at 50mA if the XBee RS-232 adapter is direct DC powered. If battery powered, driving Auxiliary Power Output more than a second will quickly drain your batteries!

Example Python to detect CTS status

```
def show_CTS_status( digital_mask, digital_data):
    """
    Decode and display the CTS info from 'IS' response words
    Return True if asserted, else False
    """
    if( digital_mask & 0x0040):
        print 'CTS input configured, value = ',
        if (digital_data & 0x0040):
            print 'Low/Dropped'
            return False
        else:
            print 'High/Asserted'
            return True
    else:
        print 'CTS input ignored (not configured)'
    return False
```

XBEE sensors

Product description

The XBee Sensor provides real-time data on temperature, humidity, and light, with the data being transmitted through wireless communications in an XBee network infrastructure. Compact size and battery power enable XBee Sensors to be dropped into facilities easily and unobtrusively while providing reliable communications. Applications include building automation, environmental monitoring, security, asset monitoring, and more.

The readings are of modest accuracy, suitable for environmental monitor but not likely suitable for control systems (See the Accuracy Section below)

There are currently two XBee Sensor product options available:

- XBee Sensor /L/T: Integrated ambient light and temperature sensors
- XBee Sensor /L/T/H: Integrated ambient light, temperature, and humidity sensors

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Python Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

iDigi Dia configuration and programming examples

Python programming examples

Configuration settings

During system start up or configuration, you require:

- D1, D2 and D3 should be set to 2, enabling analog input to the 3 sensors.
- DH and DL should be set to your XBee coordinator (your CPX gateway) or the XBee device to receive the data productions.
- P1 (DIO11) should be set to 3, enabling digital input on the battery monitor pin.

Operational settings

Although you can poll the XBee Sensors, this requires the XBee to be awake most of the time and you battery life will be limited to a few months. You may find having it wake once per 10 or 15 minutes the best solution.

The assumed operation is to place the XBee Sensor into sleep, then have it send the data unsolicited to the XBee node listed in the DH/DL settings.

To enable sleeping operation longer than 1 minute:

- Set IR to 0xFFFF, which effectively disables the 'sample rate' setting.
- Set WH to 125 to allow the sensor hardware to stabilize for 125msec before the XBee reads the analog inputs (note: not all XBee modules support the WH parameter)
- Set the SN/SP pair to enable sleeping for the desired time period.
- SP is the number of 10msec periods to sleep
- SN is the number of SP-periods to sleep for.
- For example, SP=2000 and SP=30 leads to an approximately 600 seconds data productions (30 x 20 second periods or 10 minutes).
- *The SN/SP within your gateway and all XBee routers MUST be at least as large as the values placed into the XBee Sensor product, or you will find the routers (the parents) de-associate the sleeping XBee Sensor (the child) while it sleeps, and thus reject the periodic data productions.*

Formulas

Temperature:

```
temp_C = (mVanalog - 500.0)/ 10.0
mVanalog = (ADC2/1023.0) * 1200
```

Humidity:

```
hum = (((mVanalog * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
mVanalog = (ADC3/1023.0) * 1200
```

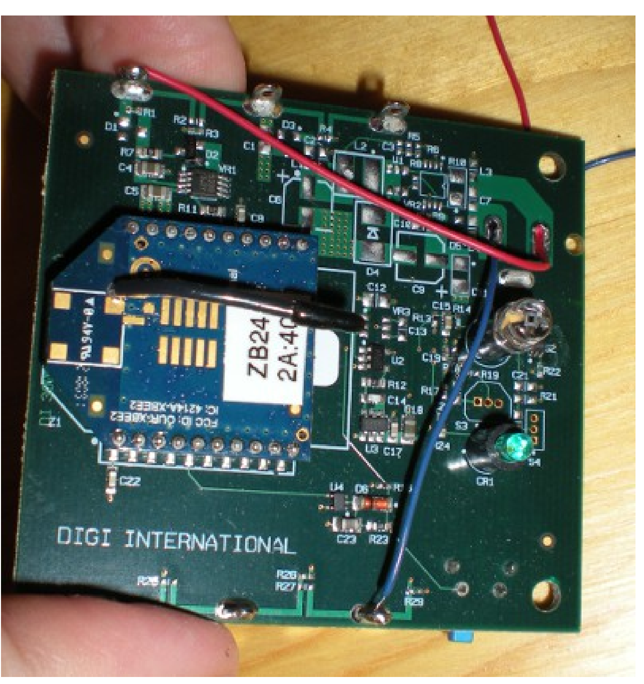
Light:

```
brightness = (ADC1) /1023.0) * 1200
```

Hardware information

Power options

Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.6v up to 6.0v. For testing purposes you can use a stable 5vdc supply, attaching your power leads to the battery tabs as shown in the photo (red is +, blue is -). The wires pass through, but do not connect to the barrel jack socket-holes. This jack is only connected electronically in the 9-30vdc version of this PCB (which is not available).



XBee module support

The XBee L/T/H/ Sensor requires either the 'End Device AT' or 'Router AT' firmware on the XBee. Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.7v up to 6.0v. For testing purposes, you can use a stable 5VDC supply.

Given the need to be low power and to sleep, not all XBee modules are appropriate for this product.

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-A	802.15.4 on 2.4Ghz	Pending				
XB24-B	ZNet 2.5 on 2.4Ghz	Yes		ZNet 2.5 Router/End Device AT, such as 1247	0x2000E	0x2000D

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes		Zigbee End Device AT, such as 2864	0x3000E	0x3000D
XB09-DM	DigiMesh on 900Mhz	Pending				
XB24-DM	DigiMesh on 2.4Ghz	Pending				
XB08-DP	Point-to-Multipoint on 868Mhz	No	Consumes too much power	N/A	N/A	
XB09-DP	Point-to-Multipoint on 900Mhz	Pending				

Notes:

Although the XBee XB24-ZB family includes a firmware named "Zigbee Router/End Device Sensor", that firmware is for the 1-wire Sensor Adapter - **it is NOT for the Sensor /L/T or /L/T/H**. Installing this firmware will result in bad values being read.

Accuracy

XBee L-T-H sensor adapter

The /L/T/H product is NOT calibrated by Digi. This means multiple units placed side-by-side out of the box will show a higher than desired variability. However, the sensor readings are fairly linear and modest software calibration can greatly improve the accuracy.

Temperature

The stated accuracy per the datasheet (and component supplier) is +/- 2 DegC (+/- 3.6 DegF).

They are not designed for industrial control, and even the temperature within a normal room varies by many degrees based on drafts, heat or cold sources, and how far the sensor is above the floor. Anyone who looks at a thermostat, reads the '72' and believes the entire room is a perfectly constant 72 degrees Fahrenheit is being foolish.

Ad-hoc tests show that a simple fixed offset added or subtracted to the readings allow them to be used in normal building automation with satisfactory result. For example, a test of four units showed that adding constants to the raw 0-1023 value allowed all four to return the same temperature to within +/- 0.25 DegC most of the time and to within +/- 0.75 DegC all of the time. The magnitude of these binary constants within this test of four units were (-4, 14, -15 and 5). You could use a float constant scaled as DegC or DegF instead.

These constants were calculated by taking 5 readings over five hours, then examining the average deviation from the desired value. For example, one unit returned the values 739, 735, 734, 700 and 702 when the expected values were 738, 732, 732, 690, and 694. Thus adding a -4 (subtracting 4) from the value received resulted in a better value. The expected value was calculated based on the temperature reading of a third party device 'trusted' as correct.

Humidity

The stated accuracy per the datasheet (and component supplier) is:

Interchangeability: +/- 5 %RH (0%RH to 59%RH) and +/- 8 %RH (60%RH to 100%RH)

Accuracy: +/- 3.5%RH

This means if you take a dozen factory-fresh units, allow them to stabilize within a 70%RH environment, then you may see readings range from 62%RH to 78%RH. This is the 'Interchangeability' clause.

However, if the user does modest linear software calibration (primarily fixed offset), then the same dozen devices can show the 70%RH as 66.5%RH to 73.5%RH. This is the 'Accuracy' clause.

Light

The light sensor in the adapter returns values from 0 to about 1100 based on light intensity - it does NOT return any standard measure, and the actual readings vary based on the opacity of the label applied and so on. It can be used to easily sense that a room is brighter or darker. For example, you can use it to turn security lights on when the sun goes down - or to turn off room heaters and computer displays when the overhead lights are turned off, indicating that the room is not occupied.

Note These are simple, low-cost "light intensity" sensors that are intended to be used for a wide variety of applications. However, they were never intended to measure the lux of a particular scene. For this reason, we're not able to provide a formula to convert light intensity to lux.

You could not use it (for example) to test that a workbench has an exact luminous intensity of "X" cd/m2. If you require such measurements, you should use a standard light sensor with a 0-10v or 4-20mA signal in a AIO adpater.

Also note that the duration of the actual light detection is short, so it works best with either sun light or non-flickering incandescent or DC-powered halogen lamps. Measure light with standard fluorescent light will result in the light value varying over a range of values (many percentage points). This does not prevent your system from detection 'the room is light' or 'the room is dark', but you will need to accommodate this variation with a hysteresis or dead-band calculation.

Digi Products

These pages discuss or summarize specific Digi Products.

Designing a sleeping XBee sensor

How to design a good sleeping XBee sensor product

This Wiki page outlines best practice at a higher-level functional level. It does NOT explain hardware design, and exists because some Digi XBee customers create wonderful designs - and a few have created stupid, unusable designs.

The basic assumption of the design outlined is that you integrate a PIC or other small controller to actually manage your sensor, then use standard serial communications to the XBee module.

Pin-sleep vs cycle-sleep

Pin-sleep - your PIC controls XBee

Primarily used for Xbee with 802.15.4, ZNet, Zigbee or Point-Multipoint technology.

Advantages

- Offers the lowest power and longest battery life
- Allows PIC to take samples without powering up radio - could send less data when process is idle

Disadvantages

Your customers will require one powered fully-awake 'router' device for each eight to twelve sleeping devices

Your PIC cycle time won't be accurate enough to use DigiMesh and sleeping routers

Cycle-Sleep - The XBee Controls Sleep

Can be used with any XBee technology - critical for DigiMesh with sleeping routers.

Advantages

- Xbee suitable for DigiMesh have adjustable sleep-clocks accurate enough to keep all nodes waking at the same time
- Powered/Fully-awake routers are NOT required

Disadvantages

The accurate clock control increases power consumed while sleeping

So there is a trade-off between these two designs. You must consider what features are important - here are two ideas:

- If your devices will tend towards long sleeps and require no interaction, then the pin-sleep and hassle of powered routers is likely the best design. A tank farm or collection of chemical drums is an example of this system.
- If your devices will tend to be active every 10 or 15 minutes, and if the devices cooperate in a system, then the slightly worse battery life of a DigiMesh system with sleeping routers is likely the best design. An irrigation system is an example of this design.

An ideal example

The ideal XBee-based sleeping sensor/device does the following:

- The XBee has Zigbee End-Device API firmware.
- The PIC takes a sample using its own logic and power
- The PIC wakes the XBee and sends a serial packet in the API code 0x10 form to either MAC 00:00:00:00:00:00:00:00 or optionally a user loaded MAC. For Zigbee this goes to the "coordinator" - the Digi gateway.
- The PIC keeps the XBee awake for a user-defined time - probably in the 10 to 30 second range. This allows the host to react to the data sent, issuing other requests to the PIC or XBee.

This design can be greatly improved by having the PIC send multiple readings within one packet. Minimizing the number of transmissions by sending as much data in each packet maximizes battery life. For example, a sensor taking hourly readings could send the last 8 hourly readings every 4 hours - or 9 hourly readings every 3 hours. In the unlikely event of packet loss, this 'history' within each send allows the host to transparently recover from the problem without any "retry" scheme requiring more RF transmissions.

The use of API mode is not required. You could use AT-mode, however starting with the API-mode allows more future flexibility. For example, API-mode would enable optionally 'routing' two samples to two diverse nodes for added redundancy. AT-mode would not permit this.

Example data production

The world has moved away from the old master/slave poll/response paradigms. While there are many sets of jargon, perhaps the easiest to adopt is the producer/consumer model. Your sleeping sensor is a **producer of data**. The gateway and ultimately your customer's host application is a **consumer of data**. When your sensor wakes, it sends a data production out and in truth does not care if anything consumed it. In truth, it is the consumer (and YOU) who needs to solve network problems - not your small, battery powered sensor.

Here is an example data production of roughly 52 bytes which assumes the last 8 readings are repeated.

Size	Description
2 to 4 bytes	Header and device status, which indicates that this is a data production. It may include a rev or data format code to allow different formats.
2 to 3 bytes	Running counter - your sys-ticks. This acts as a sequence count, plus helps the gateway application relate the data samples to real time of day
2 bytes	Sample Rate, perhaps the number of seconds (or sys-ticks) between samples
2 bytes	Sensor voltage. There is no reason to take this 'per sample' - once per production is fine. If you only want to send this once every 10 productions to save the ADC sampling power, define a "null" value like 0x0000 or 0xFFFF.
8 x 5 bytes	Eight Samples, each sample perhaps 5 bytes. Perhaps the sample consists of two 16-bit values and 1 byte of status and reading quality info.

Size	Description
1 byte	Simple XOR or addition checksum. It probably isn't required since the mesh includes many levels of checksum and retry already. But it won't hurt and will make some users feel more comfortable.

Notes:

- You might chose to save the readings in FLASH, but it might not be worth the power. Assuming some memory remains valid during sleep, your sensor could maintain a prepared message, where the 7 old samples are shifted over before the newest sample is inserted.
- There is no reason to support sending less than 8 during 'start-up'. In theory that occurs once in 3 to 5 years. Instead, define unused samples as 0x00 00 00 00 00 or 0xFF FF FF FF FF.
- There is no reason to number or time-stamp the samples. The first sample is assumed 'now', with the other seven being one sample-time in the past. If the sample rate changes, then design your PIC firmware to zero/null out the old samples. Since many host applications blindly reload configuration settings in sensors each time the application restarts, your PIC firmware must be smart enough to detect the host rewriting an unchanged sample rate, and not clear out the old data needlessly.

What NOT to do

Do NOT have the PIC wake the Xbee, then wait silently for a serial poll

This is a bad/unusable design! In contrast, your PIC and XBee MUST SAY SOMETHING anytime it wakes up. It could send an "I am Awake" message and expect the host to send a serial request/poll, however that wastes battery life. Since your PIC must say something, why not just send the most recent data?

Do NOT use broadcast

In mesh, **Broadcast = very_bad**. Yes, Zigbee has a broadcast and 1 or 2 Zigbee nodes can survive with broadcasts, but you will produce a non-scalable solution. Digi has had many customers IGNORE this, thinking broadcast is the easy way out ... they all had to redesign their protocols and systems. Either learn from their mistakes - or repeat them.

Do NOT design things to require tight timing

Avoid protocols with complex procedures where the host must do a series of steps in a tightly timed sequence. For example, a two-step reboot where the host sends the reboot command, the device sends an "Are you sure?" response and the host must send a special second confirmation within a few seconds. This design is NOT scalable. If the mesh has dozens or hundreds of devices, it is not possible for the host to reliably send a tightly timed sequence of messages because other devices are talking and also asking for service. Worse-case your device may end up in an unstable state due to lack of host follow-through; best-case the host can never accomplish the task because it can never meet the timing requirements.

Do NOT design unstable test modes

Do not require the host to reconfigure the device into a "broken mode" to initiate a test. If the host goes offline, the device may become unreachable. For example one product required the host to turn off sampling, risking the device sleeping silently (see the first DO NOT above) - and in this situation if

the host got busy, the device was lost to the mesh. Instead, any request for a test should be a one-shot, where the device temporarily changes settings as required, plus starts a timer. If the host fails to complete the full test or transaction, then the device should reboot with the older operational steady-state parameters, returning to normal operations.

Do NOT treat the sample and mesh-report timers as independent

The mesh-report "timer" should be some multiple of the sample timer, so in reality you send a report after "X-th" sample is taken.

- An example of a bad design: device takes a sample every hour, and sends the last 6 samples every 3 hours. While this sounds good, the problem occurs if the time of the last sample is not related to the time of report. It is easy for the host to assume the newest of the 6 samples occurred "now" and the older 5 occurred 1 sample period ago, yet if the sample timer is independent of the report timer, then the samples might have been taken 10 seconds ago - or 59 minutes ago. It is impossible to recreate an accurate history this way.
- So a good design would have the report "timer" be checked after any sample. In this example, after the third new sample the report should be triggered. Thus the host can reasonable assume the samples occurred as-of "now". So the best configuration setting for reports is NOT how often, but after how many samples. So one sample per hour, report after 3 samples would be the better design.

Questions / FAQ

Do I need to check for collisions or busy radio traffic when I wake the radio?

No. The Xbee does this for you. You send your serial message into the XBee, and it 'fits' the message into the mesh.

Can I have the host 'Acknowledge' the data, and the PIC retry if no 'Acknowledge' is seen?

You can, but you will NOT find it useful. The XBee already does intelligent RF sense to avoid collisions and 1 retry to the parent/peer node. If the host fails to see the first send of the data, then a retry within a minute probably will ALSO fail. This is the nature of a smart mesh. Plus what if someone powers the host off for a few days? Do you really want all of your sensors retrying and cutting their battery life in half?

Won't the sleeping node's parent hold a few message for it?

Yes, but only for 28 seconds. Thus if your sensor sleeps for 1 hour and wakes up, there is little probability that any queued messages are waiting for it.

Can I use cyclic sleep on the XBee, then have the Xbee wake up my PIC?

Yes, you can. This has the big advantage of enabling support for DigiMesh and sleeping routers. It has the small disadvantage of requiring the XBee to be awake even when the PIC discovers it has nothing to say.

I don't want to use addresses; I just want to treat it like RS-485

Sorry, you can't do this. The mesh can be organized in a "virtual star/hub design", where end-devices reply to the central node without addressing (in Zigbee, the Digi Gateway). However, the gateway will need to send unicast messages to the MAC address of a single end-device.

DigiMesh products

DigiMesh XBee technology

Digi produces several families of DigiMesh XBee module. The most important feature in determining support within XBee adapters and gateways is the foot-print of the Xbee.

- Most existing Digi Adapters and Gateways support only the 20-pin through-hole footprint, for which only DigiMesh at 2.4 Ghz and 900 Mhz is available.
- Newer XBee families have moved to a SMT format and include 865 Mhz, 868 Mhz and 900 Mhz.

Existing DigiMesh gateways

- Digi ConnectPort X2
  - ConnectPort X2 - Industrial with DM 2.4 GHz to Ethernet (metal case, extra memory)
  - ConnectPort X2 - Industrial with DM 900 MHz to Ethernet (metal case, extra memory)
  - Note: as of March 2012 there are no X2 Commercial models with DigiMesh (lower cost, plastic case, less memory)
  - Note: as of March 2012 there are no X2 models with Wifi and DigiMesh
- Digi ConnectPort X4
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & cellular (GSM/Edge)
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & Wi-Fi
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet & cellular (GSM/Edge)
  - Note: as of March 2012 there are no X4H or X4 IA models with DigiMesh
  - Note: as of March 2012 there are no X4 CDMA cellular models with DigiMesh
  - Note: as of March 2012 there is no SMT support within the X4, so no 865 Mhz or 868 Mhz DigiMesh support
  - Note: as of March 2012 there are no X5 vehicle models with DigiMesh

Existing DigiMesh adapters

- XBee-PRO DigiMesh 2.4 Range Extender
- Note: as of March 2012 there is no DigiMesh 900 Mhz Range Extender

- XBee-PRO DigiMesh Adapters
  - XBee-PRO DigiMesh 2.4 GHz, RS-232 adapter
  - XBee-PRO DigiMesh 2.4 GHz, RS-485 adapter
  - XBee-PRO DigiMesh 2.4 GHz, Digital IO adapter
  - XBee-PRO DigiMesh 2.4 GHz, Analog Input adapter
  - XBee-PRO DigiMesh 2.4 GHz, USB adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-232 adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-485 adapter
  - XBee-PRO DigiMesh 900 Mhz, Digital IO adapter
  - XBee-PRO DigiMesh 900 Mhz, Analog Input adapter
  - XBee-PRO DigiMesh 900 Mhz, USB adapter
  - Note: as of March 2012 there is no DigiMesh Smart Plug
  - Note: as of March 2012 there is no DigiMesh Light/Temperature (LT/LTH) Sensor
  - Note: as of March 2012 there is no DigiMesh XStick
  - Note: as of March 2012 there are no DigiMesh WatchPort Adapters

Can Digi ZigBee/XBee products be converted to DigiMesh?

Officially, No.

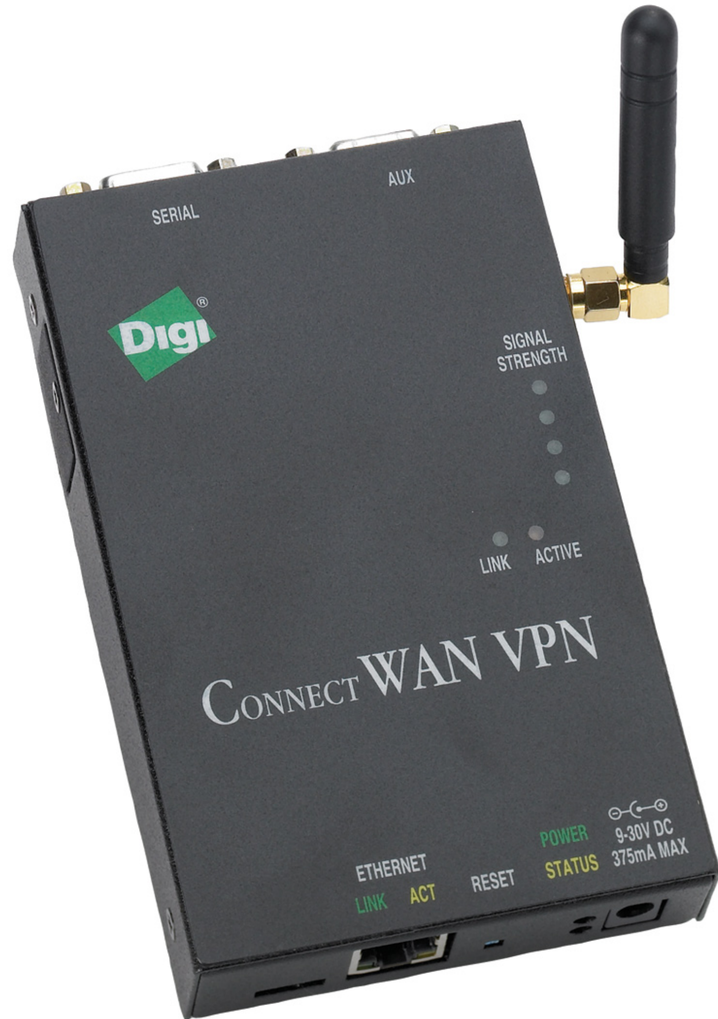
However, one can remove the 20-pin ZigBee XBee and replace it with a 20-pin DigiMesh XBee. This works fine with the RS-232/485/USB adapters, but may not work with adapters expecting analog signals because the ZigBee Xbee and DigiMesh XBee use different reference voltages. Your software will need to compensate for this, as well as the fact that there are some minor AT/DDO settings differences between Zigbee-XBee and DigiMesh-Xbee.

Digi extras

ConnectPort X2

ConnectPort X4

Connect WAN



ConnectPort WAN

Transport Family

Python migration guide for NDS 2.8 to 2.9

Existing Python applications are potentially affected by an XBee Socket API change summarized below.

The API change specifically affects the recvfrom() call, and existing application Python code should be audited for len() checks on the socket returned from recvfrom(). Testing Python applications under NDS 2.9 is also recommended to verify backward compatibility. The Dia (Device Integration Application) framework does check the length returned by recvfrom(). As a result, applications using Dia 1.1.16 and prior must upgrade to 1.1.17 (or later). No additional configuration changes should be necessary.

To understand if you are running firmware 2.8 or 2.9, see What is your product firmware level?

XBee socket change summary

Effective in the NDS 2.9, the AF_XBEE sockets have had their addressing structure extended to include an options field and a transmission_id field. When using Python this means the addressing tuple has changed from 4 elements to 6 elements in length. Backward compatibility has been preserved so long as the length of the address tuple is not checked by user source code. The new fields in the addressing tuple are considered optional and a 4-element tuple may still be used in all socket operations.

The options field is used differently depending on whether a packet is being transmitted or received. The meanings of the options are defined in constants which begin with XBS_OPT_*.

The transmission_id field is used to track information about a specific datagram transmission sent with a socket object instance's sendto() method. The transmission_id is only used when the XBS_SO_EP_TX_STATUS socket option is enabled at the XBS_SOL_ENDPOINT level. If this option is enabled, the socket will provide transmission status frames with the transmission_id field set to the value given by the user in a prior sendto() call.

Transmission information arrives in three formats: X-API type 0x89 (XBee Transmit Status Frame), X-API type 0x8b (ZigBee XBee Transmit Status Frame), and internal error codes. The cluster_id field specifies the type of information: 0x89, 0x8b, and 0x0 respectively. Any non-zero internal error code should be considered a transmission failure. The X-API frames may be decoded according to the XBee API reference documentation.

RCI request

Remote Command Interface (RCI) is a method for remote clients to control, configure, and gather statistics from Digi Connect devices. RCI is a stateless, request/response protocol. RCI uses XML and HTTP to exchange data between clients and Digi devices.

RCI over HTTP

RCI requests are sent to the device using an URI of UE/rci. For example, if the Digi Device's IP address is 192.168.1.1, then RCI requests are sent to http://192.168.1.1/UE/rci.

RCI requests are sent as an HTTP POST with the XML request of the form specified in this document. Note, due to space limitations on the device, the largest request that can be processed is 32KB. If a request is larger than this, it must be split into multiple RCI requests. RCI replies from the device are not subject to this limit.

Security is handled in the usual HTTP mechanism. The username and password must be passed to the device in the header of each HTTP request.

RCI over serial

RCI requests can also be sent over the serial port. This is useful in scenarios where a master processor is connected to the Digi Device through a serial port. This allows the master processor to configure the Digi Device as part of its configuration process, so that a separate manual configuration step for the Digi Device is eliminated. You must enable 'RCI over Serial' in either the Web Interface or the Command Line Interface before the Digi Device will accept RCI requests and return replies. The RCI over Serial option is available only on the primary port. RCI over Serial uses the DSR (Data Set Ready) serial signal. Verify that the serial port is not configured for autoconnect, modem emulation, or any other application which is dependent on DSR state changes. Note: When the Digi Device sees its DSR raised, it will set the serial port settings to 9600 baud, 8 data bits, no parity, and 1 stop bit. When DSR is lowered, the Digi Device will restore the previous serial settings.

Configure using the Command Line Interface (CLI)

1. Access the CLI using telnet or rlogin and the module's IP address. Ex:

```
telnet 192.168.1.2 -or-
rlogin 192.168.1.2
```

2. At the command prompt type:

```
 #> set rciserial state=on
```

Configure using the web user interface

1. Access the web interface by entering the module's IP address in a browser's URL window.
2. Choose Serial Ports from the Configuration menu.
3. If the device has more than one port, select Port 1.
4. If a port profile has not been selected, select Custom and click Apply.
5. Select Advanced Serial Settings.
6. Select Enable RCI over Serial (DSR) and click Apply.

RCI request/reply

An RCI XML document is identified by the XML elements rci_request and rci_reply. An RCI request specifies the XML element "rci_request" optionally with a version number. The version should match the version of RCI the client expects. The current RCI version is 1.1. If a version is not specified, the RCI version of the device is used to form the reply. Not specifying a version can cause problems when communicating with devices at different RCI versions, if the client code is not written in a version independent way. Therefore, it is highly recommended to always supply the version of RCI in requests, unless the client code has been designed to be version independent. Example of a request element:

```
<rci_request version="1.1">
```

The device will respond to requests with the element "rci_reply" along with the version number as an attribute. Example reply:

```
<rci_reply version="1.1">
```

rci_reply errors

Errors that occur at the request level will result in an error element as a sub-element of the <rci_reply>. Errors and warnings are explained below <rci_reply> errors: Error ID Description

1. Request not valid XML
2. Request not recognized
3. Unknown command

Command

The command section of the protocol indicates the action requested (or action performed in replies).

Commands are specified as sub-elements to <rci_request> and <rci_reply>.

This example requests all configuration settings:

```
    <rci_request version="1.1"> <!—Identifies the protocol and whether this is a
request or a response --
        <query_setting/> <!— request config of device -->
    </rci_request>
```

This example requests the configuration information for just boot settings and serial settings.

```
    <rci_request version="1.1">
        <query_setting>
            <boot/>
            <serial/>
        </query_setting>
   </rci_request>
```

Supported commands

COMMAND	REQUEST DESCRIPTION	RESPONSE DESCRIPTION
query_setting	Request for device settings. May contain setting group elements to subset query (only setting group subset supported. Subsetting below this level not supported).	Returns requested config settings. Requests specifying no settings groups (eg. <query_setting/>) return all settings.
set_setting	Set settings specified in setting element. Settings data required.	Empty setting groups in reply indicate success. Errors returned as specified below.
query_state	Request current device state such as statistics and status. Sub-element may be supplied to subset results.	Returns requested state. Requests specifying no groups (eg. <query_state/> return all state.
set_factory_default	Sets device settings to factory defaults. Same semantics as set_setting.	Same semantics as set_ setting.
reboot	Reboots device immediately.	None
do_command	see RCI do command	see RCI do command

Errors and warnings

Response documents may contain an element as a child of the command or data element that indicates the result of the request. More than one error or warnings may be present. Error and Warning elements:

<table>
<tr><td>error</td><td>An error occurred.</td><td>Attribute id: A numeric id specified by the parent element (the command or the data element). An error element id="0" is equivalent to no error.</td><td>Children Elements<table><tr><th>name</th><th>desc</th></tr><tr><td>desc</td><td>Optional - Text description of the error.</td></tr><tr><td>hint</td><td>Optional - Used to indicate to the client the source of the error. This will typically be set to the field name that the error.</td></tr></table></td></tr>
<tr><td>warning</td><td>Command executed, but a warning was issued.</td><td></td><td></td></tr>
</table>

Example:

```
<serial_setting>
   <error id="3">
       <hint>baud</hint>
       <desc>Value out of valid range.</desc>
   </error>
    </serial_setting>
```

Errors are required to have an id. <hint> and <desc> are optional and more than one are allowed.

Notes

RCI XML must be well-formed XML

The device parses incoming RCI requests in a sequential manner. Each XML element is parsed and acted upon as it arrives. This is not ideal behavior, but is necessary because of the inherent resource limitations of a device. Ideally, the entire XML request would be read into memory, validated, parsed and acted upon only after validation.

XML structure errors may be found after actions have been taken. For instance:

```
<rci_request version="1.0">
    <set_factory_default/>
</rci_requestBADENDTAG>
```

This request will result in an XML parse error, but since the parse error occurs after the set_factory_ defaults, the device will be set to factory defaults. Therefore, it is highly recommended that RCI requests be validated with an XML parser before being sent to the device. Using any standard parsers, such as the XML parsing in the Java SDK, to form RCI requests accomplishes this.

XML structure characters must not be sent as character data

Care must be taken to avoid accidental badly formed XML in RCI requests because of including XML structure characters, such as "<", as user entered data. Any field that accepts character data must be checked to ensure that "<" and ">" are not present (fields such as the email body of an alarm are common places this can happen). It is recommended that all instances of "<" and ">" in character data be converted to "<" and ">", which is the standard XML representation(entities) of these characters.

To use RCI to Query DIA device/channel Information

Reading device/channel information by direct HTTP to a DIA device requires a different do_command set. See Simple RCI by HTTP for working code examples.

References

https://www.digi.com/search/results?q=900005www.digi.com/search/results?q=9000056969

Raw API over ethernet to CPX2

Talking raw API frames over ethernet to the connectPort X2

The Digi ConnectPort X2 offers low-cost Ethernet access into your mesh. The CPX2 runs two very different firmwares, either of which you can flash in at will.

ConnectPort X2 ethernet Python firmware (82001596)

This firmware provides a reduced-memory platform for your Python applications. It does not include the normal Digi IA engine for Modbus/TCP to Modbus serial bridging, nor does it offer raw API frame access to the XBee. Officially, these firmwares support XBee running the 802.15.4, ZNet and ZigBee (ZB) firmwares. The Wi-Fi version of the CPX2 with Python (so WiFi to Mesh) uses firmware number 82001630.

General comments:

- The CPX2 has one-half the normal RAM of an X4/X8, plus runs at less than half the CPU speed. Therefore it is not a perfect 'low-cost version' of the X4; it does introduce limitations.
- In general, your own Python code will run fine on either X2 or X4, however many stock public Python libraries (like HTTP or SMTP servers) assume RAM is virtual and free. So users expecting to make heavy use of external Python modules will find the CPX2 more challenging.
- You will find developing on the CPX2 frustrating, so you should develop on an X4 with an eye to keeping memory usage down, then 'port' your code to the X2 for testing and implementation.
- Some people get burned thinking 'My application is so functionally simple - it will run on the CPX2'. However, 'simple' does not equal 'low memory usage'. Some very complex applications require very little memory, while some very simple applications (like parsing XML) can consume many MB of RAM.

ConnectPort X2 Ethernet firmware w/ API Access (82001631)

Notice there is NO word 'Python' - this firmware offers the normal Digi IA engine for Modbus/TCP to Modbus serial bridging, or it can be configured for raw API frame access to the XBee. The Modbus bridging functions fully for ZNet and ZigBee, and partially for 802.15.4. The raw API frame support works for ALL versions of the XBee (including the point-to-multipoint models like the 868Mhz versions for Europe). The Wi-Fi version of the CPX2 with Python (so WiFi to Mesh) uses firmware number 82001597 - the numbers look swapped, but they are correct.

General comments:

- The Modbus/TCP bridge is documented here: Modbus bridge on CPX4 and Modbus Serial Over Mesh.
- You must select either the IA Engine/Modbus Bridge, or the raw API frames - the CPX2 does not support both functions concurrently
- The CPX4 and X8 have the same IA Engine/Modbus Bridge integrated in the normal firmware, however neither the X4 nor X8 support the raw API frames.
- The raw-API frames can be sent via TCP/IP, UDP/IP, SSL/TLS or Digi Realport.

Using terminal server to test Xbee serial adapters

Using Digi TS Products to test XBee serial adapters

Testing your application on a realistic field systems is always a challenge - even if you have 20 serial end devices, you may not have the hardware to simulate realistic field behavior. Plus those 20 end devices will not purposely create bad responses, so it will be difficult to fully test your application in both good and bad situations.

An excellent way to test large numbers of serial devices is with a Digi ConnectPort TS8/TS16 terminal server, which even comes with the same embedded Python environment which you use on the X4 or X8. Thus you can create custom Python test code running directly within the terminal server to mimic your XBee devices. Some TS8/TS16 models support RS-232 only, while other models are configurable on a port-by-port basis to be RS-232, RS-422 or RS-485. Internally, Digi uses terminal servers to drive a 1000 mesh node test system with serial messages.

Example application

A Python application runs on a Digi ConnectPort X4, managing a tank farm. Once per hour, ultrasonic level sensors use Digi XBee modules set for API-mode to send the latest level readings. These level readings are timestamped and stored in non-volatile memory. Once per day the application builds a report and uploads the timestamped level data back to a central server to be used for supply forecasting and delivery planning.

Physically simulating a level with an ultrasonic level sensor is easy - point the sensor at a plywood paddle or cardboard box, which is moved manually nearer or farther away to simulate distance and

thus level. However, it is impractical to manually move paddles or boxes hourly, 24-hours per day, 7-days a week. Thus it is impractical to do long-term application testing while physically simulating 8 (or even 80) tanks requiring fresh data hourly.

Using a Digi terminal server to simulate serial end devices in the practical solution. Python code simulating tanks emptying and filling connect to the desired number of Digi XBee RS-232 adapter, then sends the appropriate serial data message (production or publish) via each XBee device hourly.

The X4 tank management application is never aware of the ruse - it receives and timestamps realistic level data hourly, 24-hours a day, 7 days a week. Better, the custom Python simulation program can randomly simulate lost messages or improperly formatted responses. If the X4 tank management application also generates alarm call-outs, the custom program simulating the tanks can also send low (or high) level responses to trigger alarms, then confirm the X4 application correctly handles the alarm situation.

Cables

Most Digi terminal server products use a 10-pin RJ-45 connector. Since the XBee RS-232 adapter has a 9-pin DTE port, use the **Digi cable part number 76000645** with the description **RJ45/DB9 Female Crossover 48**".

WVA datastream to device cloud

Example: WVA to Device Cloud

This a simple example to show how to collect data from the WVA, format it and upload the data as DataStream to Device Cloud.

To run the example bellow

- Upload the script to the WVA under the name **simple_wva.py**.
- Dropdown to the shell, Enter the command **Python simple_wva.py** and press **Enter**.

```
###########################################################################
#                                                                         #
# Copyright (c)2015, Digi International (Digi). All Rights Reserved.      #
#                                                                         #
# Permission to use, copy, modify, and distribute this software and its   #
# documentation, without fee and without a signed licensing agreement, is #
# hereby granted, provided that the software is used on Digi products only #
# and that the software contain this copyright notice,  and the following #
# two paragraphs appear in all copies, modifications, and distributions as #
# well. Contact Product Management, Digi International, Inc., 11001 Bren  #
# Road East, Minnetonka, MN, +1 952-912-3444, for commercial licensing    #
# opportunities for non-Digi products.                                    #
#                                                                         #
# DIGI SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED  #
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A         #
# PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, #
# PROVIDED HEREUNDER IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND. #
# DIGI HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,        #
# ENHANCEMENTS, OR MODIFICATIONS.                                         #
#                                                                         #
# IN NO EVENT SHALL DIGI BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,     #
# SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,  #
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF  #
# DIGI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.               #
#                                                                         #
###########################################################################

"""
    Simple WVA to Device Cloud Example
"""

import time
import json
import httplib
import traceback
from cgi import escape

import idigidata

class wva_simple(object):
    """
        this class connect to the local wva api
    """
    VERSION = "1.1"
```

```
    def __init__(self, host="127.0.0.1"):
        self.host = host
        self.connection = None
        self.json_headers = {'ACCEPT': 'application/json',
                             'Content-Type': 'application/json'}

    def collect(self, list_of_urls):
        """
            Description:
                collect data from the WVA
            Args:
                list_of_urls - a list of WVA data urls
            Returns:
                a json object
            Raises:
                (None)
        """
        data_obj = {}
        try:
            for url in list_of_urls:
                full_url = self.host + "/ws/" + url
                print "URL = %s" % full_url
                _data = self.read(url)[1]
                data_obj.update(json.loads(_data))
        except ValueError:
            print "Did not receive json data back: [[[%s]]]" % str(_data)
        except:
            print "Error while reading url: %s\n%s" % (full_url,
traceback.format_exc())
        finally:
            self.connection.close()
            self.connection = None
        return data_obj

    def read(self, partial_url):
        """
            Description:
                perform a url request
            Args:
                partial_url: a url to be read - i.e: /vehicle/dtc
                output_format: response output, The WVA support JSON(default) and
XML
            Returns:
                status: the response status from the server
                res: the response status (JSON or XML String)
            Raises:
                HTTPException: if an Error occurs when trying to read.
        """
        try:
            if not self.connection:
                self.connection = httplib.HTTPSConnection(self.host, timeout=15)
            self.connection.request("GET", "/ws/" + partial_url,
headers=self.json_headers)
            response = self.connection.getresponse()
            return response.status, response.read()
        except httplib.HTTPException, err:
            raise httplib.HTTPException(str(err) + " " + partial_url)

def wva_collect_forever():
```

```
    """
        Description:
            This is the main entry point of this example
        Args:
            (None)
        Returns:
            (None)
        Raises:
            (None)
    """
    data_list = [
        "vehicle/data/VehicleSpeed",
        "vehicle/data/EngineSpeed",
        "vehicle/data/TotalEngineHours",
        "vehicle/data/TotalDistance",
        "vehicle/data/BatteryPotential",
        "vehicle/data/FuelEconomy",
        "vehicle/data/FuelLevel",
    ]

    # initialize wva interface
    wva = wva_simple()

    while True:
        #collect
        collected = wva.collect(data_list)

        #format
        datapoints = ""
        for data in collected:
            datapoints += ("<DataPoint>"
                           "<streamId>%(streamId)s</streamId>"
                           "<data>%(data)s</data>"
                           "<timestamp>%(timestamp)s</timestamp>"
                           "</DataPoint>"
                           % {
                                "streamId": escape(data),
                                "data": collected[data]["value"],
                                "timestamp": int(time.mktime(time.strptime
(collected[data]["timestamp"],

"%Y-%m-%dT%H:%M:%SZ")) * 1000)
                           })

        datapoint_list = "<list>%s</list>" % str(datapoints)
        print datapoint_list

        #upload
        print idigidata.send_to_idigi(datapoint_list, "DataPoint/stream.xml")
        time.sleep(60)

if __name__ == '__main__':
    wva_collect_forever()
```

Watchport Camera

Watchport/V2 and Watchport/V3 (non-interlaced model) are high performance USB cameras designed for kiosks, ATMs, point-of-sale, ID badging, mobile computing, webcam, or any mission-critical application utilizing camera surveillance. They offer exceptional low light sensitivity (<1 lux), 30 fps USB frame rates at all resolutions and enhanced resolution to deliver optimal picture quality. An optional lens kit is available for complete customization. Watchport cameras are USB powered and offer Plug and Play installation for easy integration into any new or existing application.

To use the Watchport/V2 camera under Python, see Module: camera.

Features

- Ideal for photo ID badging, kiosks, ATMs, access control and general surveillance
- Only USB cameras with Windows NT drivers
- Motion detection software available to create cost-effective remote monitoring system
- Facial recognition software available to create continuous PC access control without passwords or tokens
- Low light sensitivity: <1 lux
- Adjustable resolutions
  - 1280 x 960 (software enhanced)
  - 640 x 480
  - 352 x 288
  - 320 x 240
  - 176 x 144
  - 160 x 120
  - 128 x 96
- Maximum frame rates by resolution: 30 fps at all resolutions
- Removable 4.9 mm lens; F:2.0 adjustable
- Non-interlaced model for even higher quality pictures
- Optional lens pack available
  - 3 mm wide angle (green)
  - 8 mm close-up (red)
  - 12 mm telephoto (blue)
- Optional mounts
  - Wall mount
  - Clamp mount

- Exposure controls
  - Manual and automatic gain and shutter control,
  - Automatic white balance (AWB)
  - Manual color balance
  - Color saturation
- Color formats: 16.8 million TrueColor (24 bit RGB), YUY2, UYVY
- Motion JPEG compression
- Sensor: color CCD
- Scan frequencies: 60, 50, 30, 25, 15, 12.5, 7.5, 6.25 and 4 Hz
- Shutter rate controls: 1/4 second to over 1/30,000 second

Watchport/VE

- Video encoder to connect conventional analog cameras via USB
- Converts NTSC video streams
- Composite and S-type video connections
- Mono or stereo audio with dual RCA connectors
- Same drivers as Watchport cameras
- Compatible with Watchport Vision Series software products

Related pages

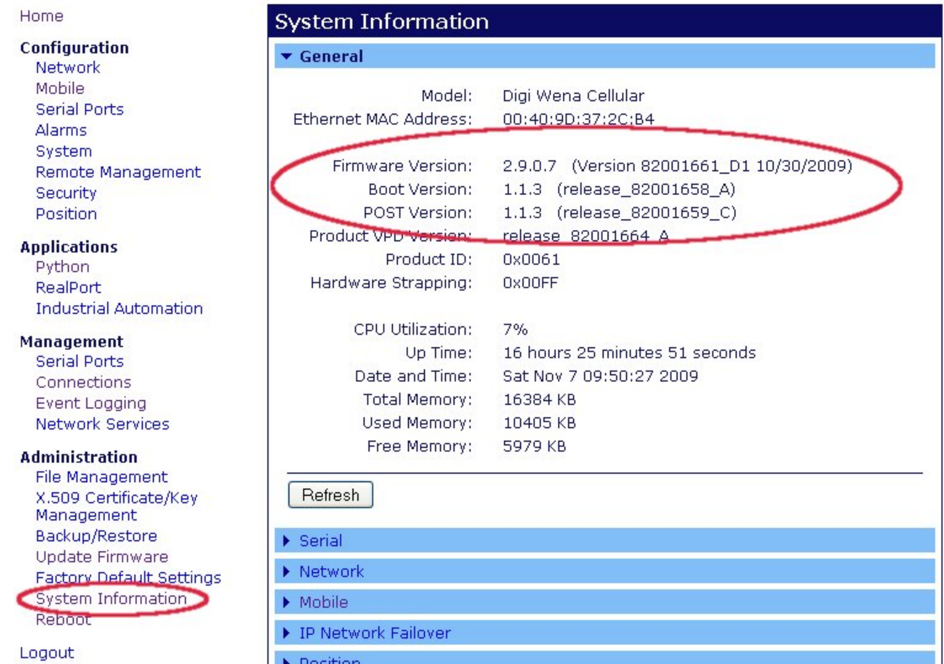
Module: camera
Card Swipe Demo

What is your product firmware level

How to learn the firmware level in your Digi product

This Wiki page covers the Digi Connect and ConnectPort products.

By web interface



Screen Shot of System Information

For a summary of the firmware on the Digi product, click the link.

By CLI/Telnet

To view the product firmware level via command line, telnet or Module: digicli, use the display device command (or disp dev).

```
#> display device
Device Information:
Product               : Digi Wena Cellular
MAC Address           : 00:40:9D:37:2C:B4
Firmware Version      : 2.9.0.7 (Version 82001661_D1 10/30/2009)
Boot Version          : 1.1.3 (release_82001658_A)
Post Version          : 1.1.3 (release_82001659_C)
Product VPD Version   : release_82001664_A
Product ID            : 0x0061Administration | System Information
Hardware Strapping    : 0x00FF
CPU Utilization       : 13 %
Uptime                : 18 hours, 6 minutes, 51 seconds
Current Date/Time     : Sat Nov  7 11:31:31 2009
Total Memory          : 16777216
```

```
Used Memory          : 10687036
Free Memory          : 6090180
```

Digi firmware terminology

Application or Python code (Also a Filesystem Image)

If you are running a Digi product with a custom Python application, this is loaded either via the web interface's Python page or as part of the initial manufacturing process when a custom filesystem is preloaded. These are not part of the Firmware or EOS, so changing firmware does not directly affect the Python files or application. You look under **Applications | Python** link to see a list of uploaded Python code.

Firmware or EOS

This is the main operating program - what makes the product into a device server or cellular router. In the image below, the EOS is listed as 2.9.0.7 and 82001661_D1. 2.9.0.7 refers the NDS family version. The code 1661 defines the hardware which this firmware runs on - in this example a Digi Connect WAN with 16MB of RAM. The 'D1' is the revision of this particular EOS build, 'D' being the major rev and '1' being the minor rev. **You should confirm your POST is the correct level before updating the firmware/EOS, as some newer firmwares rely upon changed flash or memory access features**.

You may also hear Digi people refer to terms like NDS, PS3 or GeneOS. These refer to families of products sharing common hardware, source code and therefore features. Most newer Digi products (such as those with Python) are in the NDS family. If you see a code like 'D1_SA3', this means the firmware is a Beta test-build, with the SA3 meaning it is the third SA build (or System Assurance build).

POST (like a BIOS)

The POST is the low-level code which maps the exact hardware function for the firmware, for example how to access the realtime clock or how to erase sectors of flash. The numeric and letter codes mean the same, and you'll notice the numeric codes for the EOS and POST do not match. **You should confirm your POST is the correct level before updating the firmware/EOS, as some newer firmwares rely upon changed flash or memory access features**.

BOOT and VPD

The BOOT and VPD relate to hardware and manufacturing tests, and these cannot be changed without special hardware tools. The Product Id and Hardware strapping also define hardware details built into the product - for example how many serial ports are populated and so on.

Which Digi products support Python

Partial product list:

- Digi Connect Core 9P 3G (all versions)
- Digi Connect ES (firmware 82001234_D or higher)
- Digi Connect WAN (newer units with 16MB of RAM running firmware 82001660_A - older 8MB units running 82001160 **cannot be converted to support Python**)
- Digi Connect WAN IA (newer units with 16MB of RAM running firmware 82001661_A - older 8MB units running 82001323 **cannot be converted to support Python**)
- Digi Connect WAN VPN (newer units with 16MB of RAM running firmware 82001662_A - older 8MB units running 82001253 **cannot be converted to support Python**)
- Digi ConnectPort Display (all versions)
- Digi ConnectPort TS8 (all versions)
- Digi ConnectPort TS16 (all versions)
- Digi ConnectPort WAN VPN (82001276_G or higher, 82001350_F or higher)
- Digi ConnectPort X8 (all versions)
- Digi ConnectPort X4 (all versions)
- Digi ConnectPort X4 H (NEMA 4X/IP66) (all versions)
- Digi ConnectPort X2 (when loaded with firmware 82001596 - units running firmware 82001631 do NOT have Python, but can be reflashed to run firmware 82001596. However, these units then lose the functionality unique to 82001631.)
- Digi Connect WAN 3G
- Digi Connect WAN 3G IA

How to check by the web interface:

Log into the Digi Product with your web browser, and look under the Applications heading for the Python menu item. If it is NOT there, then your product does NOT support Python.

Home

Configuration
Network
Mobile
Mesh Network
Serial Ports
Camera
Alarms
System
Remote Management
Security
Position

Applications
Python
RealPort
Industrial Automation

Home

Getting Started

Tutorial Not sure what to do ne

System Summary

Python is shown here

Model: Connect
Ethernet MAC Address: 00:40:9I
Ethernet IP Address: 192.168
Mobile IP Address: 166.136

Description: None
Contact: None
Location: None

How to check by the Telnet/Command line:

Log into the Digi Product with telnet or SSH, then enter the command "set Python ?". If you are shown the syntax for it, then it is supported. If you only see a list of valid commands - with 'Python' missing from the list, then it is not supported.

If your Digi Product supports Python, then you will see:

```
#> set Python ?
syntax: set Python [options...]
options:
    range=(1 - 4)
    state=(on|off)
    onexit=(none|restart|reboot)
    command="<program file> [arguments...]"
#>
```

If your Digi Product DOES NOT support Python, then you will see (notice no 'Python' command listed):

```
#> set Python ?

syntax: set [options...]
```

```
accesscontrol   alarm           autoconnect     buffer
camera          ddns            devicesecurity  dhcpserver
dialserv        dirp            dnsproxy        ethernet
failover        forwarding      geofence        host
hostlist        ia              mesh            mgmtconnection
mgmtglobal      mgmtnetwork     mobile          mobileppp
nat             network         passthrough     pmodem
pppoutbound     position        profile         serial
service         snmp            socket_tunnel   surelink
system          tcpserial       term            time
udpserial       user            vpn             vrrp

#>
```

Which Python Version

Different Digi products support different versions of Python. If you upload the uncompiled.PY files, then you may use any version of Python as long as you avoid features which are not supported on the Digi product (for example, you cannot use new 2.6.x features on a Digi ConnectPort X4 which only supports 2.4.3).

However, if you plan to use the iDigi/Dia framework and/or upload the compiled.PYC files, then you must use an exact match.

Python 2.4.3

Download from http://Python.org/download/releases/2.4.3/

- Digi Connect WAN Family
- Digi ConnectPort WAN Family
- Digi ConnectPort X2
- Digi ConnectPort X4
- Digi ConnectPort X4H
- Digi ConnectPort X5
- Digi ConnectPort X8

Python 2.6.1

Download from http://Python.org/download/releases/2.6.1/

Digi ConnectPort X3

Python 2.6.2

Download from http://Python.org/download/releases/2.6.2/

Digi ConnectPort LTS 8/16/32

XBee Product Codes

Digi Product Types

Adapter or boxed Xbee products purchased from Digi will come correctly configured and be displayed within the web interface and telnet node lists with a type string - such as "RS-232 Adapter" or "X4 gateway". The XBee module's "DD" parameter encodes this knowledge as a 32-bit value.

Yet if you replace the supplied Xbee module with a new one - for example to swap a higher power XBP24-B module for the supplied XB24-B module, then the device may show up as "Unspecified Device" instead. Some XBee firmware - such as the special purpose Digital or Analog I/O versions - force the "DD" value to the correct codes, while others - such as the generic AT or API firmware usable in RS-232 or RS-485 adapters will leave set DD set to zero and be shown as "Unspecified".

It is important that the "DD" value be correct, as many gateway and Python functions rely upon the "DD" value to understand how to use the device.

Python access to DD Parameter

Your Python program can read or write the DD or Digi Device Type with the ddo_get_param() and ddo_set_param() call. If the ddo_get_param() is successful, then 'result' will be a 4-byte binary string in form AA BB CC DD, where Module Device Type is 0xAABB and Product Type is 0xCCDD:

```
import zigbee

result = zigbee.ddo_get_param( "[00:13:a2:00:40:0a:07:8d]!", 'DD')
```

Forcing New DD from a Digi Gateway

You can preload the DD value correctly from XCTU when you load the XBee module firmware

You can also use telnet (or the command-line) access to a Digi X2/X4/X8 gateway to write new DD values to an active device. In the example below, the four "Unspecified" devices are RS-232 Adapters which had new XBee modules installed. The set xbee command is shown used to set new DD values, plus the WR command must be added to save the new DD value to NVRAM. The 0x03 in the upper word defines these correctly as Zigbee 2007, and the 0x0005 in the lower word defines them as RS-232 adapters.

Since the gateway does NOT expect the DD value of devices to change dynamically, you may need to reboot either the affected XBee nodes or the gateway to have the new device information show up. **Of course, setting the incorrect DD values will confuse and perhaps disable mesh applications which base functional and I/O expectations on the DD value!**

```
#> disp xbee

XBee network device list

PAN ID:           0x3261
Channel:          0x0d (2415 MHz)
Gateway address:  00:13:a2:00:40:4b:87:c7!

Node ID      Network Extended address         Product type
------------ ------- ------------------------ ------------
COORDINATOR
[0000]! 00:13:a2:00:40:4b:87:c7! X4 Gateway

ROUTERS
```

```
AN05          [5aea]! 00:13:a2:00:40:52:94:8b! Analog IO Adapter
AN06          [0c06]! 00:13:a2:00:40:52:94:b8! Analog IO Adapter
AN17          [7ad6]! 00:13:a2:00:40:52:94:ac! Analog IO Adapter
AN26          [0480]! 00:13:a2:00:40:52:94:ad! Analog IO Adapter
ANNA_ZN       [05e8]! 00:13:a2:00:40:52:94:a0! Unspecified
BELA_ZN       [794a]! 00:13:a2:00:40:34:16:20! Unspecified
CALI_ZN       [3c39]! 00:13:a2:00:40:34:16:4e! Unspecified
DEBI_ZN       [3921]! 00:13:a2:00:40:52:94:b2! Unspecified

#> set xbee address=00:13:a2:00:40:52:94:b2! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:34:16:20! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:52:94:a0! DD=0x30005 WR

#> set xbee address=00:13:a2:00:40:34:16:4e! DD=0x30005 WR

(Then after some time delay ... the types will refresh and show correctly)

#> disp xbee
...
ANNA_ZN       [05e8]! 00:13:a2:00:40:52:94:a0! RS-232 Adapter
DEBI_ZN       [3921]! 00:13:a2:00:40:52:94:b2! RS-232 Adapter
CALI_ZN       [3c39]! 00:13:a2:00:40:34:16:4e! RS-232 Adapter
BELA_ZN       [794a]! 00:13:a2:00:40:34:16:20! RS-232 Adapter
...
```

Note that the telnet "DD=" command assumes decimal unless the "0x" prefix is added. So setting "DD=30005" will instead set the DD value to 0x00007535. You need to use "DD=0x30005"

DD upper word : Module Type

16-bit	Description	FW / HW Mnemonics
0x0000	Unspecified	
0x0001	XBee 802.15.4 (Series 1)	XB24 (or XB24-A)
0x0002	XBee ZNet 2.5	XB24-B
0x0003	XBee ZB (Zigbee 2007)	XB24-ZB (or XB24-Z7)
0x0004	XBee DigiMesh 900 (900MHz)	XB09-DM
0x0005	XBee DigiMesh 2.4 (2.4GHz)	XB24-DM
0x0006	XBee 868 point to multi-point (868MHz for EU market)	XB08-DP
0x0007	XBee Point to Multi-point 900Mhz	XB09-DP
0x0008	XTend DigiMesh 900Mhz	
0x0009	XBee 802.11 Wifi	
0x000A	XBee ZB on S2C (surface mount)	
0x000B	XBee DigiMesh 900 S3B	
0x000C	XBee DigiMesh 868	

The PRO or higher wattage XBee modules use XBP instead of XB in firmware and hardware mnemonics. However the upper DD word is used to learn the general class of API command functionality within the firmware in a module, *thus do NOT attempt to assign hardware meaning to these values!*

Example code to convert these values to strings:

```
dd_upper_names = ['Bad_Code','ZB24-A','ZB24-B','XB24-ZB','XB09-DM','XB24-
DM','XB08-DP','XB09-DP']
def get_dd_upper_code_string( code):
    """Given upper/module-type word of the DD response as 16-bit integer, return
string"""
    try:
        return( dd_upper_names[code])
    except:
        return( dd_upper_names[0])
```

Important feature/limitation in the DD upper word

While some XBee technology force this value to be as expected, others allow the user to redefine the meaning. Thus you can never fully trust the DD value returned. For example, an OEM who produced products using ZNet 2.5 might load a value such as 0x00021234 into their product. After they start creating Zigbee 2007 or DigiMesh 2.4Mhz models, the SHOULD change the DD value to be 0x00031234 and 0x00051234 respectively - but they might not. ***Therefore it is safest to have your Python code use the upper-word of the gateway XBee to determine 'mesh/xbee' type, and only read the DD lower word of attached devices.***

DD lower word : Digi Product Type

0x0000	Unspecified
0x0001	ConnectPort X8 Gateway
0x0002	ConnectPort X4 Gateway
0x0003	ConnectPort X2 Gateway
0x0004	XBee Commissioning Tool
0x0005	XBee RS-232 Adapter
0x0006	XBee RS-485 Adapter
0x0007	XBee Sensor (1-wire) Adapter
0x0008	XBee Wall Router
0x0009	XBee RS-232PH (Power Harvesting) Adapter
0x000A	XBee Digital IO Adapter
0x000B	XBee Analog IO Adapter
0x000C	X-Stick
0x000D	XBee Sensor /L/T/H Adapter
0x000E	XBee Sensor /L/T Adapter

0x000F	Smart Plug
0x0010	USB Dongle
0x0011	LCD Display
0x0013	ConnectPort X5 Gateway
0x0014	Embedded Gateway
0x0015	ConnectPort X3 Gateway
0x0016	Net OS Device
0x0017	XG3 Gateway
0x0018	LTS Gateway
0x0019	CC3G Gateway
0x001A	X2 ULC Gateway
0xFF00-0xFFFF	Available for private customer use

Example code to convert these values to strings:

```
dd_lower_names = ['Unspecified','X8','X4','X2','XBee Commissioning
Tool','XBee232','XBee485',
    'XBee1W','XBee Wall Router', 'XBee232PH','XBeeDIO','XBeeAIO','X-Stick',
    'XBee /L/T/H','XBee /L/T','SmartPlug','USB Dongle','LCD Display','Undefined',
    'X5','Embbeded GWay','X3','NetOS',
          ]
def get_dd_lower_code_string( code):
    """Given lower/product word of the DD response as 16-bit integer, return
string"""
    try:
        return( dd_lower_names[code])
    except:
        return( dd_lower_names[0])
```

Currently assigned 'Private Customer Use' code

Note Although these are official, Digi is not rigidly enforcing use. Thus you may encounter other XBee users reusing these codes for other products.

Rabbit-Brand Products

0x0100	Generic Rabbit-Brand Product
0x0101	RCM4510W
0x0102	BL4S1xx
0x0103	BL4S230
0x01F0-0x01FF	Rabbit End-Customer Use

Non-Digi End Products

0x0201	Massa M3
0x0210	B&B Electronics LDVDS-XB
0x0220	SSI Embedded Systems PSU Sensor M
0x0221	SSI Embedded Systems PSU Sensor O
0x0231	PointSix Temperature Sensor
0x0240	Henny Penny 485 Adapter with PXBee
0x02B0-BF	Bejouled Solar Inverters
0x02C0-CF	Windpower
0x02D0-DF	RobustMesh, RS-232/485

If you have a device utilizing an XBee radio and desire a registered public DD value, please contact Digi support.

XBee RS-232 adapter

Product description

The XBee RS-232 Adapter provides short range wireless connectivity to any RS-232 serial device.

Note that this Wiki page documents the packaged box product, and not the RS-232 development bare board! It also covers operation when powered full-time, not when batteries are used and the product sleeps.

Assuming the serial end device is a traditional 'slave/server' which answers remote polls and is unaware of the mesh, then load the standard "AT" firmware and not the API firmware. Then use a Digi ConnectPort X gateway or another XBee module with the API firmware loaded to act as the 'master/client' and issue requests directly to end devices via MAC address, and the 'slave/server' XBee will always return responses to the MAC address of the 'master/client'.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

XBee Module Support

All Xbee modules should work in the RS-232 adapter, and they would use the standard AT or API versions. There are not (at this time) any special builds for this adapter. However, X-CTU cannot reflash firmware in this RS-232 module - you will need to move the XBee module to a USB or RS-232 development board to reflash firmware.

Module	Description	Tested	Comments	Output Defaults
XB24-A	802.15.4 on 2.4Ghz	Yes	No access to DTR or DSR	DTR and RTS asserted/high
XB24-B	ZNet 2.5 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB09-DM	DigiMesh on 900Mhz	Pending	No access to DSR	DTR is disabled/low, RTS asserted/high

Module	Description	Tested	Comments	Output Defaults
XB24-DM	DigiMesh on 2.4Ghz	Pending	No access to DTR or DSR	
XB08-DP	Point-to-Multipoint on 868Mhz	Pending		DTR is disabled/low, RTS asserted/high

Configuration to consider

These values can be set from X-CTU - or use the CLI or Web UI of a Digi ConnectPort X gateway:

- Set dest_addr (DH/DL) to the MAC of your CPX gateway or the XBee module which asts as master/client. This prevents broadcast responses.
- Set the baud rate and other port characteristics (requires X-CTU?).
- Set dio12_config (P2) to either 4 or 5; 4 (DO Low) causes DTR to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio7_config (D7) to either 4 or 5; 4 (DO Low) causes RTS to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio6_config (D6) to 3 to enable CTS as an input.
- Set dio3_config (D3) to 3 to enable CD as an input.
- Set dio1_config (D1) to 3 to enable RI as an input.

Note DSR cannot be read; there is no configuration required for it.

Pinouts

The RS-232 connector is an industry-standard DB9 male connector with a DTE configuration, similar to a PC serial port. However, there are limitations in the support for DTR/DSR. Pinouts for the connector are:

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
1	CD	Input	17	DIO3	D3	0x0008		
2	RXD	Input	3					
3	TXD	Output	2					
4	DTR	Output	4	DIO12	P2 (See Note)	0x1000	'P2\x04'	'P2\x05'
5	GND	---						
6	DSR	Input	9	DI8	Not Readable	Not Readable		
7	RTS	Output	12	DIO7	D7	0x0080	'D7\x04'	'D7\x05'

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
8	CTS	Input	16	DIO6	D6	0x0040		
9	RI	Input	19	DIO1	D1	0x0002		
9-alt	12vdc Switched Power	Output	18	DIO2	D2 (See Note)	0x0004	'D2\x05'	'D2\x04'

Note on Pin 4/DTR. Some XBee modules do not offer access to raise or lower DTR; the "P2" command is not available.

- XBee modules known **NOT** to support P2/DTR control: **XB24-A** (802.15.4), **XB24-DM**
- XBee modules known to **support** P2/DTR control: **XB24-B** (Znet2.5), **XB24-ZB**, **XB09-DM**

Note on Pin 6/DSR. None of the XBee module allow reading the level of DSR via the "IS" command.

Note on Pin 9. By default it is an input, however setting DIO2 high turns the 12vdc 50mA switched power output on and reading RI/DIO1 returns TRUE if power is on. Setting DIO2 low sets the switched power to tri-state, thus reading RI/DIO1 returns the RI-like status of pin 9. So do NOT connect RI to any device which might be damaged by a +12vdc signal - while it will NOT damage any true EIA/RS-232 compliant device, it can't be good for any device attempting to drive RI low.

Powering 'green' or 'port-powered' RS-232 devices

Some external devices (such as RFID readers or short-haul modems) attempt to draw power from the RS-232 driver circuit. The voltage output from the XBee 232 Adapter may be too low to power such external devices.

However, cross-wiring the RS-232 cable so that the 12vdc Aux-Power (pin 9) from the XBee 232 Adapter connects to the external DTE DSR input (pin 6 of DB9) or the DCE DTR input (pin 4 of DB9) provide a solid solution. A +12vdc signal is well within the RS-232 voltage signal specification, plus the approximately 12vdc 50mA supplied is more power than the 'Port Powered' device expects to tap.

Python programming examples

Polls or Requests sent to field devices: The 'master/client' XBee module should send serial data via addressed unicast with one of the API Transmit Request frames, such as 0x00, 0x01 and 0x10.

Responses or unsolicited data from field devices: If the dest_addr (DH/DL) registers have been set properly, then any serial data received from the field devices will be forwarded to the central 'master/client' XBee module. API Receive Packet frames will be received by the 'master/client' XBee module.

Driving RS-232 control signals

DTR/RTS signals are raised or lowered by sending a 3-byte command with the API Remote AT (command 0x17) - the examples below are coded as Python expects:

- To assert (or raise) the outgoing DTE signal DTR, send the AT command 'P2\x04'
- To deassert (or drop) the outgoing DTE signal DTR, send the AT command 'P2\x05'
- To assert (or raise) the outgoing DTE signal RTS, send the AT command 'D7\x04'
- To deassert (or drop) the outgoing DTE signal RTS, send the AT command 'D7\x05'

Note that these commands affect the pin within one second, yet do not save the state in FLASH. Thus a reboot of the XBee adapter puts the DTR or RTS signal back into the configured default; the factory default is low/not asserted. If it is desired to have DTR or RTS asserted upon power-up, manually set the DIO12/DIO7 parameters to 4.

Why does output DO = Low assert the RS-232 signal and DO = High drop the RS-232 signal? This is how historically TTL communications systems worked. A 5v line was assumed idle, thus pulled weakly up to 5v and representing OFF or binary zero (0). A 0v line was being actively shunted to ground by a powered transistor, thus represented ON or binary one (1). Even today, most RS-232/485 chips assume 0v = 1/on and 5v = 0/off.

Reading RS-232 control signals

To read the signal status, issue the 'IS' command. **You must DELAY at least one (1) second after issuing any of the D2/D7/P2 commands before issuing the 'IS' command or the output status won't be correctly returned in the response**. The IS command returns 6 bytes by direct API, and 5 bytes with ddo_get_param() function, so the last 5 bytes can be decoded as:

- Response[0] should equal '\x01' (is Sample Set Count)
- Digital_mask = (ord(response[1]) * 256) + ord(response[2]); shows which bits of following data are valid
- Response[3] should equal '\x00' (is Analog Data Mask - unless adapter voltage is being read)
- Digital_data = (ord(response[4]) * 256) + ord(response[5]); shows the actual I/O states

The RS-232 signals show as inverted for historical reasons, so a '0' means the signal is HIGH/ASSERTED and '1' means LOW/DROPPED

However, the Auxiliary Power Output (pin 9) shows up per digital logic levels. Therefore, a '0' means power is off and '1' means power is on. The Auxiliary Power Output can drive 12vdc at 50mA if the XBee RS-232 adapter is direct DC powered. If battery powered, driving Auxiliary Power Output more than a second will quickly drain your batteries!

Example Python to detect CTS status

```
def show_CTS_status( digital_mask, digital_data):
    """
    Decode and display the CTS info from 'IS' response words
    Return True if asserted, else False
    """
    if( digital_mask & 0x0040):
        print 'CTS input configured, value = ',
        if (digital_data & 0x0040):
            print 'Low/Dropped'
            return False
        else:
            print 'High/Asserted'
            return True
    else:
        print 'CTS input ignored (not configured)'
    return False
```

XBee RS-232 PH Adapter

Overview

The Digi Xbee RS-232 PH (Power Harvesting) Adapter is a wireless to RS-232 adapter which draws power from the attached RS-232 port, slowly charging a super-cap which functions as a short-term battery. It does not require sleeping, however it can only actively transmit from 5% to 20% of the time, which means it must be idle from 80% to 95% of the time to allow the super-cap to recharge between radio transmissions. Unfortunately the exact duty cycle of operation (or percentage of time sending/receving RF packets) is defined by the device to which the XBee RS-232 PH is connected.

In general the Digi Xbee RS-232 PH Adapter functions identically to the Digi XBee RS-232 Adapter, so refer to Digi XBee RS-232 adapter for most programming questions.

Features which are the same:

The same 9-pin DTE port, matching a standard PC computer including support for DTR/DSR, RTS/CTS, CD and RI

Features which are different:

- No support for the Auxillary Power Out on pin 9 / RI.
- Draws power from DTE inputs RXD, DSR, CTS, CD, RI; can be either high or low (either V+ or V-)
- Cannot 'talk' 100% of the time - a general rule of thumb is one poll-response per second.

XBEE sensors

Product description

The XBee Sensor provides real-time data on temperature, humidity, and light, with the data being transmitted through wireless communications in an XBee network infrastructure. Compact size and battery power enable XBee Sensors to be dropped into facilities easily and unobtrusively while providing reliable communications. Applications include building automation, environmental monitoring, security, asset monitoring, and more.

The readings are of modest accuracy, suitable for environmental monitor but not likely suitable for control systems (See the Accuracy Section below)

There are currently two XBee Sensor product options available:

- XBee Sensor /L/T: Integrated ambient light and temperature sensors
- XBee Sensor /L/T/H: Integrated ambient light, temperature, and humidity sensors

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Python Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

iDigi Dia configuration and programming examples

Python programming examples

Configuration settings

During system start up or configuration, you require:

- D1, D2 and D3 should be set to 2, enabling analog input to the 3 sensors.
- DH and DL should be set to your XBee coordinator (your CPX gateway) or the XBee device to receive the data productions.
- P1 (DIO11) should be set to 3, enabling digital input on the battery monitor pin.

Operational settings

Although you can poll the XBee Sensors, this requires the XBee to be awake most of the time and you battery life will be limited to a few months. You may find having it wake once per 10 or 15 minutes the best solution.

The assumed operation is to place the XBee Sensor into sleep, then have it send the data unsolicited to the XBee node listed in the DH/DL settings.

To enable sleeping operation longer than 1 minute:

- Set IR to 0xFFFF, which effectively disables the 'sample rate' setting.
- Set WH to 125 to allow the sensor hardware to stabilize for 125msec before the XBee reads the analog inputs (note: not all XBee modules support the WH parameter)
- Set the SN/SP pair to enable sleeping for the desired time period.
- SP is the number of 10msec periods to sleep
- SN is the number of SP-periods to sleep for.
- For example, SP=2000 and SP=30 leads to an approximately 600 seconds data productions (30 x 20 second periods or 10 minutes).
- *The SN/SP within your gateway and all XBee routers MUST be at least as large as the values placed into the XBee Sensor product, or you will find the routers (the parents) de-associate the sleeping XBee Sensor (the child) while it sleeps, and thus reject the periodic data productions.*

Formulas

Temperature:

```
temp_C = (mVanalog - 500.0)/ 10.0
mVanalog = (ADC2/1023.0) * 1200
```

Humidity:

```
hum = (((mVanalog * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
mVanalog = (ADC3/1023.0) * 1200
```

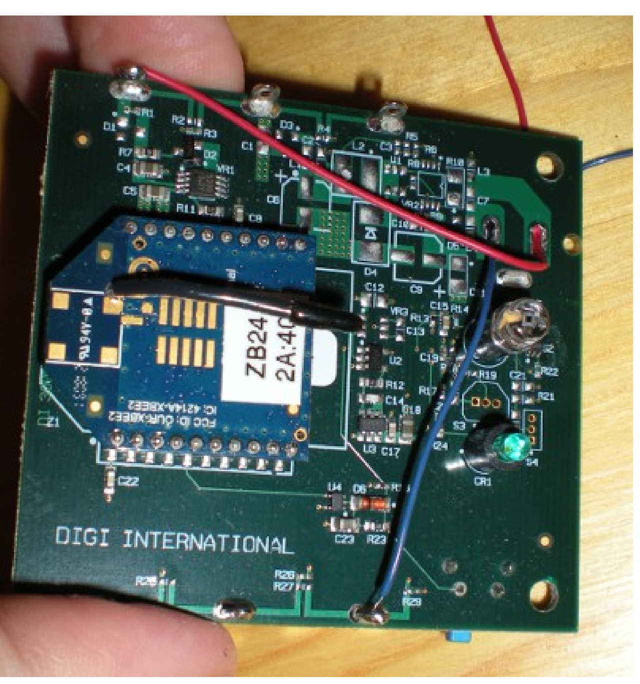
Light:

```
brightness = (ADC1) /1023.0) * 1200
```

Hardware information

Power options

Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.6v up to 6.0v. For testing purposes you can use a stable 5vdc supply, attaching your power leads to the battery tabs as shown in the photo (red is +, blue is -). The wires pass through, but do not connect to the barrel jack socket-holes. This jack is only connected electronically in the 9-30vdc version of this PCB (which is not available).



XBee module support

The XBee L/T/H/ Sensor requires either the 'End Device AT' or 'Router AT' firmware on the XBee. Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.7v up to 6.0v. For testing purposes, you can use a stable 5VDC supply.

Given the need to be low power and to sleep, not all XBee modules are appropriate for this product.

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-A	802.15.4 on 2.4Ghz	Pending				
XB24-B	ZNet 2.5 on 2.4Ghz	Yes		ZNet 2.5 Router/End Device AT, such as 1247	0x2000E	0x2000D

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes		Zigbee End Device AT, such as 2864	0x3000E	0x3000D
XB09-DM	DigiMesh on 900Mhz	Pending				
XB24-DM	DigiMesh on 2.4Ghz	Pending				
XB08-DP	Point-to-Multipoint on 868Mhz	No	Consumes too much power	N/A	N/A	
XB09-DP	Point-to-Multipoint on 900Mhz	Pending				

Notes:

Although the XBee XB24-ZB family includes a firmware named "Zigbee Router/End Device Sensor", that firmware is for the 1-wire Sensor Adapter - **it is NOT for the Sensor /L/T or /L/T/H**. Installing this firmware will result in bad values being read.

Accuracy

XBee L-T-H sensor adapter

The /L/T/H product is NOT calibrated by Digi. This means multiple units placed side-by-side out of the box will show a higher than desired variability. However, the sensor readings are fairly linear and modest software calibration can greatly improve the accuracy.

Temperature

The stated accuracy per the datasheet (and component supplier) is +/- 2 DegC (+/- 3.6 DegF).

They are not designed for industrial control, and even the temperature within a normal room varies by many degrees based on drafts, heat or cold sources, and how far the sensor is above the floor. Anyone who looks at a thermostat, reads the '72' and believes the entire room is a perfectly constant 72 degrees Fahrenheit is being foolish.

Ad-hoc tests show that a simple fixed offset added or subtracted to the readings allow them to be used in normal building automation with satisfactory result. For example, a test of four units showed that adding constants to the raw 0-1023 value allowed all four to return the same temperature to within +/- 0.25 DegC most of the time and to within +/- 0.75 DegC all of the time. The magnitude of these binary constants within this test of four units were (-4, 14, -15 and 5). You could use a float constant scaled as DegC or DegF instead.

These constants were calculated by taking 5 readings over five hours, then examining the average deviation from the desired value. For example, one unit returned the values 739, 735, 734, 700 and 702 when the expected values were 738, 732, 732, 690, and 694. Thus adding a -4 (subtracting 4) from the value received resulted in a better value. The expected value was calculated based on the temperature reading of a third party device 'trusted' as correct.

Humidity

The stated accuracy per the datasheet (and component supplier) is:

Interchangeability: +/- 5 %RH (0%RH to 59%RH) and +/- 8 %RH (60%RH to 100%RH)

Accuracy: +/- 3.5%RH

This means if you take a dozen factory-fresh units, allow them to stabilize within a 70%RH environment, then you may see readings range from 62%RH to 78%RH. This is the ‘Interchangeability’ clause.

However, if the user does modest linear software calibration (primarily fixed offset), then the same dozen devices can show the 70%RH as 66.5%RH to 73.5%RH. This is the ‘Accuracy’ clause.

Light

The light sensor in the adapter returns values from 0 to about 1100 based on light intensity - it does NOT return any standard measure, and the actual readings vary based on the opacity of the label applied and so on. It can be used to easily sense that a room is brighter or darker. For example, you can use it to turn security lights on when the sun goes down - or to turn off room heaters and computer displays when the overhead lights are turned off, indicating that the room is not occupied.

Note These are simple, low-cost "light intensity" sensors that are intended to be used for a wide variety of applications. However, they were never intended to measure the lux of a particular scene. For this reason, we're not able to provide a formula to convert light intensity to lux.

You could not use it (for example) to test that a workbench has an exact luminous intensity of "X" cd/m2. If you require such measurements, you should use a standard light sensor with a 0-10v or 4-20mA signal in a AIO adpater.

Also note that the duration of the actual light detection is short, so it works best with either sun light or non-flickering incandescent or DC-powered halogen lamps. Measure light with standard fluorescent light will result in the light value varying over a range of values (many percentage points). This does not prevent your system from detection 'the room is light' or 'the room is dark', but you will need to accommodate this variation with a hysteresis or dead-band calculation.

Digi Transport Products

The following pages cover the Digi Enterprise Routers and VPN Concentrators, including: Digi TransPort® WR Family

Expansion card Python options

Purpose

This section will list out the various options that the expansion cards for the TransPort's are capable of using. These are the generic options that are available with no real Python code around them. Later sections of this guide will incorporate various pieces of these code snippets in real world examples.

Note All of the operations that follow this note will need the 'digihw' module imported into the Python code for these operations to work properly. There are also individual guides for the expansion cards that list more details on what the cards can do that can be referred to.

Telemetry 1 Card

```
            #Digital Input
digihw.get_din(port)   #Port = 0; only 1 port exists
#Digital Output
digihw.set_dout(port, value)  #Port = 0-3; value = 0-1 (off/on)
#Relay Port
digihw.set_relay(port, value)  #Port = 0; value = 0-1 (off/on)
#Voltage Monitor - ASY 1 must be at 9600 baud for this to work
digihw.voltage_monitor()
#Temp Monitor - ASY 1 must be at 9600 baud for this to work
digihw.temperature()
```

Telemetry 2 Card

```
            #Digital I/O
digihw.gpio_set_input(port)  # Port = 1-4
digihw.gpio_set_value(port, <0|1>)  #Port = 1-4; 0 = off, 1 = on
digihw.gpio_get_value(port)  #Gets the current status of the port

#Analog I/O
digihw.aio_set_tx_loop(port, <on|off>)
digihw.aio_set_rx_loop(port, <on|off>)
digihw.aio_get_value(port)
digihw.aio_set_dac(port, value)
digihw.aio_set_ma(port, value)
```

GPS Card

```
            #GPS Location
digihw.gps_location()
```

Fleet Card

```
            #Accelerometer
digihw.accelerometer

#Digital I/O
digihw.gpio_set_input(port)
digihw.gpio_set_value(port, <0|1>)
digihw.gpio_get_value(port)

#Ignition Sense
```

```
digihw.ignition_sense()

#GPS Location
digihw.gps_location()
```

WR44/41 DC Power I/O Ports

```
            #Digital Input
digihw.wr44_gpio_get_value(port)

#Digital I/O
digihw.wr44_gpio_set_input(port)
digihw.wr44_gpio_set_value(port, <0|1>)
digihw.wr44_gpio_get_value(port)
```

FTP client

Purpose

This section will go through an example of using the TransPort as a FTP Client to retrieve a file via FTP to some FTP Server.

```
import ftplib
from ftplib import FTP

def FTPClient():
    ftp = FTP('IP_ADDRESSS_OF_SERVER')
    ftp.connect('IP_ADDRESS_OF_SERVER', '5000')
    ftp.login('username', ‘password’)
    ftp.retrlines('LIST')
    ftp.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write)
    ftp.quit()

FTPClient()
```

Important notes

1. Make sure to edit the IP address of the server so the code knows which IP to connect to.
2. For “anonymous” logins, leave the username and password blank.
3. Edit the “filename.txt” items above to match the actual file name of the file that needs to be FTP’ed onto the TransPort from the program.

Code breakdown

```
import ftplib
from ftplib import FTP
```

This section is importing the ‘ftplib’ to use the FTP library, and from ‘ftplib’ we are specifically wanting “FTP”.

```
def FTPClient():
    ftp = FTP('IP_ADDRESSS_OF_SERVER')
    ftp.connect('IP_ADDRESS_OF_SERVER', '5000')
    ftp.login('username', ‘password’)
    ftp.retrlines('LIST')
    ftp.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write)
    ftp.quit()
```

1. First, we define the program itself. (def FTPClient():)
2. A variable called “ftp” is created with the values to FTP into the Server IP address. (ftp = FTP ('IP_ADDRESS_OF_SERVER'))

3. Using the ‘ftp’ variable, connect to the IP address of the server at the given port number. (ftp.connect('IP_ADDRESS_OF_SERVER', '5000'))

   NOTE: Number 3 above is an optional line. It can be removed if the default port is being used (21).

4. Using the ‘ftp’ variable, perform a login of the server with the given username and password. (ftp.login('username', 'password'))
5. Using the ‘ftp’ variable, “LIST” the contents of the FTP server. (ftp.retrlines('LIST'))
6. Using the ‘ftp’ variable, return the file name we are asking for, open the file, and write it to the TransPort. (ftp.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write))
7. The last line quits the FTP operation. (ftp.quit())

FTPClient()

This last portion of the code tells it to run the definition that was previously created.

SFTP Client

Purpose

This section will go through an example of using the TransPort as a SFTP Client to retrieve a file via SFTP to some SFTP Server.

```
import ftplib_d
from ftplib_d import FTP_TLS

def SFTPClient():
    ftps = FTP_TLS('IP_ADDRESS_OF_SERVER')
    ftps.login('username', 'password')
    ftps.prot_p()
    ftps.retrlines('LIST')
    ftps.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write)
    ftps.quit()

SFTPClient()
```

Note “ftplib_d” is the Python 2.7 "ftplib" Python file. Python 2.6 on the TransPort does not support SFTP on its own. This file must be loaded on the TransPort before this script will run properly.

Code Breakdown

```
import ftplib_d
from ftplib_d import FTP_TLS
```

Here the code is importing the “ftplib_d” module and from that module, the function “FTP_TLS”. As mentioned in the note above, “ftplib_d” is just the Python 2.7 version of the “ftplib” library that is already on the TransPort, but the Python implementation of the TransPort (Version 2.6) does not support this function, hence the need for the 2.7 library instead. This file can be obtained from www.Python.org and just rename it to something like “ftplib_d” to call it in your code.

```
def SFTPClient():
    ftps = FTP_TLS('IP_ADDRESS_OF_SERVER')
    ftps.login('username', 'password')
    ftps.prot_p()
    ftps.retrlines('LIST')
    ftps.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write)
    ftps.quit()
```

1. First, we define the program itself. (def SFTPClient():)
2. A variable called “ftps” is created with the values to SFTP into the Server IP address. (ftps = FTP_TLS('IP_ADDRESS_OF_SERVER'))
3. Using the ‘ftps’ variable, perform a login of the server with the given username and password. (ftp.login('username', 'password')).
4. Using the ‘ftps’ variable, switch the mode into a secure communications mode. (ftps.prot_p())
5. Using the ‘ftps’ variable, “LIST” the contents of the FTP server. (ftp.retrlines('LIST'))

6. Using the ‘ftps’ variable, return the file name we are asking for, open the file, and write it to the TransPort. (ftps.retrbinary('RETR filename.txt', open('filename.txt', 'wb').write))
7. The last line quits the FTP operation. (ftps.quit())

```
SFTPClient()
```

This last portion of the code tells it to run the definition that was previously created.

SMS transport

Purpose

This SMS example will go over the very basics of how to send an SMS message via Python on the TransPort product family. Below is the sample code that will be discussed:

Source code

```
import sarcli, time


# Edit the phone number to match the destination the SMS should go to
destination = '19525551212'
# Edit the text to send in the SMS message
message = 'This is the text to send'
# Edit the number of seconds the script should sleep before running again
sleep = 10

#Do not edit anything below this line.
def SendSms(destination, message):
        cli = sarcli.open()
        command = 'sendsms ' + destination + ' "' + message + '" '
        cli.write(command)
        cli.close()

#Main loop

while True:
    time.sleep(sleep)
    print "Sleeping"
    SendSms(destination, message)
    print "Sent SMS message: " + message
```

Code breakdown

This code is importing two Python modules to use: “sarcli” and “time”. These modules are needed for: a) “sarcli” is needed to send the SMS message using the CLI of the TransPort, and b) “time” is needed to allow the “While” loop to sleep instead of sending never ending messages over and over again.

```
import sarcli, time
```

The “import” command is telling the Python environment to import specific modules or parts of modules to use later in the code. If these are missed, the rest of the code will not work.

```
# Edit the phone number to match the destination the SMS should go to
destination = '19525551212'
message = 'This is the text to send'
sleep = 10
```

This section is declaring 3 variables to use in the code: “destination”, “message”, and “sleep”. By creating these variables, it makes it easier to edit portions of the code by just having to edit the

variable once and not in every place it shows up in the code. These should be edited to match the end usage of the code.

```
#Do not edit anything below this line.
 def SendSms(destination, message):
        cli = sarcli.open()
        command = 'sendsms ' + destination + ' "' + message + '" '
        cli.write(command)
        cli.close()
```

This section is defining the ”SendSms” operation and what it will do once called by the Python environment. In this case, we are telling the code that “SendSms” will do the following in this order:

1. Declare a variable called “cli” and assign the value of “sarcli.open()” to this variable. This allows the “sarcli” module to open the CLI of the TransPort.
2. Declare a 2nd variable called “command” and assign the actual CLI command to send to the CLI of the TransPort. The CLI command in this case will be sendsms 19525551212 “This is the message to send” based on the variables that are being called out in the code.
3. Using the “cli” variable, the code will now write the variable called “command” to the CLI.
4. The “cli” variable will close the CLI once it is done writing the command.

```
#Main loop

while True:
    time.sleep(sleep)
    print "Sleeping"
    SendSms(destination, message)
    print "Sent SMS message: " + message
```

This is the main “While” loop that will run the code. While the state is “true” for this loop, it will call the module “time”, use the “sleep” option, and sleep the loop for the given variable of “sleep” that was declared earlier in the code. Once the sleep cycle ends, a message will “print” to a terminal window to the CLI of the TransPort (if open) the word “Sleeping”, immediately execute the “SendSms” definition that was established erailer, and then print another message of the terminal window of “Send SMS Message: This is the message to send”. Once it sends the SMS, the sleep cycle starts again and the process continues until told to stop.

Serial port transport

Purpose

This example will go over the basics of how to read and write data to the serial port of the TransPort via Python. The example will take data it receives on the serial port and send it back out the same port it was received on. Below is the sample code:

```
import sys, select

fr = open("asy/00", "r")
fw = open("asy/00", "w")


input = [fr]

while 1:
    inputready,outputready,exceptready = select.select(input,[],[], 1.1)

    for d in inputready:
            if d == fr:
            data = d.read()
            if data:
             fw.write(data) #Echo the data that was read
             fw.flush() #Send the data to the serial port

    print("tick")
```

Code breakdown

```
import sys, select
```

This portion of code is importing two Python modules to use: “sys” and “select”. The “sys” module is needed for some of the various parts of the code to work properly (don’t ask me which ones), and the “select” module is needed for the select operation that happens later in the code.

```
fr = open("asy/00", "r")
fw = open("asy/00", "w")

input = [fr]
```

This section is declaring two variables to use: “fr” and “fw”, and assigning those variables values that the TransPort understands for reading and writing data to the serial port at a system level. I'm not sure what the “input = [fr]” is used for but is necessary.

```
while 1:
    inputready,outputready,exceptready = select.select(input,[],[], 1.1)

    for d in inputready:
            if d == fr:
             data = d.read()
            if data:
             fw.write(data) #Echo the data that was read
             fw.flush() #Send the data to the serial port
```

```
    print("tick")
```

This is the main “While” loop of the application :

1. The for d in inputready:‘ line, the code is declaring a variable called “d” inside of the “inputready” variable that was created in the previous line.
2. The ‘if d == fr:’ is to tell the script to look for the ‘d’ variable to be equal to (==) ‘fr’ for the loop to continue.
3. If the script moved to the next line of ‘data = d.read()’, a new variable called ‘data’ is declared and is given the value of ‘d.read()’ to tell the script to read the data of the ‘d’ variable.
4. If there is data on the serial line, ‘if data:’ line will then trigger, and execute ‘fw.write(data)’ to echo the data that was read, and then use ‘fw.flush()’ to write the data to the serial port.
5. The script will then print the word ‘tick’ to a terminal window to indicate that it that it has successfully pushed data to the serial port.

Serial data SMS on Transport

Purpose

This example will go over combining the two previous examples together to make more of a real world application example. This code will look for specific messages incoming on the serial port of the TransPort, and will then send a SMS message to a specific destination based on the text it received. See the sample code below:

```
import sys, select, sarcli, time

#Setting up the structure for sending SMS messages
def SendSms(destination, message):
    cli = sarcli.open()
    command = 'sendsms ' + destination + ' " ' + message + ' " '
    cli.write(command)
    cli.close()

# Edit the phone number(s) to match the destination to send messages to and the
message
destination = '19525551212'
destination2 = '19525551212'

# Configure the various messages to SMS based off of specific ASCII strings
message = "Message 1"
message2 = "Message 2"
message3 = "Message 3"

# Edit the ASCII strings you want to search for on the serial port
data1 = "ASCII"
data2 = "ASCII2"
data3 = "ASCII3 3"

# Variables for reading the serial port - DO NOT EDIT
fr = open("ASY/00", "r")

input = [fr]

# ASY port read and SMS based off variables
while 1:
    time.sleep(0.1)
    inputready,outputready,exceptready = select.select(input,[],[], 1.1)

    for d in inputready:
        if d == fr:
            data = d.read()
            # This line will match "data1" from above and send a SMS to
            # the "destionation" and using  "message" from above.
            if (data == data1):
                SendSms(destination, message)
                print "Sending Message 1"
            elif (data == data2):
                SendSms(destination2, message2)
                print "Sending Message 2"
            # This example shows sending to multiple destinations simultaneously.
            elif (data == data3):
```

```
                SendSms(destination, message3)
                SendSms(destination2, message)
                print "Sending Message 3"
```

Code breakdown

```
import sys, select, sarcli, time
```

The code is importing all 4 of the modules from the previous 2 examples.

```
            #Setting up the structure for sending SMS messages
def SendSms(destination, message):
    cli = sarcli.open()
    command = 'sendsms ' + destination + ' " ' + message + ' " '
    cli.write(command)
    cli.close()
```

The code is defining the ‘destinations’ to send the SMS messages to, and ‘messages’ that will be sent out.

```
# Edit the ASCII strings you want to search for on the serial port
data1 = "ASCII"
data2 = "ASCII2"
data3 = "ASCII3 3"
```

This is the section that will define what strings of text the Python code is listening for that will trigger the SMS messages to be sent out. In this example, the code is looking for “ASCII”, “ASCII2”, and “ASCII3 3” to trigger sending off the various SMS messages.

```
# Variables for reading the serial port - DO NOT EDIT
fr = open("ASY/00", "r")

input = [fr]
```

The same variables as the ‘serial read/write’ example, with only the ‘read’ portion being used.

```
            # ASY port read and SMS based off variables
while 1:
    time.sleep(0.1)
    inputready,outputready,exceptready = select.select(input,[],[], 1.1)

    for d in inputready:
        if d == fr:
            data = d.read()
            # This line will match "data1" from above and send a SMS to
            # the "destination" and using  "message" from above.
            if (data == data1):
                SendSms(destination, message)
                print "Sending Message 1"
            elif (data == data2):
                SendSms(destination2, message2)
                print "Sending Message 2"
            # This example shows sending to multiple destinations simultaneously.
            elif (data == data3):
                SendSms(destination, message3)
```

```
                SendSms(destination2, message)
                print "Sending Message 3"
```

This is the main ‘while’ loop for the code. Here is what is effectively happening:

1. The loop will sleep for 0.1 seconds before reading the serial port with ‘time.sleep(0.1)’.
2. The same input items from the ‘read/write’ example are used to read all data coming in from the serial port until the ‘if’ statements are executed in the code.
3. Once the code gets to the first ‘if’ statement, it will use the variable of ‘data’ and see if the information of that variable matches what was declared as the ‘data1’ variable earlier on in the code. As long as that data matches, the code will send a SMS to the phone number declared in ‘destination’ variable and the message declared in the ‘message’ variable.
4. If ‘data’ was not equal to ‘data1’, the code will move down to the ‘elif (data == data2):’ line and see if ‘data’ matches the variable set for ‘data2’, and if it does, it sends a SMS message to the ‘destination2’ variable with the ‘message2’ variable as the message.
5. If ‘data’ was not equal to ‘data2’, the code will mode down to the last section of ‘elif (data == data3):‘ and look to see if ‘data’ is equal to ‘data3’. If they match, the code will send a SMS message to ‘destination’ with the ‘message3’ variable, and also send a message to ‘destination2’ with the ‘message’ variable. 6) If no conditions are met (or if conditions are met), the script sleeps for another ‘0.1’ seconds and reads incoming data on the serial port again, looking for the same conditions to send SMS messages based off of.

TCP server loopback transport

Purpose

This example will go over setting up a generic TCP Server within Python, and echoing out the received data back to the host that sent it.

Note This is not a good sample to run if you are going to send lots of data to the transport. This is because it's going to create a TCP socket for everytime it receives data. This will create a lot of overhead.

```
import socket

HOST = ''                    # Symbolic name meaning the local host
PORT = 5000                  # The port used by the server

def main():
    #Main While loop to allow the code to restart once the socket closes
    while 1:
        # Create the server socket.
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((HOST, PORT))
        s.listen(1)

        conn, addr = s.accept()
        print "Client %s connected" %str(addr)

        # Read data from the client and return it back until the client closes
        # the connection or it sends the 'quit' command.
        while 1:
            try:
                data = conn.recv(1024)
                if data == "quit" or data == "":
                    print "Quit condition met"
                    break
                print "Client %s sent data: %s" %(str(addr), data)
                conn.send(data)
            except:
                break

        print "Client %s disconnected" %str(addr)
        conn.close()
        s.close()


if __name__ == '__main__':
    main()
```

Code breakdown

```
import socket
```

Here the code is importing the module called "socket" which is needed to use TCP and UDP sockets within Python code.

```
HOST = ''                  # Symbolic name meaning the local host
PORT = 5000                # The port used by the server
```

Here we are declaring 2 variables: “HOST” and “PORT”. The value for “HOST” in the case is ‘’, which is symbolic for using the local host (loopback address – 127.0.0.1) as the IP. The “PORT” value will be the port number we want the code to listen in on for the incoming TCP connection.

```
def main():
    #Main While loop to allow the code to restart once the socket closes
    while 1:
        # Create the server socket.
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((HOST, PORT))
        s.listen(1)

        conn, addr = s.accept()
        print "Client %s connected" %str(addr)

        # Read data from the client and return it back until the client closes
        # the connection or it sends the 'quit' command.
        while 1:
            try:
                data = conn.recv(1024)
                if data == "quit" or data == "":
                    print "Quit condition met"
                    break
                print "Client %s sent data: %s" %(str(addr), data)
                conn.send(data)
            except:
                break

        print "Client %s disconnected" %str(addr)
        conn.close()
        s.close()
```

This is the main program definition that contains 2 different ‘while’ loops to run the code properly (def main():). Let’s break this down into steps:

1. The first while loop (while 1:) is creating a variable called ‘s’ for the incoming TCP socket (s = socket.socket(socket.AF_INET, socket.SOCK_STREAM). The “SOCK_STREAM” is for TCP where “SOCK_DGRAM” is for UDP.
2. The code then binds the ‘s’ variable to the “HOST” and “PORT” values that were declared earlier (s.bind((HOST, PORT))). In this case, “HOST” is the local host IP and “PORT” is 5000.
3. The next step is to allow the ‘s’ variable to listen for incoming socket connections, and we are only allowing 1 connection at a time (s.listen(1)).
4. The code then needs to accept the incoming connection (conn, addr = s.accept()) and prints this to a terminal screen as a status message (print "Client %s connected" %str(addr)).
5. Once a connection is established on port 5000 from a client, the 2nd while loop triggers (while 1:). The code will “try” (try:)to listen for any incoming data with a buffer up to 1024 bytes (data = conn.recv(1024)).
6. If that data is equal to ‘quit’ or blank data (if data == "quit" or data == "":), the code should print a status message (print "Quit condition met") and break out of the while loop (break).

7. If data is anything else, a message will print that the client had sent data and include the actual data sent (print "Client %s sent data: %s" %(str(addr), data)) and then actually send the data back to the same client (conn.send(data)).
8. 8) The last potion of the 2nd while loop is an exception (except:) to break (break) out of the while loop if an exception is made, such as closing the socket connection.
9. 9) Once a disconnection happens, it will print a message indicating so with the IP of the client (print "Client %s disconnected" %str(addr)), close the connection to the other side (conn.close ()), and finally close the socket (s.close()).
10. 10) Once the socket closes, the 1st while loop should trigger again and allow for a new incoming TCP socket connection on port 5000.

Telemetry 2 card digital in-to-multiple SMS example

Purpose

This section will go through an example of using the Telemetry 2 I/O card in a WR44 to watch for several Digital In pins to go high, and once it does, send an SMS message to the given phone numbers. This example will not send a second message indicating the signal went low like the previous Telemetry 1 example. Below is the code sample:

```
import sarcli, digihw, time

destination = '19525551212'
destination2 = '16515551212'

def SendSms(destination, message):
    cli = sarcli.open()
    command = 'sendsms ' + destination + ' " ' + message + ' " '
    cli.write(command)
    cli.close()

oldstateDIN = digihw.gpio_get_value(0)
print "initial value 1: " + str(oldstateDIN)
oldstateDIN2 = digihw.gpio_get_value(1)
print "initial value 2: " + str(oldstateDIN2)
oldstateDIN3 = digihw.gpio_get_value(2)
print "initial value 3: " + str(oldstateDIN3)

while True:
    print "Sleeping"
    time.sleep(10)
    print "Awake"
    stateDIN = digihw.gpio_get_value(0)
    print "Current state 1: " + str(stateDIN)
    print "Current old state 1: " + str(oldstateDIN)
    stateDIN2 = digihw.gpio_get_value(1)
    print "Current state 2: " + str(stateDIN2)
    print "Current old state 2: " + str(oldstateDIN2)
    stateDIN3 = digihw.gpio_get_value(2)
    print "Current state 3: " + str(stateDIN3)
    print "Current old state 3: " + str(oldstateDIN3)

    if not stateDIN == oldstateDIN:
        oldstateDIN = stateDIN
        if oldstateDIN == 1:
            message = 'Digital input state for port 1 is: ' + str(oldstateDIN)
            print message
            SendSms(destination, message)
            print "Sent SMS 1"
        elif oldstateDIN == 0:
            pass
            print "State is 0 on port 1 so not sending a message"

    if not stateDIN2 == oldstateDIN2:
        oldstateDIN2 = stateDIN2
        if oldstateDIN2 == 1:
            message = 'Digital input state for port 2 is: ' + str(oldstateDIN2)
```

```
            print message
            SendSms(destination2, message)
            print "Sent SMS 2"
        elif oldstateDIN2 == 0:
            pass
            print "State is 0 port 2 so not sending a message"

    if not stateDIN3 == oldstateDIN3:
        oldstateDIN3 = stateDIN3
        if oldstateDIN3 == 1:
            message = 'Digital input state for port 3 is: ' + str(oldstateDIN3)
            print message
            SendSms(destination, message)
            SendSms(destination2, message
            print "Sent SMS 3"
        elif oldstateDIN3 == 0:
            pass
            print "State is 0 on port 3 so not sending a message"
```

Code breakdown

```
import sarcli, digihw, time
```

Here the code is importing the modules needed.

```
destination = '19525551212'
destination2 = '16515551212'
```

Here we are declaring 2 variables, ‘destination’ and ‘destination2’, both of which have phone numbers associated with them. These are the numbers that will receive the SMS messages.

```
def SendSms(destination, message):
    cli = sarcli.open()
    command = 'sendsms ' + destination + ' " ' + message + ' " '
    cli.write(command)
    cli.close()
```

Here we define the “SendSms” function that will be used to send the SMS messages. See the SMS section for more details, if necessary.

```
oldstateDIN = digihw.gpio_get_value(0)
print "initial value 1: " + str(oldstateDIN)
oldstateDIN2 = digihw.gpio_get_value(1)
print "initial value 2: " + str(oldstateDIN2)
oldstateDIN3 = digihw.gpio_get_value(2)
print "initial value 3: " + str(oldstateDIN3)
```

Here the code is declaring 3 new variables that will be used: “oldstateDIN”, “oldstateDIN2”, and “oldstateDIN3”. Each of these variables is polling the initial state of the respective Digital In line on the Telemetry 2 card, and then prints that value to a terminal screen.

Let’s break the “while” loop into a few sections:

```
while True:
    print "Sleeping"
    time.sleep(10)
    print "Awake"
```

```
    stateDIN = digihw.gpio_get_value(0)
    print "Current state 1: " + str(stateDIN)
    print "Current old state 1: " + str(oldstateDIN)
    stateDIN2 = digihw.gpio_get_value(1)
    print "Current state 2: " + str(stateDIN2)
    print "Current old state 2: " + str(oldstateDIN2)
    stateDIN3 = digihw.gpio_get_value(2)
    print "Current state 3: " + str(stateDIN3)
    print "Current old state 3: " + str(oldstateDIN3)
```

This is the beginning if the while loop, where the script will first sleep for x seconds (10 in this case). Once it wakes up, 3 new variables are declared: “stateDIN”, “stateDIN2”, and “stateDIN3”, each of which are being assigned the current value of the Digital In port they are associated with. The code then prints the old state of the port since the last sleep cycle, and the new state of the port that it just read.

```
    if not stateDIN == oldstateDIN:
        oldstateDIN = stateDIN
        if oldstateDIN == 1:
            message = 'Digital input state for port 1 is: ' + str(oldstateDIN)
            print message
            SendSms(destination, message)
            print "Sent SMS 1"
        elif oldstateDIN == 0:
            pass
            print "State is 0 on port 1 so not sending a message"
```

Here the code performs a series of if/then statements to check each of the 3 ports. This section is from the first port only. It first checks to see if the new state is not equal to the old state (if not stateDIN == oldstateDIN:). If that is true, the old state variable is then assigned the value of the new state (oldstateDIN = stateDIN). If the old state (now equal to the current new state) is equal to ‘1’ (if oldstateDIN == 1:), declare a new variable called “message” that contains the message to send (message = 'Digital input state for port 1 is: ' + str(oldstateDIN)), print the message to the screen (print message), and then send the SMS message to “destination” (SendSms(destination, message)), and print out that it did so to the screen (print "Sent SMS 1"). If the state is equal to 0 instead (elif oldstateDIN == 0:), ignore the code (pass), and print the message indicating it ignored this to the screen (print "State is 0 on port 1 so not sending a message").

```
    if not stateDIN2 == oldstateDIN2:
        oldstateDIN2 = stateDIN2
        if oldstateDIN2 == 1:
            message = 'Digital input state for port 2 is: ' + str(oldstateDIN2)
            print message
            SendSms(destination2, message)
            print "Sent SMS 2"
        elif oldstateDIN2 == 0:
            pass
            print "State is 0 port 2 so not sending a message"
```

This section works just like the previous section, except it is looking at DIN port 2, and will send the SMS message to “destination2” instead.

```
    if not stateDIN3 == oldstateDIN3:
        oldstateDIN3 = stateDIN3
        if oldstateDIN3 == 1:
            message = 'Digital input state for port 3 is: ' + str(oldstateDIN3)
            print message
```

```
            SendSms(destination, message)
            SendSms(destination2, message
            print "Sent SMS 3"
        elif oldstateDIN3 == 0:
            pass
            print "State is 0 on port 3 so not sending a message"
```

This last section will perform the same operations as the previous 2 sections, except it is looking at DIN port 3, and will send a SMS to both “destination” and “destination2” instead.

Transport Python programmer's guide

Purpose of this guide

This guide introduces the Python programming language by showing how to create and run a simple Python program. It describes how to load and run Python programs onto Digi Transport devices, either through the command-line or Web user interfaces. It reviews Python modules, particularly those modules with Digi-specific behavior. This guide describes how to run the executable programs and describes program files.

What Is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

For more information on the Python Programming Language, go to http://www.Python.org/ and click the Documentation link.

The Transports use Python version 2.6.1.

Running Python

How to run a Python program on a Digi Transport router. Firstly check your firmware version on the router, we recommend using firmware version 5090 or later. Start by checking your router has Python in its firmware by following these simple steps:

Step 1: Using either a telnet or serial connection (default login/password = username/password) to the router issue the following commands

```
pycfg 0 stderr2stdout on
Python
```

Step 2: At the prompt now type the following command:

```
help()
```

The following should then be displayed:

```
Welcome to Python 2.6!  This is the online help utility.
```

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.Python.org/tutorial/.

Step 3: Type the following to exit the Python interpreter exit() Now you can upload your Python script to the router via FTP. For testing purposes you can simply run the script by using the following command:

```
Python filename.py
```

For the router to start the script automatically on Powerup / Reboot issue the following commands:

```
cmd 0 autocmd "Python filename.py"
config 0 save
```

First Python script

Lets start with a hello world program. Firstly create a text file with the following text inside:

```
print "Hello World!"
```

Save this file with a file name of myfirst.py Now FTP the myfirst.py text file onto the router and issue the following command: Python myfirst.py

The router should produce the following output:

```
OK
Hello World!
```

Miscellaneous items

Your Python code can detect when it is running on a Transport product by importing the SYS module, then testing the sys.platform variable like this:

```
if sys.platform == 'digiSarOS':
   print "Running on Digi Transport"
```

File names on the Transport are limited to 8 characters or less. There is also a 3 character limit on file type extensions. Example:

```
  myfile.py  - is valid (6 characters and 2 character file type extension)
  myLongFileName.py - is not valid
  myfile.superlongextension - is not valid (6 characters are ok but we have more
than 3 characters on the filetype extension)
```

Also, please note that the Digi ESP, which is used as a code development tool, is not aware of this limitation. That means the programmer must make sure to use short filenames on their projects in the ESP.

Python examples can be generated on the Digi ESP code development tool (Digi ESP -> file -> new -> digi Python application sample project). This tool can be downloaded from the Digi website.

RCI is a remote protocol used for configuring the Transport from an application. RCI is defined on the Digi website in the RCI command reference Guide.

Device ID can be determined via the 'ati5' command.

Other example scripts

WR44 - bus demo, Python script

Below is the complete script we are using in our Bus Demo, this script collects the status of the two Digital Inputs and the current GPS co-ordinates. The script will only send data when there is a change in either the GPS co-ordinates or one of the Digital Inputs alters state. The file has been separated into individual modules for ease of explanation however you can download the complete script here bus-demo.py The required libraries import sarcli import time import socket Modules The first module converts the NMEA GPS info to lat / long co-ordinates:

```
def Lat_Long(raw_gps):
        gps_array = []
        gps_array = (raw_gps.split(','))

        if gps_array[1] == '':
                gps_array[1] = "4791.75429"
                gps_array[2] = "N"
        if gps_array[3] == '':
                gps_array[3] = "01208.62562"
                gps_array[4] = "E"

        lat_raw = gps_array[1]
        long_raw = gps_array[3]

        lat_dd = lat_raw[0:2]
        long_dd = long_raw[0:3]
        lat_mm = lat_raw[2:]
        long_mm = long_raw[3:]
        lat_dd = int(lat_dd)
        lat_mm = float(lat_mm)
        lat = (lat_mm / 60) + lat_dd
        lat = round(lat,3)

        long_dd = int(long_dd)
        long_mm = float(long_mm)
        long = (long_mm / 60) + long_dd
        long = round(long,3)

        if gps_array[2] == "S":
                lat = '-' + str(lat)
        else:
                lat = str(lat)

        if gps_array[4] == "W":
                long = '-' + str(long)
        else:
                long = str(long)
        return [lat,long]
```

This module creates a TCP Socket to the WR44 itself and collect the GPS info:

```
def getSocket_Data(gpsHOST, gpsPORT):

  data = 'None'

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((gpsHOST, gpsPORT))
        print 'Getting current GPS co-ordinates'
        data = s.recv(50)
        return data

  except:

        print 'Error - Cannot Connect to ', gpsHOST
        return data
```

This module create a TCP Socket to the web server and sends the Data:

```
def sendSocket_Data(HOST, PORT, data):

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
        print 'Successfully sent new status and GPS to', HOST
        s.send (data)

  except:
        print 'Error - sending new data to', HOST
```

This module issues a gpio command to the Transport command line interface to collect the status of the Digital Inputs:

```
def Get_Cli():
        clidata = ""
        cli = sarcli.open()
        cli.write("gpio")
        while True:
                tmpdata = cli.read(-1)
                if not tmpdata:
                        break
                clidata += tmpdata
        cli.close()
        return clidata
```

Checks the individual status of the gpio line:

```
def gpio_check(str, pin):
        if (pin + " : OFF") in str:
                return "off"

        return "on"
```

Writes the individual status of the 3 gpio lines into the variables:

```
def gpio(gpio_str):

        idx = gpio_str.find("Output:")

        input  = gpio_str[:idx-1]
        output = gpio_str[idx:]

        in_status     = gpio_check(input, "in")
        io_in_status  = gpio_check(input, "inout")
        io_out_status = gpio_check(output, "inout")

        return in_status, io_in_status, io_out_status
```

Define constants and Arrays

```
gpsHOST = '127.0.0.1'     # The remote host
gpsPORT = 2000            # The same port as used by the server
latlong = []              # Lat Long array
oldlatlong = []           # the old lat and long array
iostatus = []             # io status array
panic_status = "off"      # set the initial panic status to off
ign_status = "off"        # set the initial alarm status to off
```

```
HOST = 'gromit.mobilemonkey.co.uk'    # The remote web server hostname
PORT = 9999              # This is the port the web server is listening on


oldlatlong.append(50.000000003)
oldlatlong.append(-1.990900000)
```

The main program loop

```
            # Main
print "To stop, type \"Python kill\""

while True:
        clidata = Get_Cli()
        iostatus = gpio(clidata)
        changed = False
        if iostatus[0] == "off":
                if panic_status == "off":
                        panic_status = "on"
                        changed = True
        else:
                if panic_status == "on":
                        panic_status = "off"
                        changed = True

        if iostatus [1] == "on":
                if ign_status == "off":
                        ign_status = "on"
                        changed = True
        else:
                if ign_status == "on":
                        ign_status = "off"
                        changed = True



        rawgps = getSocket_Data(gpsHOST, gpsPORT)
#       print 'new data line' , rawgps
        latlong = Lat_Long(rawgps)
        if latlong[0] != oldlatlong[0]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[0] = latlong[0]
        if latlong[1] != oldlatlong[1]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[1] = latlong[1]
        if changed == True:
                status = panic_status + ',' + ign_status
                data = status + ','+ str(latlong[0]) + ',' + str(latlong[1])
                print data
                sendSocket_Data(HOST, PORT, data)
        time.sleep(1)
```

Timed event, Python script

This script waits until a specific time of the day and then completes a task. The script has been separated into sections for ease of explanation however you can download the complete script here time-evn.py

```
import sarcli
import time

#Modules
#The module issues commands to the command line:

def cli(command):
           cli =  sarcli.open()
           cli.write(command)
           cli.close()


#Define constants and Arrays

event_time = "13:22"
running = True


#The main program loop

while running:

        # gets the current time ie "13:01"
        current_time = time.strftime ("%H:%M", time.localtime())

        if current_time == event_time:

                #  It is time to do something..

                print " I'm Doing something "
                command = 'setevent "time_evn.py: Its time to do something"'
                cli(command)

                time.sleep(61) # Sleep for 61 seconds so that we do not do the
event again.

        else:
                print current_time
                time.sleep(50) # Its not time go to sleep for 50 seconds.
```

Below is a telemetry card control Python script

Below is the complete script for controlling the telemetry card, this script waits for an SMS message which contains either a "Camera on" or "Camera off". After receiving the command it will process it changing the state of the relay on the Telemetry board and the reply back to the sender with a "camera now on /off" message. Download the complete script here

Note This script is offered as an example and its reliability can not be guaranteed, the router must have Firmware version 5100 or later to run. smsctrl.py

```
            #The required libraries

import threading
import sarcli
import os
import sys
import time
```

```
'''
Threads
The first thread runs continuously and checks the eventlog for 'SMS Received:'
Messages:
'''

class eventlog (threading.Thread):
    def run (self):

        # Constants
        running = True
          # This is what we are looking for in the eventlog.txt
        string_match = "SMS Received:"
        filename = "eventlog.txt"

        while running:
          # Try to open the eventlog.txt file
          try:
            file = open(filename, 'r')

          except:
           # Output to the eventlog if there is a problem opening the
eventlog.txt
           cli =  sarcli.open()
           cli.write('setevent "smsctrl:error opening file"')
           cli.close()

          # Check the eventlog.txt file for the string_match value.
          for line in file:
                if string_match in line:
                  line = line.strip('\r\n')
                  command_str = "basic 0 nv " + '"' + line + '"'
                  cli = sarcli.open()
                  cli.write(command_str)
                  cli.close()
                  break

        file.close()
        time.sleep(10)

# This is thread that turns on the Relay waits 30secs then turns it off:

class DelayOnOff (threading.Thread):
    def run (self):

        answer = SetRelay("on")
        ReplySms(answer, cmd_array[1])
        # Wait 30 seconds
        time.sleep(30)
        answer = SetRelay("off")
        ReplySms(answer, cmd_array[1])

#This module takes the eventlog item and separates the command and phone number:

def GetEvent(output_string):
  event_array = []
  command_array = []
```

```
  event_array = (output_string.split(','))
  command= str(event_array[2])
  command_array=(command.split(':'))
  command_array[1] = str(command_array[1]).lstrip()
  command_array[2] = str(command_array[2]).lstrip()
  return command_array

#This module alters the State of the Relay on the Telemetry board and returns the
relays status:

def SetRelay(state):
  clir = sarcli.open()
  if state == "on":
        try:
           clir.write("anaconda -y 1")
           clir.write('setevent "Smsctrl:Camera now on"')
           answer= "on"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"

  elif state == "off":
        try:
           clir.write("anaconda -y 0")
           clir.write('setevent "Smsctrl:Camera now off"')
           answer= "off"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"
  clir.close()
  return answer


#This module sends back an SMS reply to the originators phone number with the
status:

def ReplySms(answer, phonenum):
  clir = sarcli.open()
  if answer == "on":
            sms = 'sendsms ' + phonenum + ' "Camera now on"'

  elif answer == "off":
            sms = 'sendsms ' + phonenum + ' "Camera now off"'
  else:
            sms = 'sendsms ' + phonenum + ' " Error setting relay"'

  try:
    clir.write(sms)
  except:
    clir.write('setevent "smsctrl:error sending sms"')
    clir.close()


#Define constants and Arrays

# Constants
running = True

#Variables
```

```
completed_command = "null"
cmd_array = []

#The main program loop


# Main
print "To stop, type \"Python kill\""

# Start eventlog Thread
eventlog().start()

while running:

        cli = sarcli.open()
        try:
           cli.write("basic 0 nv")
           tmpdata = cli.read(-1)
        except:
          print "error writing command: ", tmpdata
          errormsg = 'setevent ' + '"' + tmpdata + '"'
          cli.write("errormsg")
        cli.close()
        if completed_command != tmpdata:
                print "new event: ", tmpdata
                cmd_array = GetEvent(tmpdata)
                if cmd_array[2] == "Camera on":
                        answer = SetRelay("on")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera off":
                        answer = SetRelay("off")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera on 30":
                        DelayOnOff().start()
                        completed_command = tmpdata
                else:
                        time.sleep(5)
        else:
                time.sleep(5)
```

Wake on Lan, Python script

This script sends out a Wake on Lan packet to a specific host. The script has been separated into sections for ease of explanation however you can download the complete script here wol.py.

```
            #The required libraries

import socket
import struct

#Modules
#The module sends out Wake On LAN packets

def wake_on_lan(macaddress):

    """ Switches on remote computers using WOL. """
```

```
    # Check macaddress format and try to compensate.

    if len(macaddress) == 12:
        pass

    elif len(macaddress) == 12 + 5:

        sep = macaddress[2]
        macaddress = macaddress.replace(sep, '')

   else:
        raise ValueError('Incorrect MAC address format')

    # Pad the synchronization stream.
    data = ''.join(['FFFFFFFFFFFF', macaddress * 20])
    send_data = ''

    # Split up the hex values and pack.

    for i in range(0, len(data), 2):

        send_data = ''.join([send_data,

                             struct.pack('B', int(data[i: i + 2], 16))])

    # Broadcast it to the LAN.

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    sock.sendto(send_data, ('192.168.1.255', 9))

#The main program loop

if __name__ == '__main__':

    # Use macaddresses with any seperators.
    wake_on_lan('00:40:63:FC:0C:75')
    wake_on_lan('00-40-63-FC-0C-75')

    # or without any seperators.
    wake_on_lan('004063FC0C75')
```

Supported Python modules

```
    A
        abc
        aifc
        anydbm
        array
        ast
        asynchat
        asyncore
        atexit
        audiodev
    B
        BaseHTTPServer
        Bastion
```

```
    base64
    bdb
    binascii
    binhex
    bisect
C
    CGIHTTPServer
    ConfigParser
    Cookie
    cPickle
    cProfile
    cStringIO
    calendar
    cgi
    cgitb
    chunk
    cmath
    cmd
    code
    codecs
    codeop
    collections
    colorsys
    commands
    compileall
    contextlib
    cookielib
    copy
    copy_reg
    csv
D
    DocXMLRPCServer
    datetime
    dbhash
    decimal
    difflib
    digihw
    digiwdog
    digiweb
    dircache
    dis
    doctest
    dumbdbm
    dummy_thread
    dummy_threading
E
    email
    encodings
    errno
    exceptions
F
    filecmp
    fileinput
    fnmatch
    formatter
    fpformat
    fractions
    ftplib
    functools
```

```
    future_builtins
G
    gc
    genericpath
    getopt
    getpass
    gettext
    glob
    gzip
H
    HTMLParser
    hashlib
    heapq
    hmac
    htmlentitydefs
    htmllib
    httplib
I
    ihooks
    imaplib
    imghdr
    imp
    imputil
    inspect
    io
    itertools
K
    keyword
L
    linecache
    locale
    logging
M
    MimeWriter
    macpath
    macurl2path
    mailbox
    mailcap
    markupbase
    marshal
    math
    md5
    mhlib
    mime
    mimetools
    mimetypes
    mimify
    modulefinder
    multifile
    mutex
N
    netrc
    new
    nntplib
    ntpath
    nturl2path
    numbers
O
    opcode
```

```
    operator
    optparse
    os
    os2emxpath
P
    parser
    pdb
    pickle
    pickletools
    pipes
    pkgutil
    platform
    plistlib
    popen2
    poplib
    posix
    posixfile
    posixpath
    pprint
    profile
    pstats
    pty
    py_compile
    pyclbr
    pydoc
    pydoc_topics
    pyexpat
Q
    Queue
    quopri
R
    random
    re
    repr
    rexec
    rfc822
    rlcompleter
    robotparser
    runpy
S
    SimpleHTTPServer
    SimpleXMLRPCServer
    SocketServer
    StringIO
    sarcli
    sarutils
    sched
    select
    sets
    sgmllib
    sha
    shelve
    shlex
    shutil
    site
    smtpd
    smtplib
    sndhdr
    socket
```

```
        sre
        sre_compile
        sre_constants
        sre_parse
        ssl
        stat
        statvfs
        string
        stringold
        stringprep
        strop
        struct
        subprocess
        sunau
        sunaudio
        symbol
        symtable
        sys
   T
        tabnanny
        tarfile
        telnetlib
        tempfile
        termios
        textwrap
        this
        thread
        threading
        time
        timeit
        toaiff
        token
        tokenize
        trace
        traceback
        tty
        types
   U
        UserDict
        UserList
        UserString
        unittest
        urllib
        urllib2
        urlparse
        user
        uu
        uuid
    W
        warnings
        wave
        weakref
        webbrowser
        whichdb
        X
        xdrlib
        xmllib
        xmlrpclib
    Z
```

```
        zipfile
        zipimport
        zlib
```

References

Portions of this document are reproduced from documents by Matt jameson & Jon Lyons: extranet.jrp2.com/~jpowell/gromit.mobilemonkey.co.uk/trans-Python.html

UDP echo transport

Purpose

This example will go over setting up a generic UDP Server within Python, and echoing out the received data back to a specified destination. For more detailed information on UDP Sockets in Python, please see the UDP Sockets section of the Python manual. Below is the code sample:

```
import socket, time

#This the remote IP address to send the data too
HOST = '10.10.19.80'
#This is the port the remove server is listening on
PORT = 5000


def main():
    # Create the socket and connect to the remote server.
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    #This is the port number the TransPort listens for data on and
    #accepts any IP address
    s.bind(('', 5001))

    message, udpAddress = s.recvfrom(8192)

    # Send data to the server.
    s.sendto(message, (HOST, PORT))

    s.close()

if __name__ == '__main__':
    while True:
        time.sleep(0.1)
        main()
```

Code breakdown

```
import socket, time
```

This is where the code will import the ‘socket’ and ’time’ modules into the code for use later on.

```
#This the remote IP address to send the data too
HOST = '10.10.19.80'
```

```
#This is the port the remove server is listening on
PORT = 5000
```

Here is where 2 variables are being declared called “HOST” and “PORT”. In this example, “HOST” will represent the IP address to send the echo’ed UDP data to, and “PORT” is the port number the IP address will listen for the data on.

```
def main():
    # Create the socket and connect to the remote server.
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    #This is the port number the TransPort listens for data on and
    #accepts any IP address
    s.bind(('', 5001))

    message, udpAddress = s.recvfrom(8192)

    # Send data to the server.
    s.sendto(message, (HOST, PORT))

    s.close()
```

Here is the main program information. Here we are defining the “main” program (def main():). Within this program, we are creating a variable called “s” and giving it the socket parameters that tell it to use UDP as the protocol (s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)). The next line allows the local socket to be reused immediately without needing to timeout first (s.setsockopt (socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)). We then tell the “s” variable to “bind” the socket to an incoming IP address on port 5001 (s.bind((, 5001))). The code then declares a new variable called “message” and assigns a receive buffer of 8192 bytes to it (message, udpAddress = s.recvfrom(8192)). The code will then send the “message” to the “HOST” and “PORT” that were defined earlier (s.sendto (message, (HOST, PORT))), and once that is sent, “close” the socket (s.close()).

```
if __name__ == '__main__':
    while True:
        time.sleep(0.1)
        main()
```

Here is where we run the program (if __name__ == '__main__':). While the program says “true” (while True:), it will sleep for 0.1 seconds (time.sleep(0.1)), and then run the “main” program (main()) to listen for incoming UDP data on port 5001.

Drop-in Networking Products

This category covers all Digi Drop-In Networking Products. The official product page on the digi website may be found here: http://www.digi.com/products/wireless-modems-peripherals/wireless-range-extenders-peripherals/.

Advanced Device Discovery Protocol (ADDP)

What is ADDP?

ADDP (Advanced Device Discovery Protocol) is a proprietary protocol developed by Digi International that allows devices on a local network to be found regardless of their network configuration.

How does it work?

ADDP uses a client/server model. The client is the application that is searching for devices. The server is the device that is being search for.

In the simplest terms, the client application sends out a specially formatted UDP broadcast packet on the network. ADDP servers listening for the packet, will receive it, and send an ADDP response back to the client. Once this process is complete, the client can then send configuration requests to the device. These can include things like network settings, and reboot requests.

Java library

A subset of the protocol has been implemented in Java. You can find the jar file here: ADDP Library.

The associated javadoc documentation can be found here: ADDP Java doc.

This library allows you to search synchronously, and asynchronously for devices on the network. You can then use it to reconfigure the device's network settings, or reboot the device.

Java sample application

A simple discovery sample application can be found here: AddpSample(r2010).zip

Basic usage

First, instantiate the AddpClient object.

```
AddpClient addpClient = new AddpClient();
```

Next, call SearchForDevices() and check the return value. Then get the devices, and walk the hashtable.

```
if (addpClient.SearchForDevices()) {
    AddpDeviceList deviceList = addpClient.getDevices();

    Enumaration<AddpDevice> e = deviceList.elements();
    while(e.hasMoreElements()) {
       AddpDevice device = e.nextElement();

       // do something with the device here
       System.out.println(device.toString());

       // if device is not configured for DHCP, then turn it on and reboot.
       if (device.getDHCP() == 0) {
          addpClient.setDHCP(device, true, "dbps");
          addpClient.rebootDevice(device, "dbps");
       }
```

```
    }
}
```

ConnectPort X gateways

ConnectPort X gateways aggregate and transport ZigBee/802.15.4 network traffic to central data applications over cellular, Wi-Fi or Ethernet connections. ConnectPort X gateways are a key element of Digi's Drop-in Networking family of products – a collection of hardware components that also includes our XBee® adapters, modules, extenders and bridges – which together enable distributed electronic devices to be wirelessly networked where no wired infrastructure exists or where access to an existing network is prohibited. ConnectPort X gateways feature an integrated ZigBee/802.15.4 module plus cellular (3G, 2.5G or 2G), Wi-Fi, GPS and local storage options. Several physical interfaces are available as well, including Ethernet, serial, USB and a 1-Wire sensor port that accepts Digi's Watchport® sensors (X8 model only). ConnectPort X gateways are available in multiple configurations to suit various applications and connectivity needs. For programmability, ConnectPort X gateways offer an embedded Python® engine, giving users a powerful open-source software tool to develop custom applications to run on the gateway. The Python environment is noted for its simplicity and can be used to automate a variety of control, monitoring and notification procedures.

Features

- Network protocols: UDP/TCP, DHCP
- LEDs
  - ConnectPort X2: Ethernet status, power, ZigBee link/activity
  - ConnectPort X4, X8: Ethernet status, power, cellular link/activity, signal strength (4 bars), ZigBee link/activity, Wi-Fi link/activity
- Security - ConnectPort X4, X8
  - Security - SSL tunnels, SSHv2, FIPS 197 (serial port)
  - Real-time clock
- Router/Security Features
  - NAT
  - Port forwarding
  - Access control lists (IP filtering)
- VPN Features - ConnectPort X4, X8
  - IPsec with IKE/ISAKMP
  - Multiple tunnel support
  - DES, 3DES and up to 256-bit AES Encryption
  - VPN pass-through, GRE forwarding
  - Management
  - HTTP/HTTPS web interface
  - Password access control
  - IP service port control
  - Optional secure enterprise management via Digi Connectware Manager

Interfaces

- Serial - ConnectPort X4, X8
    - 1 RS-232 DB-9M serial port
    - Throughput up to 230 Kbps
    - Full signal support for TXD, RXD, RTS, CTS, DTR, DSR and DCD
    - Hardware and software flow control
- USB
    - ConnectPort X4: 1 Powered USB Type A connector (Host)*
    - ConnectPort X8: 2 Powered USB Type A connectors (Host)*
- See www.digi.com/products/wireless/usb-peripherals/ for the list of supported USB devices
- Ethernet
    - 1 RJ-45 port
    - Standard: IEEE 802.3
    - Physical Layer: 10/100Base-T
    - Data rate: 10/100 Mbps (auto-sensing)
    - Mode: full or half duplex (auto-sensing)
- Sensor - ConnectPort X8
    - 1 RJ-45 port
- ZigBee/802.15.4
    - XBee-PRO (Freescale and Ember supported)
- Cellular
    - ConnectPort X4: EV-DO/1xRTT or EDGE/GPRS PCI Express Module (1)
    - ConnectPort X8: EV-DO/1xRTT or EDGE/GPRS PCI Express Modules (2)
- Wi-Fi (802.11b) - ConnectPort X2
    - Standard IEEE 802.11b
    - Frequency: 2.4 GHz
    - Data rate: Up to 11 Mbps w/ fallback
    - Modulation: DBPSK (1 Mbps), DQPSK (2 Mbps), CCK (11, 5.5 Mbps)
    - Transmit power: 16 dBm typical
    - Receiver sensitivity: -82 dBm @ 11 Mbps
    - Connector: 1 x RP-SMA
- Wi-Fi (802.11b/g) - ConnectPort X4, X8
    - Ad-hoc & AP Client Modes only; Access Point Mode not supported
    - Not available when gateway is configured with a cellular module

- Optional (via PCIe Module) - ConnectPort X8
  - Global Positioning System (GPS)
  - Local storage (up t**1 GB)

ConnectPort X2e

Product description

The ConnectPort X2e family is a low-cost IP to XBee gateway targeted at residential and light commercial use. Some models include WiFi. All antenna are internal. The Xbee supported is only the newer SMT modules such as the XBee ZB SMT (or S2C). As of June 2012, only ZigBee or ZB/Smart Energy are supported.

Unlike the older ConnectPort X2 or X4, the X2e is designed for use primarily with the iDigi Device Cloud. The core operation system is Linux, however as of June 2012 none of the models give customers access to add or modify the Linux code directly.

Feature list

- Small size: 3 x 3 x 1 inch (7.62 cm x 7.62 cm x 2.54 cm)
- Security: SSL tunnels
- Ethernet Port: RJ-45, 10/100Mhz, Half or Full Duplex, auto-sensing
- Optional WiFi: 802.11 b/g/n, ad-hoc and Ap Client modes only (Access Point mode not yet supported)
- Memory: 64 MB RAM, 128 MB Flash (user available memory varies by firmware and functions active)
- Power Supply: 5vdc from included AC wall transformer. Typically 1.2 watts, maximum 2.5 watts.
- Real time clock: No - NTP access is required to create an iDigi connection.
- Operating Temperature: 0° C to 40° C (32° F to 104° F)
- Python version: 2.7.1
- Development: Digi ESP for Python version 2.1 or newer

Compatibility comments

- The X2e has no CLI or Commandline interface. Python code using the Module: digicli will not run on the X2e.
- The X2e uses Python 2.7.1, whereas the X2/X4 use 2.4.3. In general, this won't cause a problem porting X2/X4 code to the X2e, but may cause problems porting in the other direction.
- The X2e SE cannot be used for development purposes. The X2e ZB has an SSH login, which does not offer root access.
- Since the X2e always uses SSL/TLS to iDigi, NTP access is required to validate security certificates, and therefore NTP is required for device operation.

Family models

ConnectPort X2e SE (Smart Energy 1.1)

Both domestic and international models include a low-power XBee Module (6.3mW/+8dBm) loaded with Smart Energy 1.1 firmware. The Python code is locked and no user login exists with high enough priority to allow users to add or modify anything on the X2e SE. To see the local web interface, you must push the black button in the upper left corner. This activates the web page for a number of minutes before disabling it. The ConnectPort X2e SE cannot be converted to an X2e ZB in the field.

ConnectPort X2e ZB (Python programmable)

Domestic models include a high-power Xbee module (63mW/+18dBm), while international models include a low-power XBee Module (6.3mW/+8dBm). SSH is used to log into the X2e, and SCP is used to copy files to and from it. The local web interface is NOT locked by the button. iDigi allows users to enable such a lock, or the button is available to Python programs. The ConnectPort X2e ZB cannot be converted to an X2e SE in the field.

Special Linux commands

If you have a model which allows SSH access, then you will find X2e-specific commands such as these.

Xbee Utilities

```
XBee utilities

syntax:
    xbee [command ...]

commands:
    refresh                           {Discover network devices}
    zigbee                            {Discover ZigBee devices}
    clear                             {Clear list and discover devices}
    id|network|extended|node|product  {Sort node list by field}

    [node] <CC>[[=]param]             {Run AT command on device:
                                      <CC> is 2 character upper case command
                                      param is <decimal>, 0x<hex>, or "string"}

    load_profile [node] file          {Load settings from file}
    factory_default [node]            {Restore factory default settings}
    restart                           {Restart gateway radio}

    child_table node                  {Display child table}
```

```
neighbor_table node                {Display neighbor table}
source_route node                  {Display source route}
route destination [source]         {Display route}

identify                           {Display identify messages}
identify node [seconds]            {Send identify message}
ping node [count size interval_ms] {Send loopback data}

fw_update [file]                   {Update gateway radio firmware}
fw_update target [updater] [file]  {Schedule remote firmware update}
fw_update target cancel            {Cancel remote firmware update}
fw_update                          {Display firmware status list}
fw_status [node]                   {Display firmware status for node}
```

Python modules

In general, the ConnectPort X2e supports a larger set of common Python modules.

Digi-specific modules NOT supported

The ConnectPort X2e does not support these commonly used modules, plus any not listed as supported should be assumed unsupported:

- Module: cwm (this module is obsolete)
- Module: digicli (Programmable X2e have a linux login, not a digi command-line)
- Module: digipowercontrol (The X2e has no ability to sleep or power off sections of hardware)
- Module: digiwdog (The X2e has no external hardware to make this reliable, so it is disabled)
- Module: digiweb (The X2e does not have a custom Digi-specific web server such the X4 does)

ConnectPort X5

Product description

The ConnectPort X5 is a heavy duty cellular gateway that bridges industrial protocols to IP networks using Python based applications. This page describes the features available on the device, links to Python documentation and associated programming samples. This page is not intended to cover device certifications, product availability and data sheets.

Please note: This product is no longer available for purchase.

Feature list

The ConnectPort X5 contains the following major features:

- IP67-compliant enclosure
- Cellular connectivity (GSM or CDMA)
- 802.11 B/G wireless
- Ethernet interface
- XBee radio connectivity
- GPS
- CAN bus interface
- J1939 industrial protocol
- J1708 industrial protocol
- J1587 industrial protocol
- Integrated power management
- Accelerometer sensor
- Temperature sensor
- Ignition detection
- Battery voltage detection

Python information

For general Python programming information on the device, check the programmer's guide Digi Python programmers guide.

General IP socket programming on the X5

(XBee radio API)

GPS sample

CAN bus sample

J1939 sample

J1708 sample

J1587 sample

Module: digipowercontrol (Power Control Python API)

Remote Power Management Demo

Accelerometer sample

Temperature sample

Battery sample

Fleet management demo

Demos and applications

One application to use with the ConnectPort X5 is the DIA.

The Device Cloud Wiki is a Python framework designed to gather information from multiple sources, provide programmatic control over the information collecting, and present the data either to a software platform such as the Device Cloud or directly to the user through a variety of methods.

Focused code samples can be found in the Code samples category

Additional annotated programming samples can be found in the ConnectPort X5 user Guide. See: http://www.digi.com/support/, select the ConnectPort family, select your ConnectPort X5 variant, and then click on the Documentation link.

DigiMesh products

DigiMesh XBee technology

Digi produces several families of DigiMesh XBee module. The most important feature in determining support within XBee adapters and gateways is the foot-print of the Xbee.

- Most existing Digi Adapters and Gateways support only the 20-pin through-hole footprint, for which only DigiMesh at 2.4 Ghz and 900 Mhz is available.
- Newer XBee families have moved to a SMT format and include 865 Mhz, 868 Mhz and 900 Mhz.

Existing DigiMesh gateways

- Digi ConnectPort X2
  - ConnectPort X2 - Industrial with DM 2.4 GHz to Ethernet (metal case, extra memory)
  - ConnectPort X2 - Industrial with DM 900 MHz to Ethernet (metal case, extra memory)

  - Note: as of March 2012 there are no X2 Commercial models with DigiMesh (lower cost, plastic case, less memory)
  - Note: as of March 2012 there are no X2 models with Wifi and DigiMesh
- Digi ConnectPort X4
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & cellular (GSM/Edge)
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & Wi-Fi
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet & cellular (GSM/Edge)
  - Note: as of March 2012 there are no X4H or X4 IA models with DigiMesh
  - Note: as of March 2012 there are no X4 CDMA cellular models with DigiMesh
  - Note: as of March 2012 there is no SMT support within the X4, so no 865 Mhz or 868 Mhz DigiMesh support
  - Note: as of March 2012 there are no X5 vehicle models with DigiMesh

Existing DigiMesh adapters

- XBee-PRO DigiMesh 2.4 Range Extender
- Note: as of March 2012 there is no DigiMesh 900 Mhz Range Extender
- XBee-PRO DigiMesh Adapters
  - XBee-PRO DigiMesh 2.4 GHz, RS-232 adapter
  - XBee-PRO DigiMesh 2.4 GHz, RS-485 adapter
  - XBee-PRO DigiMesh 2.4 GHz, Digital IO adapter
  - XBee-PRO DigiMesh 2.4 GHz, Analog Input adapter
  - XBee-PRO DigiMesh 2.4 GHz, USB adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-232 adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-485 adapter
  - XBee-PRO DigiMesh 900 Mhz, Digital IO adapter
  - XBee-PRO DigiMesh 900 Mhz, Analog Input adapter
  - XBee-PRO DigiMesh 900 Mhz, USB adapter
  - Note: as of March 2012 there is no DigiMesh Smart Plug
  - Note: as of March 2012 there is no DigiMesh Light/Temperature (LT/LTH) Sensor
  - Note: as of March 2012 there is no DigiMesh XStick
  - Note: as of March 2012 there are no DigiMesh WatchPort Adapters

Can Digi ZigBee/XBee products be converted to DigiMesh?

Officially, No.

However, one can remove the 20-pin ZigBee XBee and replace it with a 20-pin DigiMesh XBee. This works fine with the RS-232/485/USB adapters, but may not work with adapters expecting analog signals because the ZigBee Xbee and DigiMesh XBee use different reference voltages. Your software will need to compensate for this, as well as the fact that there are some minor AT/DDO settings differences between Zigbee-XBee and DigiMesh-Xbee.

Which Python Version

Different Digi products support different versions of Python. If you upload the uncompiled.PY files, then you may use any version of Python as long as you avoid features which are not supported on the Digi product (for example, you cannot use new 2.6.x features on a Digi ConnectPort X4 which only supports 2.4.3).

However, if you plan to use the iDigi/Dia framework and/or upload the compiled.PYC files, then you must use an exact match.

Python 2.4.3

Download from http://Python.org/download/releases/2.4.3/

- Digi Connect WAN Family
- Digi ConnectPort WAN Family
- Digi ConnectPort X2
- Digi ConnectPort X4
- Digi ConnectPort X4H
- Digi ConnectPort X5
- Digi ConnectPort X8

Python 2.6.1

Download from http://Python.org/download/releases/2.6.1/

Digi ConnectPort X3

Python 2.6.2

Download from http://Python.org/download/releases/2.6.2/

Digi ConnectPort LTS 8/16/32

XBee analog I/O adapter

Product description

The XBee Analog Adapter is a product designed to take analog input readings and communicate them across the XBee network. For more details of usage and configuration, please review the *Drop-in-Networking Accessories user Guide* at: digi.com/support/documentation/90000891.pdf. This Analog Adapter is designed for ZB Firmware.

Major Features of the XBee Analog Adapter:

- 4 Analog input terminals supporting Ten Volt mode, Current Loop mode, and Differential mode.
- Easy to use screw terminal connector.
- Power output of 12 Volt 50mA in mains powered mode.
- Optional battery powered for isolated locations.

Supported platforms

The XBee Analog Adapter is supported on all Digi's XBee protocols. All of Digi's ConnectPort X gateways products that are using the same XBee protocol as the adapter are compatible.

Example: If you have a gateway product with a XBee ZB radio installed, a XBee ZB Analog Adapter is compatible, but not a XBee ZNet 2.5 Analog Adapter.

There are two distinct versions of the XBee Analog Adapter hardware, each supporting a different sub-set of the Digi XBee modules. They are NOT interchangeable and putting the incorrect XBee on the incorrect adapter will return bad data. One model supports ZB and ZNet XBee only, while the other supports 802.15.4, DigiMesh and the Digi Point-to-Multipoint XBee.

Programming options

The XBee Analog adapter uses a unique function set for each of the protocols, requiring that configuration of the radio be done via the RF interface through remote AT commands.

Important: Serial communications functionality is disabled on XBee Analog and Digital I/O Adapters

For the XBee Analog and Digital I/O Adapters, the XBee firmware disables the XBee module's serial communications functionality, so that the pins normally used for serial communications can instead be used for additional I/O functionality. As a result, once an XBee module has been loaded with the firmware image specific to the XBee Analog or Digital I/O Adapters, the XBee module can no longer be configured using X-CTU, the XBee serial API, or local AT commands.

The only way to configure these adapters is by using the Ident or Ident/Reset commissioning and identification button (described in the topic "Commissioning and identity behaviors" in the Drop-in Networking Accessories user Guide) and OTA commands—either by using a ConnectPort gateway, X-CTU, or by using the remote command API of another serial-enabled module associated to the same network.

Mode and terminal configuration

The Analog Adapter's terminals must be configured in pairs. The terminals have the option of not being the same mode as the other terminal in its pair, with the exception of Differential input mode. Terminals 1 and 2 are bound together, terminals 3 and 4 are bound together.

To configure the mode for terminals 1 and 2

D8=4, D4=4, D6=4, (Enables Terminals 1 and 2 for Current Loop mode)

D8=4, D4=4, D6=5, (Enables Terminal 1 for Current Loop mode, Terminal 2 for Ten Volt mode)

D8=4, D4=5, D6=4, (Enables Terminal 1 for Ten Volt mode, Terminal 2 for Current Loop mode)

D8=4, D4=5, D6=5, (Enables Terminals 1 and 2 for Ten Volt mode)

D8=5, D4=4, D6=4, (Enables Terminals 1 and 2 for Differential mode)

All other combinations are invalid and may result in inconsistent behavior

To configure the mode for terminals 3 and 4

P0=4, D7=4, P2=4, (Enables Terminals 3 and 4 for Current Loop mode)

P0=4, D7=4, P2=5, (Enables Terminal 3 for Current Loop mode, Terminal 5 for Ten Volt mode)

P0=4, D7=5, P2=4, (Enables Terminal 3 for Ten Volt mode, Terminal 4 for Current Loop mode)

P0=4, D7=5, P2=5, (Enables Terminals 3 and 4 for Ten Volt Mode)

P0=5, D7=4, P2=4, (Enables Terminals 3 and 4 for Differential mode)

All other combinations are invalid and may result in inconsistent behavior

To enable terminals 1-4 for analog input

D0=2 (Enables Terminal 1 for analog input)

D1=2 (Enables Terminal 2 for analog input)

D2=2 (Enables Terminal 3 for analog input)

D3=2 (Enables Terminal 4 for analog input)

Sensor readings retrieval

Once the device has been configured, it is time to start polling sensor samples from the device. There are two methods of retrieving the sample, active polling and passive polling. Both methods of polling

require the user to parse the 'IS' data structure for the sensor readings. More information on the 'IS' data structure here: IS Data structure

Active polling is when an application or user on the XBee network sends the Remote AT command 'IS', which causes the Analog sensor to take a sensor reading and return the results to the requester.

Advantages

- Simple to setup
- User-Controlled timing of samples

Disadvantages

For reliable responses, requires a non-sleeping Analog Adapter.

Passive polling is when the Analog adapter periodically sends a sensor reading and send it to the configurable destination address. To setup a Analog adapter to send periodic sensor readings to an address, configure the 'IR' parameter to be the rate in milliseconds to send the reading, and the 'DH' and 'DL' addresses to match the 'SH' and 'SL' address of the XBee node desired. In cases where the adapter is also sleeping, the device sends sensor readings at least once every wake cycle. If you desire the device to send only one sensor reading per wake cycle (recommended), set the 'IR' parameter to 0xFFFF.

Advantages

Allows for sleeping Analog adapters to reliably get sensor readings to the destination address

Consistent timing of samples

Disadvantages

More difficult to setup, requires working knowledge of sleep settings.

Power output

Terminal 6 is the power out pin. It can be set in either battery pack voltage out or +12VDC at 50mA out if mains powered. To set battery pack voltage output, set dip switch 1 to the on position. To enable +12VDC output, set dip switch 2 to the on position. Setting both to the on position defaults to +12VDC output but incurs extra battery drain as well.

Once the desired power output is selected, the 'P3' setting needs to be set to 5.

Note: In order for consistent output voltage, the XBee Adapter cannot sleep. Using the power out function of this adapter in conjunction with batteries will result in significant additional battery usage.

DIA configuration and programming examples

The Device Cloud Wiki is a Python framework designed to gather information from multiple sources, provide programmatic control over the information collecting, and present the data either to a software platform such as the Device Cloud or directly to the user through a variety of methods.

The DIA supports the XBee Analog Adapter. For an example configuration file: XBee Analog I/O DIA Example.

Python programming examples

The XBee Analog Adapter may also be controlled from a Digi gateway device using the embedded XBee Python module's function calls ddo_get_param() and ddo_set_param(). If you wish to use the adapter in a passive polling situation, you must also use the sockets API to receive the sensor readings from the Analog adapter.

Below is a quick demonstration of how to send commands to configure terminals 1 and 2 to Ten Volt mode, enable them for analog input, and retrieve an analog sample.

```
from xbee import ddo_get_param, ddo_set_param

addr = '[00:13:a2:00:40:48:59:95]!'

##Enabling Ten volt mode on terminals 1 and 2
ddo_set_param(addr, 'D8', 4)
ddo_set_param(addr, 'D4', 5)
ddo_set_param(addr, 'D6', 5)

##Enabling terminals 1 and 2 for analog input
ddo_set_param(addr, 'D0', 2)
ddo_set_param(addr, 'D1', 2)

##Applying changes and writing to flash
ddo_set_param(addr, 'AC')
ddo_set_param(addr, 'WR')

##Taking a sensor sample, printing the raw binary string to STDOUT
sample = ddo_get_param(addr, 'IS')
print 'sample(%d)' % len(sample),
for by in sample:
  print '%02X' % ord(by),
print
```

Here is an example using 4-20mA on input #1 and #2. Since we enable the 50mA 12vdc power output from terminal #6, you would wire your sensor's '+' wire to terminal 6, then your '-' wire to terminal 1. Terminal 6 'sources' your current loop, and terminal 1 'sinks' it.

```
from xbee import ddo_get_param, ddo_set_param

addr = '[00:13:a2:00:40:48:59:95]!'

##Enabling Ten volt mode on terminals 1 and 2
ddo_set_param(addr, 'D8', 4)
ddo_set_param(addr, 'D4', 4)
ddo_set_param(addr, 'D6', 4)

##Enabling terminals 1 and 2 for analog input
ddo_set_param(addr, 'D0', 2)
ddo_set_param(addr, 'D1', 2)

##Enabling terminals 6 to output 50mA @ 12vdc
## make sure your enable DIP switch #2 to ON (#1 off) to
## use your external power supply to drive the 12vdc power-out.
ddo_set_param(addr, 'P3', 5)

##Applying changes and writing to flash
ddo_set_param(addr, 'AC')
ddo_set_param(addr, 'WR')

##Taking a sensor sample, printing the raw binary string to STDOUT
sample = ddo_get_param(addr, 'IS')
print 'sample(%d)' % len(sample),
for by in sample:
  print '%02X' % ord(by),
print
```

Usage tricks and hints

Hard-coding a fixed PAN ID

Hardcoding a fixed XBee PAN ID describes a simple way to semi-permanently force in a fixed PAN ID - this is especially useful for the XBee AIO and DIO adapters since XCTU cannot talk serially to them.

Reflashing the XBee

Because the AIO firmware disables the serial port on the XBee, reflashing can be challenging. Manually activate the XBee bootloader to speed the process: Bootloader to force XBee reflash.

Associating with the correct gateway

A user with multiple gateways will find the initial setup of the XBee AIO Adapter challenging. With other products, XCTU can be used to preload a profile with the desired PAN ID. However, an XBee with the AIO firmware cannot communicate with XCTU because it has no available serial port. Therefore your only option is the use the commissioning 4-button press.

The problem is that you may have multiple gateways on any single channel, so when the XBee AIO Adapter moves to a new channel, it may NOT notice your desired gateway until it has cycled through the channel list many times. For example, if you have 10 active gateways within range, they may be on only 3 or 4 channels of the 14 available.

Some simple solutions:

- Use a commissioning gateway which has had the SC/Scan Channels setting changed from default - for example to 0x0C00, which is only channels 0x15 or 0x16 and NOT the defaults down near channel 0x0D. This increases the probability that your gateway is on a channel by itself. A low-cost ConnectPort X2 makes a nice commissioning gateway.
- Leave the commissioning gateway's PAN Id 0x0 so it uses a large random PAN ID.
- If you have an XBee with an external antenna:
  - Remove the XBee's antenna to weaken its range, reducing the number of visible gateways.
  - If the gateway has an external antenna, consider linking the XBee directly by wire to the gateway's RPSMA connector. This insures your commissioning gateway's signal is the strongest which the XBee AIO Adapter can see.
- Use the commissioning 4-button press to move the XBee AIO Adapter between channels until it shows up on your commissioning gateway's node list, then manually force in other settings, followed last by your desired PAN ID. Just remember that applying the new PAN ID will cause the XBee AIO Adapter to leave association with your commissioning gateway.

Replacement parts

Six-terminal block plug

The terminal block is a Phoenix Contact MSTB 2.5/6-ST-5.08, which can be purchased directly from Phoenix Contact or from an online reseller such as DigiKey as part 227-1015-ND.

Locking power connector

Locking Power Connecter

XBee Digital I/O Adapter

Product description

The XBee Digital Adapter is a product designed to take digital input or programatically assert digital output and communicate the I/O changes across the XBee network. For more details of usage and configuration, please review the Drop-in-Networking Accessories user Guide at: http://www.digi.com/din/docs

Major Features of the XBee Digital Adapter:

- 4 Digital I/O terminals supporting open collector and pull up and pull down capabilities.
- Easy to use screw terminal connector.
- Power output of 12 Volt 50mA in mains powered mode.
- Optional battery powered for isolated locations.
- Power input 3.7-6V (different from first generation XBee adapters)

Supported platforms

The XBee Digital Adapter is supported on all Digi's XBee protocols. All of Digi's ConnectPort X gateways that are using the same XBee protocol as the adapter are compatible.

Example: If you have a gateway product with a XBee ZB radio installed, a XBee ZB Analog Adapter is compatible, but not a XBee ZNet 2.5 Digital Adapter.

Programming options

The XBee Digital adapter uses a unique function set for each of the protocols, requiring that configuration of the radio be done via the RF interface through remote AT commands.

Important: Serial communications disabled on XBee Analog and Digital I/O Adapters

For the XBee Analog and Digital I/O Adapters, the XBee firmware disables the XBee module’s serial communications functionality, so that the pins normally used for serial communications can instead be used for additional I/O functionality. As a result, once an XBee module has been loaded with the

firmware image specific to the XBee Analog or Digital I/O Adapters, the XBee module can no longer be configured using X-CTU, the XBee serial API, or local AT commands.

The only way to configure these adapters is by using the Ident or Ident/Reset commissioning and identification button (described in the topic "Commissioning and identity behaviors" in the Drop-in Networking Accessories user Guide) and OTA commands—either by using a ConnectPort gateway, X-CTU, or by using the remote command API of another serial-enabled module associated to the same network.

Mode and terminal configuration

The Digital Adapter's terminals can be configured independently for input or output operation. It is possible to set a single terminal to be in both input and output mode.

To configure a Terminal for Input

D8=3 (Enables Terminal 1 for Digital Input)

D1=3 (Enables Terminal 2 for Digital Input)

D2=3 (Enables Terminal 3 for Digital Input)

D3=3 (Enables Terminal 4 for Digital Input)

To configure a Terminal for Output

D4=5 (Enables Terminal 1 for Digital High Output)

D6=5 (Enables Terminal 2 for Digital High Output)

D7=5 (Enables Terminal 3 for Digital High Output)

P2=5 (Enables Terminal 4 for Digital High Output)

D4=4 (Enables Terminal 1 for Digital Low Output)

D6=4 (Enables Terminal 2 for Digital Low Output)

D7=4 (Enables Terminal 3 for Digital Low Output)

P2=4 (Enables Terminal 4 for Digital Low Output)

Sensor readings retrieval

Once the device has been configured, it is time to start polling sensor samples from the device. There are three methods of polling available for the Digital Adapter. An active polling, and two methods of passive polling. All forms of polling require the user to parse the IS data structure.

Active polling is when an application or user on the XBee network sends the Remote AT command 'IS', which causes the Digital sensor to take a sensor reading and return the results to the requester.

802.15.4 I/O line passing support

The I/O Line Passing function of XBee 802.15.4 Modules is not supported with the Digital I/O Adapter, as the Input Port mapping is not identical to the Output Port mapping, e.g. Terminal 2 input is D1, but output on Terminal 2 is D6. Line Passing would require that both are the same!

Advantages

- Simple to setup
- User-Controlled timing of samples

Disadvantages

- For reliable responses, requires a non-sleeping Adapter.
- Less efficient on larger networks.

The first passive polling option is to periodically transmit a sensor reading and send it to the configurable destination address. To configure this setting, set the 'IR' parameter to the rate in milliseconds to send the sensor reading, and the 'DH' and 'DL' parameters to match the 'SH' and 'SL' parameters of the desired destination. In cases where the sensor is also sleeping, enabling the 'IR' parameter causes the sensor to send at least 1 reading per wake cycle. If you desire the device to send only one sensor reading per wake cycle (recommended), set the 'IR' parameter to 0xFFFF.

Advantages

- Allows for sleeping Digital adapters to reliably get sensor readings to the destination address
- Consistent timing of samples

Disadvantages

More difficult to setup, requires working knowledge of sleep settings.

The second passive polling option is to wait for a change in state on the I/O lines. To enable this feature, the 'IC' parameter is set to a mask indicating which I/O lines to monitor for a state change. Once the desired mask is set, any changes in logical state on the specified lines will generate a sensor reading and send it to the 'DH' and 'DL' address.

Advantages

- Only XBee traffic generated is caused by actual changes in input. This typically means lower traffic.
- Can be used with sleep settings.

Disadvantages

More difficult to setup, requires working knowledge of sleep settings.

DIP switches

The adapter has several DIP switches on the underside of the unit. DIP switch 1 is the leftmost switch. When the adapter is oriented with the mounting tabs facing upwards, the DIP switches are in the ON position when the switches are positioned away from the screw-lock connector.

Switch settings are:

DIP Switch Settings

DIP Switch	Settings
1	ON = Internal Battery Powers Pin #6, OFF = battery power not used.
2	ON = External Supply Powers Pin #6, OFF = external supply not used.
3	ON = Enables a 10K pullup on terminal #1 to 3vdc.
4	ON = Enables a 10K pullup on terminal #2 to 3vdc.

Notes:

- Switches 1 and 2 should not be on at the same time, as increased parasitic battery drain will result.
- If both Switches 1 and 2 are off, then pin #6 will never supply power out.

- Pullup use is not recommended when running from battery power, owing to the constant drain on the batteries.
- Terminal #3 and #4 have no internal pullup. You may require external pullup resisters to be used.

Power output

Terminal 6 is the power out pin. It can be set in either battery pack voltage out or +12VDC at 50mA out if mains powered. To set battery pack voltage output, set dip switch 1 to the on position. To enable +12VDC output, set dip switch 2 to the on position. Setting both to the on position defaults to +12VDC output but incurs extra battery drain as well.

Once the desired power output is selected, the 'P3' setting needs to be set to 5.

Note: In order for consistent output voltage, the XBee Adapter cannot sleep. Using the power out function of this adapter in conjunction with batteries will result in significant additional battery usage.

DIA configuration and programming examples

The Device Cloud Wiki is a Python framework designed to gather information from multiple sources, provide programmatic control over the information collecting, and present the data either to a software platform such as the Device Cloud or directly to the user through a variety of methods.

Python programming examples

This Demo Application uses the Motion Detection with XBee.

Replacement parts

Six-terminal block plug

The terminal block is a Phoenix Contact MSTB 2.5/6-ST-5.08, which can be purchased directly from Phoenix Contact or from an online reseller such as DigiKey as part 227-1015-ND.

Locking power connector

Information on buying the Locking Power Connecter.

XBee display

Note This product is under development

It is not released, nor available by normal channels. Please Contact Digi if you have an interest in testing or discussing options.

Product description

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

DIA configuration and programming examples

Python programming examples

XBee RS-232 adapter

Product description

The XBee RS-232 Adapter provides short range wireless connectivity to any RS-232 serial device.

Note that this Wiki page documents the packaged box product, and not the RS-232 development bare board! It also covers operation when powered full-time, not when batteries are used and the product sleeps.

Assuming the serial end device is a traditional 'slave/server' which answers remote polls and is unaware of the mesh, then load the standard "AT" firmware and not the API firmware. Then use a Digi ConnectPort X gateway or another XBee module with the API firmware loaded to act as the 'master/client' and issue requests directly to end devices via MAC address, and the 'slave/server' XBee will always return responses to the MAC address of the 'master/client'.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

XBee Module Support

All Xbee modules should work in the RS-232 adapter, and they would use the standard AT or API versions. There are not (at this time) any special builds for this adapter. However, X-CTU cannot reflash firmware in this RS-232 module - you will need to move the XBee module to a USB or RS-232 development board to reflash firmware.

Module	Description	Tested	Comments	Output Defaults
XB24-A	802.15.4 on 2.4Ghz	Yes	No access to DTR or DSR	DTR and RTS asserted/high
XB24-B	ZNet 2.5 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes	No access to DSR	DTR and RTS disabled/low
XB09-DM	DigiMesh on 900Mhz	Pending	No access to DSR	DTR is disabled/low, RTS asserted/high

Module	Description	Tested	Comments	Output Defaults
XB24-DM	DigiMesh on 2.4Ghz	Pending	No access to DTR or DSR	
XB08-DP	Point-to-Multipoint on 868Mhz	Pending		DTR is disabled/low, RTS asserted/high

Configuration to consider

These values can be set from X-CTU - or use the CLI or Web UI of a Digi ConnectPort X gateway:

- Set dest_addr (DH/DL) to the MAC of your CPX gateway or the XBee module which asts as master/client. This prevents broadcast responses.
- Set the baud rate and other port characteristics (requires X-CTU?).
- Set dio12_config (P2) to either 4 or 5; 4 (DO Low) causes DTR to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio7_config (D7) to either 4 or 5; 4 (DO Low) causes RTS to be asserted/high upon adapter power up, while 5 causes it to be dropped/low.
- Set dio6_config (D6) to 3 to enable CTS as an input.
- Set dio3_config (D3) to 3 to enable CD as an input.
- Set dio1_config (D1) to 3 to enable RI as an input.

Note DSR cannot be read; there is no configuration required for it.

Pinouts

The RS-232 connector is an industry-standard DB9 male connector with a DTE configuration, similar to a PC serial port. However, there are limitations in the support for DTR/DSR. Pinouts for the connector are:

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
1	CD	Input	17	DIO3	D3	0x0008		
2	RXD	Input	3					
3	TXD	Output	2					
4	DTR	Output	4	DIO12	P2 (See Note)	0x1000	'P2\x04'	'P2\x05'
5	GND	---						
6	DSR	Input	9	DI8	Not Readable	Not Readable		
7	RTS	Output	12	DIO7	D7	0x0080	'D7\x04'	'D7\x05'

Pin	Function	Direction	Module Pin	DIO to rd/wr	AT Command	Mask in IS Response	to Raise/Assert	to Drop/Disable
8	CTS	Input	16	DIO6	D6	0x0040		
9	RI	Input	19	DIO1	D1	0x0002		
9-alt	12vdc Switched Power	Output	18	DIO2	D2 (See Note)	0x0004	'D2\x05'	'D2\x04'

Note on Pin 4/DTR. Some XBee modules do not offer access to raise or lower DTR; the "P2" command is not available.

- XBee modules known **NOT** to support P2/DTR control: **XB24-A** (802.15.4), **XB24-DM**
- XBee modules known to **support** P2/DTR control: **XB24-B** (Znet2.5), **XB24-ZB**, **XB09-DM**

Note on Pin 6/DSR. None of the XBee module allow reading the level of DSR via the "IS" command.

Note on Pin 9. By default it is an input, however setting DIO2 high turns the 12vdc 50mA switched power output on and reading RI/DIO1 returns TRUE if power is on. Setting DIO2 low sets the switched power to tri-state, thus reading RI/DIO1 returns the RI-like status of pin 9. So do NOT connect RI to any device which might be damaged by a +12vdc signal - while it will NOT damage any true EIA/RS-232 compliant device, it can't be good for any device attempting to drive RI low.

Powering 'green' or 'port-powered' RS-232 devices

Some external devices (such as RFID readers or short-haul modems) attempt to draw power from the RS-232 driver circuit. The voltage output from the XBee 232 Adapter may be too low to power such external devices.

However, cross-wiring the RS-232 cable so that the 12vdc Aux-Power (pin 9) from the XBee 232 Adapter connects to the external DTE DSR input (pin 6 of DB9) or the DCE DTR input (pin 4 of DB9) provide a solid solution. A +12vdc signal is well within the RS-232 voltage signal specification, plus the approximately 12vdc 50mA supplied is more power than the 'Port Powered' device expects to tap.

Python programming examples

Polls or Requests sent to field devices: The 'master/client' XBee module should send serial data via addressed unicast with one of the API Transmit Request frames, such as 0x00, 0x01 and 0x10.

Responses or unsolicited data from field devices: If the dest_addr (DH/DL) registers have been set properly, then any serial data received from the field devices will be forwarded to the central 'master/client' XBee module. API Receive Packet frames will be received by the 'master/client' XBee module.

Driving RS-232 control signals

DTR/RTS signals are raised or lowered by sending a 3-byte command with the API Remote AT (command 0x17) - the examples below are coded as Python expects:

- To assert (or raise) the outgoing DTE signal DTR, send the AT command 'P2\x04'
- To deassert (or drop) the outgoing DTE signal DTR, send the AT command 'P2\x05'
- To assert (or raise) the outgoing DTE signal RTS, send the AT command 'D7\x04'
- To deassert (or drop) the outgoing DTE signal RTS, send the AT command 'D7\x05'

Note that these commands affect the pin within one second, yet do not save the state in FLASH. Thus a reboot of the XBee adapter puts the DTR or RTS signal back into the configured default; the factory default is low/not asserted. If it is desired to have DTR or RTS asserted upon power-up, manually set the DIO12/DIO7 parameters to 4.

Why does output DO = Low assert the RS-232 signal and DO = High drop the RS-232 signal? This is how historically TTL communications systems worked. A 5v line was assumed idle, thus pulled weakly up to 5v and representing OFF or binary zero (0). A 0v line was being actively shunted to ground by a powered transistor, thus represented ON or binary one (1). Even today, most RS-232/485 chips assume 0v = 1/on and 5v = 0/off.

Reading RS-232 control signals

To read the signal status, issue the 'IS' command. **You must DELAY at least one (1) second after issuing any of the D2/D7/P2 commands before issuing the 'IS' command or the output status won't be correctly returned in the response**. The IS command returns 6 bytes by direct API, and 5 bytes with ddo_get_param() function, so the last 5 bytes can be decoded as:

- Response[0] should equal '\x01' (is Sample Set Count)
- Digital_mask = (ord(response[1]) * 256) + ord(response[2]); shows which bits of following data are valid
- Response[3] should equal '\x00' (is Analog Data Mask - unless adapter voltage is being read)
- Digital_data = (ord(response[4]) * 256) + ord(response[5]); shows the actual I/O states

The RS-232 signals show as inverted for historical reasons, so a '0' means the signal is HIGH/ASSERTED and '1' means LOW/DROPPED

However, the Auxiliary Power Output (pin 9) shows up per digital logic levels. Therefore, a '0' means power is off and '1' means power is on. The Auxiliary Power Output can drive 12vdc at 50mA if the XBee RS-232 adapter is direct DC powered. If battery powered, driving Auxiliary Power Output more than a second will quickly drain your batteries!

Example Python to detect CTS status

```
def show_CTS_status( digital_mask, digital_data):
    """
    Decode and display the CTS info from 'IS' response words
    Return True if asserted, else False
    """
    if( digital_mask & 0x0040):
        print 'CTS input configured, value = ',
        if (digital_data & 0x0040):
            print 'Low/Dropped'
            return False
        else:
            print 'High/Asserted'
            return True
    else:
        print 'CTS input ignored (not configured)'
    return False
```

XBee RS-485 adapter

Product description

Notation

The XBee RS-485 is claimed to support RS-422 and RS-485. RS-422 means 4-wire (separate TX/RX pairs). RS-485 strictly means 2-wire with the TX/RX function on a single wire pair for half-duplex only. If you want RS-485 4-wire, then configure the product for RS-422.

Pinouts

The XBee RS-485 has a 6-screw removable terminal block. Pin #1 is on the same side as the DIP switches - or if the adapter is mounted (screwed down), then if you face the open ends of the terminals, then pin #1 is on the left (opposite the power connector)

Pin	When RS-422	When RS-485
1	TxD+	Dat+ (B)
2	TxD-	Dat- (A)
3	RxD+	Not Used
4	RxD-	Not Used
5	Ground	Ground
6	12vdc Switched Power	12vdc Switched Power

Note on pin 5: This pin is used as power, signal, and reference ground. Since the Digi AC/DC supply has a floating earth, you will need to tie this to your RS-485 device (or at least the same ground reference they use). This is also the return for the 12vdc switched power.

Note on pin 6: When enabled, provides up to 50mA of 12vdc power - see programming options below

DIP Switch Settings

Switch	Comments
1	Not used
2,3,4	OFF for RS-422, ON for RS-485
5,6	ON to enable BIAS and Termination, OFF to disable

Note on line bias: Line bias is available only when external power is used, but it should always be enabled unless there is a specific reason to disable it. Enabling line bias 'quiets' a floating line, which prevents noise from generating large amounts of XBee traffic.

Note on ON setting: the lebeling on the DIP switch can be hard to see - as of Sep 2009 ***the black DIP switch block used is ON when the switch button is AWAY from the etched numbers***.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide Python Programming Environment and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

Using the Aux-Power output

The auxiliary power output can be used to selectively power up an external RS-485 equiped sensor. When the XBee RS-485 adapter sleeps, the aux-power drop.

To turn the auxiliary power on, send the AT command 'D2\x05' (set D2 to 5).

To turn the auxiliary power off, send the AT command 'D2\x04' (set D2 to 4).

To make use of this power option with sleeping sensors, you will need the following things to occur:

1. Set the XBee sleep settings as required - the length of time to remain awake will be tricky to calculate.
2. Enable the aux-power output to be true
3. if the sensor transmits its data automatically, then this will be sent over the mesh to either the MAC address loaded in the DH/DL register pair, or for Zigbee to the coordinator
4. if the sensor must be polled, then you must enable one of the I/O settings with the XBee so that it sends an dummy message to the host. Then your host application must wait for that dummy message and quickly send any serial poll required.

iDigi Dia configuration and programming examples

Python Programming Examples

XBee RS-232 PH Adapter

Overview

The Digi Xbee RS-232 PH (Power Harvesting) Adapter is a wireless to RS-232 adapter which draws power from the attached RS-232 port, slowly charging a super-cap which functions as a short-term battery. It does not require sleeping, however it can only actively transmit from 5% to 20% of the time, which means it must be idle from 80% to 95% of the time to allow the super-cap to recharge between radio transmissions. Unfortunately the exact duty cycle of operation (or percentage of time sending/receving RF packets) is defined by the device to which the XBee RS-232 PH is connected.

In general the Digi Xbee RS-232 PH Adapter functions identically to the Digi XBee RS-232 Adapter, so refer to Digi XBee RS-232 adapter for most programming questions.

Features which are the same:

The same 9-pin DTE port, matching a standard PC computer including support for DTR/DSR, RTS/CTS, CD and RI

Features which are different:

- No support for the Auxillary Power Out on pin 9 / RI.
- Draws power from DTE inputs RXD, DSR, CTS, CD, RI; can be either high or low (either V+ or V-)
- Cannot 'talk' 100% of the time - a general rule of thumb is one poll-response per second.

XBEE sensors

Product description

The XBee Sensor provides real-time data on temperature, humidity, and light, with the data being transmitted through wireless communications in an XBee network infrastructure. Compact size and battery power enable XBee Sensors to be dropped into facilities easily and unobtrusively while providing reliable communications. Applications include building automation, environmental monitoring, security, asset monitoring, and more.

The readings are of modest accuracy, suitable for environmental monitor but not likely suitable for control systems (See the Accuracy Section below)

There are currently two XBee Sensor product options available:

- XBee Sensor /L/T: Integrated ambient light and temperature sensors
- XBee Sensor /L/T/H: Integrated ambient light, temperature, and humidity sensors

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Python Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

iDigi Dia configuration and programming examples

Python programming examples

Configuration settings

During system start up or configuration, you require:

- D1, D2 and D3 should be set to 2, enabling analog input to the 3 sensors.
- DH and DL should be set to your XBee coordinator (your CPX gateway) or the XBee device to receive the data productions.
- P1 (DIO11) should be set to 3, enabling digital input on the battery monitor pin.

Operational settings

Although you can poll the XBee Sensors, this requires the XBee to be awake most of the time and you battery life will be limited to a few months. You may find having it wake once per 10 or 15 minutes the best solution.

The assumed operation is to place the XBee Sensor into sleep, then have it send the data unsolicited to the XBee node listed in the DH/DL settings.

To enable sleeping operation longer than 1 minute:

- Set IR to 0xFFFF, which effectively disables the 'sample rate' setting.
- Set WH to 125 to allow the sensor hardware to stabilize for 125msec before the XBee reads the analog inputs (note: not all XBee modules support the WH parameter)
- Set the SN/SP pair to enable sleeping for the desired time period.
- SP is the number of 10msec periods to sleep
- SN is the number of SP-periods to sleep for.
- For example, SP=2000 and SP=30 leads to an approximately 600 seconds data productions (30 x 20 second periods or 10 minutes).
- *The SN/SP within your gateway and all XBee routers MUST be at least as large as the values placed into the XBee Sensor product, or you will find the routers (the parents) de-associate the sleeping XBee Sensor (the child) while it sleeps, and thus reject the periodic data productions.*

Formulas

Temperature:

```
temp_C = (mVanalog - 500.0)/ 10.0
mVanalog = (ADC2/1023.0) * 1200
```

Humidity:

```
hum = (((mVanalog * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
mVanalog = (ADC3/1023.0) * 1200
```

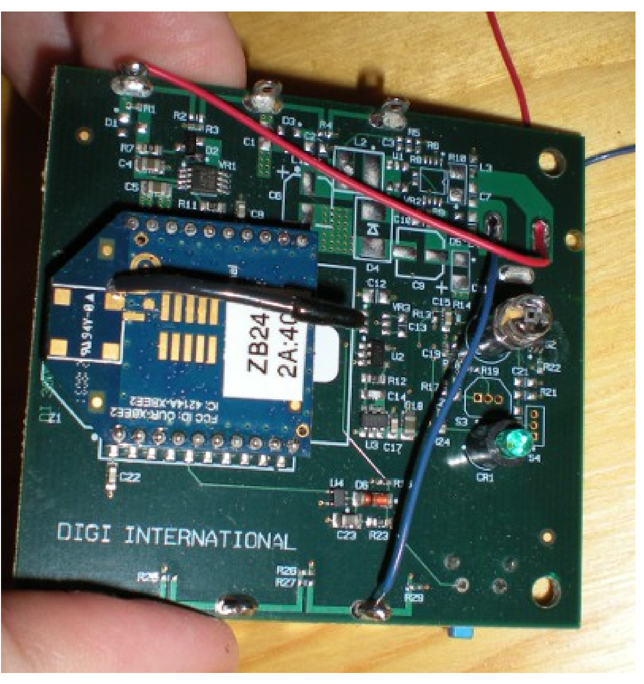
Light:

```
brightness = (ADC1) /1023.0) * 1200
```

Hardware information

Power options

Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.6v up to 6.0v. For testing purposes you can use a stable 5vdc supply, attaching your power leads to the battery tabs as shown in the photo (red is +, blue is -). The wires pass through, but do not connect to the barrel jack socket-holes. This jack is only connected electronically in the 9-30vdc version of this PCB (which is not available).



XBee module support

The XBee L/T/H/ Sensor requires either the 'End Device AT' or 'Router AT' firmware on the XBee. Although it is designed to run on 3 AA batteries, it can operated on any stable voltage supply from about 3.7v up to 6.0v. For testing purposes, you can use a stable 5VDC supply.

Given the need to be low power and to sleep, not all XBee modules are appropriate for this product.

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-A	802.15.4 on 2.4Ghz	Pending				
XB24-B	ZNet 2.5 on 2.4Ghz	Yes		ZNet 2.5 Router/End Device AT, such as 1247	0x2000E	0x2000D

Module	Description	Tested	Comments	Firmware	/L/T DD Value	/L/T/H DD Value
XB24-ZB	Zigbee 2007 on 2.4Ghz	Yes		Zigbee End Device AT, such as 2864	0x3000E	0x3000D
XB09-DM	DigiMesh on 900Mhz	Pending				
XB24-DM	DigiMesh on 2.4Ghz	Pending				
XB08-DP	Point-to-Multipoint on 868Mhz	No	Consumes too much power	N/A	N/A	
XB09-DP	Point-to-Multipoint on 900Mhz	Pending				

Notes:

Although the XBee XB24-ZB family includes a firmware named "Zigbee Router/End Device Sensor", that firmware is for the 1-wire Sensor Adapter - **it is NOT for the Sensor /L/T or /L/T/H**. Installing this firmware will result in bad values being read.

Accuracy

XBee L-T-H sensor adapter

The /L/T/H product is NOT calibrated by Digi. This means multiple units placed side-by-side out of the box will show a higher than desired variability. However, the sensor readings are fairly linear and modest software calibration can greatly improve the accuracy.

Temperature

The stated accuracy per the datasheet (and component supplier) is +/- 2 DegC (+/- 3.6 DegF).

They are not designed for industrial control, and even the temperature within a normal room varies by many degrees based on drafts, heat or cold sources, and how far the sensor is above the floor. Anyone who looks at a thermostat, reads the '72' and believes the entire room is a perfectly constant 72 degrees Fahrenheit is being foolish.

Ad-hoc tests show that a simple fixed offset added or subtracted to the readings allow them to be used in normal building automation with satisfactory result. For example, a test of four units showed that adding constants to the raw 0-1023 value allowed all four to return the same temperature to within +/- 0.25 DegC most of the time and to within +/- 0.75 DegC all of the time. The magnitude of these binary constants within this test of four units were (-4, 14, -15 and 5). You could use a float constant scaled as DegC or DegF instead.

These constants were calculated by taking 5 readings over five hours, then examining the average deviation from the desired value. For example, one unit returned the values 739, 735, 734, 700 and 702 when the expected values were 738, 732, 732, 690, and 694. Thus adding a -4 (subtracting 4) from the value received resulted in a better value. The expected value was calculated based on the temperature reading of a third party device 'trusted' as correct.

Humidity

The stated accuracy per the datasheet (and component supplier) is:

Interchangeability: +/- 5 %RH (0%RH to 59%RH) and +/- 8 %RH (60%RH to 100%RH)

Accuracy: +/- 3.5%RH

This means if you take a dozen factory-fresh units, allow them to stabilize within a 70%RH environment, then you may see readings range from 62%RH to 78%RH. This is the 'Interchangeability' clause.

However, if the user does modest linear software calibration (primarily fixed offset), then the same dozen devices can show the 70%RH as 66.5%RH to 73.5%RH. This is the 'Accuracy' clause.

Light

The light sensor in the adapter returns values from 0 to about 1100 based on light intensity - it does NOT return any standard measure, and the actual readings vary based on the opacity of the label applied and so on. It can be used to easily sense that a room is brighter or darker. For example, you can use it to turn security lights on when the sun goes down - or to turn off room heaters and computer displays when the overhead lights are turned off, indicating that the room is not occupied.

Note These are simple, low-cost "light intensity" sensors that are intended to be used for a wide variety of applications. However, they were never intended to measure the lux of a particular scene. For this reason, we're not able to provide a formula to convert light intensity to lux.

You could not use it (for example) to test that a workbench has an exact luminous intensity of "X" cd/m2. If you require such measurements, you should use a standard light sensor with a 0-10v or 4-20mA signal in a AIO adpater.

Also note that the duration of the actual light detection is short, so it works best with either sun light or non-flickering incandescent or DC-powered halogen lamps. Measure light with standard fluorescent light will result in the light value varying over a range of values (many percentage points). This does not prevent your system from detection 'the room is light' or 'the room is dark', but you will need to accommodate this variation with a hysteresis or dead-band calculation.

XBee Smart Plug

Product description

Digi's XBee Smart Plug is a intelligent outlet that can measure power usage and provide basic power management to attached appliances via the standard electrical outlet. For more details of safety and usage concerns, please review the *Drop-in-Networking Accessories user guide* at: digi.com/support/documentation/90000891.pdf.

Major features of the XBee Smart Plug:

- Programmable power relay controlling the integrated outlet
- Current sensor monitoring the power consumption of the integrated outlet
- Light sensor
- XBee network extension
- Built-in mounting tab for U.S. versions.

Supported platforms

The XBee Smart Plug uses the XBee ZB protocol. All of Digi's gateway products that use the XBee ZB protocol are compatible with the XBee Smart Plug.

Programming options

The XBee Smart Plug uses the Router AT function set. The XBee Router AT function set uses the AT commands to query, configure and control the XBee RF module in the XBee Smart Plug.

The AT commands for the XBee Smart Plug's unique features are:

Light sensor

D1=2 (Analog input)

Current sensor

D3=2 (Analog input)

Power Relay

- D4=4 (Turns outlet off)
- D4=5 (Turns outlet on)

Note Settings do not take affect without the corresponding apply changes command, which is 'AC'. See the XBee module product manual for more information.

DIA configuration and programming examples

The XBee Smart Plug is supported in the DIA (Digi Device Integration Application), a software component for quick device integration.

The Device Cloud Wiki is a Python framework designed to gather information from multiple sources, provide programmatic control over the information collecting, and present the data either to a software platform such as the Device Cloud or directly to the user through a variety of methods.

See Example Smart Plug for an example of a XBee Smart Plug configuration.

Python programming examples

The XBee Smart Plug may also be controlled from a Digi gateway device using the embedded XBee Python module's function calls ddo_get_param() and ddo_set_param(). Using these function calls, one can enable the current and light sensors, enable the power relay, and take a sensor reading.

Below is a quick demonstration of how to send such commands. The address (addr argument) used for the XBee Smart Plug varies depending on its 64-bit MAC address.

```
from xbee import ddo_get_param, ddo_set_param

addr = '[00:13:a2:00:40:48:59:95]!'

ddo_set_param(addr, 'D1', 2) ## Enabling light sensor
ddo_set_param(addr, 'D3', 2) ## Enabling current sensor
ddo_set_param(addr, 'D4', 5) ## Turning the power relay on
ddo_set_param(addr, 'AC') ## Making sure to apply changes

ddo_get_param(addr, 'IS') ## Getting a sensor reading
```

For a interactive demonstration of the XBee Smart Plug Interactive Demo provides a means to control the power relay of the XBee Smart Plug while retrieving and interpreting the sensor values into real-world units.

XBee USB adapter

Product description

The XBee USB Adapter provides short-range wireless connectivity to any USB device. Unlike an embedded wireless module, which requires design integration and development time, these off-the-shelf adapters provide instant wireless connectivity to existing USB devices. All XBee adapters can be used with Digi's ConnectPort X gateways for data aggregation and IP connectivity

The XBee USB Adapter is a bus-powered device.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible Device Cloud Wiki Application to collect and aggregate information.

iDigi Dia configuration and programming examples

Python programming rxamples

XBee Wall Router

Product description

The XBee Wall Router is a XBee network extending device that is mains powered and additionally provides temperature and light sensor input. It is designed to be used in conjunction with other XBee enabled devices. For more details of usage and configuration, please review the *Drop-in-Networking Accessories user guide* at: digi.com/support/documentation/90000891.pdf.

Major Features of the XBee Wall Router:

- Provides XBee network extension
- Light sensor
- Temperature sensor

Supported platforms

The XBee Wall Router is supported on:

- XBee-PRO ZB Wall Router (ZB or ZigBee)
- XBee Smart Energy Range Extender (SE)
- Xbee-PRO DigiMesh 2.4 Range Extender (DM24 - there is no DM 868/900Mhz versions!)

Five versions of each exist with interchangeable AC power pins for use in: US, EU, UK, Australia and Japan. All of Digi's gateway products that support ZB, SE and DM24 are compatible.

Note The units are chemically welded shut for UL safety reasons. Therefore, you cannot buy a ZB unit and for example convert it into a DM 900Mhz unit by just swapping XBee modules.

Supported firmware

- For the ZB Wall Router, run 'ZIGBEE AT Router' firmware. Newer firmware sets include a better 'ZIGBEE ROUTER AT (WALL RT)' firmware which supports 20 end-device/children nodes instead of the traditional 12.

- For the SE Range Extender, run 'SE RANGE EXTENDER' firmware.
- For the DM24 Range Extender, run 'XBEE PRO DIGIMESH 2.4 WALL_ROUTER' firmware.

In all 3 products, any normal 'router' firmware can be used.

However, never load one of the special ANALOG, DIGITAL, SENSOR, 232 or 485 firmwares since these will assume I/O pin assignments which are not true within the product - recovery may not be possible.

Programming options

The XBee Wall Router uses a Router AT command set. Due to not having serial port access, all configuration must be done via RF through remote AT commands.

Note The information below does not apply to the XBee Smart Energy Range Extender! Use the Smart-Energy framework for that product.

Enabling the sensors

Enabling the light sensor

D1=2 (Enables the DIO line tied to the light sensor for analog input)

Enabling the temperature sensor

D2=2 (Enables the DIO line tied to the temperature sensor for analog input)

Note Due to being mains powered, self heating occurs in the device and the temperature sensor may read several degrees higher Celsius then its outside environment. It is estimated that this affect is approximately 4 degrees Centigrade.

Obtaining the data

Since the Wall Router is always awake, you can poll with the remote AT request for the 'IS' value. The result is a packed binary structure defined in the various XBee manuals in sections titled "Analog and Digital IO Lines" and/or "IO Sampling".

Yet polling has two disadvantages:

- It creates two RF packets instead of one
- If the Wall-Router is offline, literally your outbound XBee channel may be limited for up to 30 seconds.

The better solution is to set the XBee IR parameter to a non-zero value - for example 60,000 (0xEA60) to cause the Wall Router to asynchronously send the same 'IS' response once per minute without the need to issue the poll. You would bind on the socket ("", 0xe8, 0xc105, 0x92) to receive all incoming 'IS' responses and decode them per the XBee documentation.

General XBee Programming Options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X [2] gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X [3] offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud

Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's[4] reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

iDigi Dia Configuration and Programming Examples

The Wall Router uses the Dia driver named XBR (xbee_xbr.py). The YML example below adds support for a Wall Router or Range Extender with the extended (MAC) address of "00:13:a2:00:40:6b:16:31!". The unit sends in its data (without polling) once per minute. In this example the device has been called hallway, so its data would exist in iDigi channels named hallway.temperature and hallway.light.

```
- name: hallway
  driver: devices.xbee.xbee_devices.xbee_xbr:XBeeXBR
  settings:
      xbee_device_manager: zb_man
      extended_address: "00:13:a2:00:40:6b:16:31!"
      sample_rate_ms: 60000
```

Python Programming Examples

Calculating the millivolts

The formulas to create real-world readings for both analog sensors requires knowing the millivolts which the Xbee within the wall-router/range-extender sees. The data sample returns the ADC value as between 0-1023, which your code needs to return to millivolts (mv). You should force the result to a floating point value.

The millivolt formula for ZB / SE:

```
mv = float(value * 1200) / 1023
```

The millivolt formula for DM (the DM VRef varies product by product - for the Range Extender it is 3.3vdc):

```
mv = float(value * 3300) / 1023
```

Calculating the light reading

The light sensor is not reading a standardized light-level. It is a number which increases with brightness and decreases with dimness.

```
light = mv
```

Calculating the temperature reading

Given the mv, we can directly calculate degree Celsius. Since the AC/DC conversion within the products tends to be self-heating, we subtract a fixed 'fudge-factor' which tends to be around 4 DegC. :

```
degc = ((mv - 500.0) / 10.0) - {fudget factor of about 4.0 degC}
```

If you wish to have degree Fahrenheit instead, you can convert the degree Celsius like this:

```
degf = (degc * 1.8) + 32.0
```

XStick

Product description

The XStick is a USB peripheral module adapter that provides short-range wireless connectivity to an XBee network.

Programming options

There are many options to consider when making wireless programmatic access to an XBee network of devices or device adapters. In broad terms, one may write a program which runs on a PC to interact with a network or one may use a gateway device, such a ConnectPort X gateway.

When using a PC, one may consider using the simplistic and easy "AT-Command" mode for the XBee attached to the computer. Although using this mode is straight-forward it does not offer one as fine of control as when using the "API mode" firmware option.

One may also consider using a Digi ConnectPort X family of gateway device to provide additional intelligence and flexibility when connecting to a network of wireless devices. The ConnectPort X offers a customizable Digi Python programmers guide and instant connectivity to the Device Cloud Management Platform. The ConnectPort X offers users the ability to choose to write their own applications from the ground-up, using Digi's and Python's reference and example information, or to use the highly extensible DIA Application to collect and aggregate information.

iDigi Dia configuration and programming examples

Python programming examples

General Python

This category covers use of Python samples and general information.

ADC values

Program to receive ADC values from ZigBee

(ZigBee routers and sensors) Receive and parse the ADC values coming from the ZigBee routers and sensors.

How does it work?

This application receives data packets from routers and end devices and prints the parsed output to the console. These data packets include Light, Temperature and Humidity values.

Test files

This sample program contains two files, Get_Router_Sensor_reading.py and RouterSensor_reading_application_notes.doc. The program file is Get_Router_Sensor_reading.py.

ADC value test sample application

The ADC Value sample application can be found here: Get_Router_Sensor_reading.zip.

Basic usage

See Application Note here: RouterSensor_reading_application_notes.zip.

Sample of Get_Router_Sensor_reading.py file:

##

```
# Copyright (c)2012, Digi International (Digi). All Rights Reserved.       #
#                                                                          #
# Permission to use, copy, modify, and distribute this software and its    #
# documentation, without fee and without a signed licensing agreement, is  #
# hereby granted, provided that the software is used on Digi products only #
# and that the software contain this copyright notice,  and the following  #
# two paragraphs appear in all copies, modifications, and distributions as #
# well. Contact Product Management, Digi International, Inc., 11001 Bren   #
# Road East, Minnetonka, MN, +1 952-912-3444, for commercial licensing     #
# opportunities for non-Digi products.                                     #
#                                                                          #
# DIGI SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED   #
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A          #
# PARTICULAR PURPOSE. THE SOFTWARE AND ACCOMPANYING DOCUMENTATION, IF ANY, #
# PROVIDED HEREUNDER IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND. #
# DIGI HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,         #
# ENHANCEMENTS, OR MODIFICATIONS.                                          #
#                                                                          #
# IN NO EVENT SHALL DIGI BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,      #
# SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS,   #
# ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF   #
# DIGI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.                #
############################################################################
```

```
import os
import sys
import socket
import struct
```

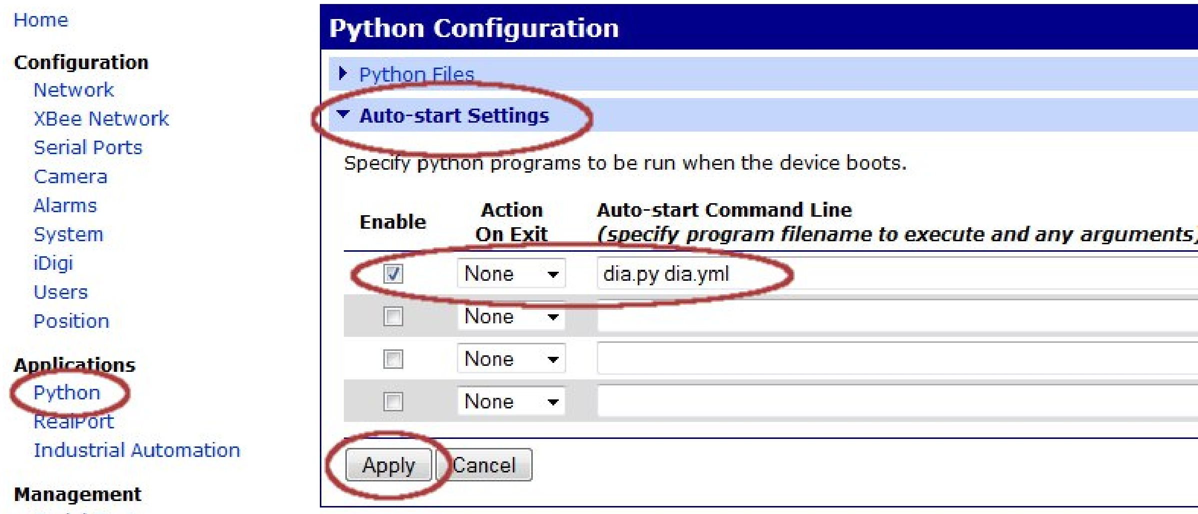
```
import time
import zigbee
import datetime
import traceback
from socket import *

humidity = 0
light = 0
temp = 0

def parse_packet(payload):
    print "parser function"

def calc_light(value):
    mv = float(value * 1200) / 1023
    return mv

def calc_temp(value):
    mv = float(value * 1200) / 1023
    degc = ((mv - 500.0) / 10.0) - 4.0
    #degf = (degc * 1.8) + 32.0
    return degc

def calc_humidity(value):
    mv = (value / 1023.0) * 1200
    humidity = (((mv * 108.2 / 33.2) / 5000 - 0.16) / 0.0062)
    return humidity

try:

    sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)
    ''' Bind to ("", 0xe8, 0xc105, 0x92)to receive all incoming 'IS' responses
    and decode them per the XBee documentation
    '''
    sd.bind(("", 0xe8, 0xc105, 0x92))
    print "socket is bound"

    while 1:
        try:
            payload, src_addr = sd.recvfrom(255)
            len_payload = len(payload)

            print datetime.datetime.now()
            src = src_addr[0][1:24]
            print "Source Address: %s" %src

            #print len_payload
            if len_payload == 8:
                fixed_byte, digital_bits, analog_set, lig, tmp = struct.unpack
(">bhbhh", payload[:8])
                light = calc_light(lig)
                temp = calc_temp(tmp)
                print "light - %s" %light
                print "temperature - %s" %temp

            elif len_payload == 10:
                b1,b2,b3,b4,b5,b6,lig,tmp = struct.unpack(">bbbbbbhh", payload
[:10])
```

```
                print b1
                print b2
                print b3
                print b4
                print b5
                print b6
#                print b7
#                print b8
#                print b9
#                print b10
#                print "fixed_byte is %d" %fixed_byte
#                print "digital_bits is %d" %digital_bits
#                print "analog_set is %d" %analog_set
                print "lig is %d" %lig
               print "temp is %d" %tmp
               light = calc_light(lig)
               temp = calc_temp(tmp)
               print "light - %s" %light
               print "temperature - %s" %temp


           elif len_payload == 12:
               print "sensor"
               print len_payload
               fixed_byte, digital_bits, analog_set, dont_know_bit1, dont_know_
bit2, lig, tmp, humidity = struct.unpack(">bhbbbhhh", payload[:12])
               light = calc_light(lig)
               temp = calc_temp(tmp)
               humidity = calc_humidity(humidity)
               print "light is %d" %light
               print "temp is %d" %temp
               print "humidity is %d" %humidity
           else:
               print "invalid payload"
               print len_payload

           print " "

        except Exception, e:
            print "Exception %s" %e
            traceback.print_exc()

except socket.error, s:
    print "socket exception: %s" %s

sd.close()
print "socket is closed"
```

Auto-start Python on a Digi gateway

By web interface

The most direct way to enable auto-start is via the device's web interface. Select the Python link on the left, then the Auto-Start Settings. You will be shown the four scripts you can auto-start. Enter the program name, including the ".py" extension. *Do not include the command "Python"!*



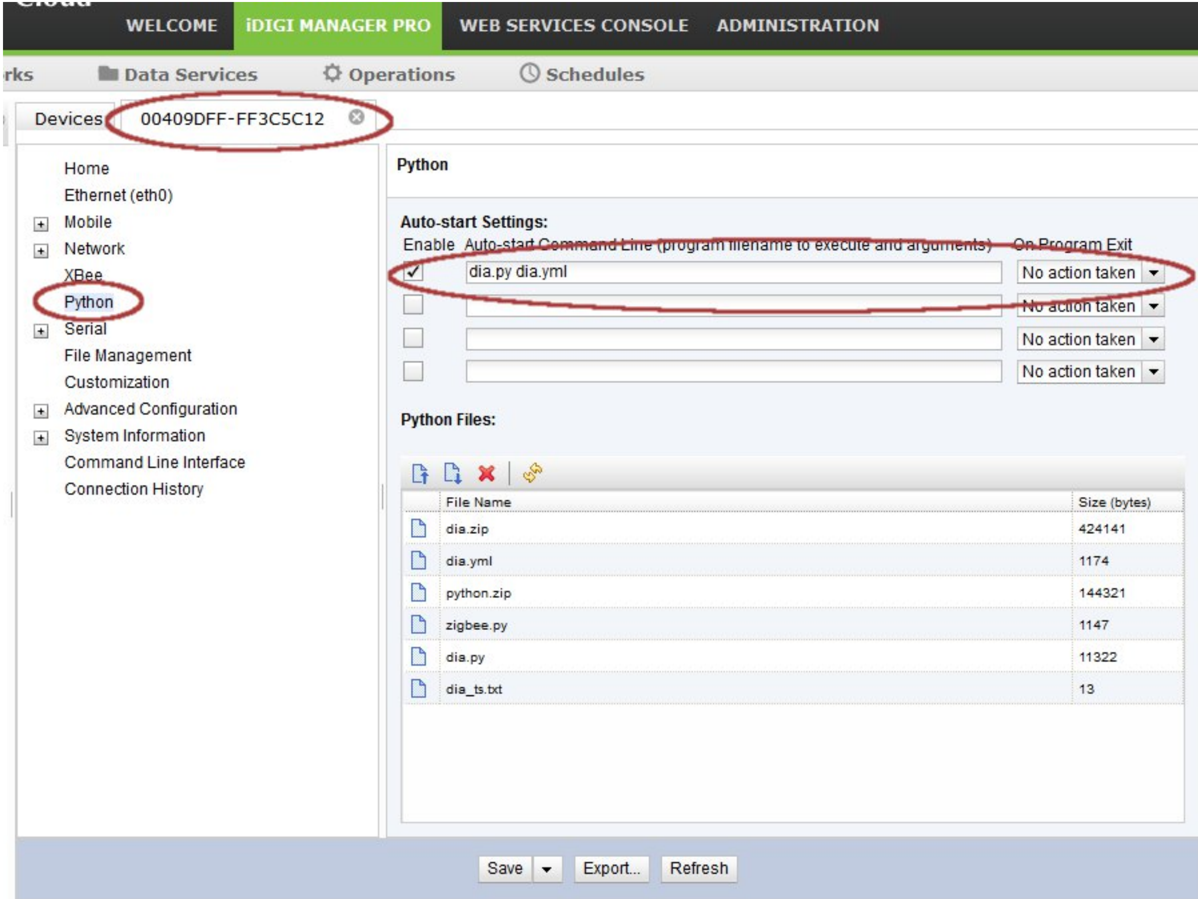
The On-Exit actions default to None, but can be set to restart the script, or reboot the gateway. **However a warning - never enable a restart/reboot option while debugging your code**, for having a simple typo in your main routine can cause the gateway to reboot in a tight loop which makes recover difficult (impossible by remote Device Cloud access). Users of the DIA framework should review this helpful change to dia.py: Rapid Reboot Detection.

Notes on ConnectPort X2e ZB

- Selecting the Python link on the left directly shows you the Auto-Start settings. Unlike on the ConnectPort X2/X4, Python files are NOT shown on this page. The Python files are accessed under the File Management link.
- The ConnectPort X2e ZB has built in Rapid Reboot Detection. The faster a reboot occurs, the longer the X2e/Linux kernel delays before auto-starting Python scripts.

By device manager

Setting auto-start via the Device Manager web console is much like setting it via the web interface. Note the above warning about the restart/reboot setting! if the device is rebooting every few seconds due to a Python error or simple typo, you will not be able to recover by Device Cloud access.



By RCI/SCI

The Python Auto-Start settings are held within the 'Python' settings group. See Digi's RCI and SCI documentation to understand how to read and write settings.

Request

```
<rci_request version="1.1">
 <query_setting>
  <Python />
 </query_setting>
</rci_request>
```

Response

```
<rci_reply version="1.1">
 <query_setting source="current" compare_to="none" encrypt="none">
  <Python><state>on</state><command>dia.py
dia.yml</command><onexit>none</onexit></Python>
  <Python
index="2"><state>off</state><command></command><onexit>nosne</onexit></Python>
  <Python
index="3"><state>off</state><command></command><onexit>none</onexit></Python>
  <Python
```

```
index="4"><state>off</state><command></command><onexit>none</onexit></Python>
 </query_setting>
</rci_reply>
```

Digi DIA notes

The DIA framework requires at least 2 special files to run - **dia.py** and **dia.zip**. You run the dia.py script to start the framework, and this file handles the diverse platform differences, mounting the dia.zip and running the framework from within. By default the dia.yml configuration is embedded in the dia.zip file. Some users prefer to manually keep a copy of dia.yml outside of the dia.zip to simplify remote browsing and changes. If this is done, you need to auto-start "dia.py dia.yml".

As of DIA version 2.0, creating a simple text file in the Python area named "nospin.txt" will enable dia.py to watch for a rapid reboot situation, which means the gateway has been rebooted more than 10 times in 20 minutes. This causes dia.py to delay launching the dia.zip files. See Also: Rapid Reboot Detection.

Digi ESP notes

The Digi ESP (Eclipse for Python) IDE use a default start file name of **dpdsrv.py**. If you use ESP to upload your Python code, you can safely put dpdsrv.py into your auto-start setting instead of dia.py.

Generally, dpdsrv.py does nothing more than run your script, which means you can often safely bypass dpdsrv.py and run your own main script. However, opening the dpdsrv.py script in an editor allows you to see what it is doing. Upon some platform (like a Windows PC), the dpdsrv.py *DOES* create a corrected search path so cannot be bypassed.

ConnectPort FTP client capabilities

Using Python's FTP client module on the ConnectPort gateway

Introduction

A lot of users may already be taking advantage of the convenient storage capabilities that the iDigi cloud has to offer.

When using iDigi to store data from your sensors, it's important to remember that the storage is only temporary and is not meant to store data indefinitely. The general protocol for using this type of storage would be to place your data "in the cloud" until a client program can use iDigi web services to retrieve the data and store remotely in some type of database, etc.

The ability to temporarily store this data using iDigi and have remote access to it using iDigi web service is extremely convenient; however, many customers have pre-existing host applicati ons assuming the remote devices FTP data files in (for example) Comma-Separated-Value format. Therefore FTP may be required for legacy reasons.

Note This Wiki page and the Python ftplib module only **cover use of a Digi gateway as an FTP client calling out to external servers**. Python.org does not seem to offer a standard FTP server module - only the FTP client. Also, the ftplib.py file is NOT by default on the Digi device. You will need to manually copy the correct version of the file to the gateway.

How can I benefit from using an FTP client from my ConnectPort?

For reasons discussed above and to further customize your application, the ability to move data to a remote storage location directly from the ConnectPort device can come in handy. Some examples might include copying sensor data to an internal (or external) FTP server for further monitoring, indefinite storage capabilities or data redundancy.

Another possibility would be for the script that is running on the ConnectPort to use its own internal logic to determine which data to send, what format to send it in, and to which particular location or website. You may wish to move the larger data files to a remote site in order to free up memory on the ConnectPort device, select sensor data only, or build your own HTML page with sensor data and upload that to a remote website, all from within the ConnectPort!

How can I include FTP client capabilities in my Python scripts?

Enter *ftplib.py*, a useful Python module that exposes one class which acts as an FTP client.

The FTP class inside of this module gives us the capability to connect and establish a client session with a remote FTP server. A lot of the FTP protocol is handled for you behind the scenes, so it's extremely easy to use.

Now let's get on to the basics of using the FTP client module.

You will first need to import the ftplib.py module so that you can import and reference it in your Python scripts.

```
from ftplib import FTP
```

Next, for FTP servers that require a login, you must fill in the typical server address, user name and password information (Change according to your login information. For anonymous login, please see below):

```
ftpServer = "ftp.myserver.com"
ftpUser = "myuser"
ftpPwd = "mypassword"
```

You may also wish to include two more variables to indicate the remote directory you wish to store the file in, as well as the name of the file(s) you will be transferring:

```
ftpDir = "/"
txtfileName = "sample.xml"
binfileName = "picture1.bmp"
```

A few things worth noting: if you leave the "ftpDir" as it is, the client will simply store the file in the initial directory you are placed in after logging into the FTP server, which is usually your user "root" directory. (Information on changing directories is discussed further down.)

The *ftplib* module only contains one class, called FTP. This class only serves one purpose and that is to implement the client side of the FTP protocol. The *FTP class* includes different methods for connecting to the FTP server (*connect*()), as well as a method for passing in login credentials (*login*()). However, using the library can be much simpler than that by passing in any login credentials as parameters when initializing the FTP client object. The connect() and login() methods are automatically called (if needed) when the object is initialized, decreasing the number of lines in your script. For example, creating the FTP client object would simply take one line and look something like this:

```
ftpClient = FTP(ftpServer, ftpUser, ftpPwd)
```

In this example, we have called to create the object, passing in the server address and login credentials. This will automatically call *connect*() and *login*() for us behind the scenes, which means we are returned an object which will serve as our client to the FTP server we are logged into.

Note For anonymous login, the last two parameters would not need to be included.)

In my example, I have called this object "ftpClient". Please note, you can do this all in one line as in the example above, or you can call these methods individually. An additional timeout value may also be passed in as the last parameter.

If we want to store the file in a directory other than the root, we simply use the cwd() method on the ftpClient object, like this:

```
ftpClient.cwd(ftpDir)
```

After we have successfully opened up an FTP session and changed directories (if needed), we will want to first obtain a file handle to the local file on the ConnectPort that we wish to upload. This file could already exist on the ConnectPort, or it could be a file that is created dynamically by another function in your script.

Note It's a good idea to wrap this up in a try/catch block, in case the file is not found! Also note we're opening the file up using the open method's "rb" parameter, since our Wiki example is going to show how to upload a binary image, as opposed to a text file:

```
fhandle = open ("WEB/Python/" + binfileName , "rb")
```

Once we have successfully obtained a file handle, we are ready to send the file over to the remote FTP site.

The FTP class exposes two types of methods, depending on the type of file you are working with (text or binary).

If you're transferring binary files, you would use the appropriate "*bin" methods. If you will be transferring a text file, you would want to use the appropriate "**lines*" methods.

Those of you already familiar with FTP communication may have already guessed at this point that the ftpClient object will see which methods you are using (binary or text) and automatically set the FTP transfer type on your behalf. This helps simplify your scripts and also eliminates extra script lines.

Here are a examples of each type of transfer:

If you wish to transfer a text file (for instance an XML file) to a remote location, you would use the "*lines" methods such as in the following example to transfer an ASCII text file:

```
ftpClient.storlines("STOR " + fileName, fhandle)
```

If transferring an image file (for instance an image file placed on the ConnectPort via an attached Watchport camera), you would use the "*bin" methods, such as the following example to transfer a binary file:

```
ftpClient.storbinary("STOR " + fileName, fhandle)
```

(The *storbinary*() method also takes an additional, option parameter called blocksize. More information on this can be found in the online documentation provided by the link below.)

Once your file has been transferred and has reached its destination, you can then close the FTP session and connection by calling the *quit*() method. Using the *quit*() method is the correct method to use to follow correct FTP protocol. By doing this you're sending a 'QUIT' message to the FTP server, which will essentially close the connection and end your session.

Note Don't forget to close the file handle to the file you have transferred as well!)

```
ftpClient.quit()
fhandle.close()
```

Optionally, you may use the close() method, such as in the following example (remembering to close both handles):

```
ftpClient.close()
fhandle.close()
```

It's important to note that both of these methods will render your current ftpClient object useless after calling. In other words, you must instantiate another FTP object for any further FTP client functionality. Until then, any subsequent methods called on your previous FTP object (ftpClient) will throw an exception.

Simple demonstration script

Now, to put this all in a script that you can copy, edit, save and try for yourself:

```
import os
import sys
from ftplib import FTP

# TODO: Change these next variables accordingly
ftpServer = "ftp.myserver.com"
ftpUser = "myuser"
ftpPwd = "mypassword"
ftpDir = "/images"
fileName = "snapshot.bmp"
```

```
print "starting..."
try:
    print "logging into ftp server..."
    print "  (this operation will time out if not successful.)"
    ftpClient = FTP(ftpServer, ftpUser, ftpPwd)
    ftpClient.cwd(ftpDir)
    fhandle = open ("WEB/Python/" + fileName, "rb")

    print "uploading file %s..." % fileName
    ftpClient.storbinary("STOR " + fileName, fhandle)
    print "done uploading!"
    print "close this ftp session and file handle..."
    ftpClient.close()
    fhandle.close()
except Exception, ex:
    print "Exception: ", ex
    sys.exit()

print "exiting..."
```

For more information, please visit: http://docs.Python.org/library/ftplib.html.

Using Digi to 'copy' file between 2 FTP servers

Here is a second example which uses the Digi device to both COPY and GZIP compress a file from a local FTP server over cellular to a remote FTP server. Notice the manual call of gc.collect, see Python garbage collection because on the Digi there is a single, persistent instance of Python running, and even discarded memory objects persist beyond the running of a single script.

Note: you will need to manually upload both ftplib.py and gzip.py.

```
import ftplib
import StringIO
import gzip
import gc

# we want to copy "192.168.196.6:/Temp/code.txt" to "myhost.com:/Temp/code.zip"
my_fil = 'code'
my_fil_src = '%s.txt' % my_fil
my_fil_dst = '%s.zip' % my_fil

server="192.168.196.6"
port=21
session=ftplib.FTP()
print 'Connecting to local FTP at %s:%d' % (server,port)
session.connect(server,port)
session.login('user','password')
session.cwd('/Temp/')
# we avoid saving temp-copy in FLASH - StringIO is like RAM file
buf = StringIO.StringIO()
print 'Fetching file %s' % my_fil_src
session.retrbinary('RETR %s' % my_fil_src, buf.write)
session.quit()

# we need to 'rewind' the RAM file
buf.seek(0)
print 'Original size is %d bytes' % len(buf.getvalue())
buf.seek(0)
```

```
print 'Compressing file %s' % my_fil
zipr = StringIO.StringIO()
# we need to use filename to put correct 'name' into ZIP file
tmpf = gzip.GzipFile(filename=my_fil_src, mode='wb', fileobj=zipr,)
tmpf.write(buf.getvalue())
tmpf.close()
# we no longer need tmpf or buf - zipr is zipped version of buf
del buf
del tmpf

# we need to 'rewind' the RAM file
zipr.seek(0)
print 'Compressed size is %d bytes' % len(zipr.getvalue())
zipr.seek(0)

server="myhost.com"
port=21
print 'Connecting to remote FTP at %s:%d' % (server,port)
session.connect(server,port)
session.login('user','password')
session.cwd('/Temp/')
print 'Storing file %s' % my_fil_dst
session.storbinary('STOR %s' % my_fil_dst, zipr)
session.quit()

# clear any garbage NOW, in case thsi script fires repeatedly
del session
del zipr
print gc.collect()
```

ConnectPort x

Python program for ConnectPort X products

ConnectPort X Test (Python program) This example program sets hex value to KY parameter.

Test files

This sample program contains two files. File name "ReadMe.doc" and "Set_ddo_param_KY_with_hexvalues.py".

ConnectPort hex value test sample application

The Set_ddo_param_KY_with_hexvalues.py Python Test sample application can be found here: Set_ddo_param_KY_with_hexvalues.zip.

Basic Usage

Provide input in values below;

1. DESTINATION = “Provide the Extended address or OUI of the node to which KY(encryption) needs to be set”
2. Value = Hex value for the KY parameter

```
Sample code;
# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'
```

Sample of Set_ddo_param_KY_with_hexvalues.py file:

```
''' This program accomplish setting hex value to KY(encryption) parameter
    without any error.
        KY(Link Key) - Set the 128-bit AES link key.
        KY parameter take either int or string as values,
        but we cannot provide hex values, this application helps in providing
        hex values using a user defined function to the KY parameter.
'''

import sys
import os
import zigbee
import xbee
from _zigbee import *
import time
import traceback

# Provide extendded address(OUI) of the node to which KY should be set
DESTINATION="00:13:a2:00:40:66:a3:02!"
# Provide the hex value
value = '0xe2c01e6b9df3ea7a33b2d7c981c04d23'

# Converts a character string of hex digits into a byte string
def hex_str_to_bin_str(hex_string):
```

```
    start_index = 0
    if hex_string[0:2] == "0x":
        start_index = 2
    result = ""
    for index in range(start_index, len(hex_string), 2):
        byte = int(hex_string[index:index+2], 16)
        print byte
        result += chr(byte)
        print result
return result

try:
        # ddo_set_param - set a Digi Device Objects parameter value.
        # KY(Link Key) - Set the 128-bit AES link key.
        zigbee.ddo_set_param(DESTINATION, "KY", hex_str_to_bin_str(value))
        # EE(Encryption Enable)-this parameter set the encryption enable setting.
      zigbee.ddo_set_param(DESTINATION, "EE", 1)
      # WR - Write parameter values to non-volatile memory so that parameter
modifications
      # persist through subsequent resets.
          zigbee.ddo_set_param(None, "WR", 1)
      print "Writing parameters, waiting five seconds..."
      time.sleep(5)

except Exception, e:
    print "Exception %s" %e
    straceback.print_exc()

print "end of the program"
```

Creating run.py reading ZIP files

Launching a serious project - a RUN.PY and pulling code from ZIP files

If you create a serious project, you'll discover that manually uploading a dozen .PY or .PYC files to your Digi product becomes tedious and error prone. Fortunately, you can pack your project files into a single ZIP file for uploading.

This example project has eleven Python files, nine of which are bundled into a single ZIP file. Then the System Path must be updated to enable the Python interpreter to locate the ZIP'd files.

A real example file list:

- **run.py** (see below) is the small script which connects all the pieces together and runs the main routine. There is nothing magic about the name - it could just as easily be zyx.py or destroy_world.py. However, naming it **run.py** will remind you in a few years which script 'runs' your project.
- **ab_logix.zip** contains the project's nine sub-scripts which parse DF1 packets, manage the Ethernet/IP socket to the PLC and so on. It can have any name, but the name will be hard-coded into the run.py. Any tool creating common ZIP files can be used - a free one is: 7-Zip Open Source Tool.
- **dcwan_config.py** is an uncompressed file used to configure the project. It could be included within the ZIP, but leaving it outside makes configuration changes easier. For this project it defines the local PLC IP address and selecting the TCP and UDP ports to listen on for cellular polls. The main scripts IMPORT this file, using constants defined to modify their behavior.
- **Python_ext.zip** is supplied with your Digi product and contains other common Python modules. You only need to include this if your code imports any of the modules - for this project, the COPY module is required.

Running the project:

```
Python run.py
```

The RUN.PY script:

```
## run.py

"""\
To run this, use command line: Python run.py (options)

Where options include:
(none yet)
"""

import os
import sys
```

```
if sys.platform.startswith('digi'):
    # on Digi, Python files are in WEB/Python
    sys.path.append(os.path.join('WEB', 'Python', 'Python_ext.zip'))
    sys.path.append(os.path.join('WEB', 'Python', 'ab_logix.zip'))
else:
    # on PC is just current directory
    sys.path.append('Python_ext.zip')
    sys.path.append('ab_logix.zip')

if __name__ == '__main__':

    import aboxy # this is our main project file, which is within ab_logix.zip
    aboxy.run_aboxy( )

sys.exit(0)
```

Estimating free flash file space

Some Digi products support the standard Python os.statvfs() function to query FLASH filesystem stats. Note that due the realities of file system overhead, allocating a file of size X, then freeing that file may NOT return the free/used statistics to the exact same condition.

Below is an example code which runs on a Digi ConnectPort X4 or Connect SP.

```
import os
import traceback

try:
    x = os.statvfs("WEB")
    print 'raw', x
    total = x[0] * x[2]
    free = x[0] * x[3]
    used = total - free
    print 'total:%0.2fMB free:%0.2fMB used:%0.2fMB' % \
        (total/1000000.0, free/1000000.0, used/1000000.0)

except:
    traceback.print_exc()
   print "perhaps this product does not have YAFFS?"
```

Gateway module checker

Python code that generates module documentation on gateways

Here is some code that automatically generates Python module documentation for a gateway. If you do any Python programming on our gateways this will probably be of interest to you.

Attached are 3 files:

- **gw_mchecker.py** – Python code that runs on a gateway, make sure to start before running pc_mreader.py
- **pc_mreader.py** – this Python code runs on the PC. It runs from a command prompt and takes one argument, the IP address or hostname of the gateway running gw_mchecker.py
- **default.css** – a style sheet that should be copied into the same folder as pc_mreader.py

How to use it:

- Copy the attached three files into a folder you want to use to document
- Via your favorite mechanism, get the gw_mchecker.py onto the gateway of your choice and execute it. Remember, for gateways without a UI, you can always telnet into them and use the 'Python x.x.x.x:gw_mchecker.py' command (where x.x.x.x is your TFTP Servers's IP address) and TFTP the file in.
- Once gw_mchecker.py is running, on your pc execute pc_mreader.py from a cmd prompt by typing 'Python pc_mreader.py x.x.x.x' where x.x.x.x is the IP address of the gateway running gw_mchecker.py.
- Watch as the data is transferred and files are created, both scripts will exit automatically when finished.

How it works:

When gw_mchecker.py executes, it goes through all the modules we say exist on Gateway (including those in Python.zip) and puts them in a list. Then it goes through each of the modules and pulls their _ _doc__ data, including the __doc__ data of those classes under each module. As it iterates over each module it pushes that data up to a connected client (established by pc_mreader.py). The code will wait to send data or further iterate the module list until a client has connected.

What you get:

As the PC client (pc_mreader.py) receives the data, it creates html pages based on the modules and classes found. The folder it sticks the data into is ./(model)/(firmware version)/ where model is the device type and the firmware version is the version of firmware the Gateway is running. In this folder is an index.htm that lays out all the module information.

Download the files here:

Module_checker.zip

HTTP basic authentication

Remote RCI with username and password

Many services can be handled by Digi's Remote Command Interface (RCI) function - for example you can query firmware levels, list file sizes and so on. However, if your remote Digi has a username and password, then you will need to include the "Basic Authentication" line in your HTTP request header.

See Module: rci and digi.com/support/documentation/90000569_G.pdf for more details about RCI.

Basic Authentication

Basic Authentication allows you to run RCI commands from either a Digi gateway based Python program (targeting localhost or 127.0.0.1) or from a remote computer. The way the username + password (in form "username:password") is hashed is NOT security, plus it is a constant so you could include only the preformed code in your Python if you username password is fixed. This base64 encoding is merely a way to encode ANY characters within HTTP, since users might have special characters like "<" or ":" or spaces in their password. Basic authentication to localhost is secure, however its use over Ethernet is not secure unless done within SSL.

```
from base64 import encodestring

def get_basic_auth( user, password):
    # return string like "Authorization: Basic cm9vdDpkYnBz\r\n"
    return "Authorization: Basic " + encodestring( '%s:%s' % (user,password))[:-
1] + "\r\n"
```

That's it. If you include this line your HTTP header, then your RCI command will succeed.

Here is an example RCI request to fetch a list of Python files, assuming the username and password is "root:dbps":

```
POST /UE/rci HTTP/1.1
User-Agent: Python/2.4 (digi)
Host: 192.168.196.14
Accept: */*
Content-Type: text/xml
Content-Length: 128
Authorization: Basic cm9vdDpkYnBz

<rci_request version="1.1">
  <do_command target="file_system">
    <ls dir="/WEB/Python"/>
  </do_command>
</rci_request>
```

Handling socket error and Keepalive

Handling TCP socket error and TCP_KEEPALIVE

The 'Net' abounds with simple 20 line examples of TCP client behavior; however they all assume the TCP server exists and that crashing due to exceptions is helpful education for you. trying to locate realistic examples often pushes you into complex examples using hundreds of lines of code to do real work.

So here is a simple-dummy example TCP client application which runs on either a PC or Digi product:

- It creates the socket
- It sets the socket timeout to 5.0 seconds, this means non-blocking and all requests will fault in 5.0 seconds unless they succeed.
- It tests and enables the TCP Keepalive - **which by default is OFF on most systems** (Windows, Linux and Digi Python) This test is NOT required; it is here merely to show how it is done.
- It tries to open (connect) to a fixed IP address and TCP port 2101, which either succeeds rapidly - or fails in 5.0 seconds.
  - "**except socket.error**" traps the error and causes the code to sleep 5 more seconds, then restart at socket creation.
- if the socket is open, it waits 5 seconds for up to 6 bytes of data, and throws an exception if socket errors or no data is received. This block has two-level error trapping:
  - "**except socket.timeout**" traps the no-data error and loops up to try receiving again. Note that a robust program design would keep track of how frequently (or long) this 'no data' continues. In many designs, a TCP socket sitting idle for more than a few minutes might be best closed.
  - "**except socket.error**" traps any remaining error, and exits the inner "while True:" and restarts creating the socket.

To use this example, set the IP address to any Digi TS/DS with TCP Sockets active. Then by powering up or down the Digi TS/DS you can cause the connect(('x.x.x.x',2101) to succeed or fail. Sending simple ASCII data into the serial port of the Digi TS/DS would enable you to force or skip the "socket.timeout" try-except clause.

TCP Keepalive

So how does your Python code understand if no data means the TCP peer is being quiet - or if the TCP socket has gone away? The normal answer is that eventually a socket error will cause the "socket.error" clause to execute. However, that could easily be hours (or forever) after the socket fails.

By default sockets in your Digi Python application (as well as Windows or Linux) open with TCP keepalives turned off, and thus it is possible your application will languish for a very long time with a dead socket open. This simple line of Python code will turn TCP Keepalives on:

```
x = sock.setsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
```

By default your Digi Python product will have a default TCP idle time of two hours, so even if you turn on TCP Keepalives don't expect your program to recover in minutes. Different TCP Keepalive settings can be entered in the Digi Web interface at **Configuration > Network > Advanced Network Settings > TCP Keep-Alive Settings**, or by telnet with the set net and show net commands. If you truly wish to force TCP Keepalive settings to always contain a smaller setting, you could use Module:digicli to force specific settings. This example sets roughly a 5 minute detection of a failed TCP socket.

```
status,results = digicli.digicli( "set net idle=240 probe_count=5 probe_
interval=10" )
```

Do NOT try to use TCP Keepalive to detect TCP socket failure more quickly than a few minutes.

People who try to set it for 5 seconds (or for milliseconds) invariably cause serious compatibility issues with other products - and invariably fail to be satisfied. If you truly require detecting a TCP socket failure in 1 second or less, which implies your TCP peers normally send data many times per second, then use non-blocking sockets with the "socket.timeout" exception to detect when no data had been received in your required time-frame. And if you accept that a TCP peer quiet for 1 second is bad, then close the socket manually and attempt recovery directly. Do not use TCP Keepalive for such short-period detection.

Example code

(Note that this code uses " **import socket** " and thus all methods and all constants such as SOCK_ STREAM must include the " **socket**. " preface. Many other example applications use " **from socket import** * ", which eliminates this needs for " **socket**. " preface. Either solution works - just be mindful of this detail if you mix-and-match sample code from diverse sources.)

```
import sys
import socket
import traceback
import time

def do_work( forever = True):

    while True:

        # start with a socket at 5-second timeout
        print "Creating the socket"
        sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout( 5.0)

        # check and turn on TCP Keepalive
        x = sock.getsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE)
        if( x == 0):
            print 'Socket Keepalive off, turning on'
            x = sock.setsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
            print 'setsockopt=', x
        else:
            print 'Socket Keepalive already on'

        try:
            sock.connect(('192.168.196.8',2101))

        except socket.error:
            print 'Socket connect failed! Loop up and try socket again'
            traceback.print_exc()
            time.sleep( 5.0)
```

```
            continue

        print 'Socket connect worked!'

        while 1:
            try:
                req = sock.recv(6)

            except socket.timeout:
                print 'Socket timeout, loop and try recv() again'
                time.sleep( 5.0)
                # traceback.print_exc()
                continue

            except:
                traceback.print_exc()
                print 'Other Socket err, exit and try creating socket again'
                # break from loop
                break

            print 'received', req

        try:
            sock.close()
        except:
            pass

        # loop back up & restart

if __name__ == '__main__':

    do_work( True)
```

Module finder

Introduction

Using modulefinder Python Tool to Determine which Files to Load

When developing Python programs to run on the Digi gateway (Digi WAN, Connect Port X2, X4, X8, etc.), it is useful to have a tool that can tell you dependancies. For most programs, determining which files should be moved onto the Digi device should be fairly simple, because most likely you are writing most program modules and content yourself. However, when using third party modules or those provided by the standard distribution, a tree of dependencies may exist, making it difficult to determine which files must be placed on the device.

The standard Python distribution provides a tool called modulefinder.py that is useful in this scenario. This tool examines imports in Python programs to build a list of modules that may be used.

Example use: ftp://ftp1.digi.com/support/images/moduleFinder_example.txt.

Sample Usage

Modulefinder_howtouse.zip

Python garbage collection

Introduction to Python memory management

Python's memory allocation and deallocation method is automatic. The user does not have to preallocate or deallocate memory by hand as one has to when using dynamic memory allocation in languages such as C or C++. Python uses two strategies for memory allocation **reference counting** and **garbage collection**.

Prior to Python version 2.0, the Python interpreter only used reference counting for memory management. Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero the object is deallocated.

Reference counting is extremely efficient but it does have some caveats. One such caveat is that it cannot handle reference cycles. A reference cycle is when there is no way to reach an object but its reference count is still greater than zero. The easiest way to create a reference cycle is to create an object which refers to itself as in the example below:

```
def make_cycle():
    1 = [ ]
    1.append(l)

make_cycle()
```

Because `make_cycle()` creates an object `1` which refers to itself, the object `1` will not automatically be freed when the function returns. This will cause the memory that `1` is using to be held onto until the Python garbage collector is invoked.

Automatic garbage collection of cycles

Because reference cycles are take computational work to discover, garbage collection must be a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations and object deallocations. When the number of allocations minus the number of deallocations are greater than the threshold number, the garbage collector is run. One can inspect the threshold for new objects (objects in Python known as *generation 0 objects*) by loading the gc module and asking for garbage collection thresholds:

```
import gc
print "Garbage collection thresholds: %r" % gc.get_threshold()

Garbage collection thresholds: (700, 10, 10)
```

Here we can see that the default threshold on the above system is 700. This means when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.

Automatic garbage collection will not run if your Python device is running out of memory; instead your application will throw exceptions, which must be handled or your application crashes. This is aggravated by the fact that the automatic garbage collection places high weight upon the NUMBER of free objects, not on how large they are. Thus any portion of your code which frees up large blocks of memory is a good candidate for running manual garbage collection.

Manual garbage collection

For some programs, especially long running server applications or embedded applications running on a Digi Device automatic garbage collection may not be sufficient. Although an application should be written to be as free of reference cycles as possible, it is a good idea to have a strategy for how to deal with them. Invoking the garbage collector manually during opportune times of program execution can be a good idea on how to handle memory being consumed by reference cycles.

The garbage collection can be invoked manually in the following way:

```
import gc
gc.collect()
```

`gc.collect()` returns the number of objects it has collected and deallocated. You can print this information in the following way:

```
import gc
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
```

If we create a few cycles, we can see manual collection work:

```
import sys, gc

def make_cycle():
l = { }
l[0] = l

def main():
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
print "Creating cycles..."
for i in range(10):
make_cycle()
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)

if __name__ == "__main__":
ret = main()
sys.exit(ret)
```

In general there are two recommended strategies for performing manual garbage collection: time-based and event-based garbage collection. Time-based garbage collection is simple: the garbage collector is called on a fixed time interval. Event-based garbage collection calls the garbage collector on an event. For example, when a user disconnects from the application or when the application is known to enter an idle state.

Recommendations

Which garbage collection technique is correct for an application? It depends. The garbage collector should be invoked as often as necessary to collect cyclic references without affecting vital application performance. Garbage collection should be a part of your Python application design process.

- Do not run garbage collection too freely, as it can take considerable time to evaluate every memory object within a large system. For example, one team having memory issues tried calling gc.collect() between every step of a complex start-up process, increasing the boot time

by 20 times (2000%). Running it more than a few times per day - without specific design reasons - is likely a waste of device resources.

- Run manual garbage collection after your application has completed start up and moves into steady-state operation. This frees potentially huge blocks of memory used to open and parse file, to build and modify object lists, and even code modules never to be used again. For example, one application reading XML configuration files was consuming about 1.5MB of temporary memory during the process. Without manual garbage collection, there is no way to predict when that 1.5MB of memory will be returned to the Python memory pools for reuse.
- Run manual garbage collection after infrequently run sections of code which use and then free large blocks of memory. For example, consider running garbage collection after a once-per-day task which evaluates thousands of data points, creates an XML 'report', and then sends that report to a central office via FTP or SMTP/email. One application doing such daily reports was creating over 800K worth of temporary sorted lists of historical data. Piggy-backing gc.collect() on such daily chores has the nice side-effect of running it once per day for 'free'.
- Consider manually running garbage collection either before or after timing-critical sections of code to prevent garbage collection from disturbing the timing. As example, an irrigation application might sit idle for 10 minutes, then evaluate the status of all field devices and make adjustments. Since delays during system adjustment might affect field device battery life, it makes sense to manually run garbage collection as the gateway is entering the idle period AFTER the adjustment process - or run it every sixth or tenth idle period. This insures that garbage collection won't be triggered automatically during the next timing-sensitive period.

Further references and reading

- Official Python documentation on the gc module: http://docs.Python.org/library/gc.html
- An article on cyclic references and Python garbage collection: http://arctrix.com/nas/Python/gc/
- Wikipedia's entry on Garbage Collection: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

SMTP Email

Python program for SMTP Email

SMTP Email Notification (Python program) This example application sends an email notification to the specified email-id's.

Test files

This sample program contains one file. File name "SMTP_email_notofication.py".

SMTP Email Notification Test Sample Application

The SMTP_email_notofication.py Python Test sample application can be found here: SMTP_email_notofication.zip.

Basic usage

Provide inputs where neccesary.

Sample of SMTP_email_notofication.py file:

```
#!/usr/bin/Python

import os
import sys
import smtplib
import traceback

##########################################################################
##########################################################################
'''###PROVIDE INPUTS HERE###'''
EMAIL_HOST = "mail.<your_domian>.com"
sender = 'M2M'  #ANY NAME
# Receivers email id
receivers = ['first_name.last_name@digi.com', 'second_name.last_name@digi.com']
message = """ From: Digi Sample <M2M@digi.com>
To: <first_name.lastname@digi.com>,<second_name.last_name@digi.com>
CC: <first_name.lastname@digi.com>
Subject: SMTP e-mail test

This is a test e-mail
"""
##########################################################################
##########################################################################
try:
      smtpObj = smtplib.SMTP('EMAIL_HOST', 25)
      smtpObj.sendmail(sender, receivers, message)
      print "Successfully sent email"
except Exception, e:
      print "Exception %s" %e
      traceback.print_exc()

print "End of the program"
```

Simple save and load to Flash

Simple save and reload of data to Flash

Warning: if you want to save more than once per MINUTE, use an external USB flash drive (or your Digi product may become inoperable rapidly)

See this Wiki page for general information and limitations in FLASH usage: How to use a USB flash drive in Python.

See this Wiki page for information on FLASH filesystem statistics: Estimating free flash file space.

Do you have a few variables you wish you could save and restore after the Digi product with Python reboots? This Wiki page covers a very simple method which saves your data as a text image of a dictionary. This allows not only a Python application to reload the data, but the user can examine the data via the web interface at will.

Actual string_file.py Code

The actual save/reload file code is trivial. Neither routine throws an exception, and you can use the print_fail parameter to enable or disable printing an error message. The routine save_string_to_file() should only fail if the FLASH filesystem is full, or if the destination filename is invalid. There is no direct way to detect a full filesystem on a Digi product. The routine load_string_from_file() should only fail if the destination filename does not exist or is invalid.

```
# constants
ROOT_PATH = "WEB/Python/"
# ROOT_PATH = "A/"

def save_string_to_file( filename, data, print_fail=True):
    """Save string to file, return byte count written"""
    filename = ROOT_PATH + filename

    try:
        fn = file( filename,'wb')
        data = str(data)
        fn.write( data)
        fn.close()
        return len(data)

    except:
        if print_fail:
            traceback.print_exc()
            print 'save_string_to_file(%s) failed' % filename

    return 0

def load_string_from_file( filename, print_fail=False):
    """Load string from file, return data or None if fails"""
    filename = ROOT_PATH + filename

    try:
        fn = file( filename,'rb')
        data = fn.read()
        fn.close()
        return data
```

```
    except:
        if print_fail:
            traceback.print_exc()
            print 'load_string_from_file(%s) failed' % filename

return None
```

Example text file saved

Here is the example text file created. While it looks like Python, it is pure text so can be viewed online from the Digi product Python tab. It can also be saved to you local PC by right-clicking the file.

```
{'lights': 9600, 'pc_stuff': 25741, 'light_active': 961, '_total_time': 142261,
'chargers': 0}
```

Example object routines to save and reload values

Here is a simple example code from an time totalizer object, which tracks the total time certain conditions are true/ON on a Digi ConnectPort X4 gateway. The reload routine uses the has_key() routine to gracefully handle poorly formed (or 'old') files.

The application saves the data every 5 minutes, and reloads the values from FLASH upon startup. This means up to five minutes of data can be lost any time the CPX4 is rebooted, however such reboots should be very rare when the program is actually installed in the field. Plus even if power is lost during blackouts, then considerably more than 5 minutes of "time" will often be lost. For example, if power is off for 4 hours, does it really matter that 245 minutes of data was lost instead of 240 minutes?

```
    def save_totals( self):
        # save the totals to NVRAM (flash)
        xdct = { '_total_time' : self.total_time }

        for lnk in self.linklist:
            xdct.update( { lnk.get_name() : lnk.get_total_time() } )

        return save_string_to_file( self.SAVE_FILE, xdct)

    def reload_totals( self):
        # attempt to reload old time totals

        data = load_string_from_file( self.SAVE_FILE)
        if data:
            # then we have something
            print 'read this', data
            try:
                data = eval(data)
                if data.has_key( '_total_time'):
                    self.total_time = int(data['_total_time'])

                for lnk in self.linklist:
                    if data.has_key( lnk.get_name()):
                        lnk.do_reset( data[lnk.get_name()] )
            except:
                traceback.print_exc()
                print 'reload of totals failed'
                return False


    return True
```

Sleep to wake on time

Using time.sleep() to wake at fixed time-of-day

iDigi/Dia scripts often cyclically sleep and wake, for example checking a sensor every hour. Yet if you just call time.sleep() with 3600 seconds, your sensor readings will slowly drift due to variability in processing and communication response times.

Most customers will eventually complain if their hourly samples are timestamped as 1:01, 2:01, 3:02, 4:03 and so on. Worse, if the Digi product restarts, then a random drift will occur so that the timestamps tomorrow might be 1:41, 2:41, 3:42 and so on. Ideally, the hourly samples can be forced at a specific time, such as on the hour or 15 minutes after every hour.

Solutions

Method #1 - Create manual time-tuple

One method is to create the desired wake-up time tuple, convert this to seconds since epoch and then subtract the current time. This nicely obtains the number of seconds until the desired wake-up time.

However, gracefully handling the roll-over of days, months and years can be complex. For example, if it is 11PM on the 28th day of the month and the desired wake-up time is 3:05:00 AM tomorrow, then the complexities of how many days are in this month (plus leap year) come into play.

Method #2 - Calculate seconds manually

The second method (show below) is to manually calculate the seconds to a future time expressed as (h,m,s), so for example 11:05:00 AM would be (11,5,0). This routine ONLY works for a wake-up time within the next 24-hours. To sleep longer than one day, add 86400 seconds for each day to remain asleep, so for example to start a sleep on Friday and wake-up on Monday, add an extra 172800 seconds to the response returned from this routine.

```
def get_secs_until_wakeup(wake_tup, now_tup=None):
    """\
    Given a desired wake-up time within next 24-hours, calculate the required
    seconds to sleep to hit that time.

    wake_tup is (h,m,s), so for example (11,5,0) to wake at 11:05:00AM
    now_tup is optional, or is a time_tuple to treat as now
    if now_tup is omitted, then it will be treated as time.localtime()
    to use UTC, call as get_secs_until_wakeup( wake_tup, time.gmtime() )
    """

    if now_tup == None:
        now_tup = time.localtime( )

    # start with seconds, always move forward
    secs = (60 + wake_tup[2]) - now_tup[5]

    # now minutes, but seconds 'ate' 1 minute already
    secs += (((60 + wake_tup[1]) - now_tup[4] - 1) * 60)

    # now hours, but minutes+seconds 'ate' 1 hour already
    secs += ((wake_tup[0] - now_tup[3] - 1) * 3600)
    if secs < 0:
```

```
        secs += 86400

    # print 'adjust is %d sec (%02d:%02d:%02d)' % (secs,(secs/3600),(secs/60),
(secs%60))
    return secs
```

The time tuple

To help understand the code, here is the structure of the standard time tuple. More information can be found at: http://docs.Python.org/library/time.html.

Notice that it includes 'day of the week', so you can easily change behavior on week-ends.

0	Year	tm_year	full 4-digits, so for example 2010
1	Month	tm_mon	range [1,12]
2	Day of Month	tm_mday	range [1,31]
3	Hour	tm_hour	range [0,23]
4	Minutes	tm_min	range [0,59]
5	Seconds	tm_sec	range [0,61]; see Python documentation
6	Day of Week	tm_wday	range [0,6], Monday is 0
7	Day of Year	tm_yday	range [1,366]
8	Daylight Savings	tm_isdst	0, 1 or -1; see Python documentation

Clean wake on minute periods

If for example you want to wake up every 5 minutes, starting at 00:00:00, then waking again at 00:05:00, then 00:10:00 and so on, below is a simple routine that calculates the seconds delay based on time periods. These periods must be a factor of 60 (including 1 and 60).

```
def secs_until_next_minute_period(period, now_tup):
    """ This routine allows a task to sleep/wake on clean time periods,
    for example, every 5 minutes starting at 00:00. It returns the seconds
    from 'now' until the next period starts.

    period must be a factor of 60, so minute in (1, 2, 3, 4, 5, 6, 10, 12, 15,
20, 30, 60)

    now_tup must be a time tuple such as returned by time.gmtime() or
time.localtime()
    """

    if period not in (1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60):
        # then is not a factor of 60
        raise ValueError("%d is not a factor of 60" % period)

    # start with seconds until next minute with tm_sec = 0
    sec_til = (60 - now_tup[5])
```

```
    if(period > 1):
        # then calc the adder for the minutes adjust
        sec_til += ((period - (now_tup[4] % period) - 1) * 60)


    # else, until next minute if fine, we are done
    return sec_til
```

Using ZIP, GZIP or compression

Compression under Python

If you want to compress (or uncompress) ZIP files, use the GZIP module - see Python.org/library/gzip.html.

Note Using compression requires considerable free memory. If your Python application is already running out of memory (for example, on an older ConnectPort X2 with 8BM RAM only), then trying to add compression will break it. You should also consider using gc.collect() frequently to free up unused memory more rapidly.

Example of how to read a compressed file

Online examples work fine - just remember to preface file name with "WEB/Python/"

```
import gzip
f = gzip.open('WEB/Python/file.zip', 'rb')
file_content = f.read()
f.close()
```

Example of how to create a compressed GZIP file

```
import gzip
content = "Lots of content here"
f = gzip.open('WEB/Python/file.zip', 'wb')
f.write(content)
f.close()
```

Example of how to GZIP compress an existing file

```
import gzip
f_in = open('WEB/Python/file.txt', 'rb')
f_out = gzip.open('WEB/Python/file.zip', 'wb')
f_out.writelines(f_in)
f_out.close()
f_in.close()
```

Example of how to GZIP compress without using FLASH files

Saving temporary FLASH files is both slow and risks file system corruption. If your goal to create a RAM-resident data object for upload to a remote FTP or HTTP server, then you can create a ZIP image directly in StringIO objects, which can be thought of as RAM-based files.

```
import StringIO
import gzip
import gc

# code here creates String object named data
data = "Lots of Data"

zipr = StringIO.StringIO()
# we need to use filename to put correct 'name' into ZIP file
tmpf = gzip.GzipFile(filename='data.orig', mode='wb', fileobj=zipr,)
```

```
tmpf.write(data)
tmpf.close()
# we no longer need tmpf or data - zipr is zipped version of data
del buf
del tmpf


# do what you want with 'zipr' here - for example push elsewhere by
# ftplib or httplib
send_file_to_host("data.zip", zipr.getvalue())

# manually push garbage collection to speed up memory recovery
del zipr
gc.collect()
```

Running code packaged within a ZIP file

You do NOT need to use GZIP to access files packed within ZIP files. You merely add the ZIP to the PythonPATH as if it were a common sub-directory to search for files.

```
import sys

# do this at the start of your main routine
sys.path.insert(0, 'WEB/Python/my_project.zip')

# at this point, imports from 'my_project.zip' should work.
import my_file
```

See Also: Loading Python programs onto a Digi device.

Which Python Version

Different Digi products support different versions of Python. If you upload the uncompiled.PY files, then you may use any version of Python as long as you avoid features which are not supported on the Digi product (for example, you cannot use new 2.6.x features on a Digi ConnectPort X4 which only supports 2.4.3).

However, if you plan to use the iDigi/Dia framework and/or upload the compiled.PYC files, then you must use an exact match.

Python 2.4.3

Download from http://Python.org/download/releases/2.4.3/

- Digi Connect WAN Family
- Digi ConnectPort WAN Family
- Digi ConnectPort X2
- Digi ConnectPort X4
- Digi ConnectPort X4H
- Digi ConnectPort X5
- Digi ConnectPort X8

Python 2.6.1

Download from http://Python.org/download/releases/2.6.1/

Digi ConnectPort X3

Python 2.6.2

Download from http://Python.org/download/releases/2.6.2/

Digi ConnectPort LTS 8/16/32

XBEE API packets

Program to design API packets

(**For ZigBee modules**) Program "Tx_Req 0x10" API packets generator for Zigbee modules(Xbee RF Modules (S1 {only DigiMesh}, S2, S3 and S8).

Test files

This sample program contains one file, "Tx_Req 0x10 packet generator.py".

XBee API Packet generator Test Sample Application

The XBee API Packet sample application can be found here: Tx_Req_0x10_packet_generator.zip.

Basic usage

Provide input where necessary.

Sample of Tx_Req 0x10 packet generator.py file:

```
########################################################################
#                   IMPORTS                                            #
########################################################################

import serial
import sys
import binascii


## Known Issue:
## 1. ONLY 1 byte length packet not transmitting



######  User Data #######
com_port = 'COM1'
dest_64bit = '0013A20040762859'
dest_16bit = 'FFFE'
rf_data = """!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`"""
#########################

def b_u(st):
    ## function to convert big-endian binary string into bytes[0-1] as int
    if len(st) == 1:
        return ord(st)
    else:
        length = len(st)
        sft = 8*(length-1)
                return (b_u(st[0])<<sft) + b_u(st[1:])




## create serial socket connection to talk with module
ser = serial.Serial(com_port, 9600, timeout=0.1, rtscts=True)

## convert RF data to hex
rf_hex = binascii.hexlify(rf_data)
```

```
##print "rf_hex=", rf_hex


                        ## calculate packet length
hex_len = hex(14 + (len(rf_hex)/2))
hex_len = hex_len.replace('x','0')
##print "hex_len=", hex_len

## calculate checksum
## 0x17 is the sum of all parameters minus 64bit & 16bit dest addr & payload
    checksum = 17


for i in range(0,len(dest_64bit),2):
    checksum = checksum + int(dest_64bit[i:i+2],16)

for i in range(0,len(dest_16bit),2):
    checksum = checksum + int(dest_16bit[i:i+2],16)

for i in range(0,len(rf_hex),2):
    checksum = checksum + int(rf_hex[i:i+2],16)

## checksum = 0xFF - 8-bit sum of bytes between the length and checksum
checksum = checksum%256
checksum = 255 - checksum
checksum = hex(checksum)
checksum = checksum[-2:]

## designing packet
tx_req = ("7E" + hex_len + "10" + "01" + dest_64bit
          + dest_16bit + "00" + "00" + rf_hex + checksum)
print "Tx packet = ", tx_req

## convert packet from hex to binary
data = binascii.unhexlify(tx_req)

## send data on serial line to module
ser.write(data)

## listin COM port for response
resp = ser.readline()

## convert response from binary to int
resp = b_u(resp)

## convert response from int to hex
resp = '%x' % resp
hex_data = resp.upper()
print "Response (in hex) = ", hex_data

## close connection
ser.close()
```

XBee sensor

Python program for XBee sensor

XBee sensor (Python program) This program gives a user information regaridng battery status of sensor.

Test files

This sample program contains one file. File name "check_battery_life_sensor.py".

XBee battery sensor test sample application

The check_battery_life_sensor.py Python Test sample application can be found here: Check_battery_life_sensor.zip.

Basic usage

Make sure that the sensors are in the network coordinator. Provide the inputs where neccesary.

Sample of check_battery_life_sensor.py file:

```
import sys
import os
import struct
import zigbee
import xbee
from _zigbee import *
import time
import traceback
from struct import *

''' Provide the extended address(OUI) of the xbee sensor'''
###################################################
DESTINATION="00:13:a2:00:40:86:cd:11!"
###################################################

def calculate_battery(bat_vol):
    mv = float(bat_vol)
    mv = ((mv * 1200)/1024)
    v = mv/1000
    v = round(v, 2)
    return v

try:
    print "reading parameters, waiting five seconds..."
    # %V- Supply Voltage. Reads the voltage on the Vcc pin. Scale by 1200/1024 to
    # convert to mV units.
    param_value = zigbee.ddo_get_param(DESTINATION, "%V")
    param_value = struct.unpack("h", param_value)
    str_param_value = str(param_value)        # converting into a string
    s_index = str_param_value.find("(")
    e_index = str_param_value.find(",)")
    bat_voltage = str_param_value[s_index+1:e_index]
    battery_mv = calculate_battery(bat_voltage)
    if (battery_mv) >= 2.8 and (battery_mv) <= 3.4:
        print "battery is in the range of 2.8 and 3.4 and the battery value" + \
        + "is %s" %str(battery_mv)
    else:
        print "low battery"

except Exception, e:
    print "Exception %s" %e
    traceback.print_exc()

print "end of the program"
```

Good Python Suggestions

These pages are not Digi-specific, but include tips and suggestions which will help you complete your work.

Commandline file upload

Easier uploading of files from Python development machines

No authentication

Here is a simple shell script to make uploading files from Linux to a ConnectPort x4 gateway easier. This allows automated uploading of files directly to a gateway device without forcing you to use your web browser by hand. This version of the script expects no access controls.

Hopefully you will find it useful.

```
            #/bin/bash

PROG=$(basename $0)
#CHANGE THIS TO MATCH YOUR SETUP:
DEFAULT=192.168.1.151

if [ -z $2 ]
then
        if [ -z $1 ]
        then
                echo "Usage: $PROG <host> <file>"
                exit
        else
                #assume only the filename was provided.
                HOST=$DEFAULT
                FILE=$1
        fi
else
#use both specified options
HOST=$1
FILE=$2
fi

#sends the file to the device

curl $HOST/Forms/Python_files_1 \
        -F "currentdirectory=WEB/Python"  \
        -F file=@$FILE
```

With Authentication

If you have setup your connectport with with a username and password use the script below:
#/bin/bash

```

#Login into a connectport and post a file to Python folder.

PROG=$(basename $0)
#CHANGE THIS TO MATCH YOUR SETUP:
DEFAULT=http://192.168.1.151

if [ -z $4 ]
then
```

```
        if [ -z $3 ]
        then
                echo "Usage: $PROG <host> <file> <username> <password>"
                exit
        else
                #assume only the filename was provided.
                HOST=$DEFAULT
                FILE=$1
                USERNAME=$2
                PASSWORD=$3
        fi
else
#use both specified options
HOST=$1
FILE=$2
USERNAME=$3
PASSWORD=$4
fi
#Login in on behalf of the user
echo login started
curl -v -c "cookies.txt" \
                -b "C1=Cc423F98b5roy9T" \
                -d
"cache=&protocol=file&username=$USERNAME&password=$PASSWORD&btLogin=Login" \
                $HOST/Forms/login_1
echo login finished
#sends the file to the device
echo upload startedcurl -v -b cookies.txt -c cookies.txt $HOST/Forms/Python_
files_1 -F "currentdirectory=WEB/Python" -F file=@$FILE
echo upload finished
```

Notes

The "currentdirectory=" path cannot be changed to a USB device (e.g. A/). If you do the upload will still succeed to the "WEB/Python" directory.

What is cURL?

Curl is a common Linux command line tool for doing URL manipulations and transfers. Invoke 'curl --help' or 'curl --manual' to get basic information about it.

Click this link for a basic cURL tutorial

Error messages

Common error messages

Many error messages are misleading, as the messages are from secondary causes. Document such messages and the results here:

Bad local file header

```
#> py dia.py
Determining platform type...Digi Python environment found.
Traceback (most recent call last):
  File "<string>", line 161, in ?
  File "<string>", line 82, in main
zipimport.ZipImportError: bad local file header in WEB/Python/dia.zip
```

The most likely cause is that the zip file has already been opened, and you are trying to open a 'new version' a second time. You need to reboot after uploading a new copy of the ZIP file.

For example, if **dia.py is already running**, then you'll always get this error. Check the 'connections' or 'who' list. Perhaps you left it running before, or have it set to auto-run.

A secondary cause might be a bad ZIP image - HTTP/web upload is not totally reliable. At times large files timeout/abort during upload and you may end up with only 320K of a 350K file. So confirm the file size of the ZIP is as expected, and/or upload dia.zip again.

Error workaround

As a workaround for this error without needing to reboot, one can clear zipimport's cached file headers. As part of one's startup script (before zip files are loaded onto sys.path call the following):

```
def clear_zipimport_cache():
    """Clear out cached entries from _zip_directory_cache"""
    import sys, zipimport
    syspath_backup = list(sys.path)
    zipimport._zip_diretory_cache.clear()
    # load back items onto sys.path
    sys.path = syspath_backup
```

Exception while uploading

Literally, the message each time DIA tried to upload data was the following:

```
iDigi_DB(idigi_db1): exception while uploading: (-6, 'The name does not resolve
for the supplied parameters. Neither nodename  nor servname were supplied. At
least one of these must be supplied.')
```

The solution (or problem) was that the DNS IP addresses within the CPX4 were NOT set properly, thus the **remote management server name (sd1-na.idigi.com)** could not be resolved, thus Device Cloud was NOT connected. Other symptoms:

- Device Cloud (or connectware manager) listed the device as disconnected.
- The web UI connections page did not list the "connectware tcp" entry

Socket.error

Socket is already open

```
Traceback <most recent call last>:
File "<string>", line 23, in ?
File "<string>", line 1, in bind
socket.error: <22, 'invalid argument'>
```

Although this error could mean a badly formed IP or port value, it also can mean that some other Python script is running and holding that specific IP+port combination open. Navigate to your ConnectPort's web UI, click on Applications->Python->Auto-start Settings, uncheck whichever script may be set to autorun, click Apply and reboot your gateway.

Syntax error (in YML file)

A very common cause is the use of tab characters. YML requires spaces only, so either use a text editor which replaces tabs with spaces, or manually make sure only spaces are use.

ValueError: failed to parse request (in RCI call)

One cause for this is 'fancy quotes' - if you cut and paste the request from a PDF or other web document, at times the quotations used are the fancy 'angled' quotes. Thus the string <rci_request version="1.1"> is really seen as <rci_request version=ö1.1ö>, which causes the RCI call to fail.

List index out of range error

A node which was part of the "index" is no longer available

```
#> Python EmbeddedKitService.py
Starting up...
Ready for incoming requests!
Discovering nodes...
Exception in thread WPANSerialEndpoint:
Traceback <most recent call last>:
  File "WEB/Python/Python.zip/threading.py", line 442, in __bootstrap
  File "WEB/Python/EmbeddedKitManager.py", line 199, in run
    nodes = self.get_bindings_hash_list<>
  File "WEB/Python/EmbeddedKitManager.py", line 496, in get_bindings_hash_list
    hash_list = [self._bindings.bindings[k].to_hash<> \
  File "WEB/Python/EmbeddedKitManager.py", line 38, in to_hash
    return {
IndexError: list index out of range
-
```

This error typically means that a node which was once associated with the parent (CP-X?) where the Python script is running is no longer available. Common causes: node has been configured for sleep parameters and is now asleep, node is powered off, node is in an unknown state. Power off parent and all associated nodes to clear out routing and neighbor tables, then power them back on and re-try the script when the nodes are available.

Handling socket error and Keepalive

Handling TCP socket error and TCP_KEEPALIVE

The 'Net' abounds with simple 20 line examples of TCP client behavior; however they all assume the TCP server exists and that crashing due to exceptions is helpful education for you. trying to locate realistic examples often pushes you into complex examples using hundreds of lines of code to do real work.

So here is a simple-dummy example TCP client application which runs on either a PC or Digi product:

- It creates the socket
- It sets the socket timeout to 5.0 seconds, this means non-blocking and all requests will fault in 5.0 seconds unless they succeed.
- It tests and enables the TCP Keepalive - **which by default is OFF on most systems** (Windows, Linux and Digi Python) This test is NOT required; it is here merely to show how it is done.
- It tries to open (connect) to a fixed IP address and TCP port 2101, which either succeeds rapidly - or fails in 5.0 seconds.
  - "**except socket.error**" traps the error and causes the code to sleep 5 more seconds, then restart at socket creation.
- if the socket is open, it waits 5 seconds for up to 6 bytes of data, and throws an exception if socket errors or no data is received. This block has two-level error trapping:
  - "**except socket.timeout**" traps the no-data error and loops up to try receiving again. Note that a robust program design would keep track of how frequently (or long) this 'no data' continues. In many designs, a TCP socket sitting idle for more than a few minutes might be best closed.
  - "**except socket.error**" traps any remaining error, and exits the inner "while True:" and restarts creating the socket.

To use this example, set the IP address to any Digi TS/DS with TCP Sockets active. Then by powering up or down the Digi TS/DS you can cause the connect(('x.x.x.x',2101) to succeed or fail. Sending simple ASCII data into the serial port of the Digi TS/DS would enable you to force or skip the "socket.timeout" try-except clause.

TCP Keepalive

So how does your Python code understand if no data means the TCP peer is being quiet - or if the TCP socket has gone away? The normal answer is that eventually a socket error will cause the "socket.error" clause to execute. However, that could easily be hours (or forever) after the socket fails.

By default sockets in your Digi Python application (as well as Windows or Linux) open with TCP keepalives turned off, and thus it is possible your application will languish for a very long time with a dead socket open. This simple line of Python code will turn TCP Keepalives on:

```
x = sock.setsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
```

By default your Digi Python product will have a default TCP idle time of two hours, so even if you turn on TCP Keepalives don't expect your program to recover in minutes. Different TCP Keepalive settings can be entered in the Digi Web interface at **Configuration > Network > Advanced Network Settings > TCP Keep-Alive Settings**, or by telnet with the set net and show net commands. If you truly wish to force TCP Keepalive settings to always contain a smaller setting, you could use Module:digicli to force specific settings. This example sets roughly a 5 minute detection of a failed TCP socket.

```
status,results = digicli.digicli( "set net idle=240 probe_count=5 probe_
interval=10" )
```

Do NOT try to use TCP Keepalive to detect TCP socket failure more quickly than a few minutes.

People who try to set it for 5 seconds (or for milliseconds) invariably cause serious compatibility issues with other products - and invariably fail to be satisfied. If you truly require detecting a TCP socket failure in 1 second or less, which implies your TCP peers normally send data many times per second, then use non-blocking sockets with the "socket.timeout" exception to detect when no data had been received in your required time-frame. And if you accept that a TCP peer quiet for 1 second is bad, then close the socket manually and attempt recovery directly. Do not use TCP Keepalive for such short-period detection.

Example code

(Note that this code uses " **import socket** " and thus all methods and all constants such as SOCK_ STREAM must include the " **socket**. " preface. Many other example applications use " **from socket import** * ", which eliminates this needs for " **socket**. " preface. Either solution works - just be mindful of this detail if you mix-and-match sample code from diverse sources.)

```
import sys
import socket
import traceback
import time

def do_work( forever = True):

    while True:

        # start with a socket at 5-second timeout
        print "Creating the socket"
        sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout( 5.0)

        # check and turn on TCP Keepalive
        x = sock.getsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE)
        if( x == 0):
            print 'Socket Keepalive off, turning on'
            x = sock.setsockopt( socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
            print 'setsockopt=', x
        else:
            print 'Socket Keepalive already on'

        try:
            sock.connect(('192.168.196.8',2101))

        except socket.error:
            print 'Socket connect failed! Loop up and try socket again'
            traceback.print_exc()
            time.sleep( 5.0)
```

```
            continue

        print 'Socket connect worked!'

        while 1:
            try:
                req = sock.recv(6)

            except socket.timeout:
                print 'Socket timeout, loop and try recv() again'
                time.sleep( 5.0)
                # traceback.print_exc()
                continue

            except:
                traceback.print_exc()
                print 'Other Socket err, exit and try creating socket again'
                # break from loop
                break

            print 'received', req

        try:
            sock.close()
        except:
            pass

        # loop back up & restart

if __name__ == '__main__':

    do_work( True)
```

Python garbage collection

Introduction to Python memory management

Python's memory allocation and deallocation method is automatic. The user does not have to preallocate or deallocate memory by hand as one has to when using dynamic memory allocation in languages such as C or C++. Python uses two strategies for memory allocation **reference counting** and **garbage collection**.

Prior to Python version 2.0, the Python interpreter only used reference counting for memory management. Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero the object is deallocated.

Reference counting is extremely efficient but it does have some caveats. One such caveat is that it cannot handle reference cycles. A reference cycle is when there is no way to reach an object but its reference count is still greater than zero. The easiest way to create a reference cycle is to create an object which refers to itself as in the example below:

```
def make_cycle():
    l = [ ]
    l.append(l)

make_cycle()
```

Because `make_cycle()` creates an object `l` which refers to itself, the object `l` will not automatically be freed when the function returns. This will cause the memory that `l` is using to be held onto until the Python garbage collector is invoked.

Automatic garbage collection of cycles

Because reference cycles are take computational work to discover, garbage collection must be a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations and object deallocations. When the number of allocations minus the number of deallocations are greater than the threshold number, the garbage collector is run. One can inspect the threshold for new objects (objects in Python known as *generation 0 objects*) by loading the gc module and asking for garbage collection thresholds:

```
import gc
print "Garbage collection thresholds: %r" % gc.get_threshold()

Garbage collection thresholds: (700, 10, 10)
```

Here we can see that the default threshold on the above system is 700. This means when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.

Automatic garbage collection will not run if your Python device is running out of memory; instead your application will throw exceptions, which must be handled or your application crashes. This is aggravated by the fact that the automatic garbage collection places high weight upon the NUMBER of free objects, not on how large they are. Thus any portion of your code which frees up large blocks of memory is a good candidate for running manual garbage collection.

Manual garbage collection

For some programs, especially long running server applications or embedded applications running on a Digi Device automatic garbage collection may not be sufficient. Although an application should be written to be as free of reference cycles as possible, it is a good idea to have a strategy for how to deal with them. Invoking the garbage collector manually during opportune times of program execution can be a good idea on how to handle memory being consumed by reference cycles.

The garbage collection can be invoked manually in the following way:

```
import gc
gc.collect()
```

`gc.collect()` returns the number of objects it has collected and deallocated. You can print this information in the following way:

```
import gc
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
```

If we create a few cycles, we can see manual collection work:

```
import sys, gc

def make_cycle():
l = { }
l[0] = l

def main():
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
print "Creating cycles..."
for i in range(10):
make_cycle()
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)

if __name__ == "__main__":
ret = main()
sys.exit(ret)
```

In general there are two recommended strategies for performing manual garbage collection: time-based and event-based garbage collection. Time-based garbage collection is simple: the garbage collector is called on a fixed time interval. Event-based garbage collection calls the garbage collector on an event. For example, when a user disconnects from the application or when the application is known to enter an idle state.

Recommendations

Which garbage collection technique is correct for an application? It depends. The garbage collector should be invoked as often as necessary to collect cyclic references without affecting vital application performance. Garbage collection should be a part of your Python application design process.

- Do not run garbage collection too freely, as it can take considerable time to evaluate every memory object within a large system. For example, one team having memory issues tried calling gc.collect() between every step of a complex start-up process, increasing the boot time

by 20 times (2000%). Running it more than a few times per day - without specific design reasons - is likely a waste of device resources.

- Run manual garbage collection after your application has completed start up and moves into steady-state operation. This frees potentially huge blocks of memory used to open and parse file, to build and modify object lists, and even code modules never to be used again. For example, one application reading XML configuration files was consuming about 1.5MB of temporary memory during the process. Without manual garbage collection, there is no way to predict when that 1.5MB of memory will be returned to the Python memory pools for reuse.
- Run manual garbage collection after infrequently run sections of code which use and then free large blocks of memory. For example, consider running garbage collection after a once-per-day task which evaluates thousands of data points, creates an XML 'report', and then sends that report to a central office via FTP or SMTP/email. One application doing such daily reports was creating over 800K worth of temporary sorted lists of historical data. Piggy-backing gc.collect() on such daily chores has the nice side-effect of running it once per day for 'free'.
- Consider manually running garbage collection either before or after timing-critical sections of code to prevent garbage collection from disturbing the timing. As example, an irrigation application might sit idle for 10 minutes, then evaluate the status of all field devices and make adjustments. Since delays during system adjustment might affect field device battery life, it makes sense to manually run garbage collection as the gateway is entering the idle period AFTER the adjustment process - or run it every sixth or tenth idle period. This insures that garbage collection won't be triggered automatically during the next timing-sensitive period.

Further references and reading

- Official Python documentation on the gc module: http://docs.Python.org/library/gc.html
- An article on cyclic references and Python garbage collection: http://arctrix.com/nas/Python/gc/
- Wikipedia's entry on Garbage Collection: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

Python self-testing code

How to create self-testing code

A common best-practice under Python is to include a self-test at the end every module - especially if the module is largely standalone. Thus the code "import bool_any" only brings in the routines, while running the module with "Python bool_any.py" runs the self tests.

This self-test code does make your files larger in both .PY and .PYC format. For small projects the added 30-50% bulk won't matter, but for larger projects you should consider using a simple Python script to parse through all your files, creating temporary copies which are truncated at an embedded text string

Simple example

Here is a simple example - the def module is of course fake, but the test is a very powerful example of solving a problem as Python likes to solve it.

```
            # file is bool_any.py

import traceback

def bool_any(data):
    # this is dummy routine; any tests expecting True will fail
    return False7

# put auto-cutting pattern here

if __name__ == '__main__':

    # here are our list of test-lists, of the form [input,output]
    # of course include as any inputs or outputs as required
    tsts = [["true",True],["false",False],["tRUe",True],["faLSe",False],
            ["t",True],["f",False],["T",True],["F",False],
            ["1",True],["0",False],["on",True],["OfF",False],
            ["23",-1],["O",-1],
            ]

    # run through all of the tests; notice use of try/except
    for tst1 in tsts:
        try:
            x = bool_any(tst1[0])
        except:
            traceback.print_exc()
            x = -1
        if( x != tst1[1]):
            print "\tError: bool_any('%s') failed" % tst1[0]
            # put an exit here; return or sys.exit(-1) as required

print "\tTest was Good"
```

RCI request

Remote Command Interface (RCI) is a method for remote clients to control, configure, and gather statistics from Digi Connect devices. RCI is a stateless, request/response protocol. RCI uses XML and HTTP to exchange data between clients and Digi devices.

RCI over HTTP

RCI requests are sent to the device using an URI of UE/rci. For example, if the Digi Device’s IP address is 192.168.1.1, then RCI requests are sent to http://192.168.1.1/UE/rci.

RCI requests are sent as an HTTP POST with the XML request of the form specified in this document. Note, due to space limitations on the device, the largest request that can be processed is 32KB. If a request is larger than this, it must be split into multiple RCI requests. RCI replies from the device are not subject to this limit.

Security is handled in the usual HTTP mechanism. The username and password must be passed to the device in the header of each HTTP request.

RCI over serial

RCI requests can also be sent over the serial port. This is useful in scenarios where a master processor is connected to the Digi Device through a serial port. This allows the master processor to configure the Digi Device as part of its configuration process, so that a separate manual configuration step for the Digi Device is eliminated. You must enable 'RCI over Serial' in either the Web Interface or the Command Line Interface before the Digi Device will accept RCI requests and return replies. The RCI over Serial option is available only on the primary port. RCI over Serial uses the DSR (Data Set Ready) serial signal. Verify that the serial port is not configured for autoconnect, modem emulation, or any other application which is dependent on DSR state changes. Note: When the Digi Device sees its DSR raised, it will set the serial port settings to 9600 baud, 8 data bits, no parity, and 1 stop bit. When DSR is lowered, the Digi Device will restore the previous serial settings.

Configure using the Command Line Interface (CLI)

1. Access the CLI using telnet or rlogin and the module's IP address. Ex:

```
telnet 192.168.1.2 -or-
rlogin 192.168.1.2
```

2. At the command prompt type:

```
 #> set rciserial state=on
```

Configure using the web user interface

1. Access the web interface by entering the module’s IP address in a browser’s URL window.
2. Choose Serial Ports from the Configuration menu.
3. If the device has more than one port, select Port 1.
4. If a port profile has not been selected, select Custom and click Apply.
5. Select Advanced Serial Settings.
6. Select Enable RCI over Serial (DSR) and click Apply.

RCI request/reply

An RCI XML document is identified by the XML elements rci_request and rci_reply. An RCI request specifies the XML element “rci_request” optionally with a version number. The version should match the version of RCI the client expects. The current RCI version is 1.1. If a version is not specified, the RCI version of the device is used to form the reply. Not specifying a version can cause problems when communicating with devices at different RCI versions, if the client code is not written in a version independent way. Therefore, it is highly recommended to always supply the version of RCI in requests, unless the client code has been designed to be version independent. Example of a request element:

```
<rci_request version="1.1">
```

The device will respond to requests with the element "rci_reply" along with the version number as an attribute. Example reply:

```
<rci_reply version="1.1">
```

rci_reply errors

Errors that occur at the request level will result in an error element as a sub-element of the <rci_ reply>. Errors and warnings are explained below <rci_reply> errors: Error ID Description

1. Request not valid XML
2. Request not recognized
3. Unknown command

Command

The command section of the protocol indicates the action requested (or action performed in replies).

Commands are specified as sub-elements to <rci_request> and <rci_reply>.

This example requests all configuration settings:

```
    <rci_request version="1.1"> <!—Identifies the protocol and whether this is a
request or a response --
        <query_setting/> <!— request config of device -->
    </rci_request>
```

This example requests the configuration information for just boot settings and serial settings.

```
    <rci_request version="1.1">
        <query_setting>
            <boot/>
            <serial/>
        </query_setting>
   </rci_request>
```

Supported commands

COMMAND	REQUEST DESCRIPTION	RESPONSE DESCRIPTION
query_setting	Request for device settings. May contain setting group elements to subset query (only setting group subset supported. Subsetting below this level not supported).	Returns requested config settings. Requests specifying no settings groups (eg. <query_setting/>) return all settings.
set_setting	Set settings specified in setting element. Settings data required.	Empty setting groups in reply indicate success. Errors returned as specified below.
query_state	Request current device state such as statistics and status. Sub-element may be supplied to subset results.	Returns requested state. Requests specifying no groups (eg. <query_state/> return all state.
set_factory_default	Sets device settings to factory defaults. Same semantics as set_setting.	Same semantics as set_ setting.
reboot	Reboots device immediately.	None
do_command	see RCI do command	see RCI do command

Errors and warnings

Response documents may contain an element as a child of the command or data element that indicates the result of the request. More than one error or warnings may be present. Error and Warning elements:

<table>
<tr><td>error</td><td>An error occurred.</td><td>Attribute id: A numeric id specified by the parent element (the command or the data element). An error element id="0" is equivalent to no error.</td><td>Children Elements<table><tr><th>name</th><th>desc</th></tr><tr><td>desc</td><td>Optional - Text description of the error.</td></tr><tr><td>hint</td><td>Optional - Used to indicate to the client the source of the error. This will typically be set to the field name that the error.</td></tr></table></td></tr>
<tr><td>warning</td><td>Command executed, but a warning was issued.</td><td></td><td></td></tr>
</table>

Example:

```
<serial_setting>
   <error id="3">
       <hint>baud</hint>
       <desc>Value out of valid range.</desc>
   </error>
    </serial_setting>
```

Errors are required to have an id. <hint> and <desc> are optional and more than one are allowed.

Notes

RCI XML must be well-formed XML

The device parses incoming RCI requests in a sequential manner. Each XML element is parsed and acted upon as it arrives. This is not ideal behavior, but is necessary because of the inherent resource limitations of a device. Ideally, the entire XML request would be read into memory, validated, parsed and acted upon only after validation.

XML structure errors may be found after actions have been taken. For instance:

```
<rci_request version=”1.0”>
    <set_factory_default/>
</rci_requestBADENDTAG>
```

This request will result in an XML parse error, but since the parse error occurs after the set_factory_defaults, the device will be set to factory defaults. Therefore, it is highly recommended that RCI requests be validated with an XML parser before being sent to the device. Using any standard parsers, such as the XML parsing in the Java SDK, to form RCI requests accomplishes this.

XML structure characters must not be sent as character data

Care must be taken to avoid accidental badly formed XML in RCI requests because of including XML structure characters, such as "<", as user entered data. Any field that accepts character data must be checked to ensure that "<" and ">" are not present (fields such as the email body of an alarm are common places this can happen). It is recommended that all instances of "<" and ">" in character data be converted to "<" and ">", which is the standard XML representation(entities) of these characters.

To use RCI to Query DIA device/channel Information

Reading device/channel information by direct HTTP to a DIA device requires a different do_command set. See Simple RCI by HTTP for working code examples.

References

https://www.digi.com/search/results?q=900005www.digi.com/search/results?q=9000056969

Transport Python programmer's guide

Purpose of this guide

This guide introduces the Python programming language by showing how to create and run a simple Python program. It describes how to load and run Python programs onto Digi Transport devices, either through the command-line or Web user interfaces. It reviews Python modules, particularly those modules with Digi-specific behavior. This guide describes how to run the executable programs and describes program files.

What Is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

For more information on the Python Programming Language, go to http://www.Python.org/ and click the Documentation link.

The Transports use Python version 2.6.1.

Running Python

How to run a Python program on a Digi Transport router. Firstly check your firmware version on the router, we recommend using firmware version 5090 or later. Start by checking your router has Python in its firmware by following these simple steps:

Step 1: Using either a telnet or serial connection (default login/password = username/password) to the router issue the following commands

```
pycfg 0 stderr2stdout on
Python
```

Step 2: At the prompt now type the following command:

```
help()
```

The following should then be displayed:

```
Welcome to Python 2.6!  This is the online help utility.
```

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.Python.org/tutorial/.

Step 3: Type the following to exit the Python interpreter exit() Now you can upload your Python script to the router via FTP. For testing purposes you can simply run the script by using the following command:

```
Python filename.py
```

For the router to start the script automatically on Powerup / Reboot issue the following commands:

```
cmd 0 autocmd "Python filename.py"
config 0 save
```

First Python script

Lets start with a hello world program. Firstly create a text file with the following text inside:

```
print "Hello World!"
```

Save this file with a file name of myfirst.py Now FTP the myfirst.py text file onto the router and issue the following command: Python myfirst.py

The router should produce the following output:

```
OK
Hello World!
```

Miscellaneous items

Your Python code can detect when it is running on a Transport product by importing the SYS module, then testing the sys.platform variable like this:

```
if sys.platform == 'digiSarOS':
   print "Running on Digi Transport"
```

File names on the Transport are limited to 8 characters or less. There is also a 3 character limit on file type extensions. Example:

```
  myfile.py  - is valid (6 characters and 2 character file type extension)
  myLongFileName.py - is not valid
  myfile.superlongextension - is not valid (6 characters are ok but we have more
than 3 characters on the filetype extension)
```

Also, please note that the Digi ESP, which is used as a code development tool, is not aware of this limitation. That means the programmer must make sure to use short filenames on their projects in the ESP.

Python examples can be generated on the Digi ESP code development tool (Digi ESP -> file -> new -> digi Python application sample project). This tool can be downloaded from the Digi website.

RCI is a remote protocol used for configuring the Transport from an application. RCI is defined on the Digi website in the RCI command reference Guide.

Device ID can be determined via the 'ati5' command.

Other example scripts

WR44 - bus demo, Python script

Below is the complete script we are using in our Bus Demo, this script collects the status of the two Digital Inputs and the current GPS co-ordinates. The script will only send data when there is a change in either the GPS co-ordinates or one of the Digital Inputs alters state. The file has been separated into individual modules for ease of explanation however you can download the complete script here bus-demo.py The required libraries import sarcli import time import socket Modules The first module converts the NMEA GPS info to lat / long co-ordinates:

```
def Lat_Long(raw_gps):
        gps_array = []
        gps_array = (raw_gps.split(','))

        if gps_array[1] == '':
                gps_array[1] = "4791.75429"
                gps_array[2] = "N"
        if gps_array[3] == '':
                gps_array[3] = "01208.62562"
                gps_array[4] = "E"

        lat_raw = gps_array[1]
        long_raw = gps_array[3]

        lat_dd = lat_raw[0:2]
        long_dd = long_raw[0:3]
        lat_mm = lat_raw[2:]
        long_mm = long_raw[3:]
        lat_dd = int(lat_dd)
        lat_mm = float(lat_mm)
        lat = (lat_mm / 60) + lat_dd
        lat = round(lat,3)

        long_dd = int(long_dd)
        long_mm = float(long_mm)
        long = (long_mm / 60) + long_dd
        long = round(long,3)

        if gps_array[2] == "S":
                lat = '-' + str(lat)
        else:
                lat = str(lat)

        if gps_array[4] == "W":
                long = '-' + str(long)
        else:
                long = str(long)
        return [lat,long]
```

This module creates a TCP Socket to the WR44 itself and collect the GPS info:

```
def getSocket_Data(gpsHOST, gpsPORT):

  data = 'None'

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((gpsHOST, gpsPORT))
        print 'Getting current GPS co-ordinates'
        data = s.recv(50)
        return data

  except:

        print 'Error - Cannot Connect to ', gpsHOST
        return data
```

This module create a TCP Socket to the web server and sends the Data:

```
def sendSocket_Data(HOST, PORT, data):

  try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
        print 'Successfully sent new status and GPS to', HOST
        s.send (data)

  except:
        print 'Error - sending new data to', HOST
```

This module issues a gpio command to the Transport command line interface to collect the status of the Digital Inputs:

```
def Get_Cli():
        clidata = ""
        cli = sarcli.open()
        cli.write("gpio")
        while True:
                tmpdata = cli.read(-1)
                if not tmpdata:
                        break
                clidata += tmpdata
        cli.close()
        return clidata
```

Checks the individual status of the gpio line:

```
def gpio_check(str, pin):
        if (pin + " : OFF") in str:
                return "off"

        return "on"
```

Writes the individual status of the 3 gpio lines into the variables:

```
def gpio(gpio_str):

        idx = gpio_str.find("Output:")

        input  = gpio_str[:idx-1]
        output = gpio_str[idx:]

        in_status     = gpio_check(input, "in")
        io_in_status  = gpio_check(input, "inout")
        io_out_status = gpio_check(output, "inout")

        return in_status, io_in_status, io_out_status
```

Define constants and Arrays

```
gpsHOST = '127.0.0.1'      # The remote host
gpsPORT = 2000             # The same port as used by the server
latlong = []               # Lat Long array
oldlatlong = []            # the old lat and long array
iostatus = []              # io status array
panic_status = "off"       # set the initial panic status to off
ign_status = "off"         # set the initial alarm status to off
```

```
HOST = 'gromit.mobilemonkey.co.uk'    # The remote web server hostname
PORT = 9999              # This is the port the web server is listening on


oldlatlong.append(50.000000003)
oldlatlong.append(-1.990900000)
```

The main program loop

```
            # Main
print "To stop, type \"Python kill\""

while True:
        clidata = Get_Cli()
        iostatus = gpio(clidata)
        changed = False
        if iostatus[0] == "off":
                if panic_status == "off":
                        panic_status = "on"
                        changed = True
        else:
                if panic_status == "on":
                        panic_status = "off"
                        changed = True

        if iostatus [1] == "on":
                if ign_status == "off":
                        ign_status = "on"
                        changed = True
        else:
                if ign_status == "on":
                        ign_status = "off"
                        changed = True



        rawgps = getSocket_Data(gpsHOST, gpsPORT)
#       print 'new data line' , rawgps
        latlong = Lat_Long(rawgps)
        if latlong[0] != oldlatlong[0]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[0] = latlong[0]
        if latlong[1] != oldlatlong[1]:
#               send the gps data and rewrite oldlatlong
                changed = True
                oldlatlong[1] = latlong[1]
        if changed == True:
                status = panic_status + ',' + ign_status
                data = status + ','+ str(latlong[0]) + ',' + str(latlong[1])
                print data
                sendSocket_Data(HOST, PORT, data)
        time.sleep(1)
```

Timed event, Python script

This script waits until a specific time of the day and then completes a task. The script has been separated into sections for ease of explanation however you can download the complete script here time-evn.py

```
import sarcli
import time

#Modules
#The module issues commands to the command line:

def cli(command):
           cli =  sarcli.open()
           cli.write(command)
           cli.close()


#Define constants and Arrays

event_time = "13:22"
running = True


#The main program loop

while running:

        # gets the current time ie "13:01"
        current_time = time.strftime ("%H:%M", time.localtime())

        if current_time == event_time:

                #  It is time to do something..

                print " I'm Doing something "
                command = 'setevent "time_evn.py: Its time to do something"'
                cli(command)

                time.sleep(61) # Sleep for 61 seconds so that we do not do the
event again.

        else:
                print current_time
                time.sleep(50) # Its not time go to sleep for 50 seconds.
```

Below is a telemetry card control Python script

Below is the complete script for controlling the telemetry card, this script waits for an SMS message which contains either a "Camera on" or "Camera off". After receiving the command it will process it changing the state of the relay on the Telemetry board and the reply back to the sender with a "camera now on /off" message. Download the complete script here

Note This script is offered as an example and its reliability can not be guaranteed, the router must have Firmware version 5100 or later to run. smsctrl.py

```
            #The required libraries

import threading
import sarcli
import os
import sys
import time
```

```
'''
Threads
The first thread runs continuously and checks the eventlog for 'SMS Received:'
Messages:
'''

class eventlog (threading.Thread):
    def run (self):

        # Constants
        running = True
          # This is what we are looking for in the eventlog.txt
        string_match = "SMS Received:"
        filename = "eventlog.txt"

        while running:
          # Try to open the eventlog.txt file
          try:
            file = open(filename, 'r')

          except:
           # Output to the eventlog if there is a problem opening the
eventlog.txt
           cli =  sarcli.open()
           cli.write('setevent "smsctrl:error opening file"')
           cli.close()

          # Check the eventlog.txt file for the string_match value.
          for line in file:
                if string_match in line:
                  line = line.strip('\r\n')
                  command_str = "basic 0 nv " + '"' + line + '"'
                  cli = sarcli.open()
                  cli.write(command_str)
                  cli.close()
                  break

        file.close()
        time.sleep(10)

# This is thread that turns on the Relay waits 30secs then turns it off:

class DelayOnOff (threading.Thread):
    def run (self):

        answer = SetRelay("on")
        ReplySms(answer, cmd_array[1])
        # Wait 30 seconds
        time.sleep(30)
        answer = SetRelay("off")
        ReplySms(answer, cmd_array[1])

#This module takes the eventlog item and separates the command and phone number:

def GetEvent(output_string):
  event_array = []
  command_array = []
```

```
  event_array = (output_string.split(','))
  command= str(event_array[2])
  command_array=(command.split(':'))
  command_array[1] = str(command_array[1]).lstrip()
  command_array[2] = str(command_array[2]).lstrip()
  return command_array

#This module alters the State of the Relay on the Telemetry board and returns the
relays status:

def SetRelay(state):
  clir = sarcli.open()
  if state == "on":
        try:
           clir.write("anaconda -y 1")
           clir.write('setevent "Smsctrl:Camera now on"')
           answer= "on"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"

  elif state == "off":
        try:
           clir.write("anaconda -y 0")
           clir.write('setevent "Smsctrl:Camera now off"')
           answer= "off"
        except:
           clir.write('setevent "smsctrl:error setting relay"')
           answer= "error"
  clir.close()
  return answer


#This module sends back an SMS reply to the originators phone number with the
status:

def ReplySms(answer, phonenum):
  clir = sarcli.open()
  if answer == "on":
            sms = 'sendsms ' + phonenum + ' "Camera now on"'

  elif answer == "off":
            sms = 'sendsms ' + phonenum + ' "Camera now off"'
  else:
            sms = 'sendsms ' + phonenum + ' " Error setting relay"'

  try:
    clir.write(sms)
  except:
    clir.write('setevent "smsctrl:error sending sms"')
    clir.close()


#Define constants and Arrays

# Constants
running = True

#Variables
```

```
completed_command = "null"
cmd_array = []

#The main program loop


# Main
print "To stop, type \"Python kill\""

# Start eventlog Thread
eventlog().start()

while running:

        cli = sarcli.open()
        try:
           cli.write("basic 0 nv")
           tmpdata = cli.read(-1)
        except:
          print "error writing command: ", tmpdata
          errormsg = 'setevent ' + '"' + tmpdata + '"'
          cli.write("errormsg")
        cli.close()
        if completed_command != tmpdata:
                print "new event: ", tmpdata
                cmd_array = GetEvent(tmpdata)
                if cmd_array[2] == "Camera on":
                        answer = SetRelay("on")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera off":
                        answer = SetRelay("off")
                        ReplySms(answer, cmd_array[1])
                        completed_command = tmpdata
                elif cmd_array[2] == "Camera on 30":
                        DelayOnOff().start()
                        completed_command = tmpdata
                else:
                        time.sleep(5)
        else:
                time.sleep(5)
```

Wake on Lan, Python script

This script sends out a Wake on Lan packet to a specific host. The script has been separated into sections for ease of explanation however you can download the complete script here wol.py.

```
            #The required libraries

import socket
import struct

#Modules
#The module sends out Wake On LAN packets

def wake_on_lan(macaddress):

    """ Switches on remote computers using WOL. """
```

```
    # Check macaddress format and try to compensate.

    if len(macaddress) == 12:
        pass

    elif len(macaddress) == 12 + 5:

        sep = macaddress[2]
        macaddress = macaddress.replace(sep, '')

   else:
        raise ValueError('Incorrect MAC address format')

    # Pad the synchronization stream.
    data = ''.join(['FFFFFFFFFFFF', macaddress * 20])
    send_data = ''

    # Split up the hex values and pack.

    for i in range(0, len(data), 2):

        send_data = ''.join([send_data,

                             struct.pack('B', int(data[i: i + 2], 16))])

    # Broadcast it to the LAN.

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    sock.sendto(send_data, ('192.168.1.255', 9))

#The main program loop

if __name__ == '__main__':

    # Use macaddresses with any seperators.
    wake_on_lan('00:40:63:FC:0C:75')
    wake_on_lan('00-40-63-FC-0C-75')

    # or without any seperators.
    wake_on_lan('004063FC0C75')
```

Supported Python modules

```
    A
        abc
        aifc
        anydbm
        array
        ast
        asynchat
        asyncore
        atexit
        audiodev
    B
        BaseHTTPServer
        Bastion
```

```
    base64
    bdb
    binascii
    binhex
    bisect
C
    CGIHTTPServer
    ConfigParser
    Cookie
    cPickle
    cProfile
    cStringIO
    calendar
    cgi
    cgitb
    chunk
    cmath
    cmd
    code
    codecs
    codeop
    collections
    colorsys
    commands
    compileall
    contextlib
    cookielib
    copy
    copy_reg
    csv
D
    DocXMLRPCServer
    datetime
    dbhash
    decimal
    difflib
    digihw
    digiwdog
    digiweb
    dircache
    dis
    doctest
    dumbdbm
    dummy_thread
    dummy_threading
E
    email
    encodings
    errno
    exceptions
F
    filecmp
    fileinput
    fnmatch
    formatter
    fpformat
    fractions
    ftplib
    functools
```

```
    future_builtins
G
    gc
    genericpath
    getopt
    getpass
    gettext
    glob
    gzip
H
    HTMLParser
    hashlib
    heapq
    hmac
    htmlentitydefs
    htmllib
    httplib
I
    ihooks
    imaplib
    imghdr
    imp
    imputil
    inspect
    io
    itertools
K
    keyword
L
    linecache
    locale
    logging
M
    MimeWriter
    macpath
    macurl2path
    mailbox
    mailcap
    markupbase
    marshal
    math
    md5
    mhlib
    mime
    mimetools
    mimetypes
    mimify
    modulefinder
    multifile
    mutex
N
    netrc
    new
    nntplib
    ntpath
    nturl2path
    numbers
O
    opcode
```

```
    operator
    optparse
    os
    os2emxpath
P
    parser
    pdb
    pickle
    pickletools
    pipes
    pkgutil
    platform
    plistlib
    popen2
    poplib
    posix
    posixfile
    posixpath
    pprint
    profile
    pstats
    pty
    py_compile
    pyclbr
    pydoc
    pydoc_topics
    pyexpat
Q
    Queue
    quopri
R
    random
    re
    repr
    rexec
    rfc822
    rlcompleter
    robotparser
    runpy
S
    SimpleHTTPServer
    SimpleXMLRPCServer
    SocketServer
    StringIO
    sarcli
    sarutils
    sched
    select
    sets
    sgmllib
    sha
    shelve
    shlex
    shutil
    site
    smtpd
    smtplib
    sndhdr
    socket
```

```
        sre
        sre_compile
        sre_constants
        sre_parse
        ssl
        stat
        statvfs
        string
        stringold
        stringprep
        strop
        struct
        subprocess
        sunau
        sunaudio
        symbol
        symtable
        sys
   T
        tabnanny
        tarfile
        telnetlib
        tempfile
        termios
        textwrap
        this
        thread
        threading
        time
        timeit
        toaiff
        token
        tokenize
        trace
        traceback
        tty
        types
   U
        UserDict
        UserList
        UserString
        unittest
        urllib
        urllib2
        urlparse
        user
        uu
        uuid
    W
        warnings
        wave
        weakref
        webbrowser
        whichdb
        X
        xdrlib
        xmllib
        xmlrpclib
    Z
```

```
zipfile
zipimport
zlib
```

References

Portions of this document are reproduced from documents by Matt jameson & Jon Lyons: extranet.jrp2.com/~jpowell/gromit.mobilemonkey.co.uk/trans-Python.html

Use Telnet to configure

Using Python and Telnet to configure a Digi device

Suppose you have 200 Digi products to configure ... do you need to log into the Web UI 200 times and try to remember to set all of the correct changes? Of course not. One option is to use the Web UI's Backup/Restore to copy configurations from 1 unit known to be setup correctly to the other 199.

Python to load a configuration

However, you can also use Python and Telnet to do the same thing. This routine sends all text lines in a file named 'config.txt' to a Digi device:

```
            #
# send_config.py
#
# use telnet to send text lines down to a Digi Product

import sys
import time
import telnetlib

if __name__ == '__main__':

    bLive = True
    settings = {
'cpxip':"192.168.1.20",'tcpport':23,'username':"root",'password':"dbps"}

    if(len(sys.argv) > 1):
        # see if we have any arguments on command line
        settings.update( { 'cpxip' : sys.argv[1] } )
    else:
        print 'This script sends the file named "config.txt" down to a Digi
product via telnet'
        print
        print 'usage:'
        print '   %s {ip of product}' % sys.argv[0]
        sys.exit( 0)

    cpxip = settings.get("cpxip","192.168.1.20")
    tcpport = settings.get("tcpport",23)
    username = settings.get("username","root")
    password = settings.get("password","dbps")
    cpxcfg = settings.get("cpxfile","config.txt")

    # first we test if the file exists
    try:
        cpxcfg_fil = open( cpxcfg, 'r' )

    except:
        print "File <%s> was not found!" % cpxcfg
        sys.exit( -1)

    print "Attempting to send file <%s> to IP:%s" % (cpxcfg,cpxip)

    try:
```

```
        tn = telnetlib.Telnet(cpxip,tcpport)
        if( len(username) > 0):
            tn.read_until( "login: ", 10)
            tn.write( username + "\n")
            tn.read_until( "password:", 10)
            tn.write( password + "\n")

except:
    print "IP Address <%s> was not found, or login failed!" % cpxip
    sys.exit( -2)

print tn.read_until( "#> ", 10)

# At this point, we should be connected.

for lin in cpxcfg_fil:
    # for each line in file, send to product
    #  print "Line=: ", lin
    tn.write( lin)
    if not lin.startswith("boot a=r"):
        print tn.read_until( "#> ", 10)
        time.sleep( 0.05)
    else:
        break

    # delay a slight bit and exit
    time.sleep( 0.5)
    tn.close()
```

Example TEXT file for a DigiOne SP or IA

```
# example Config for a Digi One SP or IA
#
# Assumes unit is nearly factory default, with an IP in the 192.168.0.x subnet
# we set the gateway, but not the IP - leave that as is
set config submask=255.255.255.0 gateway=192.168.0.1 dns=192.168.0.1
# set the serial port speeds
set line range=1 parity=N csize=8 error=ignore baud=19200 stopb=1
# enable the TCP sockets,
# with the extra option to "pack" incoming serial data until 50 msec of idle time
seen
set port range=1 dev=prn idletime=50
# set Wide-Area-Network friendly TCP Keepalive settings
set tcpip keepalive_active=on keepalive_idle=00:04:30 keepalive_byte=off
set tcpip probe_cnt=5 probe_interval=30
# Reboot the device
boot a=r
```

Using ZIP, GZIP or compression

Compression under Python

If you want to compress (or uncompress) ZIP files, use the GZIP module - see Python.org/library/gzip.html.

Note Using compression requires considerable free memory. If your Python application is already running out of memory (for example, on an older ConnectPort X2 with 8BM RAM only), then trying to add compression will break it. You should also consider using gc.collect() frequently to free up unused memory more rapidly.

Example of how to read a compressed file

Online examples work fine - just remember to preface file name with "WEB/Python/"

```
import gzip
f = gzip.open('WEB/Python/file.zip', 'rb')
file_content = f.read()
f.close()
```

Example of how to create a compressed GZIP file

```
import gzip
content = "Lots of content here"
f = gzip.open('WEB/Python/file.zip', 'wb')
f.write(content)
f.close()
```

Example of how to GZIP compress an existing file

```
import gzip
f_in = open('WEB/Python/file.txt', 'rb')
f_out = gzip.open('WEB/Python/file.zip', 'wb')
f_out.writelines(f_in)
f_out.close()
f_in.close()
```

Example of how to GZIP compress without using FLASH files

Saving temporary FLASH files is both slow and risks file system corruption. If your goal to create a RAM-resident data object for upload to a remote FTP or HTTP server, then you can create a ZIP image directly in StringIO objects, which can be thought of as RAM-based files.

```
import StringIO
import gzip
import gc

# code here creates String object named data
data = "Lots of Data"

zipr = StringIO.StringIO()
# we need to use filename to put correct 'name' into ZIP file
tmpf = gzip.GzipFile(filename='data.orig', mode='wb', fileobj=zipr,)
```

```
tmpf.write(data)
tmpf.close()
# we no longer need tmpf or data - zipr is zipped version of data
del buf
del tmpf


# do what you want with 'zipr' here - for example push elsewhere by
# ftplib or httplib
send_file_to_host("data.zip", zipr.getvalue())

# manually push garbage collection to speed up memory recovery
del zipr
gc.collect()
```

Running code packaged within a ZIP file

You do NOT need to use GZIP to access files packed within ZIP files. You merely add the ZIP to the PythonPATH as if it were a common sub-directory to search for files.

```
import sys

# do this at the start of your main routine
sys.path.insert(0, 'WEB/Python/my_project.zip')

# at this point, imports from 'my_project.zip' should work.
import my_file
```

See Also: Loading Python programs onto a Digi device.

Windows PythonPath

Telling Windows were to find code

Most programmers are familiar with how the path (or %PATH%) environment variable tells the OS where to find code to run. Python under Windows adds another variable named PythonPATH, which acts as an INCLUDE or LIB directory list. Many IDE tools (such as Mark Hammond's Pythonwin) have tools to safely edit the Windows registry entries which Python prefers, and placing the most common libraries you use here makes sense. Python under Windows finds the standard libraries by such Windows registry entries.

However, another solution is to create a collection of command-line .BAT files such as below. Place these either in your Windows PATH (C:\Windows is one place) or in the directory Windows drops you into when opening a command window.

This batch file does the following:

- Updates the CMD / DOS PATH variable (you might not require this)
- Moves you to the project directory
- Updates PythonPATH with just the directories required for this project

```
set PATH=%PATH%;C:\Python24;
h:
cd \digi\modbus_tests
set PythonPATH=C:\Python24\Lib\site-packages\serial;C:\digi\iatest\modbus
```

Java

This category explains about several Java Sample applications.

Advanced Device Discovery Protocol (ADDP)

What is ADDP?

ADDP (Advanced Device Discovery Protocol) is a proprietary protocol developed by Digi International that allows devices on a local network to be found regardless of their network configuration.

How does it work?

ADDP uses a client/server model. The client is the application that is searching for devices. The server is the device that is being search for.

In the simplest terms, the client application sends out a specially formatted UDP broadcast packet on the network. ADDP servers listening for the packet, will receive it, and send an ADDP response back to the client. Once this process is complete, the client can then send configuration requests to the device. These can include things like network settings, and reboot requests.

Java library

A subset of the protocol has been implemented in Java. You can find the jar file here: ADDP Library.

The associated javadoc documentation can be found here: ADDP Java doc.

This library allows you to search synchronously, and asynchronously for devices on the network. You can then use it to reconfigure the device's network settings, or reboot the device.

Java sample application

A simple discovery sample application can be found here: AddpSample(r2010).zip

Basic usage

First, instantiate the AddpClient object.

```
AddpClient addpClient = new AddpClient();
```

Next, call SearchForDevices() and check the return value. Then get the devices, and walk the hashtable.

```
if (addpClient.SearchForDevices()) {
    AddpDeviceList deviceList = addpClient.getDevices();

    Enumaration<AddpDevice> e = deviceList.elements();
    while(e.hasMoreElements()) {
       AddpDevice device = e.nextElement();

       // do something with the device here
       System.out.println(device.toString());

       // if device is not configured for DHCP, then turn it on and reboot.
       if (device.getDHCP() == 0) {
          addpClient.setDHCP(device, true, "dbps");
          addpClient.rebootDevice(device, "dbps");
       }
```

```
    }
}
```

DogFighter

DogFighter sample test

(**For java supported modules**) Java program; An image rendering sample.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java DogFighter test sample application

The DogFighter Test sample application can be found here: DogFighter.zip.

Basic usage

Compile, load and run program using java environment.

Sample of DogFighter.java file:

```
package com.digi.DogFighter;

import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.image.BufferStrategy;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferInt;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class DogFighter extends Canvas implements Runnable {
        private static final long serialVersionUID = 1L;

        private static final int WIDTH = 320;
        private static final int HEIGHT = 240;
        private static final int SCALE = 2;

        private boolean running;
        private Thread thread;

        private Game game;
        private Screen screen;
        private BufferedImage img;
        private int[] pixels;
//      private InputHandler inputHandler;
//      private Cursor emptyCursor, defaultCursor;
//      private boolean hadFocus = false;

        // handles the FPS counter
        private long elapsed;
```

```
        private long start_time;

        Random random = new Random();

        public DogFighter() {
                Dimension size = new Dimension(WIDTH * SCALE, HEIGHT * SCALE);
                setSize(size);
                setPreferredSize(size);
                setMinimumSize(size);
                setMaximumSize(size);

                game = new Game();
                screen = new Screen(WIDTH, HEIGHT);
                img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_
RGB);
                pixels = ((DataBufferInt) img.getRaster().getDataBuffer
()).getData();
                screen.pixels = pixels;

                // inputHandler = new InputHandler();

               // addKeyListener(inputHandler);
               // addFocusListener(inputHandler);
               // addMouseListener(inputHandler);
               // addMouseMotionListener(inputHandler);
//             emptyCursor = Toolkit.getDefaultToolkit().createCustomCursor(
//                             new BufferedImage(16, 16, BufferedImage.TYPE_INT_ARGB),
//                             new Point(0, 0), "empty");
//             defaultCursor = getCursor();
        }

        public synchronized void start() {
               if (running)
                      return;
               running = true;
               thread = new Thread(this);
               thread.start();
       }

        public synchronized void stop() {
                if (!running)
                         return;
                running = false;
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        @Override
        public void run() {
                int count = 0;
                elapsed = 0;
                start_time = System.nanoTime();
                game.init();

                //runs the game for one tick at a time
                while (running) {
```

```
                    // used to estimate FPS
                    if (count >= 200) {
                            elapsed = (System.nanoTime() - start_time) / 200;
                            start_time = System.nanoTime();
                            count = 0;
                    }

                   // run the game
                   game.doTick();

                   // render it
                   render();

                   count++;
           }

    }

    private void render() {
            // get the next frame buffer (we use triple buffering?)
            BufferStrategy bs = getBufferStrategy();
            if (bs == null) {
                    createBufferStrategy(3);
                    return;
            }

           // have the screen object render the game
           screen.render(game);

           // draw in the FPS counter
           screen.draw("FPS:" + (int) (1 / (elapsed / 1000000000.0)), 0, 0);

           // draw the screen onto the BufferedImage
           //for (int i = 0; i < WIDTH * HEIGHT; i++)
{
           //pixels[i] = screen.pixels[i];
           //}

           // get the next frame buffer
           Graphics g = bs.getDrawGraphics();
           // draw the screen to it.
           g.fillRect(0, 0, getWidth(), getHeight());
           g.drawImage(img, 0, 0, WIDTH * SCALE, HEIGHT * SCALE, null);
           g.dispose();
           // flip the buffer to the front
           bs.show();
    }


    // creates and configures the window
    public static void main(String[] args) {
            DogFighter df = new DogFighter();

            JFrame frame = new JFrame("DogFighter Sample");

            JPanel panel = new JPanel(new BorderLayout());
            panel.add(df, BorderLayout.CENTER);

            frame.setContentPane(panel);
```

```
            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setResizable(false);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);

            df.start();
    }
}
```

Etherios Jenova connector

Etherios Jenova connector sample test

(**For java supported modules**) Standalone java application via which you can send linux commands to a linux host via DC.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java Ethrios Jenova connector test sample application

The Etherios Jenova Connector Test sample application can be found here: EtheriosJenovaConnector.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Base64.java file:

```
package com.etherios.jenovaconnector;

public final class Base64<b1> {
        static private final int BASELENGTH = 128;
        static private final int LOOKUPLENGTH = 64;
        static private final int TWENTYFOURBITGROUP = 24;
        static private final int EIGHTBIT = 8;
        static private final int SIXTEENBIT = 16;
        static private final int FOURBYTE = 4;
        static private final int SIGN = -128;
        static private final char PAD = '=';
        static private final boolean fDebug = false;
        static final private byte[] base64Alphabet = new byte[BASELENGTH];
        static final private char[] lookUpBase64Alphabet = new char
[LOOKUPLENGTH];
        static {
                for (int i = 0; i < BASELENGTH; ++i) {
                        base64Alphabet[i] = -1;
                }

                for (int i = 'Z'; i >= 'A'; i--) {
                        base64Alphabet[i] = (byte) (i - 'A');
                }
                for (int i = 'z'; i >= 'a'; i--) {
                        base64Alphabet[i] = (byte) (i - 'a' + 26);
                }

                for (int i = '9'; i >= '0'; i--) {
                        base64Alphabet[i] = (byte) (i - '0' + 52);
                }

                base64Alphabet['+'] = 62;
                base64Alphabet['/'] = 63;
```

```
                for (int i = 0; i <= 25; i++)
                        lookUpBase64Alphabet[i] = (char) ('A' + i);

                for (int i = 26, j = 0; i <= 51; i++, j++)
                        lookUpBase64Alphabet[i] = (char) ('a' + j);

                for (int i = 52, j = 0; i <= 61; i++, j++)
                        lookUpBase64Alphabet[i] = (char) ('0' + j);

                lookUpBase64Alphabet[62] = (char) '+';
                lookUpBase64Alphabet[63] = (char) '/';
        }

        protected static boolean isWhiteSpace(char octect) {
                return (octect == 0x20 || octect == 0xd || octect == 0xa ||
octect == 0x9);
        }


        protected static boolean isPad(char octect) {
                return (octect == PAD);
        }

        protected static boolean isData(char octect) {
                return (octect < BASELENGTH && base64Alphabet[octect] != -1);
        }

        protected static boolean isBase64(char octect) {
                return (isWhiteSpace(octect) || isPad(octect) || isData(octect));
        }

        /**
        * Encodes hex octects into Base64
        *
        * @param binaryData
        *            Array containing binaryData
        * @return Encoded Base64 array
        */
        public static String encode(byte[] binaryData) {
                if (binaryData == null)
                        return null;

                int lengthDataBits = binaryData.length * EIGHTBIT;
                if (lengthDataBits == 0) {
                        return "";
        }

        int fewerThan24bits = lengthDataBits % TWENTYFOURBITGROUP;
        int numberTriplets = lengthDataBits / TWENTYFOURBITGROUP;
        int numberQuartet = fewerThan24bits != 0 ? numberTriplets + 1
                        : numberTriplets;
        char encodedData[] = null;

        encodedData = new char[numberQuartet * 4];

        byte k = 0, l = 0, b1 = 0, b2 = 0, b3 = 0;
        int encodedIndex = 0;
        int dataIndex = 0;
```

```
        if (fDebug) {
                System.out.println("number of triplets = " + numberTriplets);
        }

        for (int i = 0; i < numberTriplets; i++) {
                b1 = binaryData[dataIndex++];
                b2 = binaryData[dataIndex++];
                b3 = binaryData[dataIndex++];

                if (fDebug) {
                        System.out.println("b1= " + b1 + ", b2= " + b2 + ", b3= "
+ b3);
                }

                l = (byte) (b2 & 0x0f);
                k = (byte) (b1 & 0x03);

                byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                                : (byte) ((b1) >> 2 ^ 0xc0);

                byte val2 = ((b2 & SIGN) == 0) ? (byte) (b2 >> 4)
                                : (byte) ((b2) >> 4 ^ 0xf0);

                byte val3 = ((b3 & SIGN) == 0) ? (byte) (b3 >> 6)
                                : (byte) ((b3) >> 6 ^ 0xfc);

                if (fDebug) {
                        System.out.println("val2 = " + val2);
                        System.out.println("k4   = " + (k << 4));
                        System.out.println("vak  = " + (val2 | (k << 4)));
                }

                encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[val2 | (k <<
4)];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[(l << 2) |
val3];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[b3 & 0x3f];
        }

        // form integral number of 6-bit groups
        if (fewerThan24bits == EIGHTBIT) {
                b1 = binaryData[dataIndex];
                k = (byte) (b1 & 0x03);

                if (fDebug) {
                        System.out.println("b1=" + b1);
                        System.out.println("b1<<2 = " + (b1 >> 2));
                }

                byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                                : (byte) ((b1) >> 2 ^ 0xc0);

                encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
                encodedData[encodedIndex++] = lookUpBase64Alphabet[k << 4];
                encodedData[encodedIndex++] = PAD;
                encodedData[encodedIndex++] = PAD;
        } else if (fewerThan24bits == SIXTEENBIT) {
                b1 = binaryData[dataIndex];
```

```
                b2 = binaryData[dataIndex + 1];
                l = (byte) (b2 & 0x0f);
                k = (byte) (b1 & 0x03);

               byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2)
                               : (byte) ((b1) >> 2 ^ 0xc0);

               byte val2 = ((b2 & SIGN) == 0) ? (byte) (b2 >> 4)
                               : (byte) ((b2) >> 4 ^ 0xf0);

               encodedData[encodedIndex++] = lookUpBase64Alphabet[val1];
               encodedData[encodedIndex++] = lookUpBase64Alphabet[val2 | (k <<
4)];
               encodedData[encodedIndex++] = lookUpBase64Alphabet[l << 2];
               encodedData[encodedIndex++] = PAD;
        }

        return new String(encodedData);
     }

    /**
    * Decodes Base64 data into octects
    *
    * @param encoded
    *            string containing Base64 data
    * @return Array containind decoded data.
    */
    public static byte[] decode(String encoded) {
            if (encoded == null)
                    return null;

            char[] base64Data = encoded.toCharArray();

            // remove white spaces
            int len = removeWhiteSpace(base64Data);

            if (len % FOURBYTE != 0) {
                    return null;
                    // should be divisible by four
            }


            int numberQuadruple = (len / FOURBYTE);

            if (numberQuadruple == 0)
                    return new byte[0];

            byte decodedData[] = null;
            byte b1 = 0, b2 = 0, b3 = 0, b4 = 0;
            char d1 = 0, d2 = 0, d3 = 0, d4 = 0;
            int i = 0;
            int encodedIndex = 0;
            int dataIndex = 0;

            decodedData = new byte[(numberQuadruple) * 3];

            for (; i < numberQuadruple - 1; i++) {
                    if (!isData((d1 = base64Data[dataIndex++]))
                                    || !isData((d2 = base64Data[dataIndex++]))
```

```
                                    || !isData((d3 = base64Data[dataIndex++]))
                                    || !isData((d4 = base64Data[dataIndex++])))
                            return null;
                   // if found "no data" just return null
                    b1 = base64Alphabet[d1];
                    b2 = base64Alphabet[d2];
                    b3 = base64Alphabet[d3];
                    b4 = base64Alphabet[d4];

                   decodedData[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                   decodedData[encodedIndex++] = (byte) (((b2 & 0xf) << 4) | ((b3
>> 2) & 0xf));
                   decodedData[encodedIndex++] = (byte) (b3 << 6 | b4);
            }

            if (!isData((d1 = base64Data[dataIndex++]))
                            || !isData((d2 = base64Data[dataIndex++]))) {
                    return null;
                    // if found "no data" just return null
            }

            b1 = base64Alphabet[d1];
            b2 = base64Alphabet[d2];
            d3 = base64Data[dataIndex++];
            d4 = base64Data[dataIndex++];

            if (!isData((d3)) || !isData((d4))) {
                   // Check if they are PAD characters
                   if (isPad(d3) && isPad(d4)) {
                           // Two PAD e.g. 3c[Pad][Pad]
                            if ((b2 & 0xf) != 0)// last 4 bits should be zero
                                    return null;

                            byte[] tmp = new byte[i * 3 + 1];

                            System.arraycopy(decodedData, 0, tmp, 0, i * 3);
                            tmp[encodedIndex] = (byte) (b1 << 2 | b2 >> 4);

                            return tmp;
                  } else if (!isPad(d3) && isPad(d4)) {
                            // One PAD e.g. 3cQ[Pad]
                            b3 = base64Alphabet[d3];

                            if ((b3 & 0x3) != 0)// last 2 bits should be zero
                            return null;

                            byte[] tmp = new byte[i * 3 + 2];
                            System.arraycopy(decodedData, 0, tmp, 0, i * 3);
                            tmp[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                            tmp[encodedIndex] = (byte) (((b2 & 0xf) << 4) | ((b3
>> 2) & 0xf));

                            return tmp;
                    } else {
                            return null;
                            // an error like "3c[Pad]r", "3cdX", "3cXd", "3cXX"
where X is
                            // non data
                    }
```

```
            } else {
                    // No PAD e.g 3cQl
                    b3 = base64Alphabet[d3];
                    b4 = base64Alphabet[d4];

                    decodedData[encodedIndex++] = (byte) (b1 << 2 | b2 >> 4);
                    decodedData[encodedIndex++] = (byte) (((b2 & 0xf) << 4) |
((b3 >> 2) & 0xf));
                    decodedData[encodedIndex++] = (byte) (b3 << 6 | b4);
            }
            return decodedData;
        }

        /**
        * remove WhiteSpace from MIME containing encoded Base64 data.
        *
        * @param data
        *            the byte array of base64 data (with WS)
        * @return the new length
        */

         protected static int removeWhiteSpace(char[] data) {
                 if (data == null)
                         return 0;


                 // count characters that's not whitespace
                int newSize = 0;
                int len = data.length;

                for (int i = 0; i < len; i++) {
                        if (!isWhiteSpace(data[i]))
                                data[newSize++] = data[i];
                }

                return newSize;
        }


}
```

GetConnectTankAttributes

GetConnectTankAttributes sample test

(**For java supported modules**) This application gets the connect tank attributes from the device cloud and displays it to the user.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java GetConnectTankAttributes test sample application

The GetConnectTankAttributes Test sample application can be found here: GetConnectTankAttributes.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.DCT;
//imports
import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JT
xtArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

/* Brief description of the terms used in the program.
- JPanel - JPanel is a generic lightweight container
- JLabel - A JLabel object can display either text, an image, or both
- JTextField -  JTextField is a lightweight component that allows the editing of
a single line of text
- JButton - An implementation of a "push" button when it is clicked an event
should takes place
- JTable - The JTable is used to display and edit regular two-dimensional tables
of cells
- Font - Setting Font for the Components
*/

public class Main extends JFrame {


       /**
        *
        */
        private static final long serialVersionUID = 1L;

        private JTextField txtUsername;
        private JTextField txtPassword;
        private JTable tblDevices;
        private JTable tblAttributeList;
        private DeviceListTableModel devicesTableModel;
        private AttributeTableModel AttributeListTableModel;

        private String username;
        private String password;
        private String highlightedDevice;
        public String indicator = null;
        public String login_indicator = null;
        final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
        final JLabel credentialsInforLabel = new JLabel("Enter Device Cloud
credentials, click Connect and please wait for few seconds!!");
        private static final Color cl_black = new Color(21, 45, 60);
        private static final Color cl_dkgray = new Color(110, 110, 110);
        private static final Color cl_grn = new Color(12, 130, 68);
        public Font label = new Font("Times New Roman", Font.BOLD, 15);
        final JLabel wrongDeviceInfo;
        JPanel devicesScrollPanePanel = new JPanel();
        JPanel attributeScrollPanePanel = new JPanel();
        JPanel motherPanel = new JPanel();

        JPanel basePanel = new JPanel();
        JPanel extraPanel = new JPanel();
```

```
        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public Font fontDevice = new Font("Times New Roman", Font.PLAIN, 13);
        public Font fontAttribute = new Font("Times New Roman", Font.PLAIN, 15);
        public final JLabel waitInfo = new JLabel("Please select Connect Tank
device and wait!!");


        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }
        //constructor of the Main class which is responsible for the GUI
        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 950);

                setTitle("Digi Connect Tank");

                //adding main panel to the main window
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));
                credentialsInforLabel.setSize(2, 2);
                credentialsInforLabel.setFont(font_bold);
                credentialsInforLabel.setForeground(cl_grn);

                // panel which holds the user info
                JPanel credentialsPanel = new JPanel();
                credentialsPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5,
5));
                credentialsPanel.add(credentialsInforLabel);
                // adding Panels to the mother panel
                motherPanel.add(credentialsPanel);
                // adding basePanel which holds other panels to main panel i.e.,
mother panel
                motherPanel.add(basePanel);
```

```
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                //userPanel which holds username and username textfield
                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
                // declaring a Label
                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);
                userLabel.setFont(label);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);

                //passPanel which holds password label and password textfield
                JPanel passPanel = new JPanel();
                //adding passPanel to basePanel
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
                passLabel.setFont(label);

                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);

                //Panel which holds URL combo box
                JPanel dropListConnectPanel = new JPanel();
                //adding panel to basic panel
                basePanel.add(dropListConnectPanel);
                final JLabel correctDeviceInfo = new JLabel("Latest values sent
by the Connect Tank to Device Cloud!");
                correctDeviceInfo.setSize(5, 5);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(cl_grn);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select Connect Tank and wait for few seconds!");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(cl_grn);

                waitInfo.setVisible(false);
                waitInfo.setSize(5, 5);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(cl_grn);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(cl_grn);
                wrongCredentialsInfo.setFont(font_bold)        ;
```

```
                    //combo box which has list of URL's
                    JComboBox dropDownURLList = new JComboBox();
                    dropDownURLList.addItem(listURLItems[0]);
                    dropDownURLList.addItem(listURLItems[1]);
                    //adding URL's list to panel
                    dropListConnectPanel.add(dropDownURLList);
                    dropDownURLList.setSize(50,25);
                    dropDownURLList.setFont(label);

                    // declaring a button
                        JButton btnConnect = new JButton("Connect");
                        //adding button to the panel
                        dropListConnectPanel.add(btnConnect);
                        btnConnect.setOpaque(true);
                        btnConnect.setSize(new Dimension(50, 25));
                        //set font to button
                        btnConnect.setFont(label);

                        //adding scroll panel to main motherPanel
                        motherPanel.add(devicesScrollPanePanel
                        devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));
                        //adding user info to panel
                        devicesScrollPanePanel.add(wrongCredentialsInfo);
                        devicesScrollPanePanel.add(waitInfo);
                        //declaring a scroll pane
                        JScrollPane devicesScrollPane = new JScrollPane();
                        //adding scroll pane to panel
                        devicesScrollPanePanel.add(devicesScrollPane);
                        //table which displays all the devices in teh user account
                        tblDevices = new JTable();
                        tblDevices.setFillsViewportHeight(true);
                        tblDevices.setPreferredScrollableViewportSize(new Dimension
(800, 150));
                        devicesScrollPane.setViewportView(tblDevices);
                        tblDevices.setColumnSelectionAllowed(true);
                        // object of deviceLidtTableModel
                        devicesTableModel = new DeviceListTableModel();
                        tblDevices.setModel(devicesTableModel);
                        tblDevices.setRowHeight(22);
                        //set font for the table
                        tblDevices.setFont(fontDevice);
                        // Setting column width
                        tblDevices.getColumnModel().getColumn(0).setPreferredWidth
(140);
                        tblDevices.getColumnModel().getColumn(1).setPreferredWidth
(300);
                        tblDevices.getColumnModel().getColumn(2).setPreferredWidth
(125);
                        tblDevices.getColumnModel().getColumn(3).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(4).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(5).setPreferredWidth
(124);
                        tblDevices.getColumnModel().getColumn(6).setPreferredWidth
(124);
```

```
                    //adding terminalOuter to main motherPanel
                    motherPanel.add(terminalOuter);

                    terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                    terminalOuter.add(cmdInputPanel);
                    cmdInputPanel.add(correctDeviceInfo);
                    cmdInputPanel.add(wrongDeviceInfo);

                    terminalOuter.add(attributeScrollPanePanel);

                    JScrollPane attributeListScrollPane = new JScrollPane();

                    attributeScrollPanePanel.add(attributeListScrollPane);

                   //table which displays attributes of a selected device
                   tblAttributeList = new JTable();
                   tblAttributeList.setFillsViewportHeight(true);
                   tblAttributeList.setPreferredScrollableViewportSize(new
Dimension(600, 300));
                   attributeListScrollPane.setViewportView(tblAttributeList);
                   tblAttributeList.setColumnSelectionAllowed(true);
                   AttributeListTableModel = new AttributeTableModel();
                   tblAttributeList.setModel(AttributeListTableModel);
                   tblAttributeList.getColumnModel().getColumn
(0).setPreferredWidth(0);
                   tblAttributeList.setRowHeight(40);
                   tblAttributeList.setFont(fontAttribute);
                   tblAttributeList.getColumnModel().getColumn
(1).setPreferredWidth(130);
                   tblAttributeList.getColumnModel().getColumn
(2).setPreferredWidth(60);


                   terminalOuter.setVisible(true);

                   //listener for Connect button
                   btnConnect.addActionListener(new ActionListener() {
                           public void actionPerformed(ActionEvent arg0) {
                                   //method to call after clicking Connect button
                                   btnConnect_onClick();
                           }
                   });

                   //listener for dropdownlist combo box
                dropDownURLList.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                            chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                    }
                  });

                //listener for devices table
                    tblDevices.addMouseListener(new java.awt.event.MouseAdapter()
{
                            public void mouseClicked(java.awt.event.MouseEvent e)
{

                                    int row=tblDevices.rowAtPoint(e.getPoint());
```

```
                                    correctDeviceInfo.setVisible(false);
                                    wrongDeviceInfo.setVisible(false);

                                    highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                    try {
                                            // calls connect_cloud()
                                            indicator =
AttributeListTableModel.connect_cloud(username, password,chosenServer,
highlightedDevice);
                                    if(indicator.compareTo("correct device") !=
0){

                                            waitInfo.setVisible(false);
                                            correctDeviceInfo.setVisible(false);
                                            wrongDeviceInfo.setVisible(true);
                                            tblDevices.removeAll();
                                            tblAttributeList.removeAll();

                                            }
                                    else
                                    {
                                            correctDeviceInfo.setVisible(true);
                                    }
                                    }
                                    catch(Exception e1){
                                            System.out.println(e1);
                                    }
                                    System.out.println(indicator);

                            }
                    });

                    this.pack();

                    this.setVisible(true);

        }
        // this method is called when you click Connect button
        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                wrongCredentialsInfo.setVisible(false);
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        waitInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }
```

```
}
```

IDigiMonitorSample

iDigiMonitorSample sample test

(**For java supported modules**) This application is a simple UI to manage device cloud monitors.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java iDigiMonitorSample test sample application

The iDigiMonitorSample Test sample application can be found here: IDigiMonitorSample.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.monitor;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputSt
ea
;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.zip.InflaterInputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import com.digi.utils.Base64;

public class Main extends UIMain implements MonEvent {
        private static final long serialVersionUID = 1L;

        private ProcessingThread processor;
        private String monitor_ids[];
        private Monitor monitor;

        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
```

```
	}

	public Main() {
		super();

		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				if (processor != null)
					processor.shutdownThread();

				if (monitor != null)
					monitor.kill();

				// wait for threads to shut down.
				try {
					Thread.sleep(1000);
				} catch (InterruptedException e1) {
				}

				System.exit(0);
			}
		});

		btnConnect.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent arg0) {
				btnConnect_OnClick();
			}
		});

		btnCreate.addActionListener(new ActionListener() {

			@Override
			public void actionPerformed(ActionEvent arg0) {
				btnCreate_OnClick();
			}
		});

		btnDelete.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnDelete_OnClick();
			}
		});

		btnStartStop.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnStartStop_OnClick();
			}
		});

		btnUpdate.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				btnUpdate_OnClick();
			}
		});
```

```
        // hide left panel until we have some login info.
        pnlControls.setVisible(false);

        // set up worker threads/objects
        processor = new ProcessingThread();
        processor.start();
        monitor = new Monitor();
        monitor.addListener(this);

}

// some basic logging functions
private void log(String msg) {
        txtLog.append(msg);
        txtLog.setCaretPosition(txtLog.getText().length());
}

private void logln(String msg) {
        log(msg + "\r\n");
}

// checks to see if we have the required information to do some work
private boolean hasUserInfo() {
        String user = txtUsername.getText();
        String pass = new String(txtPassword.getPassword());
        if (user.isEmpty() == true || pass.isEmpty() == true)
                return false;

        return true;
}

// connects to idigi and updates the list
private void btnConnect_OnClick() {
        if (hasUserInfo()) {
                processor.doRefreshMonitorList();

                pnlControls.setVisible(true);
                btnConnect.setVisible(false);
        }
}

// tells the processor to create a new monitor
private void btnCreate_OnClick() {
        if (hasUserInfo())
                processor.doCreateMonitor();
}

// tells the processor to delete the current monitor
private void btnDelete_OnClick() {
        if (hasUserInfo())
                processor.doDeleteMonitor();
}

// toggles the state of the monitor class
private void btnStartStop_OnClick() {
        if (hasUserInfo()) {
                if (monitor.isRunning()) {
                        log("\r\nStopping monitor...");
                        btnStartStop.setText("Start Watching");
```

```
                                monitor.stop();
                                logln("done!");
                        } else {
                                logln("\r\nStarting monitor...");
                                btnStartStop.setText("Stop Watching");
                                monitor.start(cboServer.getSelectedItem
().toString(),
                                                txtUsername.getText(),
                                                new String
(txtPassword.getPassword()),
                                                Integer.parseInt((String)
cboMonitor.getSelectedItem()));


                        }

                }

        }

        // refreshes the monitor list.
        private void btnUpdate_OnClick() {
                if (hasUserInfo())
                        processor.doRefreshMonitorList();
        }

       // Called by Monitor when data is written to the socket.
        @Override
        public void onDataWrite(byte[] raw, int length) {
                logln("\r\nData Written:");
                for (int i = 0; i < length; i++) {
                        log(String.format("%02X", raw[i]) + " ");
                }
                logln("");

        }

        // Called by Monitor when data is read from the socket
        @Override
        public void onDataRead(byte[] raw, int length) {
                logln("\r\nData Read:");

                // displays the raw data in hex.
                for (int i = 0; i < length; i++) {
                        log(String.format("%02X", raw[i]) + " ");
                }
                logln("");

                // 0x03 is the packet type 'PublishMessage'
                // Everything after byte 16 is XML, so pull that data out
                // convert to String, and display.
                StringBuilder xml = new StringBuilder();

                if (raw[1] == 0x3 && raw[10] == 0x01) // if we're using gzip

        // compression, decompress and

        // show to screen
                {
```

```
                        try {
                                InputStream inputstream = new InflaterInputStream
(
                                                new ByteArrayInputStream(raw, 16,
raw.length));

                                byte[] decompressed_buffer = new byte[2000];
                                int len;
                                while (inputstream.available() == 1) {
                                        len = inputstream.read(decompressed_
buffer);

                                        for (int i = 0; i < len; i++)
                                                xml.append(String.format("%c",
decompressed_buffer[i]));
                                }
                                inputstream.close();

                        } catch (Exception ex) {
                                System.out.println(ex);
                        }

                }

                else if (raw[1] == 0x3) {
                        for (int i = 16; i < length; i++) {
                                xml.append(String.format("%c", raw[i]));
                        }
                }

                logln(utils.parseXML(xml.toString()));
        }

        // performs the monitor deletion (called ONLY from the context of the
        // processor thread)
        private void deleteMonitor() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                logln("\r\nDeleting monitor " + (String)
cboMonitor.getSelectedItem()
                               + "...");
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor/" + (String)
cboMonitor.getSelectedItem();
                URL url;
                try {
                        logln("Sending HTTP DELETE to " + path);

                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("DELETE");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
```

```
+ ":" + Password)
                                                                    .getBytes
()));

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }

                        logln("Response:");
                        logln(xml.toString());
                        refreshMonitorList();

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }

        // performs the monitor update (called ONLY from the context of the
        // processor thread)
        private void refreshMonitorList() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                logln("\r\nRefreshing monitor list...");
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor";
                URL url;
                try {
                        logln("Sending HTTP GET to " + path);
                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("GET");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
+ ":" + Password)
```

```
                                                      .getBytes
()));

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {
                                                Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }
                        logln("Response:");
                        logln(xml.toString());
                        parseMonitorList(xml.toString());

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }

        // (called ONLY from the context of the processor thread)
        private void createMonitor() {
                String Username = txtUsername.getText();
                String Password = new String(txtPassword.getPassword());
                String path = "http://" + cboServer.getSelectedItem().toString()
                                + "/ws/Monitor";
                String gzip = new String("none");

                logln("\r\nCreating monitor...");
                URL url;
                try {
                        CreateMonitor cm = new CreateMonitor();
                        cm.setVisible(true);
                        String topics = cm.getTopics();

                        // strip out gzip
                        if (topics.contains("gzip")) {
                                if (topics.contains(",gzip"))
                                        topics = topics.replace(",gzip", "");
                                else
                                        topics = topics.replace("gzip", "");
                                gzip = "gzip";
                        }

                        if (topics.isEmpty()) {
```

```
                                logln("No topics selected... stopped.");
                                return;
                        }
                        logln("Sending HTTP POST to " + path);

                        url = new URL(path);
                        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

                        conn.setDoOutput(true);
                        conn.setRequestMethod("POST");
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", "text/xml");
                        conn.setRequestProperty(
                                        "Authorization",
                                        "Basic "
                                                        + Base64.encode((Username
+ ":" + Password)
                                                                        .getBytes
()));

                        String payload = "\r\n<Monitor>"
                                        + "<monTopic>"
                                        + topics
                                        + "</monTopic>"
                                        +
"<monTransportType>tcp</monTransportType>"
                                        + "<monFormatType>xml</monFormatType>"
                                        + "<monBatchSize>1</monBatchSize>"
                                        + "<monCompression>"
                                        + gzip
                                        + "</monCompression>"
                                        +
"<monBatchDuration>0</monBatchDuration><monStatus>ACTIVE</monStatus>"
                                        + "</Monitor>";

                        logln("HTTP POST Payload:");
                        logln(utils.parseXML(payload));
                        logln("Sending....");

                        long packetsize = payload.length();

                        conn.setRequestProperty("Content-Length", packetsize +
"");

                        OutputStream out = conn.getOutputStream();

                        out.write(payload.getBytes());
                        out.close();

                        InputStream in = new BufferedInputStream
(conn.getInputStream());

                        byte[] buff = new byte[1024];
                        int read;

                        StringBuilder xml = new StringBuilder();

                        logln("Waiting for response...");
```

```
                        while ((read = in.read(buff)) != -1) {
                                if (read > 0) {
                                        xml.append(new String(buff, 0, read));
                                } else {
                                        try {
                                                Thread.sleep(100);
                                        } catch (Exception ex) {
                                        }
                                }
                        }

                        logln("Response:");
                        logln(xml.toString());

                        refreshMonitorList();

                } catch (MalformedURLException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        // parses the XML to pull out the monitor ID's
        private void parseMonitorList(String XML) {
                try {
                        DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
                        DocumentBuilder db = dbf.newDocumentBuilder();

                        // create the XML document.
                        Document dom = db.parse(new ByteArrayInputStream
(XML.getBytes()));
                        Element elem = dom.getDocumentElement();

                        NodeList nl = elem.getElementsByTagName("Monitor");

                        cboMonitor.removeAllItems();

                        if (nl != null) {
                                monitor_ids = new String[nl.getLength()];

                                for (int i = 0; i < monitor_ids.length; i++) {
                                        cboMonitor.addItem
(utils.getTextFromElement(
                                                        ((Element) nl.item(i)),
"monId"));
                                }

                                }


                } catch (Exception e) {
                }
        }

        // This is the thread that does all the data processing.
        // It is done this way so that the UI does not hang while
        // waiting for HTTP transactions to complete, and parsing to execute.
```

```
	private class ProcessingThread extends Thread {
		private int _command;
		private boolean _run;

		public synchronized void shutdownThread() {
			_command = 1;
		}

		public synchronized void doRefreshMonitorList() {
			_command = 2;
		}

		public synchronized void doCreateMonitor() {
			_command = 3;
		}

		public synchronized void doDeleteMonitor() {
			_command = 4;
		}

		@Override
		public void run() {
			super.run();

			_run = true;
			_command = 0;
			while (_run) {
				try {
					switch (_command) {
					case 0:
						Thread.sleep(100);
						break;
					case 1:
						_run = false;
						break;
					case 2:
						refreshMonitorList();
						_command = 0;
						break;
					case 3:
						createMonitor();
						_command = 0;

						break;
					case 4:
						deleteMonitor();
						_command = 0;
						break;
					}
				} catch (Exception ex) {
				}
			}
		}
	}

}
```

IOT Demo TradeShow

IOT_Demo_TradeShow sample test

(**For java supported modules**) This application lists attributes and provides a way to turn the FAN ON and OFF.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java IOT_Demo_TradeShow Test Sample Application

The IOT_Demo_TradeShow Test sample application can be found here: IOT_Demo_TradeShow.zip.

Basic usage

Compile, load and run program using java environment.

Sample of Main.java file:

```
package com.digi.etherios;

import javax.swing.BoxLayout;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;import javax.swing.JTable;
import javax.swing.ScrollPaneConstants;
import javax.swing.UIManager;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.FlowLayout;
import javax.swing.JScrollPane;
import javax.swing.event.TableModelEvent;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

import com.digi.etherios.DataTableModel;
import com.digi.etherios.DeviceListTableModel;
```

```
import java.awt.SystemColor;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Scanner;

public class Main extends JFrame {

       /**
        *
        */
        private static final long serialVersionUID = 1L;

        private JTextField txtUsername;
        private JTextField txtPassword;
        private JTextField txtRefreshInterval;
        private JTable tblDevices;
        private JTable tblAttributeList;

        private DeviceListTableModel devicesTableModel;
        private DataTableModel dataTableModel;

        private String username;
        private String password;
        private String interval;
        private String highlightedDevice;
        public String indicator = null;
        public String login_indicator = null;
        final JLabel wrongCredentialsInfo = new JLabel("You have entered wrong
credentials!");
        private static final Color cl_black = new Color(21, 45, 60);
        private static final Color cl_btn_grn = new Color(10, 148, 54);
        private static final Color cl_dkgray = new Color(110, 110, 110);
        private static final Color cl_ltgray = new Color(153, 153, 153);
        private static final Color cl_white = new Color(255, 255, 255);
        final JLabel wrongDeviceInfo;
        JPanel devicesScrollPanePanel = new JPanel();
        JPanel cmdOutputScrollPanePanel = new JPanel();
        JPanel motherPanel = new JPanel();

        JPanel basePanel = new JPanel();
        JPanel extraPanel = new JPanel();

        JPanel terminalOuter = new JPanel();
        JPanel cmdInputPanel = new JPanel();
        public static String result = null;
        public static String res = null;
        private static String[][] data = null;
        private static int rowCount = 0;
        public Font font = new Font("Times New Roman", Font.PLAIN, 15);
        public Font font_bold = new Font("Times New Roman", Font.BOLD, 20);
        public final JLabel waitInfo = new JLabel("Please select device on which
IOT_demo program is running and wait for few seconds!!");
        boolean clickedonce = false;
```

```
        private static String[] ValueList = null;

        private String chosenServer = "login.etherios.com";
        String[] listURLItems = {"login.etherios.com", "login.etherios.co.uk"};


        public static void main(String[] args) {
                Main m = new Main();
                m.setVisible(true);
        }

        public Main() {
                setBackground(SystemColor.control);
                setResizable(false);
                try {
                        UIManager.setLookAndFeel
(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception ex) {
                }

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                this.setSize(800, 750);

                setTitle("Digi Connect Tank");
                //getContentPane().setLayout(new GridLayout(0, 1, 0, 0));
                getContentPane().add(motherPanel);
                motherPanel.setLayout(new BoxLayout(motherPanel,BoxLayout.Y_
AXIS));

                motherPanel.add(basePanel);
                basePanel.setBackground(SystemColor.control);
                basePanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JPanel userPanel = new JPanel();
                basePanel.add(userPanel);
                userPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

                JLabel userLabel = new JLabel("Username");
                userLabel.setForeground(cl_black);
                userLabel.setBackground(cl_dkgray);
                userLabel.setSize(60,25);
                userPanel.add(userLabel);

                txtUsername = new JTextField();
                userPanel.add(txtUsername);
                txtUsername.setColumns(10);


                JPanel passPanel = new JPanel();
                basePanel.add(passPanel);

                JLabel passLabel = new JLabel("Password");
                passPanel.add(passLabel);
```

```
                txtPassword = new JPasswordField();
                passPanel.add(txtPassword);
                txtPassword.setColumns(10);
               JPanel dropListConnectPanel = new JPanel();
                basePanel.add(dropListConnectPanel);
                JPanel intervalPanel = new JPanel();
                basePanel.add(intervalPanel);

                final JLabel correctDeviceInfo = new JLabel("Latest values of
selected device in Device Cloud and will be refreshed based on interval!!");
                correctDeviceInfo.setSize(4, 4);
                correctDeviceInfo.setVisible(false);
                correctDeviceInfo.setFont(font_bold);
                correctDeviceInfo.setForeground(Color.BLACK);

                wrongDeviceInfo = new JLabel("You have selected wrong device!
Please select device on which IOT_demo program is running");
                wrongDeviceInfo.setVisible(false);
                wrongDeviceInfo.setSize(5, 5);
                wrongDeviceInfo.setFont(font_bold);
                wrongDeviceInfo.setForeground(Color.BLACK);


                waitInfo.setVisible(false);
                waitInfo.setSize(7,7);
                waitInfo.setFont(font_bold);
                waitInfo.setForeground(Color.BLACK);

                wrongCredentialsInfo.setVisible(false);
                wrongCredentialsInfo.setSize(5, 5);
                wrongCredentialsInfo.setForeground(Color.black);

                JComboBox dropDownURLList = new JComboBox();
                dropDownURLList.addItem(listURLItems[0]);
            dropDownURLList.addItem(listURLItems[1]);
            dropListConnectPanel.add(dropDownURLList);
            dropDownURLList.setBackground(cl_dkgray);
            dropDownURLList.setForeground(cl_black);
            dropDownURLList.setSize(50,25);

            JLabel intervalLabel = new JLabel("  Interval in min");
                intervalLabel.setForeground(cl_black);
                intervalLabel.setBackground(cl_dkgray);
                intervalLabel.setSize(60,25);
                dropListConnectPanel.add(intervalLabel);

                txtRefreshInterval = new JTextField();
                dropListConnectPanel.add(txtRefreshInterval);
                txtRefreshInterval.setColumns(5);

                JButton btnConnect = new JButton("Connect");
                dropListConnectPanel.add(btnConnect);

                Font f = new Font("Arial", Font.BOLD, 13);
                btnConnect.setOpaque(true);
                btnConnect.setBackground(cl_dkgray);
                btnConnect.setForeground(cl_black);
                btnConnect.setSize(new Dimension(50, 25));
```

```
                btnConnect.setFont(f);

                motherPanel.add(devicesScrollPanePanel);
                devicesScrollPanePanel.setLayout(new FlowLayout
(FlowLayout.CENTER, 5, 5));

                devicesScrollPanePanel.add(wrongCredentialsInfo);
                devicesScrollPanePanel.add(waitInfo);
                JScrollPane devicesScrollPane = new JScrollPane();
                devicesScrollPanePanel.add(devicesScrollPane);


                tblDevices = new JTable();
                tblDevices.setFillsViewportHeight(true);

                tblDevices.setPreferredScrollableViewportSize(new Dimension(800,
150));

                devicesScrollPane.setViewportView(tblDevices);
                tblDevices.setColumnSelectionAllowed(true);
                devicesTableModel = new DeviceListTableModel();
                tblDevices.setModel(devicesTableModel);
                tblDevices.setRowHeight(22);
                tblDevices.getColumnModel().getColumn(0).setPreferredWidth(140);
                tblDevices.getColumnModel().getColumn(1).setPreferredWidth(300);
                tblDevices.getColumnModel().getColumn(2).setPreferredWidth(125);
                tblDevices.getColumnModel().getColumn(3).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(4).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(5).setPreferredWidth(124);
                tblDevices.getColumnModel().getColumn(6).setPreferredWidth(124);
                motherPanel.add(terminalOuter);

                terminalOuter.setLayout((new BoxLayout(terminalOuter,
BoxLayout.PAGE_AXIS)));
                terminalOuter.add(cmdInputPanel);
                cmdInputPanel.add(correctDeviceInfo);
                cmdInputPanel.add(wrongDeviceInfo);
                terminalOuter.add(cmdOutputScrollPanePanel);

                JScrollPane cmdOutputsListScrollPane = new JScrollPane();

                cmdOutputScrollPanePanel.add(cmdOutputsListScrollPane);

                tblAttributeList = new JTable();
                tblAttributeList.setFillsViewportHeight(true);

                tblAttributeList.setPreferredScrollableViewportSize(new Dimension
(600, 350));


                cmdOutputsListScrollPane.setViewportView(tblAttributeList);
                tblAttributeList.setColumnSelectionAllowed(true);


                dataTableModel = new DataTableModel();
                tblAttributeList.setModel(dataTableModel);
                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(0);
                tblAttributeList.setRowHeight(40);
```

```
                tblAttributeList.setFont(font);

                tblAttributeList.getColumnModel().getColumn(0).setPreferredWidth
(60);
                tblAttributeList.getColumnModel().getColumn(1).setPreferredWidth
(130);
                tblAttributeList.getColumnModel().getColumn(2).setPreferredWidth
(60);


                terminalOuter.setVisible(true);

                btnConnect.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                                btnConnect_onClick();
                        }
                });


            dropDownURLList.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        chosenServer = new String( (String) ((JComboBox)
e.getSource()).getSelectedItem() );
                }
            });

                tblDevices.addMouseListener(new java.awt.event.MouseAdapter(){
                        public void mouseClicked(java.awt.event.MouseEvent e){


                                if (clickedonce  == false){
                                        clickedonce = true;

                                        int row=tblDevices.rowAtPoint(e.getPoint
());
                                        correctDeviceInfo.setVisible(false);
                                        wrongDeviceInfo.setVisible(false);


                                        highlightedDevice = tblDevices.getValueAt
(row,1).toString();
                                        try {
                                                indicator =
dataTableModel.connect_cloud(username, password, chosenServer,
highlightedDevice);
                                                BasicThread refreshThread = new
BasicThread(username, password, chosenServer, highlightedDevice,
interval,dataTableModel);
                                                refreshThread.start();

                                        if(indicator.compareTo("correct device")
!= 0){
                                                System.out.println("indicator :
"+indicator);
                                                correctDeviceInfo.setVisible
(false);
                                                wrongDeviceInfo.setVisible(true);
                                                tblDevices.removeAll();
                                                tblAttributeList.removeAll();
```

```
                                                }
                                        else
                                        {
                                                correctDeviceInfo.setVisible
(true);

                                        }
                                }
                                        catch(Exception e1){
                                                System.out.println(e1);
                                        }
                                        System.out.println(indicator);

                                }
                                else {
                                        System.out.println("Clicked already");
                                }
                        }


                });
                this.pack();

                this.setVisible(true);
        }

        public void btnConnect_onClick() {
                username = txtUsername.getText();
                password = txtPassword.getText();
                interval = txtRefreshInterval.getText();
                login_indicator = devicesTableModel.update(username,
password,chosenServer);

                if(login_indicator.compareTo("correct credentials") == 0){
                        wrongCredentialsInfo.setVisible(false);
                        waitInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                }

                else if(login_indicator.compareTo("bad credentials") == 0){
                        wrongCredentialsInfo.setVisible(true);
                        wrongDeviceInfo.setVisible(false);
                        tblDevices.removeAll();
                        tblAttributeList.removeAll();
                }
        }

}
```

LibDeviceCloud

LibDeviceCloud sample test

(**For java supported modules**) This program is a library for communicating to the device cloud in java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibDeviceCloud Test Sample Application

The LibDeviceCloud Test sample application can be found here: LibDeviceCloud.zip.

Basic usage

Compile, load and run program using java environment.

Sample of MonitorTest.java file:

```
package com.digi.devicecloud.test;

import java.io.IOException;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

import com.digi.devicecloud.DeviceCloud;
import com.digi.devicecloud.MonitorConfig;
import com.digi.devicecloud.Result;
import com.digi.devicecloud.WsMonitor;
import com.digi.devicecloud.monitor.MonConnectionRequestPacket;
import com.digi.devicecloud.monitor.MonConnectionResponsePacket;
import com.digi.devicecloud.monitor.MonPacket;
import com.digi.devicecloud.monitor.MonPublishMessagePacket;
import com.digi.devicecloud.monitor.TcpMonitor;
import com.digi.devicecloud.monitor.TcpMonitorListener;
import com.digi.json.JsonArray;
import com.digi.json.JsonObject;

public class MonitorTest implements TcpMonitorListener {
        private static String username = "";
        private static String password = "";
        private static String hostname = "login.etherios.com";
        private static final int MONITOR_ID = 109537;

        private DeviceCloud cloud = new DeviceCloud(hostname, username,
password);


        public static void main(String[] args) {
                MonitorTest test = new MonitorTest();

                try {
                        test.listenMonitor();
```

```
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

        while (true) {
                try {
                        Thread.sleep(10000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

}


private void start() {
        try {

                WsMonitor monitorWs = new WsMonitor(cloud);

                List<Result> results = monitorWs.list();

                for (int i = 0; i < results.size(); i++) {
                        Result result = results.get(i);

                        JsonArray items = result.getItems();

                        for (int j = 0; j < items.size(); j++) {
                                JsonObject obj = items.getJsonObject(j);

                                System.out.println(obj.display());
                        }
                }

                // create a monitor
                // createMonitor();
                // listen to monitor
                // listenMonitor();

                // while (true) {
                // Thread.sleep(1000);
                // }

        } catch (Exception ex) {
                ex.printStackTrace();
        }
}


private void createMonitor() throws IOException, ParseException {
        WsMonitor monitorWs = new WsMonitor(cloud);

        MonitorConfig config = new MonitorConfig();

        // using compresion
        config.setCompression(MonitorConfig.COMPRESSION_ZLIB);
        // using JSON
        config.setFormatType(MonitorConfig.FORMAT_JSON);
```

```
                // listening to Device Core changes
                config.setTopic(MonitorConfig.TOPIC_DEVICE_CORE);
                // using TCP/IP
                config.setTransportType(MonitorConfig.TRANSPORT_TYPE_TCP);

                // create the monitor
                Result result = monitorWs.create(config);

                // print the monitor ID
                System.out.println(result.getDataAsString());
        }


        private void listenMonitor() throws InterruptedException {
                // create a new monitor with a high timeout
                TcpMonitor monitor = new TcpMonitor(900000);

                // add myself as a listener
                monitor.addListener(this);

                // start the monitor
                monitor.start("login.etherios.com", 3200, false);

                // create the request packet
                MonConnectionRequestPacket request = new
MonConnectionRequestPacket(
                username, password, MONITOR_ID);

                // send the packet
                monitor.sendPacket(request);

        }


        @Override
        public void tcpMonitorIncommingPacket(TcpMonitor tcpMonitor, MonPacket
packet) {
                System.out.println("Recieved: " + packet.getType());

                if (packet.getType() == MonPacket.TYPE_PUBLISH_MESSAGE) {
                        MonPublishMessagePacket publish =
(MonPublishMessagePacket) packet;

                        try {
                                System.out.println(new JsonObject(new String(
                                                publish.getPayload())).display
());
                        } catch (ParseException e) {
                                e.printStackTrace();
                        }
                }

                if (packet.getType() == MonPacket.TYPE_CONNECTION_RESPONSE) {
                        MonConnectionResponsePacket response =
(MonConnectionResponsePacket) packet;

                        System.out.println("Connection Response: " +
response.getStatus());
                }
```

```
        }


        @Override
        public void tcpMonitorConnected(TcpMonitor tcpMonitor) {
                // TODO Auto-generated method stub

        }
}
```

LibFastDb

LibFastDb sample test

(**For java supported modules**) This program is an interface for easily accessing MySQL from java.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibFastDb Test Sample Application

The LibFastDb Test sample application can be found here: LibFastDb.zip.

Basic usage

```
Compile, load and run program using java environment.
```

Sample of FastDb.java file:

```
package com.digi.fastdb;

import java.lang.reflect.Field;
import java.sql.PreparedStatement;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.digi.fastdb.utils.utils;

public class FastDb {
        private static FastDb _instance = null;

        private Set<String> _tables = new HashSet<String>();


        public static FastDb getInstance() {
                if (_instance == null) {
                        _instance = new FastDb();
                }
                return _instance;
        }


        private FastDb() {

        }


        public boolean objectExists(Object object) throws Exception {
                checkTables(object.getClass());
                return utils.objectExists(object);
        }


        public void writeNewObject(Object object) throws Exception {
```

```
                checkTables(object.getClass());
                utils.writeNewObject(object);
        }


       /**
        * Attempts to push an update to the store based on the object's primary
key
        *
        * @param object
        * @throws Exception
        */
        public void pushObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.pushObject(object);
        }

       /**
        * If the object exists.. then the object is updated. Otherwise a new one
is
        * created.
        *
        * @param object
        * @throws Exception
        */
        public void saveObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.saveObject(object);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement) throws Exception {
                checkTables(klass);
                return getObjects(klass, statement, true);
        }


        public <T> List<T> getObjects(Class<T> klass, PreparedStatement
statement, boolean includeChildren) throws Exception {
                checkTables(klass);
                return utils.getObjects(klass, statement, includeChildren);
        }


        public void deleteObject(Object object) throws Exception {
                checkTables(object.getClass());
                utils.deleteObject(object);
        }


        public void removeLink(Object parent, String fieldName, Object
fieldValue) throws Exception {
                checkTables(parent.getClass());

                Field field = parent.getClass().getDeclaredField(fieldName);

                utils.deleteLink(parent, field, fieldValue);
        }
```

```
	private void checkTables(Class<?> klass) throws Exception {
		String tableName = utils.getTableName(klass);

		if (_tables.contains(tableName)) {
			return;
		}

		if (utils.tableExists(tableName)) {
			_tables.add(tableName);
			return;
		}

		List<String> schemas = utils.generateTableSchema(klass);
		for (int i = 0; i < schemas.size(); i++) {
			utils.createTable(schemas.get(i));
		}


	}
}
```

LibJil

LibJil sample test

(**For java supported modules**) This program is an interpreter for the JIL language.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJil Test Sample Application

The LibJil Test sample application can be found here: LibJil.zip.

Basic usage

Compile, load and run program using java environment.

Sample of JilProgram.java file:

```
package com.digi.jil;

import java.util.Hashtable;

import com.digi.jil.lang.JilStatement;
import com.digi.json.JsonObject;

public class JilProgram {
        private JilStatement _rootNode;
        private Hashtable<String, JilStatement> _labels;


        public JilProgram(JsonObject source) throws Exception {
                _labels = new Hashtable<String, JilStatement>();
                _rootNode = JilStatementFactory.newStatement(this, source);
                _rootNode.setOid("r");
        }


        public void step(JilContext jilContext) throws Exception {
                String oid = jilContext.getStatementOid();

                if (oid == null) {
                        jilContext.setStatementOid("r");
                        oid = "r";
                }

                JilStatement statement = find(oid);

                JilResult result = statement.execute(jilContext);

                switch (result.state) {
                // there was a horrible error
                        case JilResult.RESULT_ERROR:
                                handleError(result, jilContext);
                                break;
                        case JilResult.RESULT_GOTO:
```

```
                                handleGoto(result, jilContext);
                                break;
                        case JilResult.RESULT_NEXT_SIBLING:
                                handleNextSibling(result, jilContext);
                                break;
                        case JilResult.RESULT_COMPLETE:
                                handleComplete(result, jilContext);
                                break;
                }
        }

        public void registerLabel(JilStatement label) {
                _labels.put(label.getLabelName(), label);
        }


        public JilStatement lookupLabel(String labelName) {
                if (_labels.containsKey(labelName)) {
                        return _labels.get(labelName);
                }

                return null;
        }


        private void handleComplete(JilResult result, JilContext jilContext)
throws Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_COMPLETE);
        }


        private void handleError(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_FAILED);
        }


        private void handleGoto(JilResult result, JilContext jilContext) throws
Exception {
                jilContext.setStatementOid(result.nextOid);
                jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
                // TODO, set other stuff !?
        }


        private void handleNextSibling(JilResult result, JilContext jilContext)
throws Exception {
                // find next command
                String oid = result.nextOid;

                if (oid == null || "r".equals(oid)) {
                        // we are done!
                        jilContext.setStatementOid(oid);
                        jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_
COMPLETE);
                        return;
                }

                oid = JilUtils.nextSibling(oid);
```

```
		JilStatement statement = find(oid);
		while (statement == null) {
			oid = JilUtils.parseParent(oid);

			if (oid == null || "r".equals(oid)) {
				// we are done!
				jilContext.setStatementOid(oid);
				jilContext.setExecutionStatus(JilStatement.EXEC_
STATUS_COMPLETE);
				return;
			}

			oid = JilUtils.nextSibling(oid);

			statement = find(oid);
		}

		jilContext.setStatementOid(oid);
		jilContext.setExecutionStatus(JilStatement.EXEC_STATUS_IDLE);
	}


	public JilStatement find(String oid) {

		return _rootNode.find(oid);
	}

}
```

LibJson

LibJson sample test

(**For java supported modules**) This program is a JSON parser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibJson Test Sample Application

The libJson Test sample application can be found here: LibJson.zip

Basic usage

Compile, load and run program using java environment.

Sample of JsonArray.java file:

```
package com.digi.json;

import java.text.ParseException;
import java.util.LinkedList;
import java.util.List;

public class JsonArray {
        private List<Object> _objects = new LinkedList<Object>();


        public JsonArray() {

        }


        public JsonArray(String string) throws ParseException {
                JsonTokenizer tokenizer = new JsonTokenizer(string);
                fromTokenizer(tokenizer);

        }


        public JsonArray(JsonTokenizer tokenizer) throws ParseException {
                fromTokenizer(tokenizer);
        }


        private void fromTokenizer(JsonTokenizer tokenizer) throws ParseException
{
                if (!tokenizer.isNextTokenStartArray()) {
                        throw new ParseException("Not an array!", -1);
                }
                // pop the open brace
                tokenizer.nextToken();

                // check for empty array
                if (tokenizer.isNextTokenFinishArray()) {
```

```
                tokenizer.nextToken();
                return;
        }

        while (true) {
                _objects.add(tokenizer.parseValue());

                // if not comma, then it should've been a ']'
                if (!tokenizer.nextToken().equals(",")) {
                        break;
                }
        }

}


public int size() {
        return _objects.size();
}


public Object get(int index) {
        return _objects.get(index);
}


public Object get(String path) {
        List<String> dir = JsonTokenizer.parsePath(path);

        return get(dir);
}


protected Object get(List<String> directions) {
        if (directions.size() == 0)
                return this;

        String value = directions.remove(0);

        value = value.replace("[", "");
        value = value.replace("]", "");

        int item = Integer.parseInt(value);

        Object obj = _objects.get(item);
        if (obj instanceof JsonObject) {
                return ((JsonObject) obj).get(directions);
        }
        else if (obj instanceof JsonArray) {
                return ((JsonArray) obj).get(directions);
        }

        return obj;
}

public String getString(int index) {
        return (String) get(index);
}
```

```
        public JsonObject getJsonObject(int index) {
                return (JsonObject) get(index);
        }


        public JsonArray getJsonArray(int index) {
                return (JsonArray) get(index);
        }


        public int getInt(int index) {
                return Integer.parseInt(getString(index));
        }


        public long getLong(int index) {
                return Long.parseLong(getString(index));
        }


        public float getFloat(int index) {
                return Float.parseFloat(getString(index));
        }


        public double getDouble(int index) {
                return Double.parseDouble(getString(index));
        }


        public Object set(int index, Object value) {
                while (index >= _objects.size())
                        _objects.add(null);

                return _objects.set(index, value);
        }


        protected void set(List<String> directions, Object value) throws
ParseException {
                if (directions.size() == 0)
                        throw new ParseException("Invalid path", 0);

                String item = directions.remove(0);

                item = item.replaceAll("\\[", "");
                item = item.replaceAll("\\]", "");

                int index = Integer.parseInt(item);

                if (directions.size() == 0) {
                        set(index, value);
                }
                else {
                        // already exists
                        if (_objects.size() > index) {
                                Object obj = _objects.get(index);
```

```
                                if (obj instanceof JsonObject) {
                                        JsonObject jo = (JsonObject) obj;

                                        jo.put(directions, value);
                                        return;
                                }
                                else if (obj instanceof JsonArray) {
                                        JsonArray ja = (JsonArray) obj;
                                        String child = directions.get(0);
                                        if (child.equals("[]"))
                                                ja.add(directions, value);
                                        else
                                                ja.set(directions, value);
                                        return;
                                }
                        }
                        else {

                                String child = directions.get(0);
                                if (child.startsWith("[") && child.endsWith("]"))
{
                                        JsonArray ja = new JsonArray();

                                        set(index, ja);

                                        if (child.equals("[]"))
                                                ja.add(directions, value);
                                        else
                                                ja.set(directions, value);

                                        return;
                                }
                                else {
                                        JsonObject jo = new JsonObject();

                                        set(index, jo);

                                        jo.put(directions, value);
                                        return;
                                }
                        }

                }

        }


        public boolean add(Object value) {
                return _objects.add(value);
        }


        public void add(int index, Object value) {
                _objects.add(index, value);
        }


        protected void add(List<String> directions, Object value) throws
ParseException {
```

```
        if (directions.size() == 0)
                throw new ParseException("Invalid path", 0);

        String item = directions.remove(0);

        if (!item.equals("[]"))
                throw new ParseException("This should never happen!", 0);

        if (directions.size() == 0) {
                add(value);
        }
        else {
                String child = directions.get(0);
                if (child.startsWith("[") && child.endsWith("]")) {
                        JsonArray ja = new JsonArray();

                        _objects.add(ja);

                        if (child.equals("[]"))
                                ja.add(directions, value);
                        else
                                ja.set(directions, value);

                        return;
                }
                else {
                        JsonObject jo = new JsonObject();

                        _objects.add(jo);

                        jo.put(directions, value);
                        return;

                }

        }

}


public Object remove(int index) {
        return _objects.remove(index);
}


protected Object remove(List<String> directions) throws ParseException {
        if (directions.size() == 0)
                throw new ParseException("Invalid path!", -1);
        String item = directions.remove(0);

        item = item.replaceAll("\\[", "");
        item = item.replaceAll("\\]", "");

        int index = Integer.parseInt(item);

        if (directions.size() == 0) {
                return remove(index);
        }
        else {
```

```
                        Object obj = get(index);

                        if (obj instanceof JsonObject) {
                                JsonObject jo = (JsonObject) obj;

                                return jo.remove(directions);
                        }
                        else if (obj instanceof JsonArray) {
                                JsonArray jo = (JsonArray) obj;

                                return jo.remove(directions);
                        }

                }
                return null;

        }


        public void clear() {
                _objects.clear();
        }


        public String display() {
                return display(0).toString();
        }


        protected StringBuilder display(int depth) {
                StringBuilder sb = new StringBuilder();

                sb.append("[\n");
                for (int i = 0; i < _objects.size(); i++) {
                        Object obj = _objects.get(i);

                        sb.append(JsonTokenizer.repeat("  ", depth + 1));
                        if (obj == null) {
                                sb.append("null");
                        }
                        else if (obj instanceof JsonObject) {
                                sb.append(((JsonObject) obj).display(depth + 1));
                        }
                        else if (obj instanceof JsonArray) {
                                sb.append(((JsonArray) obj).display(depth + 1));
                        }
                        else if (obj instanceof String) {
                                sb.append(JsonTokenizer.escapeString((String)
obj));
                        }
                        else {
                                sb.append(obj + "");
                        }

                        if (i < _objects.size() - 1) {
                                sb.append(",");
                        }

                        sb.append("\n");
```

```
                }
                sb.append(JsonTokenizer.repeat("  ", depth));
                sb.append("]");

                return sb;
        }


        public StringBuilder toStringBuilder() {
                StringBuilder sb = new StringBuilder();

                sb.append("[");
                for (int i = 0; i < _objects.size(); i++) {
                        Object obj = _objects.get(i);

                        if (obj == null) {
                                sb.append("null");
                        }
                        else if (obj instanceof JsonObject) {
                                sb.append(((JsonObject) obj).toStringBuilder());
                        }
                        else if (obj instanceof JsonArray) {
                                sb.append(((JsonArray) obj).toStringBuilder());
                        }
                        else if (obj instanceof String) {
                                sb.append(JsonTokenizer.escapeString((String)
obj));
                        }
                        else {
                                sb.append(obj + "");
                        }
                        if (i < _objects.size() - 1) {
                                sb.append(",");
                        }
                }
                sb.append("]");

                return sb;
        }

        public boolean has(int index) {
                if (index < _objects.size())
                        return true;

                return false;
        }


        protected boolean has(List<String> directions) throws ParseException {
                if (directions.size() == 0)
                        throw new ParseException("Invalid path!", -1);

                String item = directions.remove(0);

                item = item.replaceAll("\\[", "");
                item = item.replaceAll("\\]", "");

                int index = Integer.parseInt(item);
```

```
                if (!has(index))
                        return false;
                else if (directions.size() == 0)
                        return true;

                Object obj = get(index);

                if (obj instanceof JsonObject) {
                        JsonObject jo = (JsonObject) obj;

                        return jo.has(directions);
                }
                else if (obj instanceof JsonArray) {
                        JsonArray ja = (JsonArray) obj;

                        return ja.has(directions);
                }

                return false;
        }


        @Override
        public String toString() {
                return toStringBuilder().toString();

        }

}
```

LibUtils

LibUtils sample test

(**For java supported modules**) This program is a collection of utility classes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibUtils Test Sample Application

The libUtils Test sample application can be found here: LibUtils.zip.

Basic usage

Compile, load and run program using java environment.

Sample of ConfigFile.java file:

```
package com.digi.utils;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;

/**
* Loads a simple config file into memory. Config files are of the following
* form
*
* ### Comments start with a #
* key = value
*
* @author mcarver
*
*/
public class ConfigFile {
        private static Hashtable<String, ConfigFile> _configFiles = new
Hashtable<String, ConfigFile>();

        private Hashtable<String, String> _parameters;
        private String _filename;


       /**
        * Loads the config file at filename and caches the data. If the config
file
        * is already loaded, then the cache is returned.
        *
        * @param filename
        * @return
        * @throws IOException
        */
        public static ConfigFile getInstance(String filename) throws IOException
```

```
{
                if (_configFiles.containsKey(filename)) {
                        return _configFiles.get(filename);
                }

                ConfigFile config = new ConfigFile(filename);

                _configFiles.put(filename, config);

                return config;
        }


        private ConfigFile(String file) throws IOException {
                _parameters = new Hashtable<String, String>();
                _filename = file;

                loadFile();
        }


        private void loadFile() throws IOException {

                File file = new File(_filename);

                BufferedReader stream = new BufferedReader(new InputStreamReader(
                                new FileInputStream(file)));

                String line = null;
                while ((line = stream.readLine()) != null) {

                        line = line.trim();

                        if (line.startsWith("#")) {
                                continue;
                        }

                        if (!line.contains("=")) {
                                continue;
                        }

                        String name = line.substring(0, line.indexOf("=")).trim
().toLowerCase();
                        String value = line.substring(line.indexOf("=") + 1).trim
();

                        _parameters.put(name, value);
                }

                stream.close();

        }


        /**
         * Returns the value of the parameter named name. Names are case
         * insensitive.
         *
         * @param name
```

```
        * @return
        */
        public String getString(String name) {
                if (!_parameters.containsKey(name.toLowerCase())) {
                        return null;
                }

                return _parameters.get(name.toLowerCase());
        }
}
```

LibZkConfigProtocol

LibZkConfigProtocol sample test

(**For java supported modules**) This program is a protocol library ZooKeeper (zk.digi.com) which uses to communicate with its slave nodes.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java LibZkConfigProtocol Test Sample Application

The libZkConfigProtocol Test sample application can be found here: LibZkConfigProtocol.zip.

Basic usage

Compile, load and run program using java environment.

Sample of FieldField.java file:

```
package com.digi.configurepackets;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class FileField extends Field {
        public StringField fileName;
        public File file;


        FileField(DataInputStream dataIn) throws IOException {
                fileName = new StringField(dataIn);
                int fileLength = dataIn.readInt();
                file = File.createTempFile("FileField_", ".tmp");

                OutputStream out = new BufferedOutputStream(new FileOutputStream
(file));
                try {
                        int pos = 0;
                        int read = 0;
                        int packet_size = 2048;
                        if (packet_size > fileLength) {
                                packet_size = fileLength;
                        }

                        byte[] data = new byte[packet_size];
                        while (pos < fileLength) {
                                read = dataIn.read(data);
```

```
                                if (read > 0) {
                                        out.write(data, 0, read);
                                        pos += read;
                                        if (fileLength - pos < packet_size) {
                                                packet_size = fileLength - pos;
                                                data = new byte[packet_size];
                                        }
                                }
                                else if (read == 0) {
                                        try {
                                                Thread.sleep(100);
                                        } catch (InterruptedException e) {}
                                }
                                else if (read < 0) {
                                        throw new IOException(
                                                        "Could not read entire file from
stream!");
                                        }
                                }

                                if (pos != fileLength) {
                                        throw new IOException(
                                                        "There was an error reading the
file from the data stream!");
                                }
                        } catch (IOException ex) {
                                out.close();
                                file.delete();
                                out = null;
                        } finally {
                                if (out != null) {
                                        out.close();
                                }
                        }

        }


        public FileField(File file) {
                fileName = new StringField(file.getName());
                this.file = file;
        }


        public FileField(String name, File file) {
                fileName = new StringField(name);
                this.file = file;
        }


        public void delete() {
                file.delete();
        }


        @Override
        public int length() {
                return fileName.length() + 4 + (int) file.length();
        }
```

```
		@Override
	public void toDataStream(DataOutputStream dataOut) throws IOException {
		fileName.toDataStream(dataOut);
		dataOut.writeInt((int) file.length());

		InputStream in = new BufferedInputStream(new FileInputStream
(file));

		try {
			int read = 0;
			byte[] packet = new byte[2048];
			while ((read = in.read(packet)) != -1) {
				if (read > 0) {
					dataOut.write(packet, 0, read);
				}
				else if (read == 0) {
					try {
						Thread.sleep(100);
					} catch (InterruptedException e) {}
				}
			}
		} finally {
			in.close();
		}
	}


}
```

WSBrowser

WSBrowser sample test

(**For java supported modules**) This program is a webservice browser.

Test files

This sample program contains several files and the source files can be found under the /src directory.

Java WSBrowser Test Sample Application

The WSBrowser Test sample application can be found here: WSBrowser.zip.

Basic usage

Compile, load and run program using java environment.

Sample of WSB.java file:

```
package com.digi.wsb;

import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.text.ParseException;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import com.digi.json.JsonObject;
import com.digi.utils.Base64;
import com.digi.utils.misc;

public class WSB extends WsbUi {
        private static final long serialVersionUID = 1L;


       /**
        * @param args
        */
        public static void main(String[] args) {
                new WSB().setVisible(true);
        }


        public WSB() {
                super();

                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
```

```
                                System.exit(0);
                        }
                });
        }


        @Override
        public void goButton_onClick(ActionEvent e) {
                try {
                        String method = ((String) methodComboBox.getSelectedItem
()).toUpperCase();
                        String url = urlTextField.getText();
                        String username = usernameTextField.getText();
                        String password = new String
(passwordTextField.getPassword());
                        String mime = (String) mimeContentType.getSelectedItem();
                        String payload = postDataTextArea.getText();

                        if (username.equals("")) {
                                username = null;
                        }

                        if ("GET".equals(method) || "DELETE".equals(method)) {
                                getData(method, url, username, password);
                        }
                        else if ("POST".equals(method) || "PUT".equals(method)) {
                                pushData(method, url, payload, mime, username,
password);
                        }
                } catch (Exception ex) {
                        log(ex);
                }

        }


        public void log(String logging) {
                try {
                        JsonObject jobj = new JsonObject(logging);
                        logTextArea.append(jobj.display() + "\n");
                } catch (Exception e) {
                        logTextArea.append(logging + "\n");
                }

                logTextArea.setCaretPosition(logTextArea.getText().length());
        }


        public void log(Exception e) {
                log(misc.getStackTrace(e));
        }


        public WebResult getData(String method, String path, String username,
String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
```

```
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");

                        // configure the authorization
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);

                        // get the steam from the server
                        InputStream str = conn.getInputStream();
                        int size = conn.getContentLength();
                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public WebResult pushData(String method, String path, String payload,
String contentType, String username, String password) {
                URL url = null;
                HttpURLConnection conn = null;
                try {
                        url = new URL(path);
                        conn = (HttpURLConnection) url.openConnection();
                        conn.setRequestMethod(method);
                        conn.setRequestProperty("User-Agent", "Internet Access");
                        conn.setRequestProperty("Content-Type", contentType);
                        conn.setRequestProperty("Content-Length", payload.length
() + "");
                        conn.setDoOutput(true);

                        // configure the authorization
```

```
                        if (username != null && password != null) {
                                conn.setRequestProperty("Authorization", "Basic "
                                                + Base64.encode((username + ":" +
password).getBytes()));
                        }

                        log("*******************************************");
                        log("                Request");
                        log("*******************************************");
                        log(path);
                        dumpConnectionRequest(conn);
                        log(payload);

                        // write the data
                        OutputStream out = conn.getOutputStream();
                        out.write(payload.getBytes());
                        out.close();

                        // get the steam from the server
                        InputStream str = conn.getInputStream();

                        int size = conn.getContentLength();

                        byte[] rawData = misc.readAllFromStream(str, 1024, size,
3000);

                        log("*******************************************");
                        log("                Response");
                        log("*******************************************");
                        dumpConnectionResponse(conn);
                        log(new String(rawData));

                        return new WebResult(conn.getResponseCode(), rawData);

                } catch (MalformedURLException e) {
                        log(e);
                } catch (IOException e) {
                        log(e);
                } finally {
                        if (conn != null)
                                conn.disconnect();
                }

                return null;
        }


        public void dumpConnectionRequest(HttpURLConnection conn) {
                Map<String, List<String>> map = conn.getRequestProperties();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
```

```
                        }

                        log(header.trim());
                }
        }


        public void dumpConnectionResponse(HttpURLConnection conn) throws
IOException {
                Map<String, List<String>> map = conn.getHeaderFields();

                Iterator<String> itr = map.keySet().iterator();

                while (itr.hasNext()) {
                        String key = itr.next();
                        List<String> list = map.get(key);

                        String header = key + ": ";
                        for (int i = 0; i < list.size(); i++) {
                                header += list.get(i) + " ";
                        }

                        log(header.trim());
                }
        }
}
```

Modbus

Modbus starting page

A guide and index to Modbus resources in this Wiki

External resources

What is Modbus? It is a 30 year old protocol which is best described as a "remote memory table access protocol". It literally allows a client/master to go to a server/slave and ask "return 16-bit words, starting at offset 23 and include 4 words". Modbus places no interpretation on that data; it assumes the client/master and server/slave AGREE as to the meaning. So 4 words of data might consist of:

- One word with 16 status (on/off) bits
- One word of unsigned data with only valid meaning from 0 to 10,000
- Two words as a 32-bit float, however Modbus doesn't even promise WHICH of the two words is the upper word and which is lower word

The Modbus specification is here: http://www.modbus-ida.org.

Here is background info on Modbus: http://iatips.wikispaces.com/Modbus.

Python Code Samples

At present there is no complete "Modbus Library" since by itself it cannot do anything - thus one needs to define how one wants Modbus to integrate with a Python application before one can have a library. Here are Wiki pages which explain the basic concepts for creating Modbus requests under Python:

- Python CRC16 Modbus DF1 : How to calculate the 16-bit CRC used by Modbus/RTU and Allen-Bradley DF1
- How to create Modbus/RTU request in Python: Creating a basic Modbus/RTU message as a binary-string
- Modbus class design in Python: The class hiearchy related to Modbus PDU, TCP, RTU and ASCII
- Modbus Floating Points: How to encode 32-bit floating points in Modbus

Leveraging the Modbus Bridge in the WAN IA / X4 / X8 products

Some Digi products include a powerful multi-master Modbus Bridge which freely converts between Modbus/TCP, Modbus/RTU and Modbus/ASCII. Thus your Modbus Python code can be quite simple, yet have full function. This page explains the integration:

Integrating the Digi IA Modbus bridge to Python

Introduction to Modbus

A guide and index to Modbus resources in this Wiki.

Python code samples

At present there is no complete "Modbus Library" since by itself it cannot do anything - thus one needs to define how one wants Modbus to integrate with a Python application before one can have a library. Here are Wiki pages which explain the basic concepts for creating Modbus requests under Python:

- Python CRC16 Modbus DF1: How to calculate the 16-bit CRC used by Modbus/RTU and Allen-Bradley DF1
- How to create Modbus/RTU request in Python: Creating a basic Modbus/RTU message as a binary-string
- Modbus class design in Python: The class hiearchy related to Modbus PDU, TCP, RTU and ASCII
- Modbus Floating Points: How to encode 32-bit floating points in Modbus

Leveraging the Modbus bridge in the WAN IA / X4 / X8 products

Some Digi products include a powerful multi-master Modbus Bridge which freely converts between Modbus/TCP, Modbus/RTU and Modbus/ASCII. Thus your Modbus Python code can be quite simple, yet have full function. This page explains the integration:

- Integrating the Digi IA Modbus bridge to Python: Overview of leveraging existing Digi Modbus features with Python
- Modbus bridge on CPX4: Overview of the standard Digi Modbus bridge
- Modbus Example X4 Setup: Configuring a CPX4 to bridge Modbus/TCP to Modbus serial devices over Xbee networks.
- Modbus Example Ethernet Adapter: How to configure a CPX2 to enable Modbus serial requests over Xbee network to query remote Modbus/TCP slave/servers.
- Modbus Example Serial Adapter: How to configure XBee 232 and 485 adapters to enable Modbus serial requests over Xbee network to query remote Modbus serial devices.
- Understanding XBee EndPoints: To enable Modbus over Xbee to work concurrently with Python and iDigi/Dia, you must move the Modbus traffic to a new end-point (away from 0xE8). This page explains how this is done.

External resources

What is Modbus?

It is a 30 year old protocol which is best described as a "remote memory table access protocol". It literally allows a client/master to go to a server/slave and ask "return 16-bit words, starting at offset 23 and include 4 words". Modbus places no interpretation on that data; it assumes the client/master and server/slave AGREE as to the meaning. So 4 words of data might consist of:

- One word with 16 status (on/off) bits
- One word of unsigned data with only valid meaning from 0 to 10,000
- Two words as a 32-bit float, however Modbus doesn't even promise WHICH of the two words is the upper word and which is lower word

The Modbus specification is here: http://www.modbus-ida.org
Here is background info on Modbus: http://iatips.wikispaces.com/Modbus.

Create IA configuration by Python script

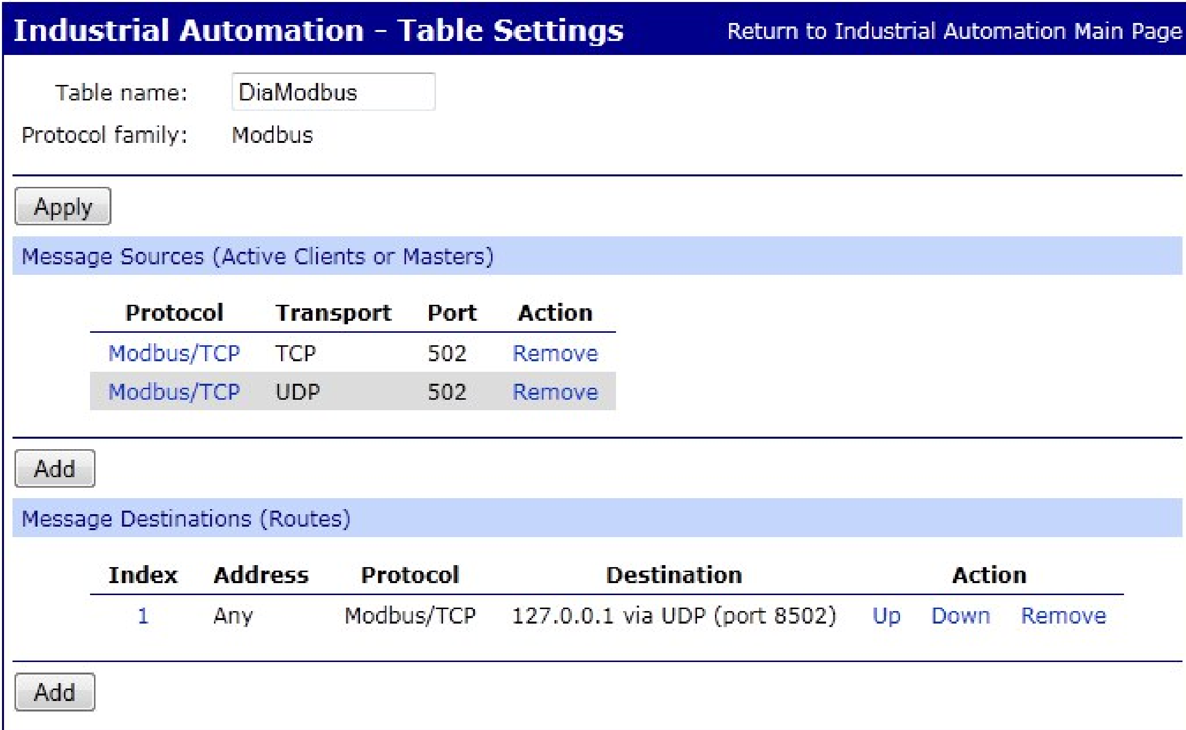
Semi-automatic creation of an IA configuration by Python script

Remotely creating an IA/Modbus routing configuration on a DigiConnect WAN IA or DigiConnectPort X4 is not rocket-science ... more like cooking without a cook book. The process is trivial if you understand it, but troublesome if you do not.

Below is a script which automatically creates a valid IA/Modbus configuration to allow Python scripts on the Digi to act as a Modbus/TCP servers. Your Python server will need to expect Modbus/TCP form requests by waiting on (binding on) localhost UDP/IP port 8502. Your script can block on this port without timeout, as the IA/Modbus engine guarantees that you will only receive correctly formatted requests without gap or inter-character delays.

Note The requests are received by UDP, yet are formatted as Modbus/TCP. They will never be Modbus/RTU.

The resulting configuration looks like this:



Preparing the script

The script is meant to be run via either telnet, or indirectly by auto-start from iDigi.

Settings

You may need to edit the configuration, which for simplicity is at the beginning of the rci_modbus.py file. Once edited and uploaded to the device, simply running it does as you requested.

LOGFILE

This setting defines if you wish output to be saved. The default is True. Set to None to suppress.

LOGFILENAME

This setting defines the text file to log (save) the report to. This is required because under iDigi you will not be able to see any output. The default is 'WEB/Python/rci_modbus_report.txt'.

FORCE_NEW_TABLE

This setting defines how safe or aggressive the script is. When set to True, it blindly clears any existing IA/Modbus configuration and create a new configuration. When set to False, it only create a new configuration is there is no existing configuration. Set it to False if you wish the script to ignore a custom IA configuration you have already created.

SERIAL_MASTER

This setting defines if you want a Modbus/RTU serial master defined for serial port #1. This does not set baud rate or MEI - you need to set that elsewhere. Set SERIAL_MASTER to True to clear any existing serial profile and create an incoming Modbus/RTU master-attached configuration. Set SERIAL_MASTER to False to ignore the serial port.

Running from Telnet or SSH

Unfortunately you cannot upload the script via Telnet or SSH. You must use either the web interface or iDigi to load the script to the Python directory.

Once uploaded, you can run the script by entering the command: "py rci_modbus.py"

You will see something like this:

```
#>
#> py rci_modbus.py

rci_modbus.py:0: SyntaxWarning: name 'LOGFILE' is assigned to before global
declaration

Testing for IA Table
req:<rci_request version="1.1"><query_setting><ia_table/></query_setting></rci_
request>
rsp:<rci_reply version="1.1"><query_setting source="current" compare_to="none"
encrypt="none"><ia_ta
ble><state>off</state></ia_table><ia_table index="2"><state>off</state></ia_
table><ia_table index="3
"><state>off</state></ia_table><ia_table index="4"><state>off</state></ia_
table><ia_table index="5">
<state>off</state></ia_table><ia_table index="6"><state>off</state></ia_
table><ia_table index="7"><s
tate>off</state></ia_table><ia_table index="8"><state>off</state></ia_table><ia_
table index="9"><sta
te>off</state></ia_table><ia_table index="10"><state>off</state></ia_table><ia_
table index="11"><sta
te>off</state></ia_table><ia_table index="12"><state>off</state></ia_table><ia_
table index="13"><sta
te>off</state></ia_table><ia_table index="14"><state>off</state></ia_table><ia_
table index="15"><sta
te>off</state></ia_table><ia_table index="16"><state>off</state></ia_
table></query_setting></rci_rep
ly>
```

```
First IA Table is inactive - no table.

Creating new IA Table
>>Send Next Request
req:<rci_request version="1.1"><set_setting><ia_
table><state>on</state><name>DiaModbus</name><family
>4</family><route_list>0</route_list></ia_table></set_setting></rci_request>
sp:<rci_reply version="1.1"><set_setting><ia_table /></set_setting></rci_reply>
>>Send Next Request
req:<rci_request version="1.1"><set_setting><ia_
master><state>on</state><active>on</active><protocol

>modbustcp</protocol><type>tcp</type><table>0</table><ipport>502</ipport><messag
e_timeout>5500</mess
age_timeout><slave_timeout>1000</slave_timeout><char_timeout>50</char_
timeout><idle_timeout>0</idle_
timeout><reconnect_timeout>5000</reconnect_timeout><master_
priority>medium</master_priority><modbus_
error_response>off</modbus_error_response><modbus_broadcast>disable</modbus_
broadcast><modbus_fixed_
address>0</modbus_fixed_address></ia_master></set_setting></rci_request>
rsp:<rci_reply version="1.1"><set_setting><ia_master /></set_setting></rci_reply>
>>Send Next Request
req:<rci_request version="1.1"><set_setting><ia_master
index="2"><state>on</state><active>on</active

><protocol>modbustcp</protocol><type>udp</type><table>0</table><ipport>502</ippor
t><message_timeout>
5500</message_timeout><slave_timeout>1000</slave_timeout><char_timeout>50</char_
timeout><idle_timeou
t>0</idle_timeout><reconnect_timeout>5000</reconnect_timeout><master_
priority>medium</master_priorit
y><modbus_error_response>off</modbus_error_response><modbus_
broadcast>disable</modbus_broadcast><mod
bus_fixed_address>0</modbus_fixed_address></ia_master></set_setting></rci_
request>
rsp:<rci_reply version="1.1"><set_setting><ia_master index="2" /></set_
setting></rci_reply>
>>Send Next Request
req:<rci_request version="1.1"><set_setting><ia_
route><state>on</state><active>on</active><protocol>

modbustcp</protocol><type>directip</type><transport>udp</transport><table>0</tabl
e><protocol_address
>0-255</protocol_address><name>to_
Python</name><ipaddress>127.0.0.1</ipaddress><ipport>8502</ipport>
<message_timeout>2500</message_timeout><slave_timeout>1000</slave_timeout><char_
timeout>50</char_tim
eout><idle_timeout>0</idle_timeout><reconnect_timeout>5000</reconnect_
timeout><modbus_error_response
>on</modbus_error_response><modbus_broadcast>replace</modbus_broadcast><modbus_
fixed_address>0</modb
us_fixed_address></ia_route></set_setting></rci_request>
rsp:<rci_reply version="1.1"><set_setting><ia_route /></set_setting></rci_reply>
```

The contents of the LOGFILENAME (by default "rci_modbus_report.txt") will contain a copy of the output.

After successfully running, you can:

1. Delete the rci_modbus.py
2. Delete rci_modbus_report.txt

Rebooting is optional since the script does not continue to run after completion.

Running from iDigi

Via iDigi, upload the script to the Python directory, then set it to auto-run. It will not interfere with other auto-run scripts (such as Dia).

Once uploaded, you must reboot the gateway, wait a few minutes and look for the LOGFILENAME (by default "rci_modbus_report.txt"). This will hold the script output, which you can open or save-as onto your PC.

After successfully running, you can:

1. Delete the rci_modbus.py
2. Remove rci_modbus.py from the auto-run list
3. Delete rci_modbus_report.txt

Rebooting is optional since the script does not continue to run after completion.

Download the code

Download and unpack this ZIP file. The file is NOT related to Dia. It is a standalone script and it will NOT run on a PC.

Rci_modbus.zip

Release Notes

rev 1.0 (June 24 2011)

Initial release. Expects NDS firmware level 2.12 or higher

Determine MTU

How to determine the maximum payload size (MTU) on a Mesh network.

Currently none of the Digi mesh systems automatically support the fragmentation or reassembly of packets larger than the Maximum Transmission Unit or MTU. Thus, the actual maximum packet size is an important fact for your program to know. Unfortunately it varies based upon underlying protocols and might be 72, 75, 84, 100 or even more than 200 bytes. Plus you need to take into account any 'options' enabled which consume part of this payload space, such as source routing or encryption headers as these reduce the data payload remaining for actual data transfer.

AT mode

In AT mode this is hidden from you - if you stream 200 bytes into an XBee serial port, it will automatically break this into as many packets as required. For example, if the MTU is 66, then it will most likely be sent as 3 packets of 66 bytes and a 4th packet of 2 bytes. If the MTU was 84 bytes, then it will most likely be sent as 2 packets of 84 bytes and a 3rd packet of 32 bytes.

API mode

In API mode you must directly honor the MTU - if you send an API frame with 84 data bytes into an XBee module set up for ZB-2007 and encryption, you will receive an error response that the frame is too long.

On newer Digi firmware there is a AT command "NP" which returns the maximum number bytes of RF payload per packet. Unfortunately, not all firmware supports this command, plus enabling some stack options causes the "NP" response to be invalid.

An indirect solution

If you have a Python program talking to an XBee coordinator set to API mode, and at the remote end you have serial devices using AT mode to return responses, then there is an indirect method to determine the MTU. As example, consider sending Modbus/RTU requests to a remote device which returns Modbus/RTU responses. Reading 100 registers of data sends an 8-byte request and receives a 205-byte response. The indirect method relies upon the fact that the remote XBee in AT mode will break the 205 byte response up into several packets, many set to the actual MTU in effect.

To use this method, the sender (your Python program) starts with a safe MTU limit for sending which is lower than all known MTU - 50 might be a good starting point. Thus to send a 100 byte Modbus request, your code would need to break it into two 50-byte packets to be submitted to the Zigbee UDP-like SendTo() socket call. The trick for this indirect method is to monitor the size of all responses in the Zigbee UDP-like RecvFrom() socket call. In all normal cases, the maximum size of any one packet received from the RecvFrom() will match the MTU for packets sent by SendTo. So this indirect method is a smart learning process over time; for example, the first 8 responses might have sizes of 8, 25, 66, 10, 84, 84, 32 and 52 bytes. In this situation, it is generally safe to assume the largest value seen - 84 bytes - is the current MTU given the firmware, protocol and options in use at this time.

Note You MUST provide a manual override to disable this auto-adjust of MTU - for example, the use of 'source routing' is known to cause RX/responses to be a different size than TX/transmissions. This override could be a fixed reduction - for example, have the TX MTU always set to 10 bytes less than the maximum bytes received. Alternatively this override could be a fixed TX MTU - for example, force the TX MTU to be 66 bytes and disable any auto-adjust.

ZigBee PRO/2007 fragmentation support

Xbee which are running in API mode and have firmware 2x6x (such as 0x2164 in your coordinator and 0x2364 in your remotes) can send up to 255 bytes within a single API frame using the ZigBee protocol 'fragmentation' support (see manual 9000976_D or never for the ZB XBee).

XBee running in API mode can send or receive fragmented messages. **XBee running in AT-transparent mode can ONLY receive fragmented packets**, but will always break outgoing packets into single-RF chunks.

So a Digi ConnectPort X4 gateway with an XBee coordinator running 0x2164 firmware can send Modbus/RTU serial requests to remove XBee 232/485 Adapters running with firmware 0x2264 (AT Router) firmware. The remote serial adapters will gracefully send out the Modbus/RTU request without time-gaps which might confuse Modbus slaves. However, any Modbus/RTU response larger than the XBee max (72, 75 or 84 bytes) will be sent as multiple non-fragmented packets. Fortunately the IA/Modbus bridge code within the X4 can reassemble these as required.

Device Cloud creation of IA configuration

Creating IA configurations from within Device Cloud

Under the **Advanced Configuration** section of the Device Cloud device, you will find four settings groups:

- Industrial Automation Master
- Industrial Automation Routes
- Industrial Automation Serial
- Industrial Automation Table

The order is alphabetical, unrelated to how you need to enter the data. Without rehashing the entire theory of the Digi IA Engine, it consists of a series of **Masters** (acting as *servers*) which accept, time-stamp and queue requests from remote Masters (acting as *clients*). The Master task needs to locate an entity which can answer the request. It uses protocol knowledge (the Unit ID in the case of Modbus) to search **Table** consisting of **Routes**, with each route pointing to an external resource. Routes may point to onboard serial ports, IP or DNS addresses, or XBee/ZigBee nodess.

The easiest way to create a valid IA configuration is via the web UI since special wizard-like programming forces you to create the required entries in the correct order. However, you can create an IA configuration via Device Cloud. Below is an example which enables Modbus/TCP and Modbus/UDP Masters (accepting Modbus requests via IP), and routes everything to a localhost port (127.0.0.1) where a Python program acts as a Modbus slave to answer the requests.

Start under the advanced configuration

Create industrial automation table 1

We make this first to make sure any created Masters/Routes can validate their protocol is compatible with their peers. For example, a protocol X master trying to obtain responses from a protocol Y slave would fail. So the IA table acts as the mediator - if it is assigned to protocol Y, then the IA Master speaking protocol X will refuse to attach to it.

Set the following settings:

- Enable = On
- Name = SunSpec (or any text desired - it has no use other than as UI feedback)
- Family = Modbus
- Route List = 0 (0 means Industrial Automation Routes 1)
- (Save this to download to the device)

Create industrial automation master 1

Set the following settings:

- Enable = On
- Active = On
- Protocol = Modbus/TCP
- Type = TCP

- Table Index = 0 (0 means Industrial Automation Table 1)
- IP Port = 502
- (Save this to download to the device)

Create industrial automation master 2

Repeat the settings for Industrial Automation Master 1, except:

- Type = UDP
- (Save this to download to the device)

Create industrial automation routes 1

Set the following settings:

- Enable = On
- Active = On
- Protocol = Modbus/TCP
- Type = directip
- Transport = UDP (the Python code assumes Modbus/UDP to reduce resources)
- Table Index = 0
- Min/Max Protocol address = 0 and 255 - leave as is or restore
  - (Note: these 2 fields are broken - setting them works, but you will NOT see the correct values when read from the device. In the near future a single string field named 'protaddr' will be seen, in which you'll put the range as 0-255. This allows scattered ranges such as 1-5,9,20-29 as well.)
- IP Address = 127.0.0.1 (which is local-host, where Python is waiting)
- Port = 8502 (we cannot use 502 since that port is already used.
- (Save this to download to the device)

Fix the industrial automation routes 1

Hit refresh, and you'll see that the Transport and Port settings did not take properly - this is a sequence issue since the Modbus/TCP code re-initializes those values during creation of a new route. So re-enter the data:

- Transport = UDP
- Port = 8502
- (Save this to download to the device, it works this take as the route already exists)

Enable Modbus query of DIA devices

Overview

Modbus is a simple memory read/write protocol. It allows a remote SCADA or HMI host to read a block of words. What those words mean must be manually configured into the host software. The Modbus DIA server Module allows a Modbus master to read the data from a collection of XBee devices as if they were each a Modbus device. For example, twenty level sensors on twenty tanks would appear as 20 Modbus devices.

Mapping Modbus unit Id to DIA device

The Modbus server uses an ASCII text file in the FS/WEB/Python directory named mbus_map.txt, which you can edit manually as required. A reboot is required to activate any changes you made. Below is an example file. The CSV-like fields are:

- Modbus slave address / Unit Id
- DIA device name, which is used to locate the device for each Modbus poll
- The device type (name) returned by the device driver - the exact value is not important and is used primarily for debug trace information
- MAC address of the device (if applicable)

```
# DIA Modbus Server unit_id mapping as of 2010-02-08 15:55:41
1,'OUTDOOR','XBeeSensor_LT','[00:13:a2:00:40:4a:6e:6b]!'
2,'INDOOR','XBeeSensor_LT','[00:13:a2:00:40:4a:6a:1e]!'
```

Hard-coded mapping

The YML can be used to hard-code a DIA device to Modbus mapping. The YML example below assumes the devices named 'solar', 'outdoor', and 'indoor' exist. Polling any Unit Id but 1, 2 or 3 will result in a Modbus exception 0x0A.

```
- name: mbus_srv
  driver: presentations.modbus.mbdia_pres:MbDiaPresentation
  settings:
      mapping: "( (1,'solar'), (2,'outdoor'), (3,'indoor') )"
```

Auto-enumerated mapping

The YML below assumes the auto-enum driver named 'xbee_autoenum' exists. It allows automatic creation of Modbus slaves numbered 1 to 20. Note that the same mbus_map.txt file is created to make the auto-assignment consistent between gateway reboots. You can edit this file as desired. To remove a device previously mapped, delete the appropriate line of the file. New devices will be placed into the lowest Unit Id available, so if you delete Unit Id 3 when 10 are defined, the next device discovered will be placed as Unit Id 3.

```
- name: mbus_srv
  driver: presentations.modbus.mbdia_pres:MbDiaPresentation
  settings:
      mapping: "('auto', 1, 20)"
      auto_enum_name: xbee_autoenum
```

Forming Modbus responses

The Modbus server module requires the target Dia device driver to support the following methods:

get_mbus_device_type()

Returns a string name of the type. No particular meaning is assigned - it can be the class name, but does not need to be.

get_mbus_device_code()

Returns a numeric code, which is included as one of the registers in the register block response. No particular meaning is assigned - it can be the lower DD word or any other value.

export_device_id()

Returns a dictionary of values for use with the Modbus Read Device Identification command (not yet implemented in the Modbus code, but will be eventually). The dictionary keys are the numeric "Object Ids" defined by the standard, and the values are all ASCII strings. Here is an example:

```
    def export_device_id( self):
        """Return the Device Id response strings"""
        dct = { 0:'Digi', 1:'XS-B14-CB1RB', 2:'1.0', 3:'www.digi.com',
                4:'XBee Sensor', 5:'/L/T', 6:'Dia' }
        return dct
```

To summarize the Object Id items:

Object Id	Required	Description
0x00	Yes	Vendor Name
0x01	Yes	Product Code
0x02	Yes	Major/Minor Revision
0x03	Yes	Vendor URL
0x04	No	Product Name
0x05	No	Model Name
0x06	No	User Application Name

export_base_regs(req_dict)

Extract the Python dictionary formatted for the Modbus block. The input dictionary will include the actual Modbus request parsed, so you can tailor the output to fit the request. See the examples in src\devices\modbus. The caller expects you to return a list (an 'array') of 16-bit integers which custom fit the Modbus request exactly. If the Modbus request is coil/boolean based, then the return is a list of boolean True/False values.

A call to src.common.modbus.mbdia_block.mbblk_to_data() creates this array for the standard values of ['status', 'din', 'ain1', 'ain2', 'ain3', 'ain4', 'dot', 'aot1', 'aot2', 'aot3', 'aot4', 'volt', 'timestamp', 'dd', 'mac'], but nothing stops you from manually creating your own custom response without using mbblk_to_data(). See Modbus DIA block register map to understand the standard register mapping.

The req_dict will include at least keys:

- req_dict['protfnc'] = Modbus function code such as 3 or 6.
- req_dict['dst'] = Modbus Slave Address / Unit Id.
- req_dict['iofs'] = zero-based offset from the request.
- req_dict['icnt'] = object count from the request.

See the file src.common.modbus.mbus_pdu.py to review the other keys usable.

import_base_regs(req_dict)

Allows Modbus to write data - FUTURE FEATURE, SO NOT YET SUPPORTED

How to create Modbus/RTU request in Python

If you have serial Modbus/RTU slaves attached to Digi XBee serial modules, then you'll need to create the original Modbus/RTU requests to send. This is quite easily done with Python, and what follows is a suggested practice after six years of using Modbus under Python.

The CRC16 checksum

Use the crc16.py module documented here: Python CRC16 Modbus DF1.

Handling BigEndian words

I normally include these two routines right in my Modbus code module.

```
def u16_to_bestr( u):
    """Given word, return as big-endian binary string"""
    u = (int(u) & 0xFFFF)
    return( chr(u>>8) + chr(u&0xFF) )

def bestr_to_u16( st):
    """Given big-endian binary string, return bytes[0-1] as int"""
    return( (ord(st[0])<<8) + ord(st[1]))
```

Feeding Input into the routine

While your first instinct might be to create routines such as make_func3(slv,offset,count) and make_func16(slv,offset,count,data), these don't make very good use of Python - plus they don't encourage creative negative testing long-term. For example, how would you create a func-3 call with a bad CRC or too much data requested - create a second routine called make_func3_badcrc() ?

Instead I suggest you fill in a dictionary holding the inputs, with some suitably defaulted if missing. While the key names are up to you, here is the list I've used with good results:

- ['dest'] is the Unit Id or slave address
- ['func'] is the Modbus code, such as 3 or 16
- ['protErr'] is the Modbus exception code
- ['rdOffset'] is the zero-based offset, so register 4x00001 is offset zero (0)
- ['rdCount'] is the word or bit count for the function
- ['rdData'] is a tuple or list holding words read
- ['wrOffset'] is like 'rdOffset', but used for writes - or for read/write functions
- ['wrCount'] is like 'rdCount', but used for writes - or for read/write functions
- ['wrData'] is a tuple or list holding words to write
- ['badCrc'] (for example) could be added to force a bad CRC to any request created

Now we can create a read of 10 registers from 4x00001 with the call **mbrtu_make_request({'dest':1, 'func':3, 'rdOffset':0, 'rcCount':10})** Below is a quick example - note that it does NOT survive missing keys to reduce the complexity.

```
def mbrtu_make_request( inDct):

    inDct.update({'error':"", 'request':None })  # indicate no error

    dest = int(inDct.get('dest',1))  # default unit id to 1
    if((dest < 0) or (dest >255)):
        inDct.update({'error':"Bad Unit Id or Slave Address"})
        return inDct

    func = int(inDct.get('func',-1)) # default is -1/error

    req = None

    if(func in [1,2,3,4]):
                req = chr(dest) + chr(func) +
                   u16_to_bestr(inDct['rdOffset']) + u16_to_bestr(inDct
['rdCount'])

    elif( fnc == 5):
        if(inDct['wrData'][0]):
           # if first item is True, so 0xFF00 (not 0xFFFF)
            dat = 0xFF00
        else: # is False
           dat = 0x0000
        req = chr(dest) + chr(func) +
           u16_to_bestr(inDct['wrOffset']) + u16_to_bestr(dat)

    elif( fnc == 6):
       req = chr(dest) + chr(func) +
          u16_to_bestr(inDct['wrOffset']) + u16_to_bestr(inDct['wrData'][0])

    # elif( fnc == 15): not here yet

    # elif( fnc == 16): not here yet

    else:
       inDct.update({'error':"Unsupported Function"})
       return inDct

    # at this point we have the basic Modbus message
    if( req):
       crc = crc16.calcString( req, 0xFFFF)
       # oddly, we need to add the CRC as LittleEndian
       req += chr(crc&0xFF) + chr(crc>>8)
       inDct.update({'request':req})


    return inDct
```

Importing Modbus data from IO device

Many people use small Modbus I/O device to manage field sensors. There are many sources, including Acromag, Advantech, DataQ, Datanab, DGH, ICP DAS, Opto22, Phoenix Contact, and Wago to name just a few. The I/O device might have for example 4 analog inputs, 8 digital inputs and so on.

Generally, one can read a dozen Modbus registers and see all the input status in one command. This Wiki page covers how to take this raw Modbus data and upload it to Device Cloud as 'DIA Data' for access by web services.

Supported Products

The code explained on this page only works on Digi Connect devices, such as:

- Connect SP (single-port Ethernet to 232/485)
- Connect WAN Family (cellular to Ethernet or Serial)
- ConnectPort X2D (Industrial, metal-enclosure X2 Xbee mesh to Ethernet)
- ConnectPort X4, X4 IA, X4 H (Xbee mesh to Ethernet and cellular)

Specifically, it does not work on the following devices

- ConnectPort X2 A, B or C - the low-cast commercial, plastic-enclosure Xbee mesh to Ethernet
- Any ConnectPort X2 with only 8MB of RAM memory
- ConnectPort X2e Xbee mesh to Ethernet
- Any Transport model
- Older non-Python products

Install the Digi ESP for Python

You will need to configure and upload a Python application. This is not as difficult as it sounds, although the jargon and steps can be new and involves some learning curve. Fortunately, once you have a functioning set of file you can replicate the solution, bypassing many of the steps.

You start by downloading the ESP installer for Windows or Mac OS. It is a large file since it includes Python and lots of documentation. You should download the newest one (2.1.0 as of Sep-2012).

Explaining how to use is beyond the scope of this page, but in summary:

1. Run the ESP.
2. Close the **Package manager** that offers to check for updates
3. Run the **Device Manager** under Device Options. Select your gateway.

   Note The gateway must be online and accessible to create, build, and upload a configuration.

4. Create a new **iDigi Dia Project**.
5. ESP won't support smart\graphical creation of a configuration, but change to the source view allows you to cut and paste the text in (explained below).

Install the Modbus client add-on

The stock Digi ESP for Python does NOT include the Modbus Client code which will import the Modbus data. We need special Modbus code which can generate a Modbus poll (such as read 10 holding registers), parse the response and create the desired DIA channels for upload to Device Cloud.

Download the Modbus Dia Code Add-On, unzip it and manually copy into the DIA copy in the ESP Program Files directory. This adds both the [[Modbus_Dia_Client | Modbus Client] (import Modbus from Modbus devices) and the Modbus DIA server.

Configure and test the general Modbus IA engine

To maximize flexibility, the Python code assumes a Modbus bridge runs on localhost (127.0.0.1) UDP port 502. Therefore you must create a Modbus bridge table which defines a list of Modbus Unit Id mapped to a remote IP, XBee MAC address, or serial port. This use of the external bridge supports Modbus/TCP form in TCP or UDP, plus Modbus/RTU or Modbus/ASCII on serial, Xbee wireless, or encapsulated in TCP/IP or UDP/IP.

In this example, assume we have a serial Modbus/RTU I/O device, so when you create the **Message Destination**, select the serial port, not an XBee Device.

Assuming you activated the Modbus/TCP master as a **Message Source**, then the Digi gateway will respond to any remote Modbus/TCP master queries. Use an OPC server or a Modbus Master tools such as ModScan to confirm that you can read the data from your I/O device.

This test is critical - if you cannot read the I/O device by indirect Modbus/TCP through the Digi gateway, then the DIA Modbus client will not be able to read it either.

Do not go past this step until you can see your Modbus I/O device data via the Digi gateway.

Create the DIA configuration

Now things will start to get more interesting.

Example generic DIA Modbus import uploading raw Modbus data from an Acromag 951EN

Paste the text below into the 'source' view of the ESP dia.yml, overwriting all other text. As shown below, this create create DIA channels named (for example) as mbus.a1_r30001, which will hold the unsigned integer value 0-65535 which was read from the I/O device's input register 3x00001. As written, the Modbus data is pulled every 60 seconds, and uploaded to Device Cloud every 60 seconds. The Modbus client creates other channels - such as mbus_a1_error, which is True if the polls are failing. Read the [[Modbus_Dia_Client | Modbus Client Wiki Page] to learn more about channels created, as well as parse options which can scale or handle floating point data.

Of course you will need to edit the 'pollinfo' to match you I/O device. The Acromag 951EN uses Modbus function #4 to read the read-only input registers. Many other I/O devices put the same information in the holding-register memory accessed by Modbus function #3.

```
devices:
  # read inpit data from Acromag 951EN
  - name: mbus
    driver: devices.modbus.mbus_udp_device:MBusUDPDevice
    settings:
        poll_rate_sec: 60
        trace: 'debug'
        round: 3
        poll_list:
            - poll: a1
              pollinfo: { 'uid':1, 'fnc':4, 'ofs':0, 'cnt':10 }
              channels:
```

```
                - parse: { 'nam':'r30001', 'ofs':0, 'unt':'word' }
                - parse: { 'nam':'r30002', 'ofs':1, 'unt':'word' }
                - parse: { 'nam':'r30003', 'ofs':2, 'unt':'word' }
                - parse: { 'nam':'r30004', 'ofs':3, 'unt':'word' }
                - parse: { 'nam':'r30005', 'ofs':4, 'unt':'word' }
                - parse: { 'nam':'r30006', 'ofs':5, 'unt':'word' }
                - parse: { 'nam':'r30007', 'ofs':6, 'unt':'word' }
                - parse: { 'nam':'r30008', 'ofs':7, 'unt':'word' }
                - parse: { 'nam':'r30009', 'ofs':8, 'unt':'word' }
                - parse: { 'nam':'r30010', 'ofs':9, 'unt':'word' }
            - poll: d1
              pollinfo: { 'uid':1, 'fnc':2, 'ofs':0, 'cnt':6 }
              channels:
                - parse: { 'nam':'i10001', 'ofs':0, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10002', 'ofs':1, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10003', 'ofs':2, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10004', 'ofs':3, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10005', 'ofs':4, 'frm':'?', 'unt':'bit' }
                - parse: { 'nam':'i10006', 'ofs':5, 'frm':'?', 'unt':'bit' }

  - name: edp_upload0
    driver: devices.edp_upload:EDPUpload
    settings:
        interval: 60
        filename: "acro"
        sample_threshold: 9999

presentations:

  # Create a new console instance on TCP port 4146.  It can be connected
  # two by using any telnet client.
  - name: console0
    driver: presentations.console.console:Console
    settings:
        type: tcp
        port: 4146
  # Enable a web interface available on http://<ip of ConnectPort>/idigi_dia
  - name: web0
    driver: presentations.embedded_web.web:Web
    settings:
        page: idigi_dia


tracing:
    default_level: "info"
    default_handler:
    - stderr
```

Getting creative

Looking into the Acromag 951EN manual shows that the DIA channel mbus.a1_r30007 is actually the CH0 analog input and will be 0-20000 representing 0ma to 20mA. We can edit the parse line to send up data samples up as floating tagged as mA. So for example, if the signal is now 12.0mA, the original YML would upload a sample "**mbus.a1_r30007 = 12000, units = word**". The modified YML would upload a sample "**mbus.a1_CHO = 12.000, units = mA**".

```
                - parse: { 'nam':'r30006', 'ofs':5, 'unt':'word' }
                - parse: { 'nam':'CH0', 'ofs':6, 'unt':'mA', 'typ':'float',
```

```
'expr':'%d/1000.0' }
                - parse: { 'nam':'r30008', 'ofs':7, 'unt':'word' }
```

Integrating the Digi IA Modbus bridge to Python

Most Digi products support a Modbus Bridge, which allows multiple Modbus clients (or masters) to share a collection of Modbus servers (or slaves). The bridge freely 'bridges' the three Modbus encoding forms of: Modbus/TCP, Modbus/RTU, and Modbus/ASCII.

Supported Digi Products with both Modbus and Python:

- Digi Connect WANIA (Ethernet, Cellular, Serial - needs firmware 82001661 rev 'A" or higher)
- Digi ConnectPort X4 (Ethernet, Cellular, Serial, Zigbee - needs firmware rev 'D" or higher)
- Digi ConnectPort X8 (Ethernet, Cellular, Serial, Zigbee - needs firmware rev 'E" or higher)

Why link the Digi IA Modbus Bridge with Python?

Of course you could use Python to create a server task, but to match the power already built into the IA Modbus Bridge you'd need to handle:

- Three encoding forms (Modbus/TCP, Modbus/RTU, and Modbus/ASCII.)
- Both TCP/IP and UDP/IP (SSL/TLS in the works)
- Defragmentation of TCP/IP and UDP/IP Modbus packets
- Dead-client detection and disconnect

So you can use the existing Digi IA Modbus Bridge to parse and validate Modbus requests, which are then forwarded in perfect form to Python waiting on a UDP socket on localhost. Your Python code can treat the Modbus/TCP requests as simple events.

What can you do?

Poll local Modbus slave, then issue Master writes when situations

Digi Configuration Example

Using Telnet or SSH, log into a supported Digi product and enter these lines to create the basic configuration. Notice that any Unit Id (slave address) 33 to 255 are sent to your Python code.

(A Python script to automate this process is on this Wiki page: Create_IA_Configuration_by_Python_Script)

```
# misc preparation configs
set profile profile=ia
set term state=off
set serial baudrate=19200 databits=8 stopbits=1 parity=none flowcontrol=none
# table defines how requests are solved into responses
set ia table=1 state=on name=Python family=modbus accessmode=multi
set ia table=1 addroute=1
set ia table=1 addroute=2
# first route defines Modbus slaves 1 to 32 on the serial ports
set ia table=1 route=1 active=on type=serial protaddr=1-32 port=1
# second route pushes Modbus/TCP requests to localhost UDP port 49502 - for
Python
set ia table=1 route=2 active=on type=ip protaddr=0-255 protocol=modbustcp
set ia table=1 route=2 transport=udp connect=passive address=127.0.0.1
```

```
set ia table=1 route=2 ipport=49502 replaceip=off slavetimeout=1000
set ia table=1 route=2 chartimeout=50 idletimeout=0 fixedaddress=0 rbx=off
# Serial port supports multi-drop of Modbus/RTU
set ia serial=1 type=slave protocol=modbusrtu slavetimeout=1000 chartimeout=20
set ia serial=1 fixedaddress=0 rbx=off active=on
# 1st and 2nd masters Modbus/TCP on TCP and UDP port 502
set ia master=1 active=on type=tcp ipport=502 protocol=modbustcp table=1
set ia master=1 priority=medium messagetimeout=2500 chartimeout=50
set ia master=1 idletimeout=0 errorresponse=on broadcast=replace
set ia master=2 active=on type=udp ipport=502 protocol=modbustcp table=1
set ia master=2 priority=medium messagetimeout=2500 chartimeout=50
set ia master=2 idletimeout=0 errorresponse=on broadcast=replace
# 3rd master serial Modbus/RTU encapsulated in UDP/IP on port 501
set ia master=3 active=on type=udp ipport=501 protocol=modbusrtu table=1
set ia master=3 priority=medium messagetimeout=2500 chartimeout=50
set ia master=3 idletimeout=0 errorresponse=on broadcast=replace
```

Python code example (Server - receive requests)

Here is a simple example of the server portion of your Python code. It waits upon UDP port 49502 (must match the number in the Digi Modbus bridge configuration above, plus CANNOT be 502 since the Digi Modbus bridge already binds on that port under both TCP and UDP.

You would customize the routine send_request_to_user(req_tuple) to process the requests and create the responses. Remember that as configured above, you must return a response within 1 second or the Digi Modbus bridge will give up and discard interest in any response with that sequence number.

```
import traceback
import socket

UDP_BREAK = 5.0
UDP_DEF_PORT = 49502
UDP_DEF_ADDRESS = ""
UDP_RCV_SIZE = 1024

def udp_run( dct):
"""Run UDP message loop forever"""

port = dct.get('ipport', UDP_DEF_PORT )
address = dct.get('address', UDP_DEF_ADDRESS )
try:
# attempt to bind on port linked to the Digi IA Modbus bridge
udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpSock.bind((address, port))
udpSock.settimeout( UDP_BREAK)

except:
return


while True:

# because we don't block, your code can do periodic other things

try:
req, addr = udpSock.recvfrom( UDP_RCV_SIZE)

except socket.timeout: # this is hit every 5 seconds (or as you configured)
```

```
continue

except: # all other errors, we just print out details and continue
traceback.print_exc()
break
if(len(req)>0):
# then we received something - should always be true if here

# SEND IT TO USER
rsp_tuple = send_request_to_user( (req, addr, dct) )

if(rsp_tuple and (len(rsp_tuple[0]) > 0)):
# then user returned a response as (rsp,addr,dct)
rsp = rsp_tuple[0]
numBytes = udpSock.sendto( rsp, addr)
# else no response desired

req = ""
rsp = ""
rsp_tuple = ()

# endwhile(true)

udpSock.close()

return

def send_request_to_user( req_tuple):
"""Given Modbus/TCP message, return exception response bad-data

Keyword arguments:
req_tuple = the request/address pair
req_tuple[0] = binstr of the protocol request
req_tuple[1] = address from socket (TCP or UDP)
req_tuple[2] = dictionary of options

Return form:
rsp_tuple = the response/address pair
if None, ia_server will hang-up
rsp_tuple[0] = binstr of the protocol response ("" is okay - no rsp)
rsp_tuple[1] = address from req_tuple[1]
rsp_tuple[2] = dictionary of options

"""

msg = req_tuple[0]
addr = req_tuple[1]
dct = req_tuple[2]

# print 'dummy mbtcp_responder saw addr ', addr

# we'll throw excpetion if not types.StringTypes

if( len(msg) < 8):
# header is too short to know full length
return None

# Here we have full header, format = SS SS 00 00 LL LL
# SS SS is sequence number - can be anything
```

```
# 00 00 is protocol format, should be 0x0000 or protocolver
# LL LL is the length, which we allow to be up to 0xFFFF

rsp = msg[:4]
rsp += chr(0)
rsp += chr(3)
rsp += msg[6] # old unit id
rsp += chr(ord(msg[7])|0x80) # old func made to error
rsp += chr(2)


return (rsp,addr,dct)
```

Python code example (Client - send requests)

The code sample isn't included yet, but sending out your own Modbus requests is very straight forward. Since the configuration for the Digi IA Modbus bridge is waiting for Modbus/TCP form on either TCP or UDP port 502, your code should just connect to the localhost (IP=127.0.0.1) by either TCP or UDP. You should use Modbus/TCP format and vary the sequence number between polls. Blocking is okay since the Digi IA Modbus bridge always returns a response to Modbus/TCP clients - either the true response or a Modbus gateway exception such as 0x0A or 0x0B (no path or timeout) if there is no response.

```
Pending.
```

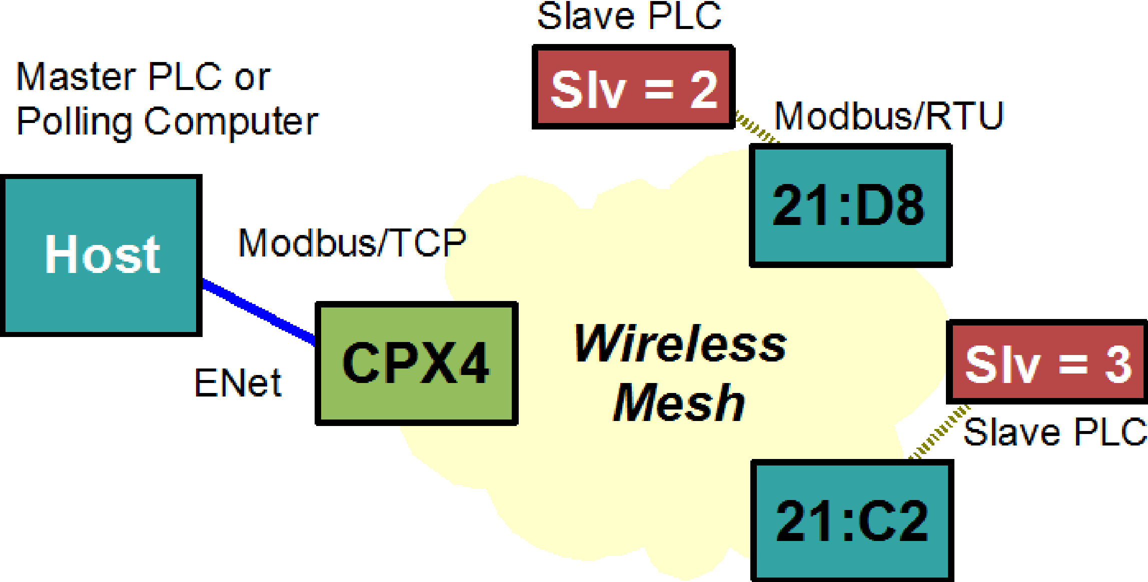
Modbus bridge on CPX4

Configuring a Modbus TCP to serial Modbus bridge Over mesh

Built into many products is the Digi IA Engine, which is a transaction processor/bridge. It allows multiple Modbus Masters (or clients) to share the same set of Modbus Slaves (servers). It works much like a print server, which allows multiple people print at the same time because it understands the notion of "a print job". Likewise, the Digi IA Engine understands the form of a Modbus transaction and gracefully interleaves requests from multiple masters, returning the correct response to the correct master.

Configuration example on Digi ConnectPort X4

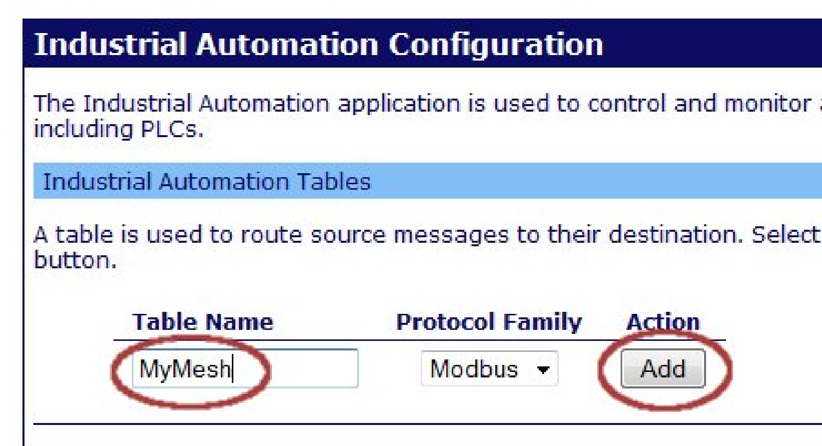
The example below will configure the CPX4 for the simple two-slave system in this drawing:



Create the IA table - link masters to slave

After making sure your CPX4 has the latest firmware, click the **Applications | Industrial Automation** link on the left. You will see this starting page to create the core protocol message routing (or mapping) table. Each table links one or more masters/clients issuing Modbus requests to one or more Modbus slaves/servers returning responses. Most people need only a single table; using multiple tables is a feature for advanced users or unusual applications.

- Click the **Applications | Industrial Automation** link on the left.
- Enter any name for your table - in this example **MyMesh**.
- Click the **ADD** button.



Message sources - who asks the Modbus questions?

After creating the table you will see the basic three-sections of the table.

- The upper section shows basic whole-table settings. For Modbus, this is only the table name. The name is a user convenience, so change as you wish.
- The center section lists the **Message Sources**. These define which remote Modbus masters/clients connections are permitted. Up to 32 masters/clients can connect at one time, and they can connect using any mix of Modbus/TCP, Modbus/RTU or Modbus/ASCII on TCP, UDP or on direct serial ports. (*At present, Modbus serial Masters cannot connect via the mesh/wireless*).
- The bottom section lists the **Message Destinations**. Each Modbus request includes a Unit Id or Slave address field, which the Digi IA Engine uses to locate which of many Modbus slaves/servers to forward the request to.

To add a new Message Source, click the middle button labeled Add.

Industrial Automation - Table Settings

Table name: MyMesh

Protocol family: Modbus

Apply

Message Sources (Active Clients or Masters)

Protocol	Transport	Port	Action
No Message Sources have been configured			

Add

Message Destinations (Routes)

Index	Address	Protocol	Destination	Action
No Message Destinations have been configured				

Add

The Message Source Settings display allows you to define incoming message types, and you can click add repeatedly to add more. In general you should add at least the Modbus/TCP on TCP/IP port 502 - even if you will be moving Modbus/RTU in UDP/IP. This makes it easier to use standard freeware/shareware tools to confirm operation.

The Source Types include:

- Receive messages from network device connecting using the network. This is any remote Master coming in from a device with an IP address. This includes local Ethernet device, remote broadband or cellular devices, or even the localhost (127.0.0.1) to enable Python scripts to talk to the Digi IA Engine.
- Receive messages from serial device connected to a serial port. This is a serial Master connected directly to a serial port on the Digi device. This does not include a serial master connected to a remote terminal or device server - those would be considered network devices. *At present it also does not include serial masters on mesh devices.*

You can see the various other options - support for any of the common Modbus protocols (Modbus/TCP, RTU, ASCII), plus IP or port settings. If you are talking over high-latency networks like cellular, then you should enable the *Hang up when source connection is idle* option to speed up recognition of broken client links.

Detailed HELP info is here

Industrial Automation - Message Source Settings

Location Settings

Source Type: Receive messages from network device connecting using the network

Network Source Settings

Protocol: Modbus/TCP

Transport: TCP

Network port: 502

Hang up when source connection is idle

Idle timeout: 0 seconds

Transaction Behavior

Message timeout: 2500 msec *(time to wait for response, including queuing delays)*

Character timeout: 50 msec *(gap between bytes or fragments of a message)*

Priority: Low Medium High

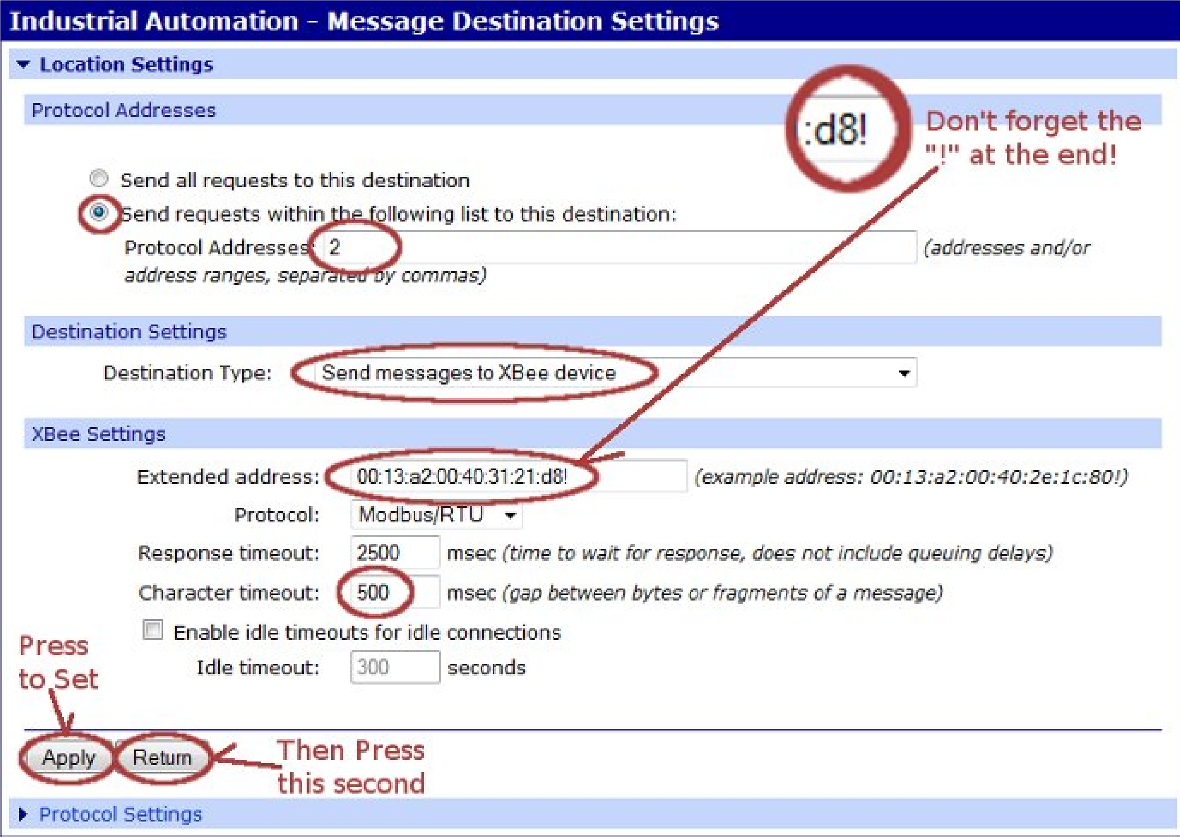
Apply Cancel

Protocol Settings

After clicking **Add**, you'll see a default dialog simplier than the one show below. As you add information, it will expand to what you see below.

1. Since we define the first destination as Modbus slave 2, we click the radio button next to " **Send requests within the following list to this destination:** "
2. Enter the number 2 - you could also enter a range such 2-10 or a scattered list such as 2,5,23,45.
3. Change the **Destination Type to Send messages to XBee device**. This causes the dialog size to double as the various destination options are exposed.
4. Enter the **Extended Address** - the MAC address. Use the format shown and remember the trailing "!"
5. Increase the Character Timeout to be from 500 msec to **5000** msec. The default of 50 msec is too short for the mesh.
6. Click the **APPLY** button - this saves this destination, but does not allow you to create a destination route.
7. Click the **RETURN** button, which returns to the table overview.

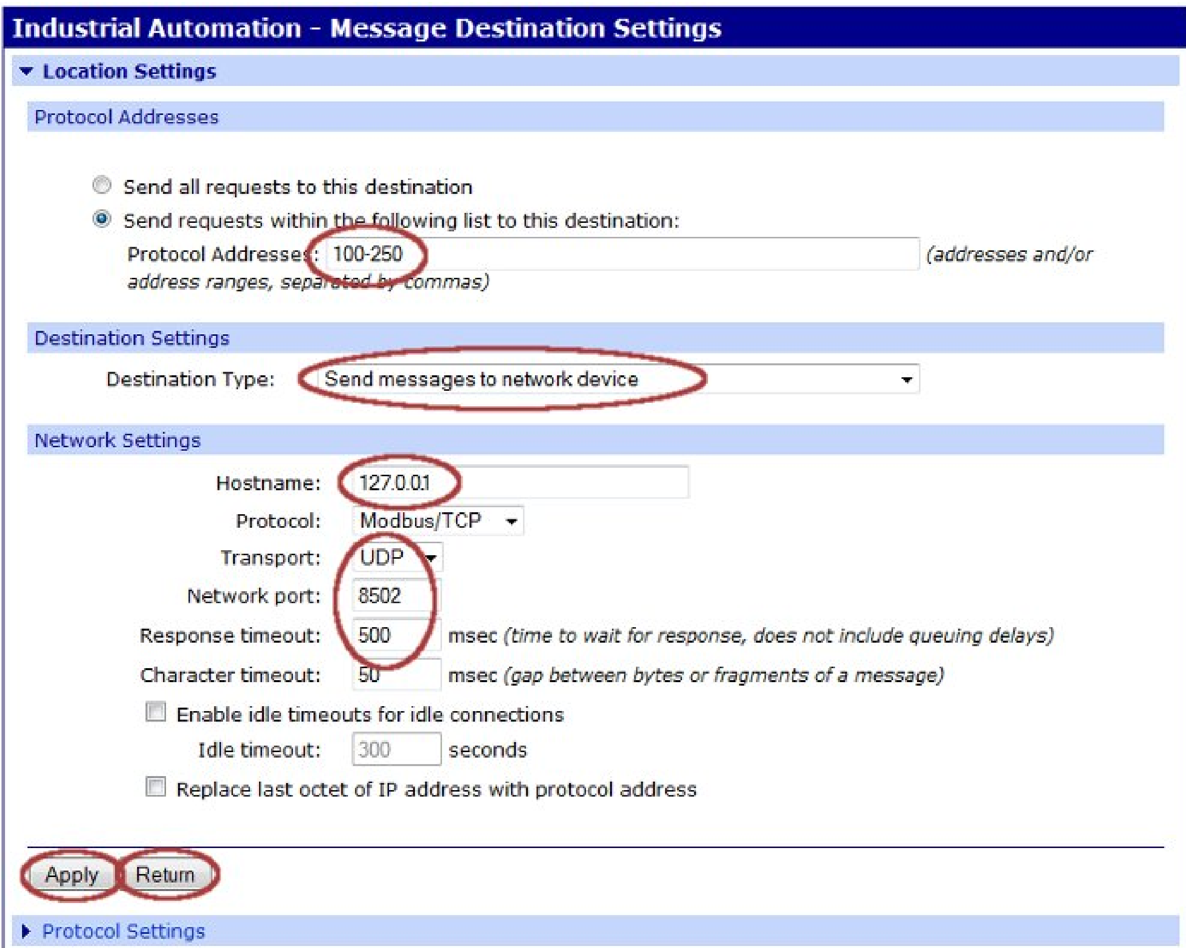
Now add the second destination for Modbus slave #3 to MAC 00:13:a2:00:40:31:21:c2! Follow these same steps as above, but enter the new address information.



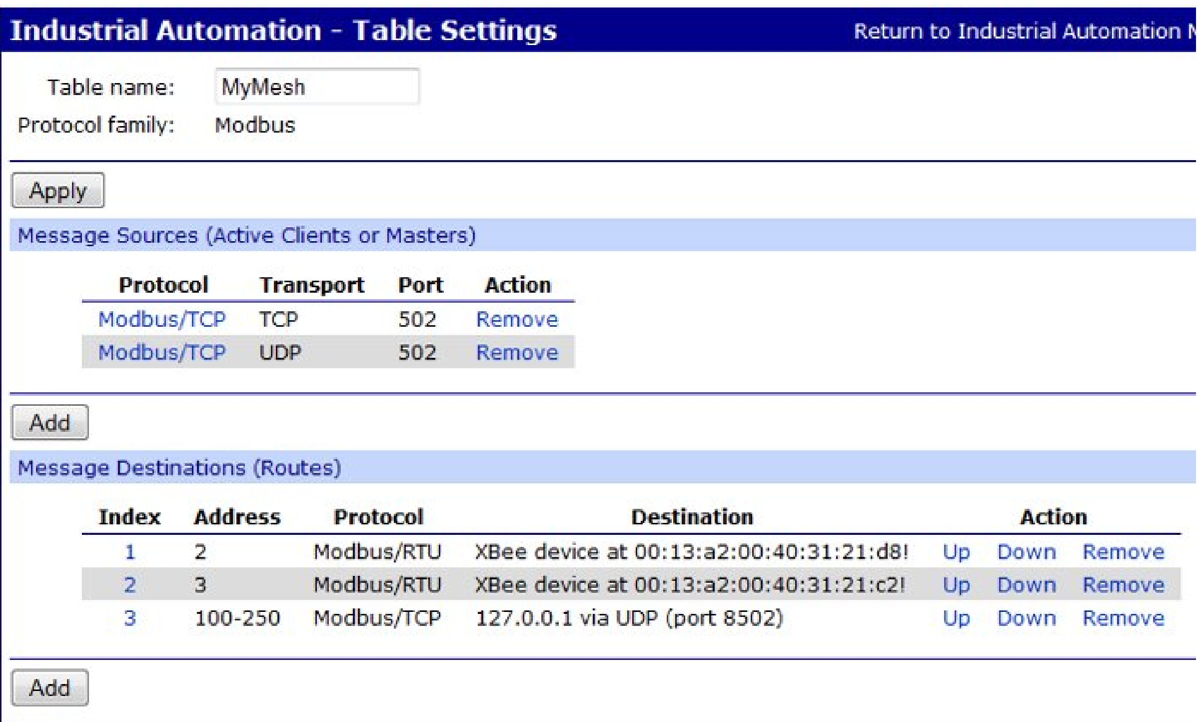
Add Python as a message destination

Technically you are now done. However, to illustrate adding custom Python support to your X4 gateway we'll add one more destination. All Modbus slaves from 100 to 250 shall be defined as destination Modbus/UDP on localhost (IP = 127.0.0.1) on UDP port 8502. *We cannot use 502 as that port is already bound to the incoming Masters!* Using UDP simplifies your Python and reduces system resources.

If your Python code binds on 127.0.0.1 UDP port 8502, then the IA Engine will pass you Modbus/TCP formatted requests which you Python code can answer.



Below is how our final table looks - notice the Up/Down actions - the IA Engine always scans the list from top to bottom, stopping at the first address match. So if you added a fourth route for Modbus slave 4 on the mesh, then your X4 would function as expected - however, if you added a fourth route for Modbus slave 144 on the mesh, then requests for Modbus slave 144 would still match on route index 3 and be handed to Python.



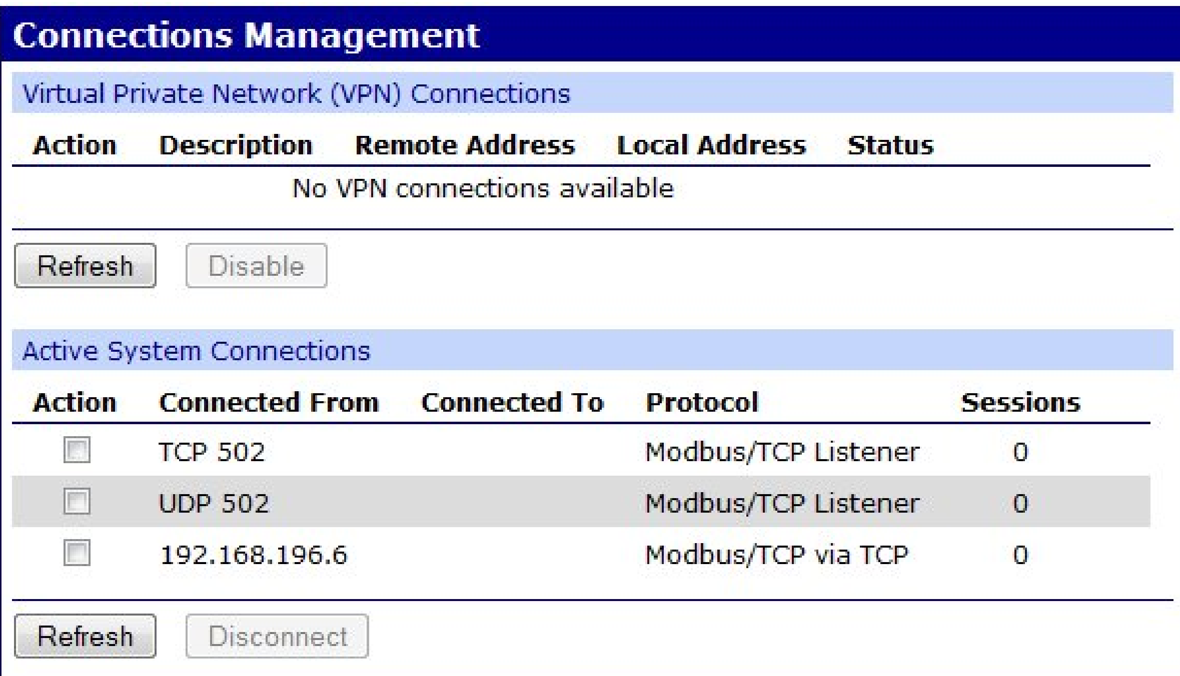
Troubleshooting

First, some common sense steps:

- Reboot after changing configuration - generally changes can be applied without rebooting. However certain actions pending can block those changes
- Confirm all of our MESH devices show up on the XBee Network display, and that they have the correct baud rates applied.
- Confirm you have the correct protocol enabled - enabling incoming Modbus/TCP but using a client sending Modbus/RTU format in TCP/IP will NOT work. You can add a new message source for Modbus/RTU in TCP.

Second, pressing the **Management > Connections** link brings up the display below. The two items listed as **Modbus/TCP Listener** should ALWAYS be present (you'll only see the one on UDP port 502 if you enabled it as a second incoming master/message source).

In the example below you will also see a third item tagged as Modbus/TCP via TCP with IP address 192.168.196.6, which means we already have one client connected.



Third, you can see a real-time trace, which is explained at this wiki page. That page highlights use of the Digi One IAP product, but the same trace functionality exists on the X4.

Fourth, if you are seeing slave timeouts, the X4 actually times all requests and displays that information in the same event trace, which is explained at this wiki page. You will want to set very long time out (up in the 10 or 15 second range), then watch these response times. You can then safely reduce the time outs to match actual performance.

Modbus DIA block register map

(Note: this feature is being tested, and will be included in a future release of DIA)

DIA Modbus register map (basic)

The block oriented Modbus server returns a fixed register map common for all supported devices. Registers which are unused (for example the analog outputs on a analog input device) shall be zero (0x0000).

Basic register map

All supported devices start with the basic register map, where any analogs are converted to fixed point integers.

Modbus	Offset	Mode	Name	Description
4x00001	0	Read-Only	--	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	status	Device Status as 16 bits, See below
4x00003	2	Read-Only	din	Digital Inputs as 16 bits, 0 if none
4x00004	3	Read-Only	ain1	Analog Input #1, 0 if none
4x00005	4	Read-Only	ain2	Analog Input #2, 0 if none
4x00006	5	Read-Only	ain3	Analog Input #3, 0 if none
4x00007	6	Read-Only	ain4	Analog Input #4, 0 if none
4x00008	7	Read-Write	dot	Digital Outputs as 16 bits, 0 if none
4x00009	8	Read-Write	aot1	Analog Output #1, 0 if none
4x00010	9	Read-Write	aot2	Analog Output #2, 0 if none
4x00011	10	Read-Write	aot3	Analog Output #3, 0 if none
4x00012	11	Read-Write	aot4	Analog Output #4, 0 if none

Modbus	Offset	Mode	Name	Description
4x00013	12	Read-Only	volt	Battery Voltage as Fixed Point, so 0 = 0.0v, 420 = 4.20v up to 65535 = 635v, 0 if none
4x00014	13	Read-Only	timestamp	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	--	High Word of UNIX style date/time of last data reading
4x00016	15	Read-Only	dd	Lower word of the XBee DD value
4x00017	16	Read-Only	mac	bytes 7-8 of IEEE MAC Address
4x00018	17	Read-Only	--	bytes 5-6 of IEEE MAC Address
4x00019	18	Read-Only	--	bytes 3-4 of IEEE MAC Address
4x00020	19	Read-Only	--	bytes 1-2 of IEEE MAC Address

Device status bits

The 16-bit device status are:

mask	bit	Description
0x0001	1	Driver Fault; if 1 then some DIA driver fault is true
0x0002	2	Device is off-line; if 1 then the device is not responding. Depending on config, normally there will be no Modbus response in this situation to simulate an unreachable slave.
0x0004	3	Low-battery signal; if 1 then device signals a low-battery condition
0x0008	4	Device Fault; if 1 then device is talking, but indicating internal error
0x0010	5	Event Sample; if 1 then the data in this sample was triggered by an event, such as alarm exception or the operator pressing the 'wake / sample' button. If 0, then this sample is a normal time cycle update.
0x00E0	6-8	Reserved
0x0100	9	If 1, then Digital Input register is valid
0x0200	10	If 1, then Analog Input registers are valid
0x0400	11	If 1, then Digital Output register is valid
0x0800	12	If 1, then Analog Output registers are valid
0xF000	13-16	Reserved

Supported devices

Digi XBee AIO Adapter (mbdia_xbee_aio driver)
Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Analog Input #1, see format below
4x00005	4	Read-Only	Analog Input #2, see format below
4x00006	5	Read-Only	Analog Input #3, see format below
4x00007	6	Read-Only	Analog Input #4, see format below
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

The fixed-point format for the registers shall be:

- type = 'raw', then the raw binary value
- type = 'off', then zero
- type = 'TenV', then voltage * 1000, so 4.23v is 4230
- type = 'CurrentLoop', then current * 1000, so 12.10mA is 12100
- type = 'Differential', then voltage * 1000, so -1.23v is 0xFB32 and +1.23v is 1230

Digi XBee /L/T/H sensor adapter (mbdia_xbee_sensor driver)
Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Temperature as Deg C, fixed point * 100, so 2300 = 23.00 DegC
4x00005	4	Read-Only	Light as direct value
4x00006	5	Read-Only	Humidity as fixed point * 100, so 7950 = 79.50 % RH, zero if this sensor does NOT have a humidity input
4x00007	6	Read-Only	Temperature as Deg F, fixed point * 100, so 7950 = 79.50 DegF

Modbus	Offset	Mode	Description
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Digi Smart Plug (mbdia_xbee_rpm driver)

Returns four analog values only - registers NOT listed in the table below are always zero.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00003	2	Read-Only	Digital Outputs: bit 0x0001 is plug status
4x00004	3	Read-Only	Current as Amps, fixed point * 100, so 230 = 2.3 Amps
4x00005	4	Read-Only	Light as direct value
4x00006	5	Read-Only	Temperature as Deg C, fixed point * 100, so 2950 = 29.50 DegC
4x00007	6	Read-Only	Always zero
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Massa M3 Ultrasonic level sensor (mbdia_massa_m3 driver)

Any registers not show below are returned as zero (0) - but that might change in the future.

Modbus	Offset	Mode	Description
4x00001	0	Read-Only	Magic Number, always 0xE801 for this register map
4x00002	1	Read-Only	Device Status as 16 bits
4x00004	3	Read-Only	Level as inches, fixed point * 100, so 1423 = 14.23 inches
4x00005	4	Read-Only	Temperature as Deg C, fixed point * 100, so 2315 = 23.15 DegC
4x00006	5	Read-Only	Target Strength, fixed point * 100, so 7500 = 75%
4x00007	6	Read-Only	Latest Event counter as word, so 1 to 65535
4x00013	12	Read-Only	Battery Voltage, fixed Point * 100, so 420 = 4.20v
4x00014	13	Read-Only	Low Word of UNIX style date/time of last data reading
4x00015	14	Read-Only	High Word of UNIX style date/time of last data reading

Note Target Strength is only 0%, 25%, 50%, 75% or 100%

Modbus DIA Client

Modbus DIA Client Driver

This driver polls remote Modbus servers/slaves for data (words or bits), which are then converted to standard DIA channels. So it imports Modbus data into DIA channels.

You can get more info and hints in the Modbus starting page.

Uses Digi IA/Modbus Engine

The normal Digi "IA Modbus Engine" is required, and the DIA will appear to it as a remote client using Modbus/TCP form in UDP (the UDP is important), so you need to enable at least the Modbus/TCP master on UDP port 502. TCP/IP could have been supported, but it consumes considerable more resources than UDP/IP without adding value.

You need to set up IA table destinations pointing at your slaves, so these can be any combination of Modbus/TCP, Modbus/ASCII or Modbus/RTU via TCP/IP, UDP/IP, serial or Zigbee mesh.

This design has the following advantages:

- Supports a wide variety of Modbus devices:
  - Modbus/TCP slaves/servers via TCP/IP or UDP/IP, via Ethernet or cellular
  - Modbus/RTU slaves/servers via TCP/IP, UDP/IP, direct serial port or XBee 232/485 Adapters
  - Modbus/ASCII slaves/servers via TCP/IP, UDP/IP, direct serial port or XBee 232/485 Adapters
- Allows transparent pass-through for remote Modbus Clients. For example, a remote Modbus/TCP client can read or write a thousand various registers in a complex Modbus device, yet the DIA configuration need only import three registers of interest
- The Digi IA/Modbus engine is a very mature product is supported by at least these Digi products:
- Digi ConnectPort X4, X4H, X8
- Digi ConnectPort TS8, TS16
- Digi Connect WAN IA, WAN VAN
- Digi ConnectPort WAN

This design has the following disadvantages:

- The Modbus DIA Driver cannot be used on Digi products unable to run the IA/Modbus engine concurrently with Python.
- Specifically, it cannot be used on:
  - Digi ConnectPort X2
  - Digi ConnectPort X3

Example YML Configuration

Below is an example YML configuration which reads two blocks of data from remote server(s)

```
  - name: mbus
    driver: devices.modbus.mbus_udp_device:MBusUDPDevice
    settings:
        poll_rate_sec: 30
        udp_peer: ('127.0.0.1',502)
        trace: 'debug'
        round: 3
        poll_list:
            - poll: in01
              pollinfo: { 'uid':1, 'fnc':3, 'ofs':0, 'cnt':20 }
              channels:
                - parse: { 'nam':'panel', 'ofs':3, 'frm':']H', 'unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }
                - parse: { 'nam':'battery', 'ofs':4, 'frm':']H', 'unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }
                - parse: { 'nam':'load','ofs':5,'frm':']H','unt':'vdc',
'typ':'float', 'expr':'(%d/1000.0)*3.25' }


            - poll: in02
              pollinfo: { 'uid':1, 'fnc':1, 'ofs':0, 'cnt':48 }
              channels:
                - parse: { 'nam':'DIN', 'ofs':8, 'frm':'?', 'unt':'valid', }
                - parse: { 'nam':'AIN', 'ofs':9, 'frm':'?', 'unt':'valid', }
                - parse: { 'nam':'DOT', 'ofs':10, 'frm':'?', 'unt':'valid', }
```

Base Device Settings

A single thread is spawned for each **MBusUDPDevice** device created. The thread sleeps and wakes on a cycle defined by the YML poll_rate_sec setting. When it wakes, it runs through a list of POLLS, running them in strict half-duplex sequential order. Modbus is not a high-speed protocol, so users must be reasonable in their selection of poll rates.

Specific settings for the base device:

```
    poll_rate_sec: 30
    udp_peer: ('192.168.196.140',502)
    trace: 'debug'
    round: 3
    poll_list:
    ...
```

poll_rate_sec

- Defines the poll rate. Note that the device attempts to prevent system drift by compensating for delays in response.
- Type integer, stated in seconds, minimum is once per 5 seconds
- Optional, default = once per 15 seconds

udp_peer

- Defines the remote host address as IP (or DNS name) plus UDp port number to send requests to
- Type string
- Optional, default = ('127.0.0.1',502) (localhost and the well-known Modbus/TCP port number, assumes links to Digi IA/Modbus Engine)

- Note that at present, the client does NOT time-out. It relies upon the server to always respond - which is true when the Digi IA/Modbus Engine is used.

trace

- Defines how chatty the driver should be on the trace output
- Type string or integer
- Optional, default = 0x0032 (fancy steady-state, start/stop events and field errors)
- See the ia_trace.py documentation for more options.

round

- Defines a global 'round()' for floating point value
- Type integer
- Optional, default = 999, magic number for ignore
- Example: a temperature with a +/- 1 deg accuracy should not be returned as '23.34876549'. Setting round: 2 will cause all floating points to rounded to 2 places, so '23.34876549' is ASCII encoded as '23.35'

poll_list

- Defines a collection (list) of Modbus block read/writes and now the data should be imported to DIA
- Required
- Example: see next section

Poll List Settings

Each **MBusUDPDevice** device can run a collection of Modbus polls. Although each MBusUDPDevice sends request to a single server, that server can interpret the Modbus Unit Id (or slave address) to obtain responses from multiple remote servers.

Specific settings for the poll objects:

```
- poll: in01
pollinfo: { 'uid':10, 'fnc':3, 'ofs':0, 'cnt':20 }
channels:
...
```

poll

- Define the name for this poll object. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness
- Example: in01 or motor or plc_b

pollinfo

- Define the Modbus parameters for this poll object. Only functions 1, 2, 3 and 4 are supported at present.

- Type Python dictionary
- Required, user is responsible to insure uniqueness
- Example: { 'uid':10, 'fnc':3, 'ofs':0, 'cnt':20 } means poll 20 holding regs starting at 4x00001 on slave unit-id 10. You should create the required destination in the Digi IA/Modbus table

pollinfo['uid']

Defines the Modbus Unit Id or slave address to use in the request

pollinfo['fnc']

Defines the Modbus function code to use - limited to function 1, 2, 3 and 4

pollinfo['ofs']

Defines the offset (zero-based). So reading Modbus register 4x00001 is offset 0, which reading coil 0x00007 is offset 6.

pollinfo['cnt']

Defines the number of registers (words) or coils (bits) to read

channels

- Defines a collection (list) of DIA channels to create from this Modbus poll object
- Required
- Example: see next section

Channel List Settings

Each Modbus poll block of data can be imported into multiple DIA channels.

Specific settings for the channel objects:

```
    - parse: { 'nam':'panel', 'ofs':3, 'frm':']H', 'unt':'vdc', 'typ':'float',
'expr':'(%d/1000.0)*3.25' }
    - parse: { 'nam':'totalEnergy', 'ofs':0, 'frm':'[L', 'unt':'Wh' }
    - parse: { 'nam':'load', 'ofs':5, 'frm':'[f', 'unt':'vdc', }
    - parse: { 'nam':'overload', 'ofs':8, 'frm':'?', 'unt':'valid', }
```

parse

- Define the import/creation of a single DIA channelect. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness
- Example: { 'nam':'totalEnergy', 'ofs':0, 'frm':'[L', 'unt':'Wh' } reads the first two registers of the poll block, treats them as a Modbus 32-int with LOW word first. It creates a DIA channel named mbus.in02_acTotalEnergy which will include a sample such as <Sample: "17294" "Wh" at "2009-10-28 15:36:30">

parse['nam']

- Defines the DIA channel name. So if the poll is named 'inv01' and channel named 'totalEnergy', then the final channel will be named 'inv01_totalEnergy'.

- Type string
- Required
- Example: four poll blocks named ['inv01','inv02','south_wing','ghouse'] could create four channels named inv01_totalEnergy, inv02_totalEnergy, south_wing_totalEnergy, and ghouse_totalEnergy.

parse['ofs']

- Defines the word or bit offset in the poll block. It is zero-based.
- Type integer
- Required

parse['frm']

- Defines how the Modbus data bytes are parsed to obtain the DIA sample.
- Type string
- Required
- Values are similar to the Python struct format
  - '?' for 1-bit coils - these can be parsed from register or coil response, but be aware the 'ofs' value is from the start of the poll block so reading the 3rd bit in the 4th register requires 'ofs' of 66 (67th bit, zero-based)
  - 'h' for 16-bit signed int import to DIA ('H' is unsigned)
  - '[i' for 32-bit signed int import to DIA ('[I' is unsigned), with the '[I' meaning LOW word is in first Modbus register, which ']I' or '>I' would be BIG word in first Modbus register
  - '[f' for 32-bit float import to DIA with '[' or ']' being used as in the 32-bit int

parse['unt']

- Defines the Unit of Measure string
- Type string
- Optional, default is (an empty string)

parse['typ']

- Over-rides the type implied by the format string
- Type string, values in ['float','int','long','bool']
- Optional, default is to use parse['frm'] to estimate channel type

parse['expr']

- An 'eval' expression to do simple value conversion of the data
- Type string, in print format such as '(%d/1000.0)*3.25'
- Optional, default is no conversion
- Example: { 'nam':'panel', 'ofs':3, 'frm':'H', 'unt':'vdc', 'typ':'float', 'expr':'(%d/1000.0)*3.25' } treats the Modbus register as a 16-bit signed integer, yet applies the formula '(value/1000.0)*3.25' to create a floating point channel sample

- Example: { 'nam':'cold', 'ofs':3, 'frm':'H', 'unt':'is_cold', 'typ':'bool', 'expr':'bool(%d<230)' } treats the Modbus register as a 16-bit unsigned integer, yet creates a boolean channel sample

Handling Sample Errors

If the Modbus server/slave did NOT respond, then the DIA sample will be an 'AnnotatedSample', which just means you'll find a new field named 'errors', which is a set[] containing the reason code which are:

- 'not_init' means the sample has never been updated even once.
- 'stale' means the last 3 or more polls have timed out.
- 'bad_calc' means some import/conversion failure occurred.

Modbus DIA server

Enabling the DIA Modbus server

The DIA Modbus server allows remote Modbus masters/clients to query your DIA devices as if they were Modbus devices. However, the data has been cached by the Digi gateway, so if the DIA device is sleeping and wakes only once per hour, then the Modbus data returned will be repeated (stale) for the entire hour.

Data models

Block Devices

Each device appears as a distinct Modbus destination with the I/O such as:

- 4 analog inputs (field to system)
- 4 analog outputs (system to field)
- 16 digital inputs (field to system)
- 16 digital outputs (system to field)

More information is on this Wiki page: Modbus DIA block register map.

How the Modbus Unit Id (or Slave Address) is mapped to DIA device is covered on this Wiki page: Enable Modbus query of DIA devices.

Channel Mapping

(Future work) You build custom Modbus maps on a channel-by-channel basis.

Robust Block Server

This version runs only on Digi gateways with the Modbus/IA Engine. DIA uses Modbus/UDP on localhost to the IA Engine, which provides a mature and robust multi-master solution.

It allows up to 32 incoming Modbus masters via:

- Modbus/TCP in TCP/IP or UDP/IP
- Modbus/RTU on gateway serial ports, or encapsulated in TCP/IP or UDP/IP
- Modbus/ASCII on gateway serial ports, or encapsulated in TCP/IP or UDP/IP
- Digi Realport on Windows set up for UDP mode, which enters the Modbus/IA Engine as a Modbus/RTU or Modbus/ASCII master encapsulated in UDP/IP

Gateway configuration

By web UI

To enable correct bridging of Modbus into the DIA, create a simple Industrial Automation configuration summarized by these steps:

- Click the Applications | Industrial Automation link on the left side of the display
- Confirm a Modbus Protocol table exists, or add one if required. The table name can be anything
- Click the Table Name to see the Table Settings page

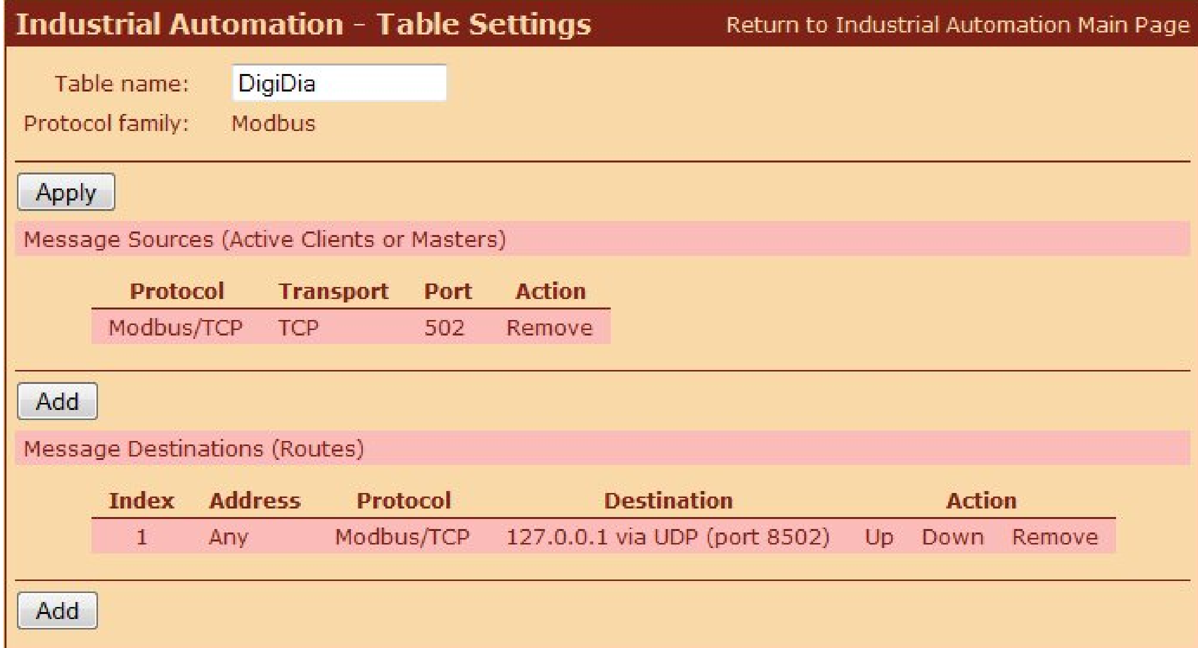
- Add at least one message source (an incoming Master/Client). This could be Modbus/TCP on Ethernet or any other selection. More than one can be created.
- Add a message destination to a Modbus/TCP in UDP/IP on IP 127.0.0.1, UDP port 8502. To start with, route all messages to this and place it in the first row (index #1).
- reboot the gateway

By CLI over Telnet or SSH

```
set ia table=1 state=on name=DigiDia family=modbus accessmode=multi
set ia table=1 ownerperiod=15000
set ia table=1 addroute=1
set ia table=1 route=1 active=on type=ip protaddr=0-255 protocol=modbustcp
set ia table=1 route=1 transport=udp connect=passive address=127.0.0.1
set ia table=1 route=1 ipport=8502 replaceip=off slavetimeout=1000
set ia table=1 route=1 chartimeout=50 idletimeout=0 lineturnmode=off
set ia table=1 route=1 fixedaddress=0 rbx=off
set ia master=1 active=on type=tcp ipport=502 protocol=modbustcp table=1
set ia master=1 priority=medium messagetimeout=2500 chartimeout=50
set ia master=1 idletimeout=0 lineturnmode=off errorresponse=on
set ia master=1 broadcast=replace
```

The Final Result

Your final IA Engine Configuration should look something like this (web colors may vary - this is a NDS web customization):



Configuring DIA

DIA YML Changes

Below is an example YML showing use of DIA 1.3 with auto-enumeration of XBee AIO and LTH sensors. Any device discovered will be automatically added to the Modbus server/slave list.

Important points:

1. You MUST use the special Modbus 'sub-class' for the DIA drivers - such devices.modbus.mbdia_xbee_aio:MBusXBeeAIO. These behave like the normal DIA drivers, but have the added Modbus calls to create the Modbus register maps
2. You must include the Modbus server presentation: presentations.modbus.mbdia_pres:MbDiaPresentation

```
- name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
  - name: xbee_autoenum
    driver: devices.xbee.xbee_devices.xbee_autoenum:XBeeAutoEnum
    settings:
        xbee_device_manager: xbee_device_manager
        short_names: True
        devices:
           - name: ain
            driver: devices.modbus.mbdia_xbee_aio:MBusXBeeAIO
            settings:
                sample_rate_ms: 60000
                power: "On"
                sleep: False
                channel1_mode: "CurrentLoop"
                channel2_mode: "CurrentLoop"
                channel3_mode: "CurrentLoop"
                channel4_mode: "CurrentLoop"
          - name: lth
            driver:     devices.modbus.mbdia_xbee_
sensor:MBusXBeeSensor
            settings:
                sleep: True
                sample_rate_ms: 15000
                awake_time_ms: 5000
presentations:
  - name: mbus_srv
    driver: presentations.modbus.mbdia_pres:MbDiaPresentation
    settings:
        mapping: "('auto', 1, 20)"
        auto_enum_name: xbee_autoenum
```

Seeing the Modbus 'Unit Id/Slave Address' assignment

The Modbus DIA server maintains a text file in the gateway's Python file systems named 'mbus_map.txt'. Every time it boots, it starts by reading this file (if it exists). Any new nodes seen are appended. You can edit this file at any time, reordering the lines.

This is a static example of mbus_map.txt, where the devices were manually entered by name in the YML as mapping: "((1,'solar'), (2,'outdoor'), (3,'indoor'))"

```
# DIA Modbus Server unit_id mapping as of 2010-03-24 15:01:56
1,'solar','XBeeAIO','00:13:a2:00:40:4b:90:24!'
2,'outdoor','XBeeSensor','00:13:a2:00:40:4a:6e:83!'
3,'indoor','XBeeSensor','00:13:a2:00:40:32:14:FA!'
```

This is a dynamic/autoenum example of mbus_map.txt

```
# DIA Modbus Server unit_id mapping as of 2010-03-26 21:46:48
1,'lth_14_c6','XBeeSensor_LT','[00:13:a2:00:40:32:14:c6]!'
2,'lth_15_1a','XBeeSensor_LT','[00:13:a2:00:40:32:15:1a]!'
```

Modbus Dia Code Add-On

Notice: this code is experimental and supplied as-is.

Download the Code

Download and unpack this ZIP file. The files will gracefully overlay the existing Dia files structure without over-writing any Dia files.

MBus_dia_04oct2012.zip

So for example, if you are using ESP, then the Dia may be in a subdirectory such as C:\Program Files (x86)\Digi\DigiPython\Dia\Dia_1.4.14 or C:\Program Files\Digi\Python\DevTools-2.1\Dia\Dia_2.1.0. You would copy the files unzipped into the appropriate directory, which will require admin privileges.

Release Notes

Sept 2010 files

Adds support for writing Dia float channels as 2 Modbus registers.

Dia 2.0/2.1 (June 2012)

Port to newer trace facility and Dia core requirements.

Dia 2.0/2.1 (Oct 2012)

Fixed a bug in the xbee_sensor file when used with Modbus.

Modbus Example Ethernet Adapter

Modbus Bridge Example - Ethernet Adapters

This is the XBee to Ethernet node portion of an Modbus Bridge example using a Digi ConnectPort X4 as the central Modbus/TCP bridge and a collection of serial and Ethernet nodes.

This page shows how to enable XBee wireless to be bridged to Modbus/TCP servers (or slaves). This is NOT directly related to use of Modbus/TCP to query remote nodes via XBee wireless.

To see how to Bridge mesh to Modbus/RTU serial slaves, see this web page: Modbus Example Serial Adapter.

Supported Models

Digi ConnectPort X2 when loaded with firmware 82001596 (Ethernet model) or 82001630 (Wi-Fi model), which supports Python.

Product Installation

Confirm that you have the correct power for your ConnectPort X2 - probably 9 to 30vdc. If you want to use an existing shared 12vdc or 24vdc supply, Digi sells a power-pigtail with the correct locking barrel connector - details are here: Locking Power Connecter.

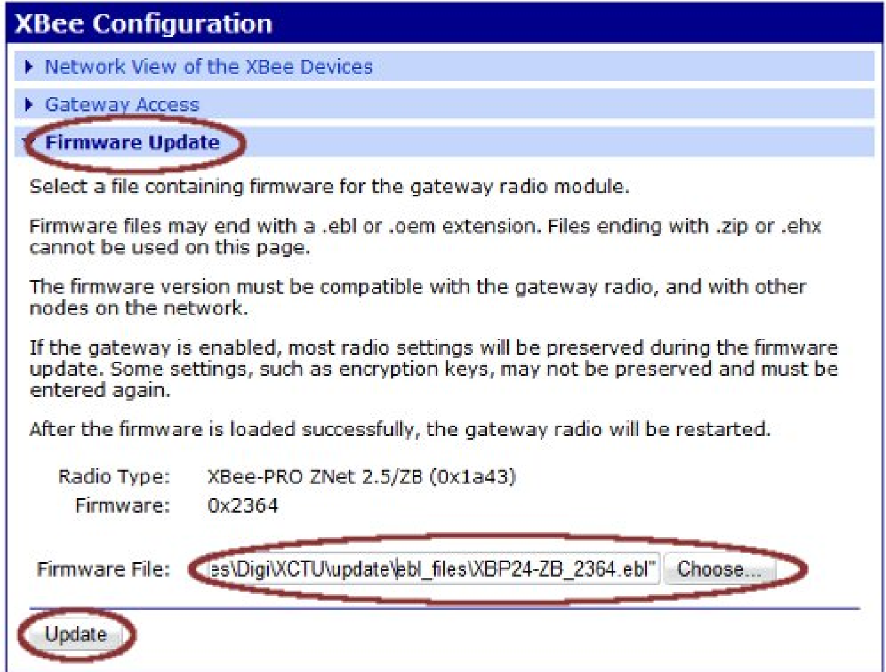
You'll also need to make sure the appropriate antenna is installed - and if the CP-X2 is to be installed inside of a metal enclosure, you'll need to plan to mount the appropriate external antenna options.

Reflash the CPX2's XBee Module

To allow Modbus/RTU traffic on the XBee wireless to bridge to local Ethernet servers, you'll need the XBee module to be flashed with the XBee Router API firmware (such as 0x2364) - NOT the default XBee Coordinator API firmware. Quite literally, you need the CPX2 to be a member of the central CPX4 bridge's mesh, so the CPX4's XBee is to be the Coordinator; NOT the CPX2's XBee.

You can either:

Use the web UI to upload the EBL file (such as XB24-ZB_2364.ebl or XBP24-ZB_2364.ebl)



If you instead dissemble the CPX2 unit and manually reflash the XBee using a serial or USB development board, then make sure you enter the settings below. These should already be handled if you used the web UI:

- Load Destination Address High (DH) with the upper 4 bytes of your CPX4 gateway XBee address
- Load Destination Address Low (DL) with the lower 4 bytes of your CPX4 gateway XBee address
- Set the lower 16-bits of the Device Type Identifier to 0x0003 to define this as a CPX2

As a second step, enter this Setting:

Set the Baud Rate to 115200 (BD=7). This will have defaulted back to 9600 after you reflashed the XBee firmware, plus XCTU might give you an error as you try to set this.

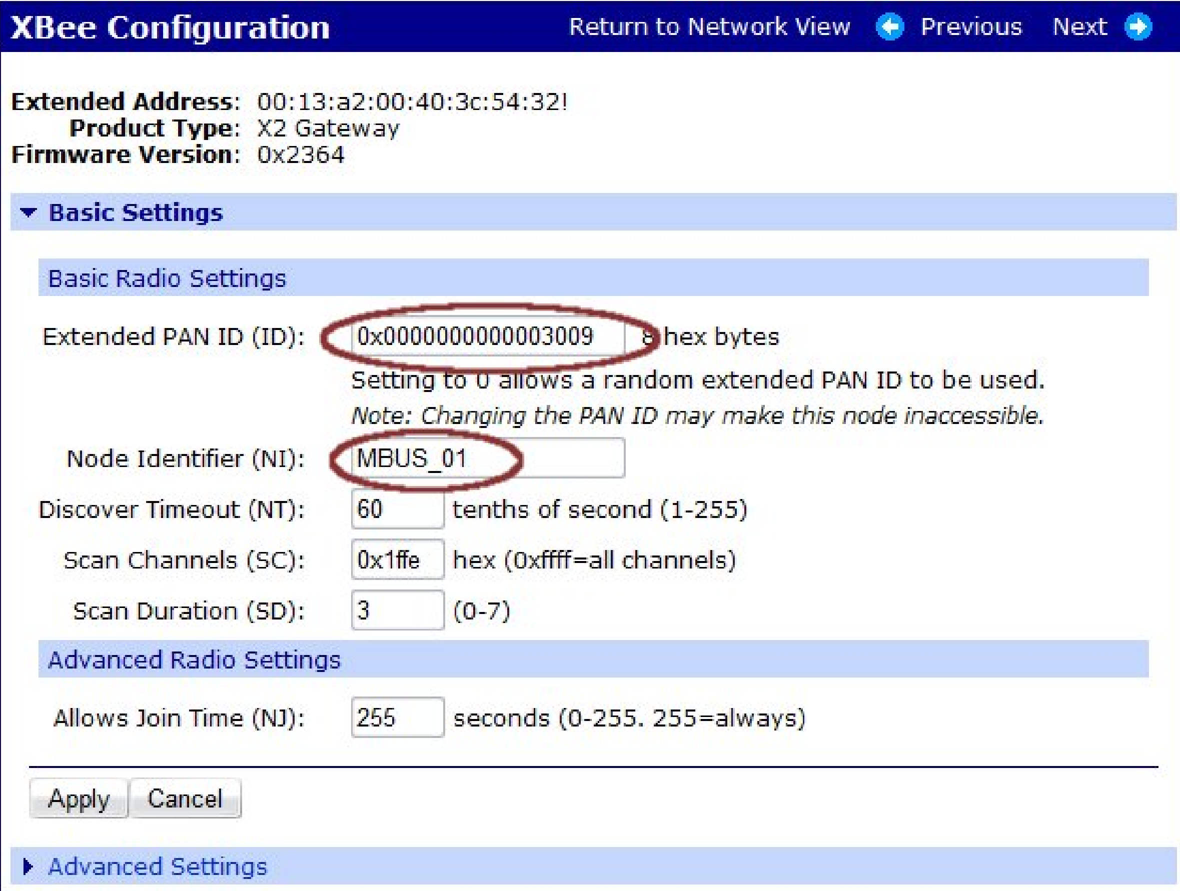
Configure the CPX2's XBee Module

If the CPX2 and XBee are properly configured, then the CPX2 will show up on both the "Network View" of the Digi ConnectPort X4 (the central bridge) and on that of the Digi ConnectPort X2. You can configure the following settings via either the CPX4 or the CPX2:

Network View of the XBee Devices

Node ID	Network Address	Extended Address	Node Type	Product Type
	[fffe]!	00:13:a2:00:40:32:16:4f!	router	
_AI_X4	[0000]!	00:13:a2:00:40:31:aa:a6!	coordinator	X4 Gateway
MBUS_01	[1c70]!	00:13:a2:00:40:3c:54:32!	router	X2 Gateway
MBUS_02	[9891]!	00:13:a2:00:40:48:a7:a4!	router	X2 Gateway
MBUS_03	[0475]!	00:13:a2:00:40:30:de:cd!	end node	RS-485 Adapter
MBUS_04	[b262]!	00:13:a2:00:40:32:15:83!	end node	RS-485 Adapter
MBUS_WR01	[f3f0]!	00:13:a2:00:40:55:6d:30!	router	Wall Router
MBUS_WR02	[0321]!	00:13:a2:00:40:55:6d:47!	router	Wall Router

Click on the appropriate Extended Address (MAC address) link and you will see the Basic settings page:



On the XBee Configuration - Basic Settings page:

- Confirm the Extended Address, Product Type and Firmware are appropriate.
- Manually force the Extended PAN ID to be your fixed value. As long as this value is zero (0x0000000000000000), then your CPX2 may associate with any ZigBee coordinator within RF range. True, it will "prefer" to remain associated to the current CPX4, yet someday it will move. So always hard-code your desired PAN ID. A phone number is a good, memorable PAN ID.
- Add a user-friendly Node Identifier - by default this will be a single SPACE character. If you are doing low-level API message formatting, then you can query device addresses based on this name, however for most users it only makes the gateway XBee display more useful.

Enabling the Python application to do the bridging

To enable the bridging of XBee wireless packets to local Modbus/TCP server/slaves, you'll need to upload the appropriate Python files to your CPX2.

Create the address.txt file

The first generation of this tool supports a single Modbus/TCP server/slave, and its IP address must be uploaded in the text file named "address.txt". The ":502" at the end is optional, but can be used to

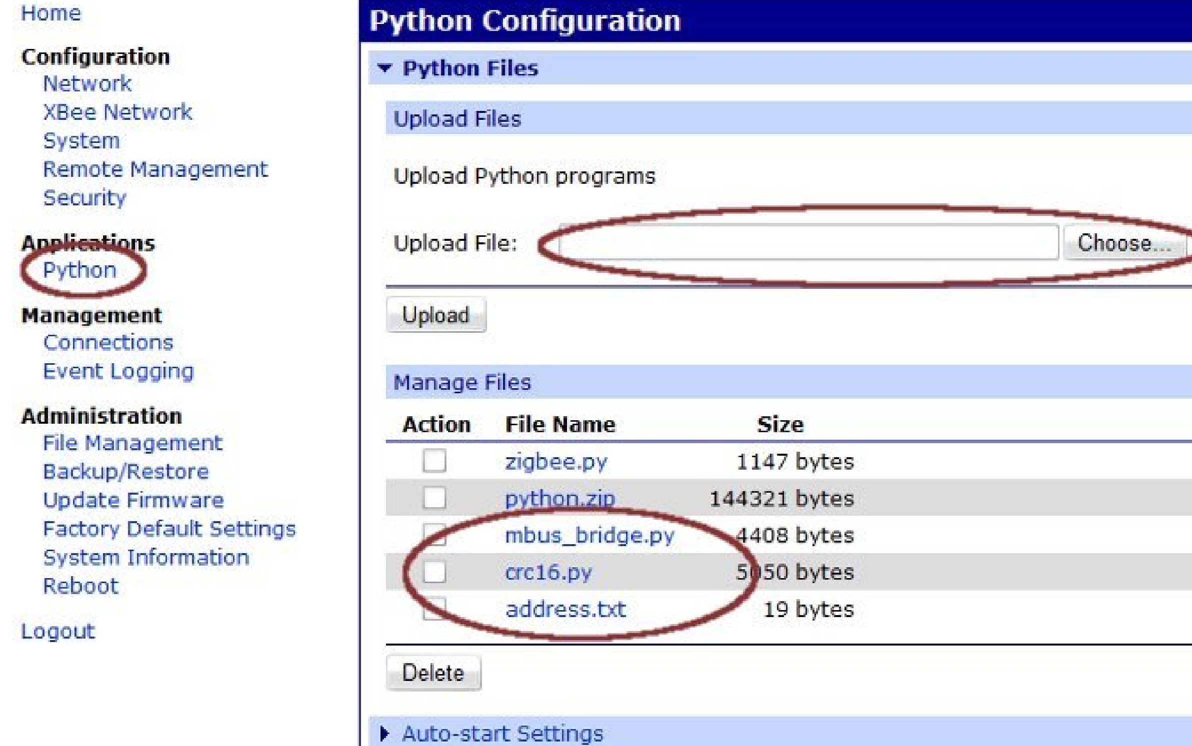
change the default TCP port number to use.

192.168.196.6:502

Uploading the Files

Click the **Python Application** link, then upload the three files: mbus_bridge.py, crc16.py and address.txt. They are contained in this ZIP file: X2_mbus_2009Sep13.zip.

The file list should look similar to that below, however the file sizes may vary:



Enable auto-start of the bridging

After uploading the files, click the Auto-Start Settings section and enter the mbus_bridge.py.

Python Configuration

▸ Python Files

▾ Auto-start Settings

Specify python programs to be run when the device boots.

Enable	Auto-start command line (specify program filename to execute and any arguments)
☑	mbus_bridge.py
☐	
☐	
☐	

Apply Cancel

Note that this Python application closes all sockets and sleeps for 60 seconds anytime there is a fault. So if you are moving Ethernet cables around or doing other disruptive things, the CPX2 might not appear to function for a minute or two.

Modbus Example Serial Adapter

Modbus Bridge Example - Xbee-Serial Adapters

This is the serial node portion of an Modbus Bridge example using a Digi ConnectPort X4 as the central Modbus/TCP bridge and a collection of serial and Ethernet nodes.

To see how to Bridge mesh to Modbus/TCP server/slaves, see this web page: Modbus Example Ethernet Adapter.

Supported Models

Any of the Digi serial adapters can be used as part of your Modbus bridge, including:

- Digi XBee RS-232 adapter (Xbee to RS-232 with DB9 DTE/Male port)
- Digi XBee RS-485 adapter (XBee to RS-422, RS-485 4-wire and RS-485 2-wire with screw terminals)
- Digi XBee RS-232 PH Adapter (Self-Powered Xbee to RS-232 with DB9 DCE/Female port)

Product Installation

Confirm that you have the correct power for your model - probably 3-6vdc (yes, sadly support for 12vdc and 24vdc was discontinued)

Confirm that your adapter has a appropriate level of XBee firmware. If you obtained the XBee adapter at the same time as the X4 gateway, then this should be true.

XBee 232 Adapter Installation

The RS-232 port is like a standard PC or computer port, so the same cables used with a PC should work.

See Digi XBee RS-232 adapter for more information.

XBee 485 Adapter Installation

Select the use of RS-422/RS-485 4-wire or RS-485 2-wire via the DIP Switches, and connect your wires. Pay special attention to the signal ground, for even if the end-device claims pure 4 or 2-wire use without ground, the Digi supplied AC/DC supplies have a floating earth and will NOT communicate reliably without the correct reference-ground connection.

See Digi XBee RS-485 adapter for more information.

Detailed Serial Settings

Although summarized information is given below, more detailed information is given on this page Setting Serial Adapter Baud Rate.

Confirm the XBee adapter is Associated

Covering the entire concept of ZigBee and association is beyond the scope of this Modbus Serial Example, however the XBee serial adapter should show the rapid LED blink of association, plus it should be included on the **Network View of the XBee Devices** on your Digi ConnectPort gateway.

If your system is the only ZigBee network within RF range, this should happen fairly trivially. Just remember that your neighbors might have ZigBee systems, so your XBee adapters might even associate to a ZigBee coordinator in another building.

If you are surrounded by active ZigBee networks, then very likely your XBee adapter will associate with the wrong system. For example, I have from 2 to 6 distinct ZigBee gateways active at any one time, so I am forced to repeatedly press the commissioning push-button 4 times until the adapter associates with the correct Digi ConnectPort. See the Chapter 'Network Commissioning and Diagnostics' in your XBee Module manual for more details.

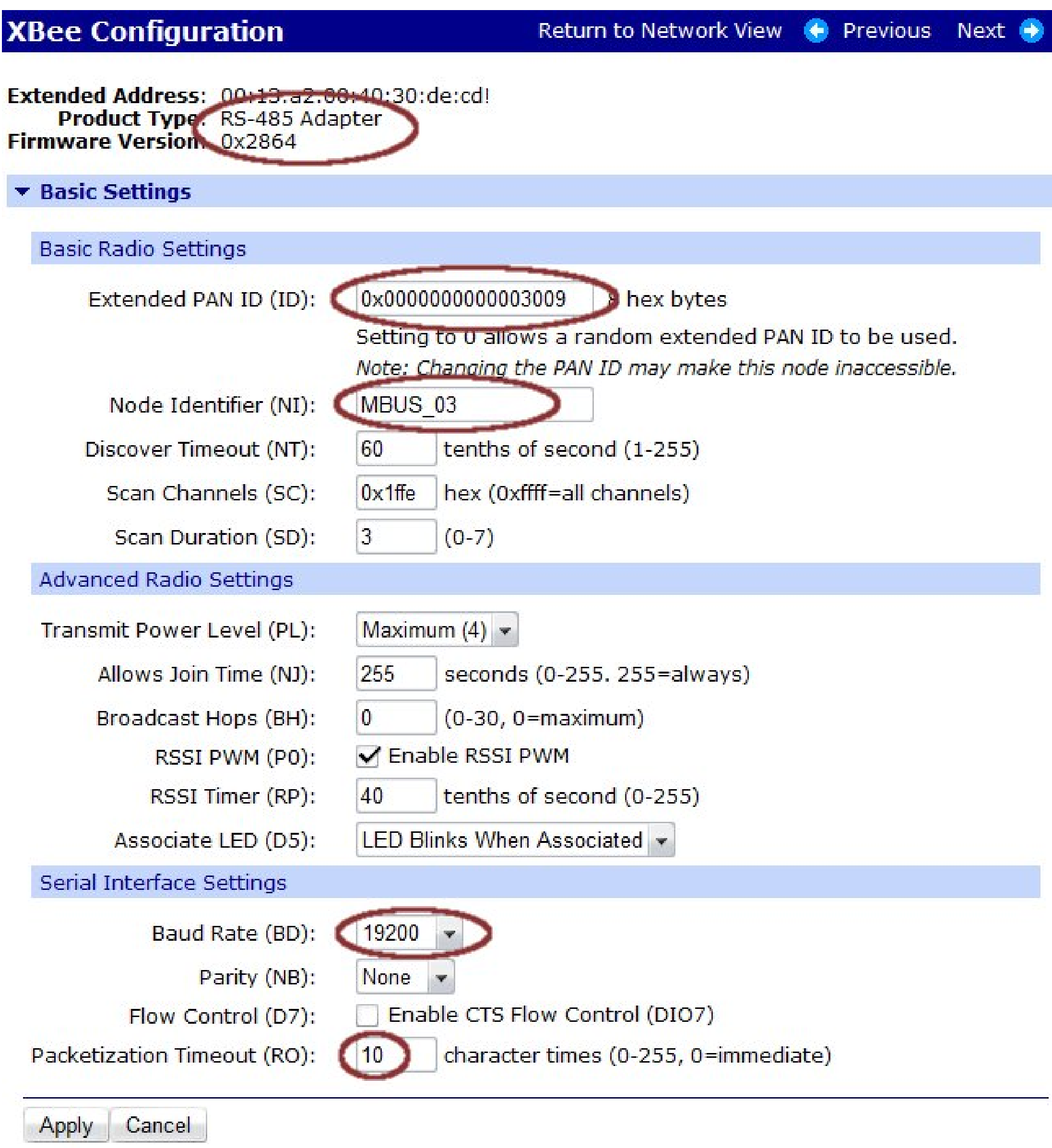
XBee Module Configuration

If your serial adapter is properly associated, it will be show within the Digi ConnectPort's **Network View of the XBee Devices**, plus the 16-bit network Address will NOT be [fffe].

Network View of the XBee Devices

Node ID	Network Address	Extended Address	Node Type	Product Type
	[fffe]!	00:13:a2:00:40:32:16:4f!	router	
_AI_X4	[0000]!	00:13:a2:00:40:31:aa:a6!	coordinator	X4 Gateway
MBUS_01	[1c70]!	00:13:a2:00:40:3c:54:32!	router	X2 Gateway
MBUS_02	[9891]!	00:13:a2:00:40:40:a7:a4!	router	X2 Gateway
MBUS_03	[0475]!	00:13:a2:00:40:30:de:cd!	end node	RS-485 Adapter
MBUS_04	[b262]!	00:13:a2:00:40:32:15:83!	end node	RS-485 Adapter
MBUS_WR01	[f3f0]!	00:13:a2:00:40:55:6d:30!	router	Wall Router
MBUS_WR02	[0321]!	00:13:a2:00:40:55:6d:47!	router	Wall Router

In this example, we'll be working on the node named MBUS_03, so click the blue link-enabled Extended (or MAC) Address to pull up the XBee information for this device. If the device is sleeping or no longer associated, then you will NOT see the dialog show below, but instead will see a warning that "*Settings are not configurable for this device*." This does NOT truly mean that the setting are not configurable, but that an attempt to fetch such settings failed.



On the XBee Configuration - Basic Settings page:

- Confirm the Extended Address, Product Type and Firmware are appropriate.
- Manually force the Extended PAN ID to be your fixed value. As long as this value is zero (0x0000000000000000), then your adapter may associate with any ZigBee coordinator within RF range. True, it will "prefer" to remain associated to the current X4, yet someday it will move. So always hard-code your desired PAN ID. A phone number is a good, memorable PAN ID.
- Add a user-friendly Node Identifier - by default this will be a single SPACE character. If you are doing low-level API message formatting, then you can query device addresses based on this name, however for most users it only makes the gateway XBee display more useful.

- Set the correct Baud Rate and parity. (Sorry, at the moment the Digi Xbee modules do NOT support 7 data bits, so it is not possible to support Modbus/ASCII at 7,E,2)
- The default Packetization Timeout of 3 characters is fine per the Modbus serial standard, however I tend to relax it to 10 (or 25) character times since the added few msec of delay is less significant at an application level than prematurely fragmented response packets.

Remember to press the APPLY button to send the settings down the to XBee Adapter!

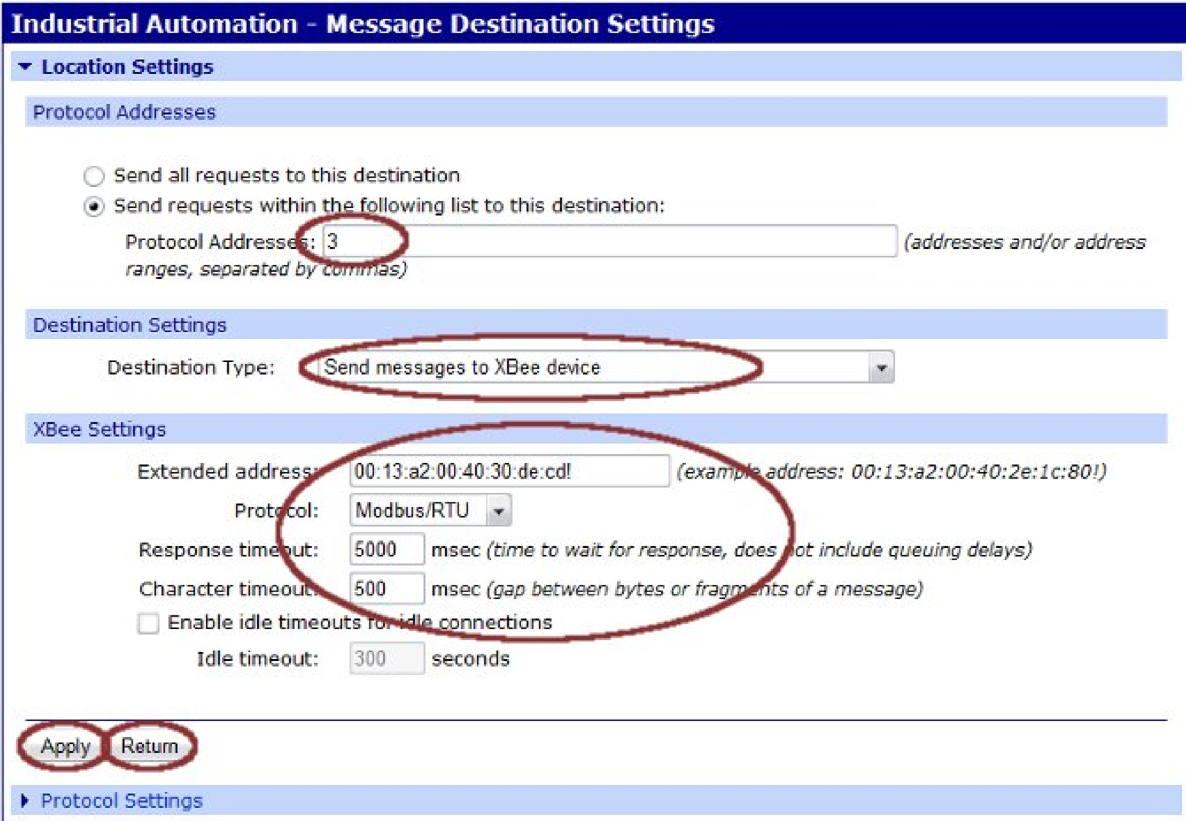
(***Late Breaking info - screen shot to be added***) : If you are using the Digi XBee RS-485 adapter and configuring it through the web UI, uncheck the CTS flow control check-box (turn it off) on the **Basic Settings** page, then go to the **Advanced Settings** page and put a seven (7) into the **DIO7 configuration (D7)**: setting.

If your adapter is set to sleep, pressing the commissioning push-button once (the Node Identification function) will wake the adapter for 30 seconds, which is enough time to READ these settings. Before trying to APPLY or save any changes, press the commissioning push-button again to gain another 30-seconds of wake time.

IA Engine Configuration

At this point your XBee serial adapter should be properly associated, with the correct basic settings forced down and saved to the adapter.

Now add the Modbus Message Destination (or IA route) to the Digi ConnectPort gateway Modbus configuration. See this page if you require step-by-step instructions on adding the IA Message Destination: Modbus bridge on CPX4.



The relevant information on this page are:

- Most likely you only want a subset of Modbus messages to go to this XBee serial adapter. So click the radio button to enable only forwarding selected requests to this destination.
  - If there is a single Modbus device on this XBee adapter, then just enter the correct Modbus serial slave address (the value 3 is shown in the example above).
  - If there are more than 1 Modbus device on this XBee adapter, then enter the Modbus serial slave address as either a range (such as "10-15") or as a scattered list (such as "7,10,15,16")
  - Note that there is NO relationship between the table index (in this example 3) and the Protocol Addresses (which in this example is also 3). The index only defines the order in which the Modbus IA Engine scans to locate the correct destination to answer the Modbus request.
  - If you need to adjust/change the slave address - see below.
- **Destination Type** must be Send messages to XBee device.
- The Extended address must be manually entered as required - there is no automatic way to have this information entered, but you can carefully open a **Network View** in another browser tab and carefully do a cut-and-paste of the address show. If you change the Xbee endpoint, then add the new endpoint after the '!', so if the endpoint is changed away from 0xe8 to 0xe9 to avoid port-bind conflict with Python or iDigi/Dia, then in this example enter the MAC as **00:13:a2:00:40:30:de:cd!E9**.
- Select the appropriate **Protocol** of Modbus/RTU or Modbus/ASCII. *Remember that 7,E,2 cannot be supported with Modbus/ASCII - please complain to Digi sales if you find this as annoying as I do. The Xbee hardware supports it, but Product Management does not believe anyone wants to use such settings.*
- The **Response timeout** must be at least 5000 msec (5 seconds) because the ZigBee mesh at times does route discover and other housekeeping which will cause responses to be returned in about 5 seconds. If you cannot tolerate 5 seconds timeouts, then do NOT use ZigBee - or do NOT use Modbus/RTU. Unfortunately, Modbus serial offers no support for detecting stale responses. So setting the Response timeout to 1 second might result in automation system misoperation as stale (old) data is misapplied to the wrong registers.
- The **Character timeout** should be 500 msec or larger. Again, ZigBee is not a high-performance Modbus transport; the nature of the multi-path, fault-tolerant mesh means transactions take a variable amount of time to complete.

Remember to press the APPLY button to save the settings within the gateway.

To enter a NEW message destination, press the RETURN button to return to previous menu. Editing the values on the current page and pressing APPLY a second time merely CHANGES the values in this current destination - it does not create a new route.

Handling Modbus slave address issues

Suppose you have 10 Modbus/RTU serial devices and they all have the slave address one (1). Can you still use a Digi ConnectPort to bridge to them without forcing a change in the end device?

Yes, you can. From the network standpoint they will all need to have unique Modbus unit id or slave addresses, however the Digi Modbus IA Engine can map or fix-up the address on the outgoing requests. So as example you might chose to call the 10 devices slave address 50 to 59.

Clicking on the **Protocol Settings** section of the Message Destination. You will see the dialog below. By enabling the "Override the Modbus unit address ..." option, you can enter the **Fixed Address 1** on every route. This causes the route for slave address 50 to swap in the value 1 over the XBee mesh, and to restore the address 50 in the responses.

Industrial Automation - Message Destination Settings

Location Settings

Protocol Settings

Override the Modbus unit address on incoming requests with specified unit address:

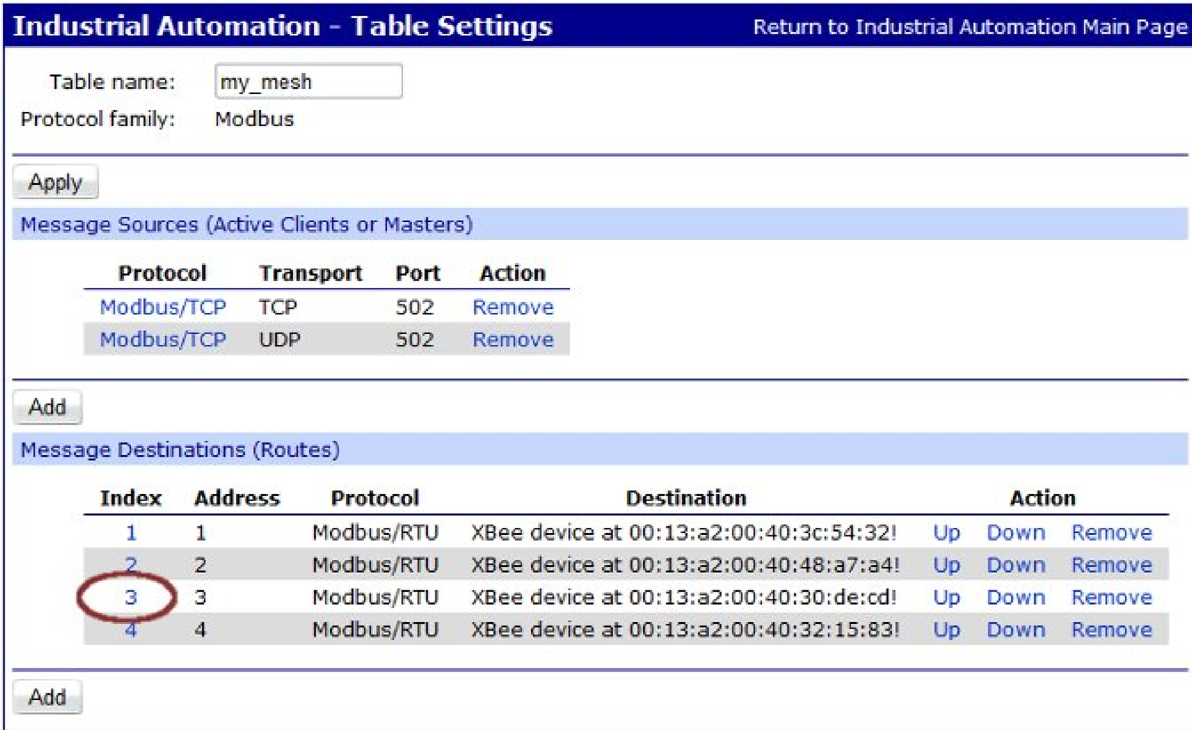
Fixed address: 1

Enable Report By Exception (XMIT) handling.

Apply Return

Summary

This completes your addition of an Xbee serial adapter for use with Modbus serial. The Digi ConnectPort gateway IA Table Settings summary will look like this. Click on the index number to re-open the settings page for this destination. Remember that the IA engine scans this list from top to bottom, sending the message to the first destination with a matching Modbus protocol address. So if you enter the address a the range 0-255 in the first route of 4, none of the other 3 will ever be used.


Industrial Automation - Table Settings
Return to Industrial Automation Main Page
Table name: my_mesh
Protocol family: Modbus
Apply
Message Sources (Active Clients or Masters)
Protocol Transport Port Action
Modbus/TCP TCP 502 Remove
Modbus/TCP UDP 502 Remove
Add
Message Destinations (Routes)
Index Address Protocol Destination Action
1 1 Modbus/RTU XBee device at 00:13:a2:00:40:3c:54:32! Up Down Remove
2 2 Modbus/RTU XBee device at 00:13:a2:00:40:48:a7:a4! Up Down Remove
3 3 Modbus/RTU XBee device at 00:13:a2:00:40:30:de:cd! Up Down Remove
4 4 Modbus/RTU XBee device at 00:13:a2:00:40:32:15:83! Up Down Remove
Add

Modbus Example X4 Setup

Modbus XBee Bridge Example - Setting up the X4

This real-world example includes the following features:

- Digi ConnectPort X4 (CPX4) as a Modbus/TCP bridge. Remote Modbus/TCP clients can access the X4 via either Ethernet or cellular, and the CPX4 fetches the Modbus responses via Xbee mesh
- Digi XBee 485/232 Adapters link serial Modbus/RTU slaves into the bridging system
- Digi ConnectPort X2 (CPX2) link Ethernet-based Modbus/TCP slaves into the bridging system
- Digi Wall Routers (or any powered node) can extend and fortify the robustness of the bridgign system
- Other CPX4 can be located remotely via long-range serial Point-to-Multipoint radios (such as XTend RF Modem). Modbus/RTU can be tunneled up to 40 miles away to other Xbee mesh groups.

Portions of this example can be review on these pages:

- **Setting up an CPX4 as the Modbus/TCP bridge/gateway** (This page)
- Setting up an Modbus Example Serial Adapter for Modbus/RTU slaves
- Setting up a Modbus Example Ethernet Adapter for Modbus/TCP Ethernet slaves
- Setting up Wall Routers
- Tunnelling Modbus between CPX4 (info to be added in the future)

ConnectPort X4 Installation

Setting an Appropriate IP into the CPX

You first need to get an appropriate IP into your ConnectPort gateway, which most likely is a fixed IP on your local IP subnet. Some ConnectPort gateway products default to DHCP clients, which other default to 192.168.1.1. Fortunately, the Digi Device Discovery tool allows you to locate and connect to your Digi device without changing the IP in your computer.

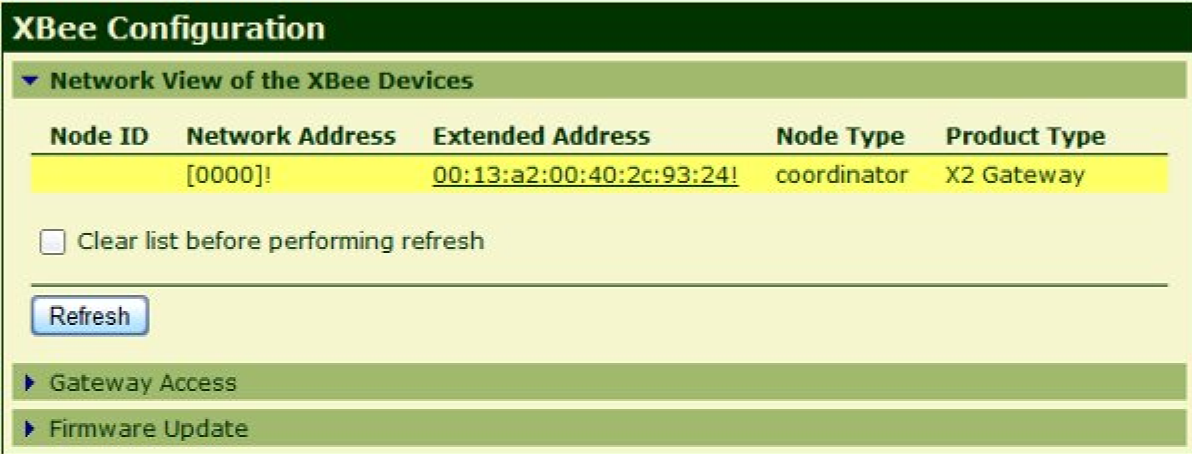
- Confirm that you have the correct power for your model - the default supplied is 12 vdc and the CPX4 support 9 to 30 vdc.
- Connect the CPX4 via an Ethernet Hub or Switch to a PC running Windows; A "Cross-Cable" will probably work, however some notebooks have problems negotiating Ethernet hardware modes with other end-devices.
- Turn off the Windows firewall on your Windows computer
- Install the Digi Device Discover tool (see support.digi.com - It is part number 40002256).

- To repeat! **Make sure your Windows firewall is off**, then you can discover Digi devices with IP of 0.0.0.0 (no IP), 169.254.x.x (auto-ip), and 192.168.1.1 (a hard-coded IP which might NOT match your own subnet). The Digo Device Discovery tool will allow you to hard-code a suitable IP (if required) and after a Digi device reboot, you can log into the web interface.

Setting an Appropriate XBee PAN id into the CPX

By default your gateway's XBee network comes up with an unknown or random PAN id - think of it as your 'network call-name'. This is fine if you are the only person in a few miles using ZigBee - and if you only have 1 such gateway. However, as soon as there are two Xbee networks within RF sight of each other, you'll need to force them to build separate PANs (or network-groups). Even if today you have the sole gateway and everything builds fine, at any moment your neighbor or a colleague might add a second system, and then any power outage risks you losing XBee devices to other networks. **So you always want to hard-code in your own unique PAN id!**

Click the **Configuration > XBee Network** link at the left, and you will see something like this:



Note These screen shots are from a ConnectPort X2, not an X4 - the same concepts apply. Also the colors are modified by a custom CSS (or cascading style sheet) - which is a little known feature of the Digi ConnectPort line.

- You'll see the dialog below, under which you'll want to:
  - Enter a fixed unique Extended PAN ID - a phone number works well
  - Enter a Node Identifier. Since these are sorted in alpha order, starting it with an '_' forces your gateway to always be listed at the top of your list.
  - Press the Apply button to save your settings.

XBee Configuration
Return to Network View
Previous
Next
Extended Address: 00:13:a2:00:40:2c:93:24!
Product Type: X2 Gateway
Firmware Version: 0x2163
Basic Settings
Basic Radio Settings
Extended PAN ID (ID): 0x0000009529123400 8 hex bytes (Setting to 0 allows a random extended PAN ID to be used)
Node Identifier (NI): _QubiLyn
Discover Timeout (NT): 60 tenths of second (0-252)
Scan Channels (SC): 0x1ffe hex (0xffff=all channels)
Scan Duration (SD): 3 (0-7)
Advanced Radio Settings
Transmit Power Level (PL): Maximum (4)
Allows Join Time (NJ): 255 seconds (0-64. 255=always)
Apply
Cancel
Advanced Settings

Modbus Floating Points

Moving 32-bit Floating Points under Modbus

Unfortunately, the history of 32-bit floating points under Modbus is one of ad-hoc solutions. It is impossible to define how any new Modbus device you obtain will handle 32-bit floating points.

Method #1: two 16-bit holding registers, low-word first

The Modicon 984E PLC with the enhanced math module upgrade began the wide use of 32-bit floating points under Modbus. Because these PLC used little-endian processors and the 32-bit floating points were placed directly in the normal 4x read/write holding register space, 32-bit floating points moved under Modbus as two consecutive registers, the low-word in the first register. So the bytes within the words are still big-endian, while the words are psuedo-little-endian.

Method #2: two 16-bit holding registers, high-word first

Other vendors approaching the same problem took to heart the "Modbus is Big-Endian" mantra and returned 32-bit floating points as two consecutive registers, with the high-word in the first register.

Method #3: one 32-bit special register, 32-bit big-endian

Still other vendors used a gray area within the Modbus standard, encoding 32-bit floating points in a manner neither allowed nor forbidden by the standard. They reasoned that since the read response includes a byte count, the master/client could understand that a read of 10 registers resulting in 40 bytes of data meant that the registers were 32-bit and not 16-bit. Thus these vendors return 32-bit floating points as double-sized (32-bit) registers in true big-endian format. These vendors tend to predefine ranges of registers types, so while reading 10 holding registers starting at 4x00001 returns 20 bytes, reading 10 holding registers starting at 4x07001 returns 40 bytes.

Bottomline: Method should be configurable

Any master/client wishing to read 32-bit floats from any arbitrary Modbus slave/server must allow user configuration to select which of these 3 methods to assume. In reality, method #1 is the most universally supported. In fact, most slave/server devices natively supporting methods #2 or #3 will include a user setting perhaps labeled "Modicon Compatibility Mode" to force method #1. Without this setting, the devices cannot be integrated with the widely used Modicon 984, Quantum and Momentum PLC.

Python Code

This code fragment converts a 32-bit floating point into a 4-byte binary string in one of three forms:

```
if(form in ['l','L']):
  # then in true Little-Endian form
  # example: 100.0 is 0x00 0x00 0xC8 0x42
  return struct.pack("<f", data)

else: # one of the big-endian forms
  fval = struct.pack(">f", data) # get true big-endian

  if(form in ['m','M']):
  # then in modbus lo-word first form
  # example: 100.0 is 0x00 0x00 0x42 0xC8
```

```
    return fval[2]+fval[3]+fval[0]+fval[1]

    # else in true Big-Endian form
    # example: 100.0 is 0x42 0xC8 0x00 0x00
    return fval
```

See Also

Here is a good online calculator which shows the 32-bit and 64-bit hex form of IEEE floating point values. It also breaks out the components, showing the exponent and significand fields. This information helps you decode misaligned or unknown binary strings.

http://babbage.cs.qc.edu/IEEE-754/Decimal.htmll

Modbus Serial Over Mesh

Modbus Serial Over Wireless and Mesh

In many ways a mesh (or point-to-multipoint) works much like an Ethernet network. Small UDP/IP-like data packets can be moved to a specific remote device. At the same time, the actual contents of the message is not of importance to the wireless nodes - they could be ASCII or Chinese text - or serial Modbus.

Limitations with Modbus

- Some device must be able to map the Modbus/RTU slave address (aka Modbus/TCP Unit Id) to the network address of the wireless mesh. The nature of mesh makes broadcast a non-scalable solution, so the mesh cannot mimic an RS-485 multi-drop. This is not a serious problem, as Digi has off-the-shelf solutions to map Modbus/TCP through the mesh to serial Modbus devices.
- The historical Modbus/RTU definition of a 3.5 character gap as end-of-message has caused problems with radio/wireless for decades, as the technology moved to 'packet networks' designs. Since the normal radio packets size is in the 64 to 128 range, invariably artificial time gaps may appear in the remote Modbus/RTU serial request. These gaps cause the slave device to discard the halves of the request as badly formed packets.
  - Fortunately, the latest **Digi XB24-ZB firmware supports the marking of packets as fragments**, which allows the remote XBee-to-serial device to reassemble the Modbus/RTU request to be sent gap-free. This firmware is coded as 2x6x, so as the example the X4 coordinator might run Zigbee Coordinator API version 2162, while the XBee RS-232 adapters would run Zigbee Router AT version 2262. Note that the fragmentation support only works in the host to field direction - technically, only API-enabled XBee knows how large the oversized packet is, thus can create the fragmentation headers. The AT-enabled XBee never knows how large the serial message being received will be, thus it sends as several small stand-alone packets.
  - An alternative solution is to use a Master/OPC Server tool which allows limiting the Modbus read and write size. For example any full-function OPC server allows the user to define the maximum read/write size as 30 registers, which allows full requests and responses to fit within one radio packet.
- Enabling the X4/X8's built-in Modbus routing function cannot share the mesh with Python programs.
- If the Modbus function opens the mesh, then Python programs will not be able to access the mesh directly.
- Conversely, if a Python program opens the socket(AF_ZIGBEE, ...) to control the mesh access, the the Modbus routing cannot use the mesh.

- However to speak pure Modbus over the Mesh, your Python program can act as a normal Modbus master and connect via TCP or UDP to localhost. This nicely offloads the timing and error recover from your Python code and places it upon the mature IA Engine. See Integrating the Digi IA Modbus bridge to Python for discussion and examples of such linking

What can be Done with Modbus

- Either Modbus/RTU or Modbus/ASCII can be moved over wireless. Given the maximum byte count per packet is limited to the range of 50 to 100 bytes, Modbus/RTU works better since it accomplishes the job with less than half the bytes of Modbus/ASCII.
- The IA Engine enabled Digi gateways such as the CPX4/X8 allow up to 32 remote Modbus masters to connect via:
  - Modbus/TCP in TCP/IP or UDP/IP, from either Ethernet or cellular IP
  - Modbus/RTU in TCP/IP or UDP/IP, from either Ethernet or cellular IP
  - Modbus/ASCII in TCP/IP or UDP/IP, from either Ethernet or cellular IP
  - Modbus/RTU or Modbus/ASCII on the gateway's serial port (but Masters may not attach by Zigbee/Mesh at this time)
- The IA Engine then uses the Modbus slave address/unit id to look up the MAC address of the XBee node servicing that Modbus serial device.
- The IA Engine also manages the timing and the multi-master contention.
- Python programs can also leverage the IA Engine to offload the complex timing and error recover. Literally, a Python program using Modbus/TCP form on a localhost UDP socket will always receive either a complete response or a timeout with Modbus exceptions 0x0A or 0x0B. See this Wiki page for discussion and examples of such linking

Timing Issues and Modbus

Gap Times - Intercharacter Timeout

Traditional Modbus/RTU serial devices have either a fixed gap detect of about 3-4 msec, or one which can be configured up to 1 second or more. Unfortunately a mesh with route discover and retries, gaps between packs could reach upwards towards 5 seconds!

Sending Modbus requests fitting within a single packet is always safe, as the remote XBee module send the serial data cleanly to the slave. Use of the newer Xbee firmware with fragmentation support improves this as it allows the Xbee to hold and send the full Modbus request upon receipt of the full packet. Yet this further delays the final response since the entire request does not begin its byte-by-byte serial shifting until all fragments are accounted for - thus the serial slave might not start receiving an oversized request for nearly 400-500 msec after the host sent the request. Keep this in mind in your timing estimations!

Although the Digi gateway cannot control the gap-timeout in the remote slave device, the Modbus drivers receiving the master requests or slave responses eliminate the gap-timeout by intelligently estimating the response size. This allows a very large gap timeout to be configured, yet bypassed when messages of known Modbus functions are handled. Therefore a Modbus/RTU function 3 request is known to be 8 bytes in length, while the function 3 response is 5 bytes longer than the byte count being returned. More information on this intelligent handling of Modbus is explained here.

Response times

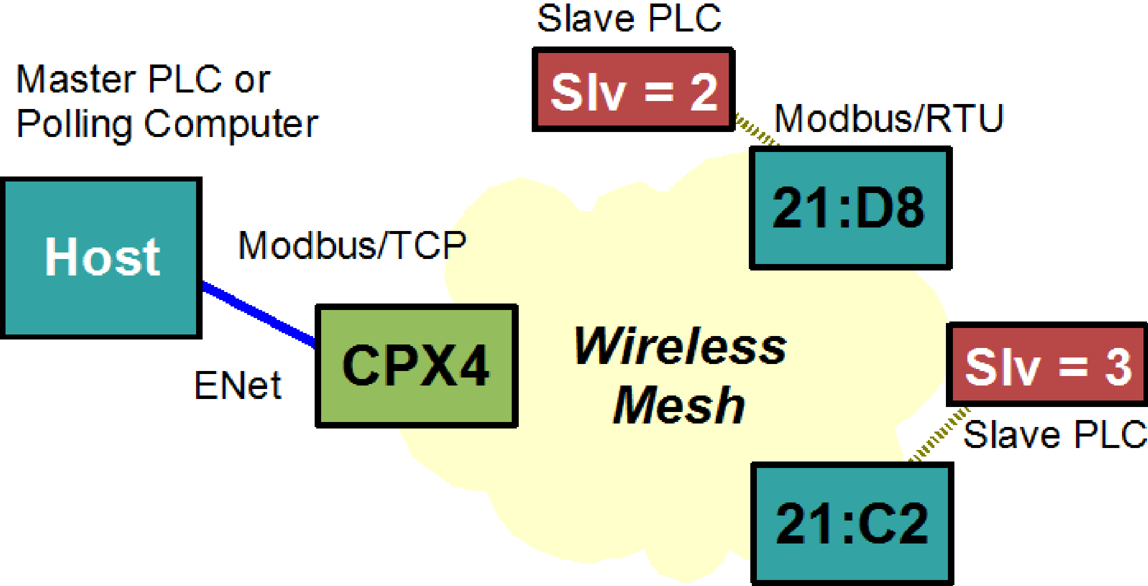
Traditional Modbus/RTU serial devices have response timeouts in the 500 msec to 1 second range, and in general a fast embedded device will begin responding in the 10 to 100 msec. However, over a mesh responses may come back more slowly given that messages may be store-and-forwarded through many router, plus retries and mesh route discovery can all add latency.

Seeing Actual Response Times

Fortunately the Digi CPX4 gateway retains basic response statistics, so you can run the system for a week with very long timeouts, then examine the statistics to discover the actual performance. How to access this information is on the digitips wikispace.

Simple Example

Below is a simple example of a Modbus/TCP master wishing to poll two Modbus/RTU serial slaves via a Digi ConnectPort X4 gateway. Only the last 2 digits of the 8-byte Xbee extended MAC addresses are shown.



The actual configuration is shown on this wiki page, but in summary to accomplish this, you need to configure the following things:

- Enable incoming Modbus/TCP messages on TCP/IP port 502
- Enable incoming Modbus/TCP messages on UDP/IP port 502 (recommended, but optional)
- Enable a Modbus destination table to hold the address/unit-id to MAC address mapping
- Add a route to Modbus slave address 2 to mesh extended address [00:13:a2:00:40:4a:21:d8]!
- Add a route to Modbus slave address 3 to mesh extended address [00:13:a2:00:40:4a:21:c2]!

Note that to maximize the flexibility of the gateway design, the X4/X8 do not attempt to configure or verify the settings in the remote mesh node - it might not even be a Digi products. Thus you must configure baud rates and other settings using a tool external to the IA Engine. If the remote serial adapter is a Digi product, you can use XCTU or Setting Serial Adapter Baud Rate.

Note that currently Digi's XBee modules only support 8 data bits, Even/Odd/None/Mark parity and 1 stop bit. XBee firmware to release this summers adds support for 2 stop bits, but you still cannot use 7,E,2 or 7,O,2.

Modbus starting page

A guide and index to Modbus resources in this Wiki

External resources

What is Modbus? It is a 30 year old protocol which is best described as a "remote memory table access protocol". It literally allows a client/master to go to a server/slave and ask "return 16-bit words, starting at offset 23 and include 4 words". Modbus places no interpretation on that data; it assumes the client/master and server/slave AGREE as to the meaning. So 4 words of data might consist of:

- One word with 16 status (on/off) bits
- One word of unsigned data with only valid meaning from 0 to 10,000
- Two words as a 32-bit float, however Modbus doesn't even promise WHICH of the two words is the upper word and which is lower word

The Modbus specification is here: http://www.modbus-ida.org.
Here is background info on Modbus: http://iatips.wikispaces.com/Modbus.

Python Code Samples

At present there is no complete "Modbus Library" since by itself it cannot do anything - thus one needs to define how one wants Modbus to integrate with a Python application before one can have a library. Here are Wiki pages which explain the basic concepts for creating Modbus requests under Python:

- Python CRC16 Modbus DF1 : How to calculate the 16-bit CRC used by Modbus/RTU and Allen-Bradley DF1
- How to create Modbus/RTU request in Python: Creating a basic Modbus/RTU message as a binary-string
- Modbus class design in Python: The class hiearchy related to Modbus PDU, TCP, RTU and ASCII
- Modbus Floating Points: How to encode 32-bit floating points in Modbus

Leveraging the Modbus Bridge in the WAN IA / X4 / X8 products

Some Digi products include a powerful multi-master Modbus Bridge which freely converts between Modbus/TCP, Modbus/RTU and Modbus/ASCII. Thus your Modbus Python code can be quite simple, yet have full function. This page explains the integration:

Integrating the Digi IA Modbus bridge to Python

Modbus ZB Fragmentation Support

Enabling ZigBee Fragmentation Support for Modbus

By default, ZigBee networks move only 60 to 80 bytes per packet. This becomes a problem during Modbus serial encapsulation because a large message written over ZigBee will arrive in many smaller fragments, and might suffer an unknown time-gap between the serial bytes actually sent to the Modbus/RTU slave. If those time-gaps are larger than a few milliseocnds, then the Modbus/RTU slave might reject the fragments (the partial messages) one-by-one as having invalid CRC. For example, writing 200 bytes will result in at least 3 separate RF packets being sent.

If you are running XBee firmware 2x6x (for example XBee firmware 0x2164 within the CPX4 and 0x2264 within the XBee 232 or 485 Adapter), then the Digi IA/Modbus engine can make use of ZigBee's fragmentation support.

This allows the X4 to send the Modbus request with flags set to instruct the XBee 232 or 485 Adapter to HOLD all data until all fragments arrive. So in the example of a 200 byte write, then XBee 232 or 485 Adapter might receive the first fragment of 70 bytes, and understand that more is coming. It would then receive perhaps another 70 bytes, then the final 60 bytes and only start sending the serial Modbus/RTU message once all fragments have been received.

Note that this WILL slow down end-to-end communications since no serial data is sent to the Modbus/RTU slave until all bytes have moved across the RF link.

Via web Interface

At the moment this setting is not available within the web Interface.

Via Telnet or CLI

Using telnet to log into the ZigBee-enabled Digi gateway (such as an X4 or X8), running the **show ia** command will display your Modbus configuration:

```
#> show ia
set ia table=1 state=on name=my_mesh family=modbus accessmode=multi
set ia table=1 ownerperiod=15000

set ia table=1 addroute=1
set ia table=1 route=1 active=off type=zigbee protaddr=1 protocol=modbusrtu
set ia table=1 route=1 address=00:13:a2:00:40:0a:49:ad! max_mtu=auto
set ia table=1 route=1 slavetimeout=5000 chartimeout=500 idletimeout=0
set ia table=1 route=1 lineturnmode=off fixedaddress=0 rbx=off

set ia table=1 addroute=2
set ia table=1 route=2 active=off type=zigbee protaddr=2 protocol=modbusrtu
set ia table=1 route=2 address=00:13:a2:00:40:0a:14:4d! max_mtu=auto
set ia table=1 route=2 slavetimeout=5000 chartimeout=500 idletimeout=0
set ia table=1 route=2 lineturnmode=off fixedaddress=0 rbx=off

IA Serial: Nothing Configured

set ia master=1 active=off type=tcp ipport=502 protocol=modbustcp table=1
set ia master=1 priority=medium messagetimeout=5500 chartimeout=50
set ia master=1 idletimeout=0 lineturnmode=off errorresponse=on
set ia master=1 broadcast=replace

set ia master=2 active=off type=udp ipport=502 protocol=modbustcp table=1
```

```
set ia master=2 priority=medium messagetimeout=5500 chartimeout=50
set ia master=2 idletimeout=0 lineturnmode=off errorresponse=on
set ia master=2 broadcast=replace
```

Notice the setting **max_mtu=auto** in both ZigBee route. This means the IA/Modbus Engine will break large Modbus packets into sizes based on the actual RF performance. You want to enter this command:

```
set ia table=1 route=1 max_mtu=255
set ia table=1 route=2 max_mtu=255
```

You can also enter a single line as "**set ia table=1 route=1-2 max_mtu=255"**.

Warnings

If either the XBee firmware in the gateway or in the XBee 232/485 adapter is too old to support fragmentation, then setting max_mtu=255 might will cause total communications failure.

Also, the fragmentation support only functions from "API-enabled" XBee to other API or AT-enabled XBee. As of September 2009 the XBee AT Router or End-Device firmware RECEIVES and handles fragmentation packets, but never sends them. Thus large Modbus READ responses from the serial adpater to the CPX gateway will always be fragmented, and the Digi IA/Modbus Engine will reassemble the Modbus response at the application level.

Modbus class design in Python

Designing a class system for Modbus

The Modbus protocol is actually 3 forms of encoding of the same core command - referred to by [modbus-ida.org] as a Protocol Data Unit or PDU. Once encoded, the messages become (per Modbus-IDA) an Application Data Unit or ADU. Unfortunately this usage of terms is confusing in an ISO/OSI world since the ADU is the smallest core user data, and not one padded with protocol-specific overhead. But then Modbus is older than ISO/OSI.

Here are example bytes for the function 3 call to node 1 to read 10 words starting at 4x00123 (the 123rd word, which is 122 on the wire)

- **Modbus PDU** as hex, 0x01 03 00 7A 00 0A
- **Modbus/RTU** as hex, 0x01 03 00 7A 00 0A (CRCx2)
- **Modbus/ASCII** as hex, 0x3A,30 31,30 33,30 30,37 41,30 30,30 41 (BCC) 0D 0A (is ASCII string ":0103007A000A??\r\n"
- **Modbus/TCP** as hex, 0x12 23 00 00 00 06 01 03 00 7A 00 0A

So the natural class design for any Modbus server or client consists of a base class called **modbus_pdu**, plus three children classes called **modbus_tcp**, **modbus_rtu**, and **modbus_ascii**.

Modbus Server Routines

The actual TCP or UDP handling routines are fairly standard; loop, accept connection and requests, then return response. So the first Modbus-specfic routine for our modbus_tcp and modbus_serial classes to provide is a **test_if_complete_message(bytes_seen)**.

- For class modbus_tcp we first confirm we have a full 6 byte header, then we can parse the length field and confirm we have the correct number of bytes of attached modbus_pdu. Be careful though - over wide-area-network TCP segments might become fragmented, so you cannot assume an entire request arrives at one time. Also, some applications concatenate (pipeline) multiple requests into one segment, so a TCP packet with 36 bytes of data might contain three complete 12-byte requests.
- For class modbus_rtu the task is more difficult. The formal definition for Modbus/RTU ending with a pause of 3.5 'character times' is meaningless over wide-area-network. The best solution - since this is YOUR server - is just to make a function-aware tool. You know which functions you support, so confirm each supported function is the correct length
- For class modbus_ascii the task is simpler again; anything starting with a ":" character and ending in the two characters "\r\n" is considered complete.

Modbus on Digi Products

Products (families) With Modbus Bridge Capability

Abreviation	Products and Families
DOIAP	Digi One IAP] and One One IAP HAZ
DOIA	Digi One IA]
DOSP	Digi One SP or SP IA]
TSn	Digi PortServer TS1 to TS16]
WANIA	Digi Connect WAN IA]
DCME	Digi Connect SP, WiSP, ME, EM, WiME, WiEM
CPX2	Digi ConnectPort X2 Ethernet to Mesh Gateway - use 82001631 firmware instead of 82001596]
WiCPX2	Digi ConnectPort X2 WiFi to mesh Gateway - use 82001597 firmware instead of 82001630]
CPX4	Digi ConnectPort X4 and X8 Gateway]
CP4x4	Digi ConnectPort 4x4 (4 serial, 4 Ethernet Switch ports)

Note The Digi One SP-IA has no Modbus Bridge function. It is a standard Digi One SP with a DIN-rail bracket and power pig-tail only.

Matrix (see feature description below Matrix)

Feature	DOIAP	DOIA	DOSP	TSn	WANIA	DCME	CPX4
Multi-Master	Yes	Yes	**-NO-**	Yes	Yes	Yes	Yes
Max Active TCP Sockets	64	32	1	64	32	32	32
Max Routes	128	32	**-NO-**	128	32	32	32
Modbus/TCP to Modbus/RTU	Yes	Yes	**-NO-**	Yes	Yes	Yes	Yes
Modbus/TCP to Modbus/ASCII	Yes	Yes	**-NO-**	Yes	Yes	Yes	Yes

Feature	DOIAP	DOIA	DOSP	TSn	WANIA	DCME	CPX4
Modbus/RTU to Modbus/ASCII	Yes	Yes	**-NO-**	Yes	Yes	Yes	Yes
All Modbus in UDP/IP	Yes	Yes	**-NO-**	Yes	Yes	Yes	Yes
Bridge Modbus with Digi RealPort	Yes	Yes	**-NO-**	Yes	UDP-only	UDP-only	UDP-only
Serial Modbus via Zigbee	**-NO-**	**-NO-**	**-NO-**	**-NO-**	**-NO-**	**-NO-**	Yes
Python Integration	**-NO-**	**-NO-**	**-NO-**	**-NO-**	Yes	**-NO-**	Yes

Note Python in newer WANIA with 16MB of RAM only.

Feature Notes

- **Multi-Master:** Means the Digi acts as a proxy-Master/Slave. Remote Masters see the Digi as a slave. Remote Slaves see the Digi as a master. The Digi accepts requests from many Master concurrently, queues and time-stamps these requests, and tries to obtain responses. The Digi product is fully "Modbus Transaction" aware and keeps the Masters requests straight and doesn't mix them up.
- **Max Active TCP Sockets:** This defines several limits in a Modbus Bridge. Incoming TCP/IP master connections and outgoing TCP/IP slave connections compete for these sockets. So as example the DOIAP can have 64 active TCP-Masters and no active TCP-Slaves ... or 32 active TCP-Masters and 32 active TCP-Slaves ... or no active TCP-Masters and 64 active TCP-slaves. Commonly OPC servers open ONE SOCKET for every slave on an RS-485 multi-drop, this also limits the number of slaves such a Master can access. So for example, the DOIAP could support 64 serial slaves and the DOIA could support 12 only.
- **Modbus/TCP to Modbus/RTU:** Allows Modbus/TCP masters to poll Modbus/RTU slaves; Also Modbus/RTU masters to poll Modbus/TCP slaves. The Modbus/RTU can be by direct serial or encapsulated within TCP/IP or UDP/IP.
- **Modbus/TCP to Modbus/ASCII:** Allows Modbus/TCP masters to poll Modbus/ASCII slaves; Also Modbus/ASCII masters to poll Modbus/TCP slaves. The Modbus/ASCII can be by direct serial or encapsulated within TCP/IP or UDP/IP.
- **Modbus/RTU to Modbus/ASCII:** Allows Modbus/RTU masters to poll Modbus/ASCII slaves; Also Modbus/ASCII masters to poll Modbus/RTU slaves. The Modbus/RTU or ASCII can be by direct serial or encapsulated within TCP/IP or UDP/IP.
- **All Modbus in UDP/IP:** Modbus/TCP, RTU, and ASCII can be moved as freely by UDP/IP as by TCP/IP. Some competitors' products only support TCP/IP, not UDP/IP.

- **Bridge Modbus with Digi RealPort:** Host applications can open virtual RealPort serial ports, which indirectly encapsulates Modbus/RTU or Modbus/ASCII into TCP/IP. The Digi products marked "YES" allow this Modbus to be treated as a normal serial Modbus master feeding into the bridge. The Digi products marked "-NO-" do support Digi RealPort, however this function has not yet been modified to allow the "Modbus Bridge" code to act as a middle-man ... the Digi RealPort "plumbs" directly to the product serial ports bypassing the Bridge.
- **Serial Modbus via Zigbee:** Support a route-type of Zigbee, which allows a Modbus 8-bit Unit Id (or slave address) to be mapped to a Zigbee MAC address. It is assumed a serial Modbus slave is attached to the Zigbee node.
- **Python Integration:** Support builtin Python environment, which can easily be linked via 'localhost' ports to build simple Modbus server/client tools.

Product Feature Notes

- **Digi One IA:** The next firmware release for the Digi One IA will MATCH the functionality of the DCWAN family, so the number of active sockets will go up to 32, etc.
- **Digi One SP and SP IA:** This product is just here for completeness. It has no protocol awareness, so can only move serial Modbus via Digi RealPort or as encapsulated in TCP/IP or UDP/IP. The Digi One SP IA runs the same firmware as the Digi One SP - what gives it the "IA" name is the DIN-rail bracket supplied and the power pig-tail instead of a normal 110vac "wall-wart" power supply.

Python CRC16 Modbus DF1

Module: CRC16

Both the Modbus/RTU and DF1 protocols use the same CRC16 calculation, however you'll note one is 'forward' and one 'reverse' because Modbus starts with a CRC of 0xFFFF, which DF1 starts with 0x0000. The module below uses a 256-word look-up table of partially prepared answers to greatly reduce the system load.

Example

See the "__name__ == __main__" self-test portion of the module for examples of Modbus/RTU and DF1 usage.

```
            #*        File: CRC16.PY
#*              CRC-16 (reverse) table lookup for Modbus or DF1
#*

INITIAL_MODBUS = 0xFFFF
INITIAL_DF1 = 0x0000

table = (
0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040 )

def calcByte( ch, crc):
```

```
    """Given a new Byte and previous CRC, Calc a new CRC-16"""
    if type(ch) == type("c"):
        by = ord( ch)
    else:
        by = ch
    crc = (crc >> 8) ^ table[(crc ^ by) & 0xFF]
    return (crc & 0xFFFF)

def calcString( st, crc):
    """Given a bunary string and starting CRC, Calc a final CRC-16 """
    for ch in st:
        crc = (crc >> 8) ^ table[(crc ^ ord(ch)) & 0xFF]
    return crc

if __name__ == '__main__':

    # test Modbus
    print "testing Modbus messages with crc16.py"
    print "test case #1:",
    crc = INITIAL_MODBUS
    st = "\xEA\x03\x00\x00\x00\x64"
    for ch in st:
        crc = calcByte( ch, crc)
    if crc != 0x3A53:
        print "BAD - ERROR - FAILED!",
        print "expect:0x3A53 but saw 0x%x" % crc
    else:
        print "Ok"

    print "test case #2:",
    st = "\x4b\x03\x00\x2c\x00\x37"
    crc = calcString( st, INITIAL_MODBUS)
    if crc != 0xbfcb:
        print "BAD - ERROR - FAILED! ",
        print "expect:0xBFCB but saw 0x%x" % crc
    else:
        print "Ok"

    print "test case #3:",
    st = "\x0d\x01\x00\x62\x00\x33"
    crc = calcString( st, INITIAL_MODBUS)
    if crc != 0x0ddd:
        print "BAD - ERROR - FAILED!",
        print "expect:0x0DDD but saw 0x%x" % crc
    else:
        print "Ok"

    print
    print "testing DF1 messages with crc16.py"

    print "test case #1:",
    st =
"\x07\x11\x41\x00\x53\xB9\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    # DF1 uses same algorithm - just starts with CRC=0x0000 instead of 0xFFFF
    # note: <DLE><STX> and the <DLE> of the <DLE><ETX> pair NOT to be included
    crc = calcString( st, INITIAL_DF1)
    crc = calcByte( "\x03", crc) # final ETX added
    if crc != 0x4C6B:
        print "BAD - ERROR - FAILED!",
```

```
        print "expect:0x4C6B but saw 0x%x" % crc
    else:
        print "Ok"

# end file
```

If you wonder how the initial table is created, this routine can create it dynamically. However just starting with the predefined table as a read-only (immutable) tuple is faster and uses less memory.

```
def init_table( ):
    # Initialize the CRC-16 table,
    #   build a 256-entry list, then convert to read-only tuple
    global table

    lst = []
    for i in range(256):
        data = i << 1
        crc = 0
        for j in range(8, 0, -1):
            data >>= 1
            if (data ^ crc) & 0x0001:
                crc = (crc >> 1) ^ 0xA001
            else:
                crc >>= 1

        lst.append( crc)

    table = tuple( lst)
    return
```

Setting Serial Adapter Baud Rate

Setting the Xbee RS-232/485 Serial Adapter settings such as Baud Rate

Although the title of this page is setting the serial baud rate, it is designed to explain the full serial-specific setup. You should also review the appropriate XBee Module Product manual to understand the capability for buffers, flow control and Transparent-vs-API modes of operation.

This Digi support page lists the most recent XBee Product manuals.

The General Paradigm

In most serial protocol systems - such as Modbus or Allen-Bradley/DF1 - the remote XBee RS232/485 Adapters are loaded with the **Router AT-Mode** firmware. Any serial messages they receive over the RF are send out the serial pins - either to the attached microcontroller, or via RS-232/485 drivers to an external device.

Since the attached device does not understand how to address packets on the mesh, the XBee returns any serial response received either to the node addressed by the DH/DL setting, or by default "Mesh Aggregator". In Znet and Zigbee this means the coordinator in the Digi gateway.

See Serial encap by API for a fuller discussion of encapsulating legacy protocols over mesh.

Using XCTU

Loading Firmware

XCTU allows you to reload the Xbee firmware, as well as set the parameters (often referred to as AT commands/settings).

The Digi XBee RS-232 Adapter and XBee RS-485 Adapter expect the normal Router AT firmware. The Digi XBee RS-232PH Adapter (PH = Port Harvesting) requires a special firmware build since it must function as an end-device and sleep some percentage of the time.

If your attached device is programmable and can understand the Digi/MaxStream API protocol, then you can use the Router API firmware.

With very specialized host firmware you can also set the XBee serial adapters to deep-sleep and power remote devices from battery. For example, the adapter could sleep for one hour, then wake and issue a dummy I/O sample to the host. The host must be waiting for this message, then send the serial poll and expect a serial response. While this sounds easy, in practice it is difficult to get the timing correct.

Restoring the Product Code/DD Parameter

If you reflash an Xbee module, the Product Code or DD parameter is reset to zero. You should restore the lower 16-bit to the values below. If you leave this set to zero, then your X4 Web interface will display the device as Unspecified and not allow you to set the serial baud rate and other settings.

It is also possible that some host programs will NOT work properly with an adapter with a product type of zero - for example, if the program expects to confirm RS-485 setting, it might require than the lower 16-bits of the DD parameter be 0x0006.

DD lower word for Digi Product Type of XBee serial adapters:

0x0000	Unspecified
0x0005	XBee RS-232 Adapter
0x0006	XBee RS-485 Adapter
0x0009	XBee RS-232PH (Power Harvesting) Adapter

The full list and discussion of Product Codes in on this wiki page

Setting the Baud Rate / BD Parameter

The Baud rate is the BD setting, with standard baud rates defined as the codes 0 to 7.

0x0000	1200 bps
0x0001	2400 bps
0x0002	4800 bps
0x0003	9600 bps - factory default
0x0004	19200 bps
0x0005	38400 bps
0x0006	57600 bps
0x0007	115200 bps

Notes:

- Use of other codes or support of non-standard baud rates varies by product, so ask Digi sales support for details.
- XBee for use within a Digi gateway generally should be set to 115200 (code 7), however if it is set for 9600 (default) the gateway will detect and change the baud rate to 115200
- Changing the baud rate might force you to change the baud rate in XCTU

Setting the Serial Parity / NB Parameter

The Serial Parity is the NB command, with standard settings defined as the codes 0 to 3.

0x0000	No Parity- factory default
0x0001	Even Parity
0x0002	Odd Parity
0x0003	Mark Parity (bit is always logical 1)

Notes:

Currently it is not possible to set 7,E,2 or 7,O,2 in the XBee

Setting the Stop Bits / SB Parameter

The Stop Bits is a the NB command - not all XBee support this (see notes)

0x0000	1 stop bit - factory default
0x0001	2 stop bits (disables use of MARK parity)

Notes:

This command makes debut in XB24-ZB firmware versions 2x6x - so available SUMMER 2009

Setting the Data Bits / Not Possible

As of May 2009, it is not possible to set 7 data bits in the XBee, so is not possible to set 7,E or 7,O in the XBee. However, host software can manually set the 8th parity bit and send as 8,N.

Setting the Packetization Timeout / RO Parameter

The XBee receiving serial data buffers the incoming data until one of this becomes true:

- A full packet worth of data is buffered and can be sent.
- The RO number of charcter times of inter-character silence has been seen.

Note that RO is character times - not milliseconds! It is also not how many bytes per packet, so do not set to 6 if you expect 6 byte packets.

The default of 3 means roughly 3 msec at 9600 baud, but 25 msec at 1200 baud. The default of 3 works for any dedicated processor which can sustain a steady flow of serial bytes. If you are having packets sent prematurely, consider bumping RO to 20, then 50 if the problem continues.

Setting the Flow Control / D7 Parameter

For RS-232 and RS-485 outbound flow (or hardware duplex) control is accomplished by the DOI7 pin.

0x0000	n/a	No Flow Control
0x0001	RS-232	CTS flow control
0x0006	RS-485	Transmit enable on low signal
0x0007	RS-485	Transmit enable on high signal

See the most recent XBee Product manuals for a fuller discussion of RTS/CTS flow and RS-485.

If you are using the XBee RS-485 adapterDigi XBee 485 Adapter and configuring it through the web UI, uncheck the CTS flow control check-box (turn it off) on the **Basic Settings** page, then go to the **Advanced Settings** page and enter a seven (**7**) into the **DIO7 configuration (D7)**: setting.

Using the Gateway Web Interface

If your XBee serial adpater is properly associated with your Digi gateway - and the correct DD values are loaded into your adapters, then you can set the serial settings from the comfort of the Web interface. The product types should be listed as RS-232 Adapter, RS-232 PH Adapter, or RS-485 Adapter. If this is true, then you will see a Network View of XBee Devices looking like this:

XBee Configuration

▼ Network View of the XBee Devices

Node ID	Network Address	Extended Address	Node Type	Product Type
_Gateway	[0000]!	00:13:a2:00:40:30:96:c3!	coordinator	X4 Gateway
N01	[74a8]!	00:13:a2:00:40:4a:21:d8!	end node	RS-232 PH Adapter
N02	[f53c]!	00:13:a2:00:40:4a:21:c2!	end node	RS-232 PH Adapter
N03	[165c]!	00:13:a2:00:40:4a:21:d5!	end node	RS-232 PH Adapter
N04	[4f84]!	00:13:a2:00:40:4a:22:02!	end node	RS-232 PH Adapter
N05	[b1df]!	00:13:a2:00:40:4a:21:c4!	end node	RS-232 PH Adapter
N06	[0814]!	00:13:a2:00:40:4a:21:ca!	end node	RS-232 PH Adapter
N07	[6f50]!	00:13:a2:00:40:4a:21:db!	end node	RS-232 PH Adapter
N08	[c433]!	00:13:a2:00:40:4a:21:d4!	end node	RS-232 PH Adapter
N09	[16fe]!	00:13:a2:00:40:4a:21:c7!	end node	RS-232 PH Adapter
N10	[37da]!	00:13:a2:00:40:4a:21:cf!	end node	RS-232 PH Adapter
N11	[a01c]!	00:13:a2:00:40:4a:21:ce!	end node	RS-232 PH Adapter
N12	[f451]!	00:13:a2:00:40:4a:21:c3!	end node	RS-232 PH Adapter
N13	[afe8]!	00:13:a2:00:40:52:29:d5!	end node	RS-232 PH Adapter
N14	[b94a]!	00:13:a2:00:40:4a:21:cd!	end node	RS-232 PH Adapter
n15	[1a4c]!	00:13:a2:00:40:3e:14:8b!	router	RS-232 Adapter
N16	[33b6]!	00:13:a2:00:40:52:18:a5!	router	RS-232 Adapter

Note The sorting by Node ID is a new feature in the March 2009 X4/X8 firmware - the CLI disp mesh command retains the old order to prevent breaking automated scripts. It is not case sensitive - notice the one mistaken lower-case 'n' in the above example. Placing the '_' on the gateway id causes it to be listed first.

Click (select) the desired adapter, and you will see a web page showing **Basic Settings**, including Basic Radio Settings, Advanced Radio Settings, and **Serial Interface Settings**. If you receive an error that the device has no configurable parameters, then either it is sleeping or has left the network - in both of this situations the adapter is still listed in the Xbee route tables, but does not respond to parameter reads.

The settings are explained above on this page, plus the Web UI help quotes the Xbee Product Manual PDFs quite closely - that is the little blue "?" patiently waiting for your attention in the upper right hand corner of the page. remember to hit the APPLY button after making your changes.

Serial Interface Settings

Baud Rate (BD): 9600

Parity (NB): None

Flow Control (D7): ☐ Enable CTS Flow Control (DIO7)

Packetization Timeout (RO): 3 msecs (0-255, 0=immediate)

Note The Packetization Timeout (RO) parameter is mistaken listed as msec - this will be fixed in future gateway firmware.

Understanding XBee EndPoints

The way Digi implimented the XBee/ZigBee end-points in the XBee extensions to the Python socket API implies that they function like TCP or UDP source and destination port numbers. However unlike TCP or UDP which treats each src/dst port pair as unique conversations, XBee/ZigBee assigns each distinct 'destination end-point' to a single application (or handler) for message delivery. The handler is free to manually assign a meaning to the 'source end-point' in received packets, but it is NOT likely unique.

For example, a Modbus/TCP server can bind on TCP port 502. When five clients connect, each uses a destination port of 502 and a unique 'source port' (per IP address pair). Five 'sockets' may be created, and each of the five server-tasks spawned is blissfully unaware that other clients are active.

In contrast, an XBee server can bind on an end-point such as 0xE8, but five remote devices sending to destination end-point 0xE8 will appear to use the same 'socket'. Additionally, they all may claim a source end-point of 0xE8. The Xbee server code must manually process and distribute messages based on extended address or other criteria.

Which XBee technologies support end-points?

You can use the upper word of the DD setting to distinguish XBee which support end-points and those which do not.

Upper DD Word	Name	End-Point Support	Description
0x0000	Unspecified	Unknown	This means the XBee is mis-configured
0x0001	802.15.4	No	Non-mesh star/peer configured nodes on 2.4GHz
0x0002	ZNet 2.5	Yes	Digi's older pre-ZigBee firmware
0x0003	Digi Zigbee	Yes	Digi's firmware supporting Zigbee 2007
0x0004	Digi Mesh 900	Yes	Digi's proprietary mesh on 900Mhz
0x0005	Digi Mesh 2.4	Yes	Digi's proprietary mesh on 2.4Ghz
0x0006	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 868Mhz
0x0007	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 900Mhz

Notes:

- Use the DD value in the Digi gateway's XBee - you cannot trust that the remote XBee has a valid DD setting!
- Even if the XBee does NOT support end-points, Python may allow you to bind on a specific end-point like 0xE8. However, you will never receive any responses on 0xE8 because all responses appear to be targeted at end-point 0x00.

Managing end-points in XBee

Under XBee AT-transparent mode

So a concrete example, sending a packet with (src=0xE9 dst=0xE8) will NOT automatically result in return-responses being received as (src=0xE8 dst=0xE9) - you need to explictely change the AT settings in the remote XBee to return serial data to destination end-point 0xE9.

Therefore, to move your Python script bind end-point away from 0xE8 (the default), then do the following:

- Your Python script binds on an alterative end-point such as 0xE9, 0x01, or 0x41.
- Manually set all required remote XBee with an AT setting DE=0xE9 (or as required). You can do this by:
  - Under XCTU, relevant Xbee firmwares (such as ZigBee Router AT 0x2264) will have a closed/collapsed sub-option named "ZigBee Addressing" under Adressing. Click on it to open and you'll see the DE/SE/CI options.
  - Newer 2.9.x Digi gateway firmwares expose the DE command under the advanced settings for remote XBee nodes.
  - Use standard "remote AT command" API frames or ddo_set_param() calls to force DE to the desired value.

Note that changing the XBee's SE setting does NOT allow moving serial encapsulation away from 0xE8 - it only changes the source end-point cliamed by AT-Transparent responses. The remote XBee in AT-transparent mode will only forward data out the serial port which is targetted at 0xE8.

Under XBee API mode

When running XBee firmware in API mode, then most source/destination end-points are passed through transparently. It is up to the serial-attached CPU to parse and assign meaning to the end-points. Therefore, an external CPU might gracefully and automatically return responses to requests with (src=0xE9 dst=0xE8) as (src=0xE8 dst=0xE9).

End-point numbers to avoid

Although no exact list is available, you should avoid using these end-points:

End-Point	Description
0x00	ZigBee Device Object end-point - reserved by ZigBee stack
0xDC to 0xEE	Reserved for Digi use
0xEF to 0xF0	Reserved for other vendor use
0xF1 to 0xFE	Reserved by ZigBee Alliance / for other uses
0xFF	Digi treats as 'any end-point' (a wild-card) during Python socket operations

End-points 0x01 to 0xDB should be free to use. However, the rules for picking end-point numbers is much like that for TCP/UDP port numbers - be flexible and ready to change if required. For example, you might use TCP port 2101 to tunnel data to a Digi device server. It works, but TCP port 2101 is officially assigned for use as "RTCM-SC104", which is a Radio Technical Commission for Maritime

Services standard used to move GPS data via TCP/IP. Microsoft also overloads this port for use with RPC-based MQIS and Active-Directory Lookups. The same may be true of XBee end-points - even if you reuse a preassigned end-point it is only a problem if you require some other tool or function with a conflicting use.

The examples above used 0xE9, which Digi claims is reserved, so might be used for some other purpose in the future. If you select end-point 0x77, some other vendor might create a product requiring a Python script binding on end-point 0x77. So design your system to allow the end-point number selected to be easily changed as required.

EndPoints in the Modbus/IA Engine

You can use the Digi Modbus/IA engine to route Ethernet-based Modbus/TCP requests to remote Modbus/RTU serial devices via XBee RS-232 adapter and XBee RS-485 adapter. Configuring the XBee module is covered in Modbus Example Serial Adapter.

However, the default Modbus/IA behavior is to use ZigBee endpoint 0xe8, which will conflict with most sample Python programs and the Device Cloud/DIA framework. This conflict is because only 1 task can bind on (or register to receive) incoming XBee packets on endpoint 0xe8.

Therefore you should move the Modbus/IA traffic away from endpoint 0xe8 and use another, such as 0xe9. This is easier than trying to change the behavior of a Python application in an unknown number of places:

- In the remote XBee adapter, set DE as explained above to E9 (or to your desired value)
- In the IA Modbus unit id/slave address mapping table, add the new Xbee endpoint after the '!', so MAC looks like as 00:13:a2:00:40:30:de:cd!E9. Note that this does NOT change the 'destination' endpoint in the outgoing Modbus request - this remains 0xe8 for XBee module reasons. Instead, it causes the Modbus/IA engine to bind on incoming endpoint 0xE9 instead of 0xE8.

NET+OS

This Category contains NET+OS information and example applications.

ASM assembly code

Program to execute assembly code in NET+OS

ASM TEST (For NET+OS 7.4.2 - 7.5.2 modules) To showcase how to execute Assembly code in NETOS.

How does it work?

Test files

This sample program contains two files, root.c and appconf.h. The main function is in root.c.

ASM test sample application

The ASM Test sample application can be found here: ASMSample.zip.

Basic usage

Compile, load and run program using NET+OS environment.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

void assembly_delay();

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;

}
#define LOOP_MAX (25)
void applicationStart (void)
{
           int loopIdx = 0;
            printf ("Hello World!\n");

           // continually call a routine that will execute one instruction 4
times in one asm call
           for (loopIdx = 0; loopIdx < LOOP_MAX; loopIdx++)
           {
               assembly_delay();
           }

           // now execute the same assembly directly
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");
           asm volatile("mov   r0, r0\n\t");

          // now do it in one call
          asm volatile(
          "mov        r0, r0\n\t"
```

```
        "mov        r0, r0\n\t"
        "mov        r0, r0\n\t"
        "mov        r0, r0\n\t"
        );


        printf("Test done\n");


        tx_thread_suspend(tx_thread_identify());
    }

        void assembly_delay()
              {

                 asm volatile("mov   r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t"
                     "mov    r0, r0\n\t");
              }
  }
```

NET OS 9P9360 external RTC

Program for 9P9360 external RTC (DS1337) in NET+OS

NET+OS 9P9360 External RTC (For NET+OS 7.4.2 - 7.5.2 modules) This example adds support for the onboard DS1337 RTC on the 9P9360 in NET+OS.

Test files

This sample program contains six files. The main function is in root.c.

9P9360 External RTC (DS1337) Test Sample Application

The 9P9360 External RTC (DS1337) Test sample application can be found here: 9P9360_External_RTC_(DS1337).zip.

Basic usage

```
Copy rtc_drv.custom over
\netos\src\bsp\devices\ns9xxx\ns9360\rtc\rtc_drv.c
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include "rtc.h"
#include <time.h>

void applicationTcpDown (void)
{
        static int ticksPassed = 0;

        ticksPassed++;
}

static NaRtcTime_t rtc_time;

static char* const WEEKDAYS[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday","Friday", "Saturday"};

static char* const MONTHS[12] = {"January", "Febuary", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};

void applicationStart (void)
{
        NaStatus rc;
        char* data;
        time_t t;

        while(1)
        {
                t = time(NULL);
                data = ctime(&t);
```

```
                printf("time(): %s", data);
                rc = naRtcGetTime(0, &rtc_time);
                switch (rc)
                {
                        case NASTATUS_SUCCESS:
                                // Wednesday, December 16, 2009 at 12:00:00
                                // day of the week/month index starts at 1, our
arrays start at 0
                                printf("Date/Time: ");
                                printf("%s, ",WEEKDAYS[rtc_time.dayOfWeek-1]);
                                printf("%s %i, %i at", MONTHS[rtc_time.month-1],
rtc_time.date, rtc_time.year);
                                printf(" %i:%i:%i\n", rtc_time.hours, rtc_
time.minutes, rtc_time.seconds);
                                break;
                        case NASTATUS_RTC_NOT_INITIALIZED:
                                printf("The real time clock is not
initialized.\n");
                                break;
                        case NASTATUS_RTC_FAIL:
                                printf("The real time clock was unsuccesffully
updated with the new time. It's given invalid rtc_time.\n");
                                break;
                        case NASTATUS_RTC_NOT_AVAILABLE:
                                printf("The real time clock doesn't support this
action.\n");
                                break;
                }
                tx_thread_sleep(500);
        }

        printf ("Hello World!\n");
        tx_thread_suspend(tx_thread_identify());
}
```

NET OS Appkit Rio

Program for controlling the Rabbit RIO chip in NET+OS

Appkit RIO Test (For NET+OS 7.4.2 - 7.5.2 modules) A driver for controlling the Rabbit RIO chip.

Test files

This sample program contains eight files. The main function is in root.c.

CPU test sample application

The Appkit Rio Driver Test sample application can be found here: RIO_Appkit_Driver.zip.

Basic usage

```
Test Hardware setup on RIO APPKIT board:
--With an LED (or Scope). Connect ground (neg) to P4.9(GND). Connect VCC(pos)
      to P4.8(RC0P0). This LED is where the majority of all the tests will be
      performed
--With a STDP(single throw, double poll) switch. Connect the common pin to
      P4.4(RC1P0). Connect one poll to VCC and the other poll to GND. This pin
      will be used for all the input tests.
BSP SETUP

Digi Connect ME 9210
Change the following in gpio.h:
#define BSP_GPIO_MUX_SERIAL_A             BSP_GPIO_MUX_SERIAL_2_WIRE_UART
#define BSP_GPIO_MUX_IRQ_1                BSP_GPIO_MUX_USE_PRIMARY_PATH
#define BSP_GPIO_MUX_IRQ_1_CONFIG                         BSP_GPIO_MUX_IRQ_FALLING_EDGE

Right Click on your project. Go to properties. Select Net+OS from the list.
Under bsp_sys.h change 'Dialog Port' and 'STDIO Port' to 'Serial Port C'

ConnectCore 9P 9215
Change the following in gpio.h:
#define BSP_GPIO_MUX_IRQ_3                BSP_GPIO_MUX_USE_3RD_ALTERNATE_PATH
#define BSP_GPIO_MUX_IRQ_3_CONFIG                         BSP_GPIO_MUX_IRQ_FALLING_EDGE
```

Known issues

The Input/Output pins on the RIO APPKIT board are all floating. This means that any pin not pulled up or down and set for an input will report random state changes due to cross talk and other interference.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

#include "rio_appkit.h"


void applicationTcpDown (void)
```

```
{
        static int ticksPassed = 0;
        ticksPassed++;
}

void TestGPIO();
void TestPWM();
void TestGPIOInput();
void TestHardwareReset();
void TestSoftReset();
void TestPPM();
void TestInterrupt();

void applicationStart (void)
{
#if (PROCESSOR == ns9215)
       /*
        * ConnectCore 9P9215 Connections
        * GPIO67 = X21.C9 - Reset Line
        * GPIO101(IRQ3) = X21.D16 - Interrupt Line (IRQ3)
        */
        naRIOInit(5,67,EXTERNAL3_INTERRUPT);
#elif (PROCESSOR == ns9210)
       /*
        * Connect ME 9210 Setup
        * GPIO13 = P3.20 - Reset Line
        * GPIO2 = P3.12 - Interrupt Line (IRQ1)
        */
        naRIOInit(5,13,EXTERNAL1_INTERRUPT);
#endif

        printf("Starting RIO Test\n");

        TestHardwareReset();
        TestSoftReset();
        TestGPIO();
        TestPWM();
        TestGPIOInput();
        TestPPM();
        TestInterrupt();

    tx_thread_suspend(tx_thread_identify());
}

void TestGPIO()
{
        int port,pin;
        int x;
        int retval;

        printf ("Starting GPIO Test...\n");
        for (x=0; x < 20; x++)
        {
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, TRUE);

                printf("All are high\n");
                tx_thread_sleep(10);
```

```
                for (port=0; port<8; port++)
                        for (pin=0;pin<4; pin++)
                                retval = naRIOSetOutput(port, pin, FALSE);

                printf("All are low\n");
                tx_thread_sleep(10);
        }
        printf ("GPIO Test complete...\n");
}

void TestPWM()
{
        int retval;
        int x,i;
       /*
        * 16666666 = 60hz
        *  20000000 = 50hz
        *  10000000 = 100hz
        */
        long period = 10000000;

        printf ("Starting PWM Test.\n");
        retval = naRIOReset(TRUE);
        retval = naRIOSetPrescaler(period);

        retval = naRIOSetPWM(0,0,period,0);
        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 100; x++)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
                for (x = 100; x >=1 ; x--)
                {
                        retval = naRIOUpdatePWM(0,0, period * x/100.0);
                        tx_thread_sleep(1);
                }
        }
        printf("PWM Test Complete...\n");
}

void TestHardwareReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Hardware Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(TRUE);
        printf("RIO has been reset!\n");
```

```
        printf("Hardware Reset Test complete...\n");
        tx_thread_sleep(500);
}

void TestSoftReset()
{
        int retval;
        BYTE port, pin;

        printf("Start Software Reset Test. All Pins...\n");

        for (port=0; port<8; port++)
                for (pin=0;pin<4; pin++)
                        retval = naRIOSetOutput(port, pin, TRUE);

        printf("All pins should be high for 5 seconds. Then the RIO should
reset.\n");
        tx_thread_sleep(500);
        retval = naRIOReset(FALSE);
        printf("RIO has been reset!\n");
        printf("Software Reset Test Compelte...\n");
        tx_thread_sleep(500);
}

void TestGPIOInput()
{
        BOOL val = FALSE;
        BOOL last = FALSE;
        int retval;
        int x;

        retval = naRIOReset(TRUE);

        printf("Starting GPIO Input Test Port 1, Pin 0...\n");

        retval = naRIOSetInput(1,0);
        for (x=0; x<20; x++)
        {
                // wait for a state change
                while ( retval == last)
                        val = naRIOReadInput(1,0);

                printf("Pin is now %d\n", retval);
                last = retval;
        }

        printf("GPIO Input Test Complete...\n");
}

void TestPPM()
{
        int retval;
        long period = 10000000;
        int x,i;

        naRIOReset(TRUE);

        printf("Starting PPM Test. Port 0, Pin 0...\n");
        retval = naRIOSetPrescaler(period);
```

```
        /* set pins 2,3 on port0 to a PWM */
        retval = naRIOSetPWM(0,2,period,period/3);
        retval = naRIOSetPWM(0,3,period,period/3);

        /* set pin0 to a PPM with a phase shift */
        retval = naRIOSetPPM(0,0,period,0,period/3);

       /*
        * set pin1 (the dead pin) to a GPIO.
        * pin1's match register is being used,
        * so it can only be used for GPIO
        */
        retval = naRIOSetOutput(0,1,FALSE);

        for (i = 0; i < 10; i++)
        {
                for (x = 1; x <= 360; x++)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
                for (x = 360; x >=1 ; x--)
                {
                        retval = naRIOUpdatePPM(0,0, x/2, x*(period/720));
                        tx_thread_sleep(1);
                }
        }

        printf("PPM Test Complete...\n");
}

void InterruptCallBack(int port, int pin)
{
        int retval;
        printf("Callback called on port: %d, pin: %d\n", port, pin);
        retval = naRIOResetGPIOInterrupt(1,0);
}

void TestInterrupt()
{
        int retval;
        printf("Starting Interrupt Test. Port 1, Pin 0...\n");
        retval = naRIOReset(TRUE);
        retval = naRIOGPIOInterrupt(1,0, InterruptCallBack);
}
```

NET OS CPU

Program to read/write CPU registers in NET+OS

CPU TEST (For NET+OS 7.4.2 - 7.5.2 modules) Shows how to read/write directly from CPU registers in NET+OS.

Test files

This sample program contains two files, root.c and readme.txt. The main function is in root.c.

CPU test sample application

The CPU Test sample application can be found here: Read-Write_CPU_Registers.zip.

Basic usage

Compile, load and run program using NET+OS.

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"

volatile unsigned long *reg_ptr = (unsigned long *) 0xA0902008;

void applicationStart (void)
{
        printf ("Register Value: 0x%X\n", *reg_ptr);
        *reg_ptr = (*reg_ptr) | 0x6C;
        printf ("Register Value: 0x%X\n", *reg_ptr);

    tx_thread_suspend(tx_thread_identify());
}

void applicationTcpDown (void){    static int ticksPassed = 0;    ticksPassed++;}
```

NET OS Ping

Program to generate a ping from a NET+OS module

PING TEST (For NET+OS 7.4.2 - 7.5.2 modules) Developed to generate a ping from the NET+OS development module.

How does it work?

PING TEST is a client based application set up to run on a NET+OS development module. It sends ping requests to a specific IP address entered in the root.c file.

Test files

This sample program contains three files, ping.c, ping.h and root.c. The main function in ping.c pings a target IP address. In the file ping.h you can specify the amount of system ticks per second.

Ping test sample application

A simple Ping Test sample application can be found here: Ping_test.zip.

Basic usage

First, open the root.c file and enter the ipaddress you would like to ping and save file

```
retval = ping("10.4.110.1");
```

Second, Build application, load the application into the embedded module and run it.

Sample of root.c file:

```
{
        int retval;

        while(1)

        {
                retval = ping("10.4.110.1");

                if(retval == -1)
                {
                        printf("Error\n");
                }
                else if(retval == 0)
                {
                        printf("No response\n");
                }
                else
                {
                        printf("Response heard\n");
                }
        }
```

```
        return;
}
```

NET OS Telnet Session

Program for Telnet_Customized_SessionID in NET+OS

NET+OS Telnet Session ID (For NET+OS 7.4.2 - 7.5.2 modules) Showcase how to customize telnet session ID using NETOS APIs. Sample application showing working of a Telnet Server.

Test files

This sample program contains six files. The main function is in root.c.

Telnet Session ID Test Sample Application

The Telnet Session ID Test sample application can be found here: Telnet_Customized_SessionID.zip.

Basic usage

Before Building and debugging please check the following in bsp_cli.h

```
1. #define BSP_CLI_TELNET_ENABLE   FALSE
2. #define BSP_CLI_ENABLE  FALSE

Debug/Run :

Open command prompt and
$telnet (ipaddress) 5000
login :
password :


Enter any characters and hit enter.
You can start more than one telnet session.
|Open another command prompt and :
$telnet (ipaddress) 5000

This application will display the below details to serial port for each session:

Session ID
Bytes Received for the particular Session
Username of the session.

Charlie & Bob :) :)
```

Sample of root.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <tx_api.h>
#include "appconf.h"
#include <sysAccess.h>


#include "telnet.h"
#include <bsp_api.h>


/*
```

```
* Set this to 1 to run the manufacturing burn in tests.
*/
int APP_BURN_IN_TEST = 0;


void applicationStart (void)

{
        int ret_telnet;
        unsigned long port_count = 0;
        iconfig_ptr.max_entries = TELNET_MAX_SERVERS;

    /*  Add Username digi password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "digi", "sysadm", NASYSACC_LEVEL_RW, NULL);

    /*  Add Username sysadm password sysadm to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "sysadm", "sysadm", NASYSACC_LEVEL_R, NULL);

    /*  Add Username debug password debug to the System Access Database   */
    NAsetSysAccess (NASYSACC_ADD, "debug", "debug", NASYSACC_LEVEL_R, NULL);

    /*
         *Initializing Telnet Server .
        */
    if( (ret_telnet = TSInitServer(&iconfig_ptr)) != SUCCESS){
    printf("Return value : %d\n",ret_telnet);
            if(ret_telnet == TS_NO_MEMORY)
                    printf("Unable to allocate memory.\n");
            else if(ret_telnet == TS_INVALID_PARAMETER)
                    printf("The value in parameter is invalid.\n");
            else if(ret_telnet == TS_SYSTEM_ERROR)
                    printf("An internal error occurred.\n");
            else
                    printf("Can't open without calling 
TSInitServer.\nExiting........\n");
            }
    /*
        *Configuring Telnet Server
        *Here only one server is configured.
        * If you want you can configure one more.
        * passing port_count+1
        */

        if(setup_telnet_server(port_count) != 0){
                printf("Telnet Server Setup Failed\n");
                return;
        }

        TS_t *list;
        while(1)
        {
                tx_thread_sleep(5*NABspTicksPerSecond);

                if(tx_semaphore_get(&semaphore_ptr,TX_WAIT_FOREVER ) != TX_
SUCCESS){
                        printf("Semaphore Get Failed\n");
                        return;
                }
                list = head;
```

```
                while(list != NULL)
                {
                        printf("\nSession ID : %d\n",list->ts_id);
                        printf("Bytes Received in this Session : %d\n",list-
>byte_recv);
                        printf("Username : %s\n",list->username);
                        list = list->ts_nxt;
                }
                if(tx_semaphore_put(&semaphore_ptr) != TX_SUCCESS){
                        printf("Semaphore Put Failed\n");
                        return;}
                }

        }

}

void applicationTcpDown (void)

{
    static int ticksPassed = 0;

    ticksPassed++;
/*
* Code to handle error condition if the stack doesn't come up goes here.
*/
}
```

Programmable XBee

The Programmable XBee is an XBee PRO ZB with a second onboard Freescale processor, which allows the customer to add custom firmware directly to the XBee. The default processor is an HCS08 (MC9S08QE32CFT) with 32K Flash and 2K RAM. Plugging the XBee with you own code into an existing XBee 232/485/DIO Adapter creates a powerful, custom-function ZB node.

Programmable XBee - getting started

What is the programmable XBee?

Throughout history, Digi XBee customers have asked the question, "Can I add custom code to the XBee firmware?" The answer has been "No" for the simple reason that there is rarely enough free RAM or FLASH available for users to use.

The Programmable XBee solves this problem by including a second Freescale processor on the XBee, use largely under the customer's complete control. The default processor is an HCS08 (MC9S08QE32CFT) with 32K Flash and 2K RAM. In 2011, another XBee Programmable version will become available with 128K Flash and 8K RAM.

Getting started

- Order the dev kit - it is currently listed on the XBee module page, not the Drop-In-Network Dev kit page: XBee-PRO ZB ZigBee RF Modules The kit includes 2 Programmable XBee modules, the USB-based debugger, a USB dev board and other hardware. You will need at least 2 free USB ports to use the kit.
- Look over the setup information in [Getting Started Guide now included in SDK]

Limitations

Currently, this programmable XBee is only available as ZigBee with PRO 100mw RF power ratings. Someday programmable XBee versions may be produced for DigiMesh (etc), but at present they are not committed to become products.

During full operation with RF activity, the programmable XBee draws about 15mA more power.

Useful online information

- The product page: XBee-PRO ZB Programmable Development Kit page, under Product Support.
- The *Getting Started Guide*: Now included in the SDK
- Using an additional XBIB from a different kit for the "Building an Advanced Application" exercise.
- What's happening underneath The Basic Framework of the main function in your first Programmable XBee Application.

XBee bootloader menu

Program to bypass bootloader menu

XBee bypass bootloader menu (Xbee program) This sample application skips and bypasses freescale MCU & directly access radio.

Test files

This sample program contains several files. Program is in file "main.c".

Bypass bootloader menu Test Sample Application

The bootloader menu Test sample application can be found here: Bypass_bootloader_menu.zip.

Basic usage

```
APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250.
```

Sample of bootloader bypass menu:

```
#include <xbee_config.h>
/* #include <types.h>
#include <xbee/platform.h>*/


void main(void)
{
/*       sys_hw_init();
        sys_xbee_init();
        sys_app_banner(); */
        //APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250
        sys_reset(APP_CAUSE_BYPASS_MODE);
}
```

XBeeComm

Program to communicate XBeeComm through Com port with Windows 7

XBeeComm Sample (For Windows 7 PC) This application can communicate with the Xbee module from the PC through the COM port. Using the different buttons on the form we can get the different Xbee parameters like pan.

How does it work?

Using this C sharp application , we can communicate with the Xbee module from the PC. Using the different buttons on the form we can get the different Xbee parameters like pan id, version of the software.

Test files

This sample program contains nine files.

XBeeComm Test Sample Application

The XBeeComm Test sample application can be found here: XbeeComm.zip.

Basic usage

Compile, load and run program using Windows Tools.

Sample Program.cs file:

```
#using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace app2
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Rockwell Allen-Bradley

These pages explain use of Rockwell Allen-Bradley protocols under Python and for Digi products:

Importing Rockwell data to Dia and iDigi

Rockwell client to import data to Dia and iDigi

This driver polls remote Rockwell PLC for data (PCCC files), which are then converted to standard Dia channels. It can handle words, floats and bits, plus simple calculations can be done during import. This client is modeled after the Dia Modbus client, so is configured and used in much the same way.

Support / requirements

Rockwell PLC support

Rockwell PLC use a diverse collection of legacy protocols, which are fairly complex when compared to the simplicity of Modbus. The first version is targeted at Ethernet/IP PLC which can:

Support	PLC	Protocol	Comments
Yes	SLC5/05	PCCC in Ethernet/IP	SLC500 commands via unconnected_send
Yes	MicroLogix	PCCC in Ethernet/IP	SLC500 commands via unconnected_send
NO	PLC5E	PCCC in Ethernet/IP	(needs PLC5 commands)
NO	CompactLogix	PCCC in Ethernet/IP	(needs enhanced e-path)
NO	ControlLogix	PCCC in Ethernet/IP	(needs enhanced e-path)

Future plans

- Adding configurable PCCC command selection will add PLC5 support (yet no equipment to test).
- Adding configurable E-Path to define slot/chassis routing will add CompactLogix/ControlLogix support.
- Adding Symbolic-Segment support will allow polling NAMED TAGS in CompactLogix/ControlLogix.
- Would like to add serial DF1-FD/RM support for direct Digi serial ports and XBee serial adapters.

Requirements

- Dia 1.4.x or higher (released July 2011 - requires the 'tracer' functionality)

Digi Product Support

This client should run on any Digi product with Python, Dia support and an Ethernet port, including but not limited to:

- Digi ConnectPort X2/X2D
- Digi ConnectPort X4/X4H
- Digi Connect SP/WiSP

Configuration

Unlike the Modbus client which leverages the Digi IA/Modbus Engine, the Rockwell Client uses direct Python socket calls. Therefore, all configuration is done via Python and a standard Dia YML file. Of course, the Digi device will require the appropriate IP settings such as gateway IP, plus DNS if name service is expected.

Example YML configuration

This example YML configuration which reads five 32-bit floating point values from the file named "F8" in a Rockwell PLC at IP 10.9.92.22. This is repeated every 15 seconds, with all floating points rounded to 3 significant decimal places.

```
- name: ab
  driver: devices.vendors.rockwell.eip_pccc_client:EipPcccDevice

   settings:
       poll_rate_sec: 15
       target: "10.9.92.22"
       trace: 'debug'
       round: 3
       poll_list:
             - pollname: in01
               pollinfo: { 'elm':'F8:0', 'cnt':5 }
               channels:
                 - parse: { 'nam':'first',  'ofs':0,  'frm':'<f', 'unt':'1st' }
                 - parse: { 'nam':'second', 'ofs':4,  'frm':'<f', 'unt':'2nd' }
                 - parse: { 'nam':'third',  'ofs':8,  'frm':'<f', 'unt':'3rd' }
                 - parse: { 'nam':'fourth', 'ofs':12, 'frm':'<f', 'unt':'4th' }
                 - parse: { 'nam':'fifth',  'ofs':16, 'frm':'<f', 'unt':'5th' }
```

Base device settings

A single thread is spawned for each **EipPcccDevice** device created. The thread sleeps and wakes on a cycle defined by the YML **poll_rate_sec** setting. When it wakes, it runs through a list of POLLS, running them in strict half-duplex sequential order.

Specific settings for the base device:

```
poll_rate_sec: 15
poll_clean_minutes: 0
target: "10.9.92.22"
trace: 'debug'
round: 3
poll_list:
  ...
```

poll_rate_sec

- Defines the poll rate in seconds. No attempt is made to compensate for system drift.
- Type integer, stated in seconds, minimum is once per 5 seconds.
- Optional, default = once per 15 seconds.

poll_clean_minutes

- ALternate setting to poll_rate_sec, which it overrides if non-zero.
- Polls in integral minutes, attempting to synchronize polling with real time. For example, setting "**poll_clean_minutes: 15**" will cause the PLC to be polled when clock minutes are 0, 15, 30 and 45 minutes after the hour.
- Must an be integer within this set: (1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60), so from once per minute to once per hour.
- Optional, default = 0 so disabled, enabling poll_rate_sec to be used.

target

- Defines the remote host address as IP (or DNS name).
- Type string
- Required, no default
- No port number is settable since all Rockwell PLC expect to talk on TCP port 44818 only.

trace

- Defines how chatty the driver should be on the trace output.
- Type string in set ('debug','info','warning',error').
- Optional, default = 'info', which shows basic events and data imported.

round

- Defines a global 'round()' for floating point value.
- Type integer
- Optional, default = 999, magic number for ignore
- Example: a temperature with a +/- 1 deg accuracy should not be returned as '23.34876549'. Setting round: 2 will cause all floating points to rounded to 2 places, so '23.34876549' is ASCII encoded as '23.35'.

poll_list

- Defines a collection (list) of PLC block read/writes and now the data should be imported to iDigi Dia
- Required
- Example: see next section

Poll list settings

Each **EipPcccDevice** device can run a collection of polls, which are sent to a single server.

Specific settings for the poll objects:

```
  - pollname: in01
    pollinfo: { 'elm':'F8:0', 'cnt':5 }
    channels:
      ...
```

pollname

- Define the name for this poll object. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness.
- Example: in01 or motor or plc_b.

pollinfo

- Define the PCCC parameters for this poll object. Only File types 'N', 'B' and 'F' are supported at present.
- Type Python dictionary
- The above example means poll 5 elements from a file named "F8."

pollinfo['elm']

Defines the PCCC file name, such as "B3", "N20", or "F75". These are defined within the RSLogix ladder logic program loaded and should be familiar to the PLC users.

pollinfo['cnt']

Defines the number of elements to read.

Note The actual number of bytes moved is defined by file type. N and B-files elements are 2 bytes, while F-file elements are 4 bytes.

channels

- Defines a collection (list) of Dia channels to create from this Rockwell poll object
- Required
- Example: see next section

Channel list settings

Each Rockwell poll block of data can be imported into multiple Dia channels. Literally, the poll imports a byte string, which the channel section breaks up byte-by-byte. In the above example, 5 floats would return 20 bytes of data, so ofs 0-19.

Specific settings for the channel objects:

```
  - parse: { 'nam':'panel', 'ofs':3, 'frm':'>H', 'unt':'vdc', 'typ':'float',
'expr':'(%d/1000.0)*3.25' }
  - parse: { 'nam':'totalEnergy', 'ofs':0, 'frm':'<L', 'unt':'Wh' }
  - parse: { 'nam':'load', 'ofs':5, 'frm':'<f', 'unt':'vdc', }
  - parse: { 'nam':'overload', 'ofs':8, 'frm':'?', 'unt':'valid', }
```

parse

- Define the import/creation of a single Dia channelect. This name is used to tag the output channels of this poll. Use only letters, numbers and underscore.
- Type string
- Required, user is responsible to insure uniqueness
- Example: { 'nam':'totalEnergy', 'ofs':0, 'frm':'<L', 'unt':'Wh' } reads the first 4 bytes of the poll block, treats them as a 32-int in standard Rockwell little-endian form (LOW byte/word first). It creates a Dia channel named ab.in02_acTotalEnergy which will include a sample such as <Sample: "17294" "Wh" at "2009-10-28 15:36:30">

parse['nam']

- Defines the Dia channel name. So if the poll is named 'inv01' and channel named 'totalEnergy', then the final channel will be named 'inv01_totalEnergy'.
- Type string
- Required
- Example: four poll blocks named ['inv01','inv02','south_wing','ghouse'] could create four channels named inv01_totalEnergy, inv02_totalEnergy, south_wing_totalEnergy, and ghouse_totalEnergy.

parse['ofs']

- Defines the byte or bit offset in the poll block. It is zero-based.
- Type integer
- Required
- This is BYTE offset, not word or element offset. You will need to the appropriate math.

parse['frm']

- Defines how the data bytes are parsed to obtain the Dia sample.
- Type string
- Required
- Values are similar to the Python struct format. Rockwell's standard data format is pure little-endian, whereas most Digi use big-endian. Therefore you should generally in the '<' little-endian modifier.
  - '?' for 1-bit coils - these can be parsed from B, F or even F files, but be aware that the 'ofs' value is from the start of the poll block so reading the 3rd bit in the 4th register requires 'ofs' of 66 (67th bit, zero-based).
  - '<h' for 16-bit signed int import to Dia ('<H' is unsigned)
  - '<i' for 32-bit signed int import to Dia ('<I' is unsigned)
  - '<f' for 32-bit float import to Dia

parse['unt']

- Defines the Unit of Measure string
- Type string
- Optional, default is (an empty string)

parse['typ']

- Over-rides the type implied by the format string
- Type string, values in ['float','int','long','bool']
- Optional, default is to use parse['frm'] to estimate channel type

parse['expr']

- An 'eval' expression to do simple value conversion of the data
- Type string, in print format such as '(%d/1000.0)*3.25'
- Optional, default is no conversion
- Example: { 'nam':'panel', 'ofs':3, 'frm':'<H', 'unt':'vdc', 'typ':'float', 'expr':'(%d/1000.0)*3.25' } treats the data as a 16-bit signed integer, yet applies the formula '(value/1000.0)*3.25' to create a floating point channel sample
- Example: { 'nam':'cold', 'ofs':3, 'frm':'<H', 'unt':'is_cold', 'typ':'bool', 'expr':'bool(%d<230)' } treats the data as a 16-bit unsigned integer, yet creates a boolean channel sample

Statistics and errors

Special channels

In addition to the channels created in the YML (such as in01_first or motor.run), several special channels will be created.

device.error

The general driver status will be held in a channel named error (such as "ab.error"). It will be True if the target is not connected, or is returning Rockwell protocol errors.

device.pollname_error

Each poll block (pollinfo section) will include its own error (such as "ab.in01_error"). It will be True if that poll block or any of the parse items failed. *Note that it may remain False is the general device.error channel is True*.

device.pollname_statistics

Each poll block (pollinfo section) will also include a statistics channel (such as "ab.in01_statistics"), which is a list of numbers. Currently there are 6 numbers in this order, but more may be added:

- Count of requests sent
- Count of requests with no response received
- Count of requests containing a Rockwell protocol error
- Minimum seconds required to obtain a response

- Sliding average seconds required to obtain a response (avg = 90% previous, plus 10% latest)
- Maximum seconds required to obtain a response (not affected by time-outs)

All the statistics automatically reset to 0 when request count passes 9,999,999 polls. This includes the minimum, average, and maximum response values.

No response errors

If the Rockwell server/slave did NOT respond, then the Dia samples remain as-is and go stale.

Download the Dia Python code

Note Requires Dia 1.4.10 or newer!

This ZIP file contains a subdirectory named eip_diff, the contents of which can be merged into your dia subdirectory.

Rockwell_import_2011aug31.zip

For example, if you have ESP and Dia version 1.4.10, then

- Copy everything from eip_def\src into {Program Files}\Digi\Digi\DigiPython\Dia\Dia_1.4.10\src.
- Copy everything from eip_def\cfg into {Program Files}\Digi\Digi\DigiPython\Dia\Dia_1.4.10\cfg.

The basic Rockwell driver file is

src\devices\vendors\rockwell\eip_pccc_client.py

Utilities used by the main driver

- src\devices\vendors\rockwell\eip.py
- src\devices\vendors\rockwell\cip.py
- src\devices\vendors\rockwell\pccc_util.py
- src\devices\vendors\rockwell\sleep_aids.py

This is an example YML configuration file

- cfg\eip_poller.yml

Although neither documented nor supported, the main code (eip.py and cip.py) can run independently of the Dia - see the test routines _test_eip_cmds.py, _test_pccc_util.py for custom usage.

Python CRC16 Modbus DF1

Module: CRC16

Both the Modbus/RTU and DF1 protocols use the same CRC16 calculation, however you'll note one is 'forward' and one 'reverse' because Modbus starts with a CRC of 0xFFFF, which DF1 starts with 0x0000. The module below uses a 256-word look-up table of partially prepared answers to greatly reduce the system load.

Example

See the "__name__ == __main__" self-test portion of the module for examples of Modbus/RTU and DF1 usage.

```
            #*        File: CRC16.PY
#*              CRC-16 (reverse) table lookup for Modbus or DF1
#*

INITIAL_MODBUS = 0xFFFF
INITIAL_DF1 = 0x0000

table = (
0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
0x0A00, 0xCAC1, 0xCB81, 0x0B40, 0xC901, 0x09C0, 0x0880, 0xC841,
0xD801, 0x18C0, 0x1980, 0xD941, 0x1B00, 0xDBC1, 0xDA81, 0x1A40,
0x1E00, 0xDEC1, 0xDF81, 0x1F40, 0xDD01, 0x1DC0, 0x1C80, 0xDC41,
0x1400, 0xD4C1, 0xD581, 0x1540, 0xD701, 0x17C0, 0x1680, 0xD641,
0xD201, 0x12C0, 0x1380, 0xD341, 0x1100, 0xD1C1, 0xD081, 0x1040,
0xF001, 0x30C0, 0x3180, 0xF141, 0x3300, 0xF3C1, 0xF281, 0x3240,
0x3600, 0xF6C1, 0xF781, 0x3740, 0xF501, 0x35C0, 0x3480, 0xF441,
0x3C00, 0xFCC1, 0xFD81, 0x3D40, 0xFF01, 0x3FC0, 0x3E80, 0xFE41,
0xFA01, 0x3AC0, 0x3B80, 0xFB41, 0x3900, 0xF9C1, 0xF881, 0x3840,
0x2800, 0xE8C1, 0xE981, 0x2940, 0xEB01, 0x2BC0, 0x2A80, 0xEA41,
0xEE01, 0x2EC0, 0x2F80, 0xEF41, 0x2D00, 0xEDC1, 0xEC81, 0x2C40,
0xE401, 0x24C0, 0x2580, 0xE541, 0x2700, 0xE7C1, 0xE681, 0x2640,
0x2200, 0xE2C1, 0xE381, 0x2340, 0xE101, 0x21C0, 0x2080, 0xE041,
0xA001, 0x60C0, 0x6180, 0xA141, 0x6300, 0xA3C1, 0xA281, 0x6240,
0x6600, 0xA6C1, 0xA781, 0x6740, 0xA501, 0x65C0, 0x6480, 0xA441,
0x6C00, 0xACC1, 0xAD81, 0x6D40, 0xAF01, 0x6FC0, 0x6E80, 0xAE41,
0xAA01, 0x6AC0, 0x6B80, 0xAB41, 0x6900, 0xA9C1, 0xA881, 0x6840,
0x7800, 0xB8C1, 0xB981, 0x7940, 0xBB01, 0x7BC0, 0x7A80, 0xBA41,
0xBE01, 0x7EC0, 0x7F80, 0xBF41, 0x7D00, 0xBDC1, 0xBC81, 0x7C40,
0xB401, 0x74C0, 0x7580, 0xB541, 0x7700, 0xB7C1, 0xB681, 0x7640,
0x7200, 0xB2C1, 0xB381, 0x7340, 0xB101, 0x71C0, 0x7080, 0xB041,
0x5000, 0x90C1, 0x9181, 0x5140, 0x9301, 0x53C0, 0x5280, 0x9241,
0x9601, 0x56C0, 0x5780, 0x9741, 0x5500, 0x95C1, 0x9481, 0x5440,
0x9C01, 0x5CC0, 0x5D80, 0x9D41, 0x5F00, 0x9FC1, 0x9E81, 0x5E40,
0x5A00, 0x9AC1, 0x9B81, 0x5B40, 0x9901, 0x59C0, 0x5880, 0x9841,
0x8801, 0x48C0, 0x4980, 0x8941, 0x4B00, 0x8BC1, 0x8A81, 0x4A40,
0x4E00, 0x8EC1, 0x8F81, 0x4F40, 0x8D01, 0x4DC0, 0x4C80, 0x8C41,
0x4400, 0x84C1, 0x8581, 0x4540, 0x8701, 0x47C0, 0x4680, 0x8641,
0x8201, 0x42C0, 0x4380, 0x8341, 0x4100, 0x81C1, 0x8081, 0x4040 )

def calcByte( ch, crc):
```

```
    """Given a new Byte and previous CRC, Calc a new CRC-16"""
    if type(ch) == type("c"):
        by = ord( ch)
    else:
        by = ch
    crc = (crc >> 8) ^ table[(crc ^ by) & 0xFF]
    return (crc & 0xFFFF)

def calcString( st, crc):
    """Given a bunary string and starting CRC, Calc a final CRC-16 """
    for ch in st:
        crc = (crc >> 8) ^ table[(crc ^ ord(ch)) & 0xFF]
    return crc

if __name__ == '__main__':

    # test Modbus
    print "testing Modbus messages with crc16.py"
    print "test case #1:",
    crc = INITIAL_MODBUS
    st = "\xEA\x03\x00\x00\x00\x64"
    for ch in st:
        crc = calcByte( ch, crc)
    if crc != 0x3A53:
        print "BAD - ERROR - FAILED!",
        print "expect:0x3A53 but saw 0x%x" % crc
    else:
        print "Ok"

    print "test case #2:",
    st = "\x4b\x03\x00\x2c\x00\x37"
    crc = calcString( st, INITIAL_MODBUS)
    if crc != 0xbfcb:
        print "BAD - ERROR - FAILED! ",
        print "expect:0xBFCB but saw 0x%x" % crc
    else:
        print "Ok"

    print "test case #3:",
    st = "\x0d\x01\x00\x62\x00\x33"
    crc = calcString( st, INITIAL_MODBUS)
    if crc != 0x0ddd:
        print "BAD - ERROR - FAILED!",
        print "expect:0x0DDD but saw 0x%x" % crc
    else:
        print "Ok"

    print
    print "testing DF1 messages with crc16.py"

    print "test case #1:",
    st =
"\x07\x11\x41\x00\x53\xB9\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    # DF1 uses same algorithm - just starts with CRC=0x0000 instead of 0xFFFF
    # note: <DLE><STX> and the <DLE> of the <DLE><ETX> pair NOT to be included
    crc = calcString( st, INITIAL_DF1)
    crc = calcByte( "\x03", crc) # final ETX added
    if crc != 0x4C6B:
        print "BAD - ERROR - FAILED!",
```

```
        print "expect:0x4C6B but saw 0x%x" % crc
    else:
        print "Ok"

# end file
```

If you wonder how the initial table is created, this routine can create it dynamically. However just starting with the predefined table as a read-only (immutable) tuple is faster and uses less memory.

```
def init_table( ):
    # Initialize the CRC-16 table,
    #   build a 256-entry list, then convert to read-only tuple
    global table

    lst = []
    for i in range(256):
        data = i << 1
        crc = 0
        for j in range(8, 0, -1):
            data >>= 1
            if (data ^ crc) & 0x0001:
                crc = (crc >> 1) ^ 0xA001
            else:
                crc >>= 1

        lst.append( crc)

    table = tuple( lst)
    return
```

SMS Service

Introduction to SMS service

Short message service (SMS)

Short Message Service (SMS) is a communication service standardized in the GSM mobile communication system, which moves 140 octets (8-bit bytes) of data per message - however various encoding schemes reduces this in actual practice. For example, an email-to-SMS gateway might add 20 octets of header encoding the 'From:' email header field as text, so the actual text payload size drops towards 100 octets.

Summary of Wiki pages

Understanding if your Digi product supports SMS

Digi cellular products with GSM cellular modems and firmware version 2.9.0 and above can support SMS messaging from Python.

See What is your product firmware level.

Enabling SMS support within Digi products

By default SMS is disabled, so you need to enable support for SMS.

See Sms enabling support.

Understanding the Digi SMS Python module

In Summary:

- You must enable SMS handling in the base Digi product using the web interface or other means
- Register a Python callback using the digisms.Callback()
- Use digisms.send() to send messages

See Module: digisms.

Sending SMS from your host tool

Host applications can use email to send SMS messages via standard IP connections. This wiki page presents a sample Python script using gmail.com or other ISP email services.

See SMS host sends

Sample Digi-to-Digi SMS demo

Sample Python code showing sending messages, and displaying messages received.

See Sms sample Digi to Digi

Sample using SMS to open reverse TCP socket to host

(Wiki page to Be Added Soon)

Sample Dia driver sending SMS

(Wiki page to Be Added Soon)

Module: digisms

Provides an interface to allow the Digi Cellular SMS handling functionality to pass SMS messages to Python applications for processing. By default SMS is disabled, so you might use the web interface (or other methods) to enable support for SMS. Your cellular data plan must support SMS - by default most data plans DO NOT.

Receiving SMS Messages

The module contains a single type named Callback that is initialized with a callback function as follows:

```
digisms.Callback(cb) -> handle
```

Returns a handle to be used for deregistration later. The callback will be handled as long as the callback handle exists. If the handle goes out of scope or is deallocated, the callback will be unregistered.

Callback functions should be of the form:

```
cb(SMSMessage)
```

An 'SMSMessage' object contains these members:

message

Contents of the message

source_addr

Source address that sent the message

timestamp

Service Center assigned timestamp

The function's return value is ignored.

Transmitting SMS messages

The 'digisms' module provides a 'digisms.send' function to transmit an SMS message over the cellular network.

The send routine is of the form: > send(destination, message)

destination

A string that contains the phone number to direct the message to.

message

The message to send. Exceeding the maximum character length of a single SMS message will cause the gateway to transmit multiple messages.

Limitations & Suggestions

Remember, call back functions will be executed in another thread. Shared data will need to be protected from race conditions due to concurrent access.

Availability

Products which support this module

This feature is available as of firmware revision 2.9.0 for cellular gateways with GSM modules. (See What is your product firmware level.)

Products which DO NOT support this module

This feature is not currently supported on:

- Digi ConnectPort X3

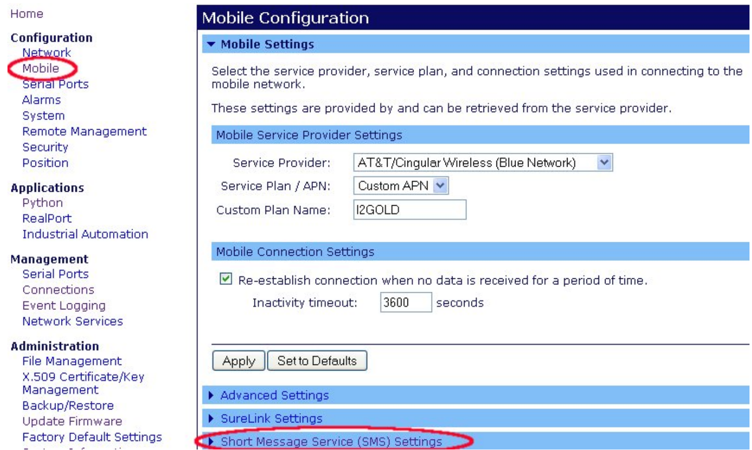
Sms enabling support

Enabling SMS support within a Digi cellular product

By default, SMS support is disabled. You also need a cellular data plan with SMS support added - most machine-to-machine data plans do NOT include SMS support.

Support for SMS has been added to the 2.9.x firmware release for NDS. See also Wiki Page: What is Your Product Firmware Level?

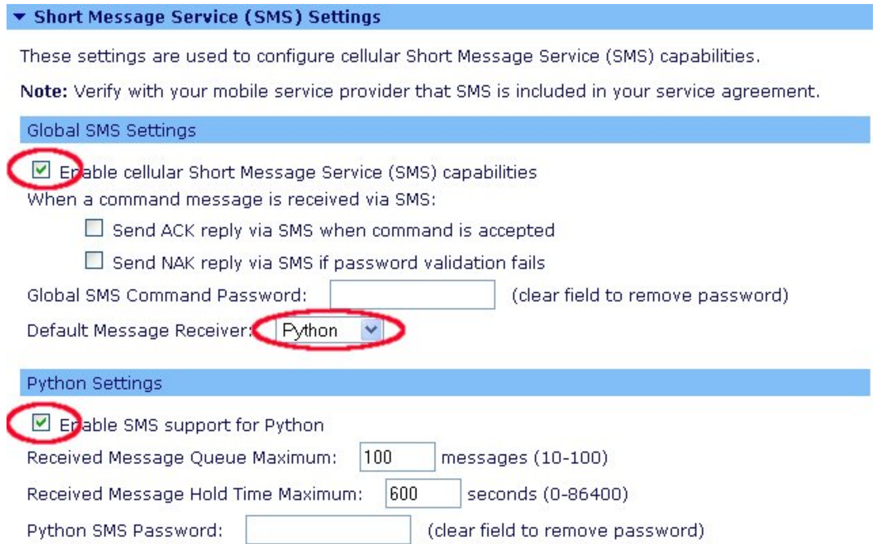
By web interface



Select the **Mobile | Short Message Service (SMS) Settings**. This link will be missing if:

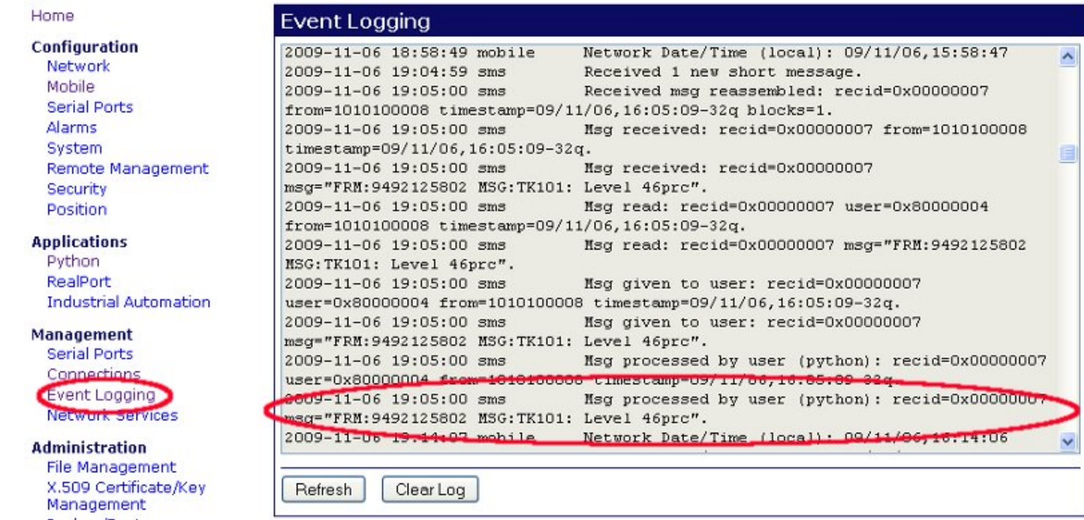
- The firmware is older than 2.9.0 See Wiki Page: What is your product firmware level?
- The cellular module is not supported yet (or lacks SMS ability).
- There is not cellular module installed, or the one installed is not functioning

To enable SMS for Python



- Click the check box to **Enable cellular Short Message Service (SMS) capabilities** for the device.
- Change the Default Message Receiver to **Python**.
- Click the check box to **Enable SMS support for Python**.

Seeing an event trace of received SMS messages

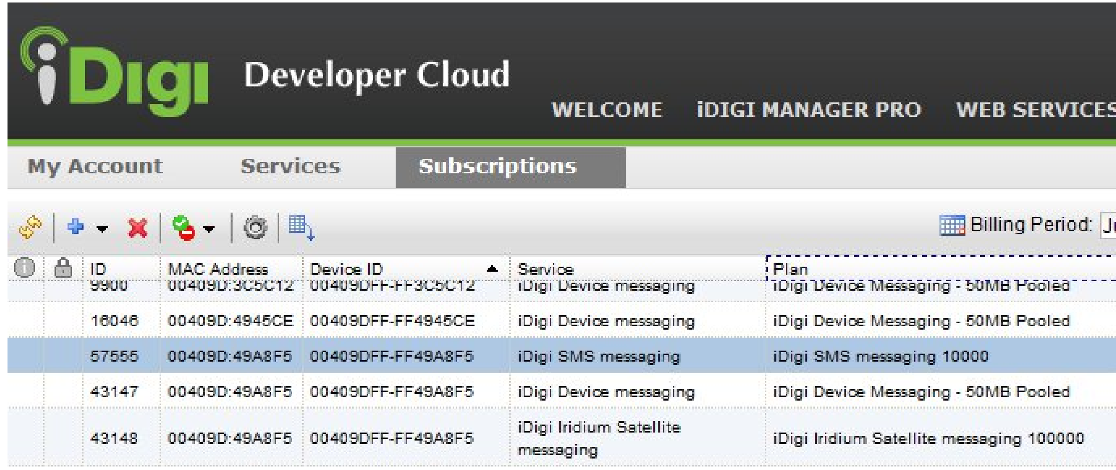


The underlying Digi cellular function logs SMS activity. Click the **Management | Event Logging** link to see an real time log of cellular events.

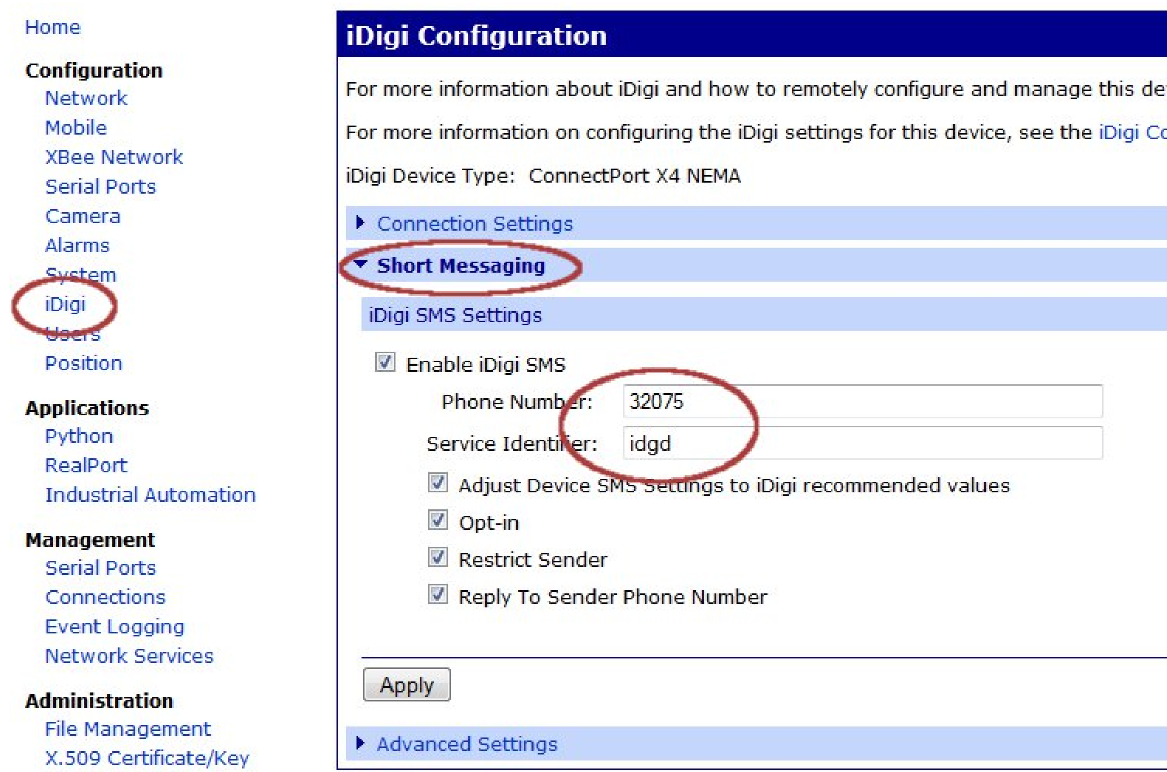
Enabling 'Dia Data' upload by SMS to iDigi

The iDigi/Dia Python framework can upload data via SMS and Satellite. Although the transport format is drastically different from the normal HTTP/XML format, then end result within iDigi looks the same.

Adding iDigi subscription



By web interface



For your data to reach iDigi you need to enter the correct phone number and 'Service Identifier' codes. By default, a new Digi device should default to correctly send data to an iDigi production account (aka: my.idigi.com). However, to send a developer.idigi.com account you need to change the Service Identifier:

- my.idigi.com (production) Phone Number = 32075, Service Identifier = idgp
- developer.idigi.com : Phone Number = 32075, Service Identifier = idgp

SMS host sends

Hosts sending SMS messages via Email

Limitations on machine-to-machine SMS Messaging

You can send SMS messages from your cell phone, or a human user can send SMS from many free public web sites. However, when you ask about sending SMS messages via automated IP-enabled systems, you tend to get simplistic answers which ignore the severe limitations placed upon public SMTP (email) servers to prevent relaying SPAM. If you want a PC or Python script to send SMS on a device without phone/data service, then your options are much more limited (and costly). Searching the web returns solutions which are:

- Specific to a single cell-phone provider and subject to change without notice.
- Or by adding an old-fashioned analog phone modem to the computer, thus gaining direct SMS support.
- Or buying a third-party tool which uses a private forwarding service.
- Or subscribing to a pay-per-message SMS forwarding service (typically costs $0.10 to $0.25 per message)

Simple method to send SMS messages via email

This page covers a simple way to generate public SMS messages without using a direct phone-service. The SMS messages can be targeted at your Digi Python-enabled device to trigger function, responses or to enable SMS testing. It makes use of your own ISP or even a gmail account. Some things to note:

- To send email from a simplistic client, you might need to use a different 'server' host name than you use with a full-function email client - for example my own ISP has me enter 'smtp.1and1.com' into a Windows client such as thunderbird or eudora, but requires me to use 'mrelay.perfora.net' for a more automated send-only client.
- You may be required to use a port other than the normal SMTP port of 25 - for example, both gmail.com and performa.net require use of port 587.
- The 'source address' of the SMS message ***MIGHT*** be meaningless since the code included is attached by the SMTP service.
  - In some systems, special logic within the host SMTP-to-SMS tool will automatically return an email response to the original email sender if an SMS response is sent to that code. It is logical to assume this has a short time-out of a few minutes, after which the SMS response will be dropped or misdirected.
  - In others, the 'From:' header field of the email must be used to define the return SMS address, and your Python code will need to decode this as part of the actual text message.

[Disclaimer: use of gmail accounts for commercial activities such as machine-to-machine may require service agreements with Google. This example code is provided for educational and testing purposes only.]

```
            # this sample code is designed to run on a host/PC.
# Minor changes may be required to run it on a Digi gateway
```

```
import smtplib

smtpserver = 'smtp.gmail.com'
AUTHREQUIRED = 1 # if you need to use SMTP AUTH set to 1
smtpuser = 'my_account@gmail.com'  # for SMTP AUTH, set SMTP username here
smtppass = 'my_password'  # for SMTP AUTH, set SMTP password here
# note that your password is stored as PLAIN TEXT in this script,
# but is sent over the public network encrypted by SSL/TLS

# as of Nov-2009 this is the form of an AT&T SMS destination
RECIPIENTS = ['555134567@txt.att.net']

msg = 'From: 5557654321\r\n'  # put a valid SMS response number here (If you
want)
msg += '\r\n'                 # add a blank line to demarcate between HEADER and
BODY
msg += 'TK101: Level 46prc\r\n' # here is the message - without the blank line
this
                                # would be discarded since it looks like invalid
header

mailServer = smtplib.SMTP('smtp.gmail.com',587)
mailServer.set_debuglevel( True)
mailServer.ehlo()
mailServer.starttls()
mailServer.ehlo()
mailServer.login(smtpuser, smtppass)
mailServer.sendmail(smtpuser, RECIPIENTS, msg)
mailServer.close()
```

Example output

Enabling SMS with default handler of Python, then running this script below results in the output below that (remember that the actual 'message from' is meaningless and the 'FRM:' was created from the SMTP email header

```
import digisms
import time

def ping_callback(a):
  print """\

Message from: %s
at: %s
====================
%s
====================
""" % (a.source_addr, a.timestamp, a.message)
  return

h = digisms.Callback(ping_callback)
while( True):
    print '.'
    time.sleep( 15.0)
```

```
Message from: 1010100006
at: 09/11/06,14:44:13-32q
```

```
====================
FRM:5557654321
MSG:TK101: Level 46prc
====================
```

Notice that the entire text message of "**FRM:5557654321\r\nMSG:TK101: Level 46prc\r\n**" arrives as the text body of the message. Replying to the source address of 1010100006 will have no useful effect.

Sms sample Digi to Digi

SMS Sample showing Digi Python sending and receiving messages

Using these building blocks you can create simple poll-response systems. One of the nice things about SMS is it allows sleeping systems, where the messaging system buffers messages on your behalf.

Sending messages

This script sends a block of messages to remote SMS peers, which may be other Digi devices.

```
import digisms
import time

addresses = ["15551234567", "15551234568"]
num_messages = 10

for address in addresses:
  for i in xrange(num_messages):
    cur_time = time.strftime("%a, %d %b %Y %H:%M:%S", time.gmtime())
    msg = "%s: Message %s" % (cur_time, i)
    digisms.send(address, msg)
    time.sleep( 1.0)
```

Receiving messages

This script sleeps forever, display any SMS messages received. Note that is this script exits, the callback might go out of scope and message receipt stops.

```
import digisms
import time

def ping_callback(a):
print """\

Message from: %s
at: %s
====================
%s
====================
""" % (a.source_addr, a.timestamp, a.message)

h = digisms.Callback(ping_callback)

while( True):
    print '.'
    time.sleep( 15.0)
```

TLS/SSL

This category covers TLS and SSL communications with Digi Products.

Tlslite

This is a basic tutorial on how to install tlslite on a Digi ConnectPortX Gateway for TLS / SSL secure communications. This allows use of newer SSL v3.1/TLS 1.1 on ConnectPort, which by default only supports SSL v3.0/TLS 1.0. It also allows richer support for SSL certs than is native to Python 2.4.3. TLSLite is written completely in Python.

A special thanks goes out to clohfink and DigiGuy42 over at the Digi Forums.

Download tlslite

Download a copy of the open source tlslite package. This article uses tlslite version 0.3.8.

Remove unneeded modules (optional)

- Unzip the tlslite package and navigate to the "tlslite-0.3.8\tlslite-0.3.8\tlslite" subfolder (this is the root directory for the tlslite package).
- Inside the "Integration" folder, remove any modules you don't need such as: IMAP, SMTP, POP3, Asyncore, etc...
- Remove the corresponding import statements (based on which modules you deleted from the "Integration" folder) from the "api.py" file located in the root directory.

Add missing Python modules

Note Depending on which modules you removed from the previous step, the required Python files that your installation will need will vary.

- Zip up the contents of your tlslite root directory into a file (tlslite.zip).
- Upload this file to your gateway's Python directory (you can use the web interface to easily upload).
- Telnet into your gateway from a command prompt and attempt to import the tlslite package with the following command/code:

```
            #> Python

>>> import sys
>>> sys.path.append("WEB/Python/tlslite.zip")
>>> from tlslite import *
```

This will return import errors for missing modules like the error below:

```
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "WEB/Python/tlslite.zip/api.py", line 52, in ?
  File "WEB/Python/tlslite.zip/SharedKeyDB.py", line 7, in ?
  File "WEB/Python/tlslite.zip/BaseDB.py", line 3, in ?
ImportError: No module named anydbm
```

You will need to add the missing modules shown in errors like the one above to your "tlslite.zip" file. You can get these missing modules from your local installation of Python (Digi Gateways currently use

Python 2.4.3). On a Windows machine, for example, the local Python directory would be "C:\Python24\Lib". So for the above error you would copy the file "C:\Python24\Lib\anydbm.py" to your "tlslite.zip" file, upload it again to your gateway, reboot it, and then attempt to import the package again. Continue to attempt to import tlslite, adding the required files as necessary to your zip file (don't forget to re-upload and reboot), until you are able to successfully import the package.

Note You will definitely need the "dumbdbm.py" file copied from your local Python installation to your "tlslite.zip" file. This resolves the following error:

```
Traceback (most recent call last):
File "<console>", line 1, in ?
File "WEB/Python/tlslite.zip/api.py", line 52, in ?
File "WEB/Python/tlslite.zip/SharedKeyDB.py", line 7, in ?
File "WEB/Python/tlslite.zip/BaseDB.py", line 3, in ?
File "WEB/Python/tlslite.zip/anydbm.py", line 62, in ?
ImportError: no dbm clone found; tried ['dbhash', 'gdbm', 'dbm', 'dumbdbm']
```

Good luck!

References

This tutorial originated from a thread posted on the Digi Forums. It was originally seeking help on how to install PyOpenSSL (which is not currently possible), and led to tlslite as the solution of choice since it is written in pure Python. You can check out the thread here.

Third Party Devices

This category covers information about Third Party products that can be connected / used with Digi Gateway Products. The products are typically used in Demos.

Device Cloud easy demo

Purpose

This page is supposed to introduce you into a sample Device Cloud application. These are the special points that are demonstrated in this demo:

- The demos shows the advantages of Device Cloud and our Drop in Networking products:

```
  - Build local intelligence with the Python programming engine that works even
without Cellular Connectivity
  - Connect to the Devices easily via Device Cloud without knowing the IP address
```

- The local intelligence toggles power control adapter either based on sensor info or I/O toggle (Push button)
- Remote connection to the demo is done via a hosted website that includes the access via webservices (XML Messages)

```
  This website shows as well the Energy information provided by the smartplug
(load, work, Current, Voltage..)
```

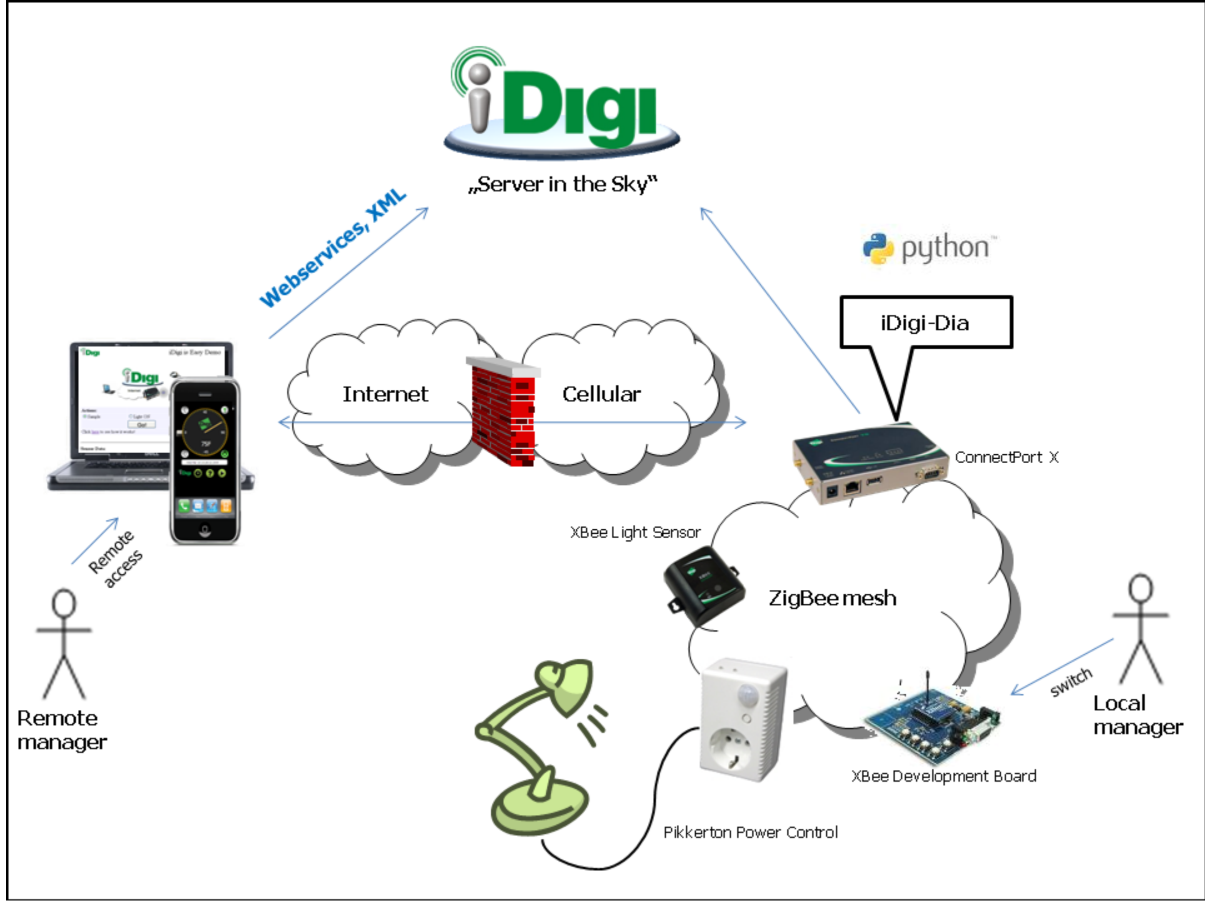
- It is also possible to connect to the device via a smartphone, showing this website.

Requirements

- ConnectPort X gateways (CPX) with cellular connectivity
- Power control device (here Third Party device: Pikkerton router)
- XBEE sensors
- XBee Development Board
- Sample power plugged device (e.g. lamp)

Introduction

The Demo configuration is shown as below:



Files

The ConnectPort X gateways (CPX) has to be equipped with these Python files:

- dia.zip (generated with make command)
- dia.py
- DigiXBeeDrivers.zip
- Python.zip
- zigbee.py

To generate the dia.zip shown above, you also need the complete DIA demo folder which can be downloaded in the last section of this page.

Getting started

1. first step...
2. second step...

How it works / How it looks

1. Local intelligence

- Switch 2 on the XBee Development Board turns on the power control adapter. => Light on.
- Switch 3 turns it off
- Switch 4 activates the "auto mode": If the /L/T/H Sensor Adapter measures a light value below a specific level, the power control is turned on, otherwise it is turned off.
- LED 1 represents the state of the power control adapter.
- LED 2 represents the state of the auto mode.

Please visit IDigi Easy Demo Details - Support for iDigi Easy Demo for details.

2. Remote Access

   - You can access to the power control information via the Device Cloud server and using any browser-enabled device.
   - A dedicated web site allows you to remotely change the state of the demo (switching on/off manually or (de-)activating the auto mode.

Here is a screenshot of the iDigi Easy Demo web interface:

Interesting code

We will give you a short introduction in the key source code, soon with more detail.

Beside the standard DIA library, you need three new pieces:

```
 - device driver for the smartplug
 - presentation that contains the local intelligence
 - the .yml file, here is an example
```

Source

IDigiEnergyDemo_Extrafiles.zip

GE Ventostat CO2 ZigBee monitor

Ventostat® Wall Mount CO2, Humidity and Temperature Transmitters

- Vendor Product Page: https://www.instrumart.com/products/18180/telaire-ventostat-series-co2-monitor
- Product Brochure: http://www.instrumart.com/assets/Ventostat-B_datasheet.pdf
- Model Number: T8100-HD-ZB (you won't find on web site)

GE Sensing Telaire Products sells a Ventostat variant with Digi ZigBee - with the S2C SMT Em357 module. You will need to talk to GE sales to obtain this because they have a variety of wireless variants from many sources, yet do not include on their web site to reduce customer-confusion.

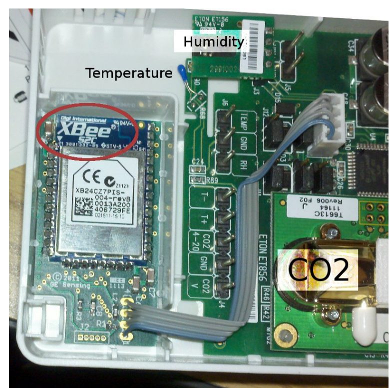
In a nutshell, it is like a thermostat which also measures the carbon-dioxide in the room, which allows the ventilation system to increase or decrease energy use based on room occupancy. Why is that important? Easy - consider my office cubical, which is probably much like your own working space. On weekends the CO2 level is about 400PPM, which is nearly outdoor quality. Great ... but in summer, that means my employer is paying too much to suck in fresh (hot) air which it must cooled down - just so that the people who ARE NOT AT WORK can feel better. In contrast, during the week the CO2 level climbs to 800-1000PPM, which perhaps means they do not intake enough fresh air.

The 'VentoStat' allows the ventilation system to increase fresh air intact when more people are 'breathing' inside, and reduce fresh air intact when less (or no) people are 'breathing' inside. Tests by the US DOT have shown this can cut an easy 17% off the energy costs related to ventilation, giving payback in months. Is CO2 dangerous? Not in levels under several tens of thousands of PPM. Some people might complain about 'stuffiness' when it climbs over 1500 or 2000 PPM, but in truth CO2 is more a 'canary gas' used as an indirect measure, in that as CO2 climbs, so does body-odor, humidity, perfume and food smells - all things which make employees feel the room is stuffy and unhealthy.

The GE VentoStat isn't cheap - expect it to be in the $350 range for low volume. And while you can buy 'air quality' sensors for as little as $50, I have been told there are some major difference between the GE Ventostat and such sensors:

- The $50 sensors tend to measure hydro-carbons, meaning vehicle fumes, new-carpet smells (etc) which is not directly related to human occupancy like CO2 level is.
- The $50 sensors tend to be catalyst-film based, so wear out or need 'refresh' in a few years, whereas the infrared method used by GE claims a minimum of 10 years maintenance-free use.

Ventostat Python code



Example usage

But enough of the sales fluff. Even in a home, one can make use of the Ventostat to manage the low-power fan-only mode in modern furnaces. For example, I have a 3-zone HVAC system at home with a 96% AFUE furnace. The fan-only mode is published as 75-watts - and since I have power monitoring in my home circuit panel, I can confirm that it really is about 75-watts. However, since my home has a fixed-open fresh-air intake, running the fan when not required can make rooms unnecessarily warm or cold in spring or fall. It also is a waste of money to run the fan-only mode when the windows are open.

So one can use a GE ventostat (or similar air-quality product) to feed a CO2 level into the Device/DIA, then use an alarm function to enable/disable the HVAC fan-only mode. The details of this are unfortunately outside the scope of DIA since you also need a DIA-driver for your HVAC system. My own furnace uses RCS TR16 the RCS TR16 thermostats based on RS-485, which I talk wirelessly to via Robust DataComm Xbee/485 adapters which support the 16vac supplied by my furnace to the thermostats.

Theory

The Ventostat uses a simple ASCII protocol. For example, to query all 3 readings at once, you send the string '$33$88\r\n' and you will receive a response such as '719, 35.3, 23.3\r\n', which is CO2, humidity and temperature respectively. Therefore the Digi XBee is in 'AT Transparent Router' mode, and the core DIA driver is based upon a common XBee serial driver which handles the details for sending and receiving strings.

Note: be careful when reflashing the XBee, for the Ventostat's serial port must be (19200,8,N,1), which is not the Xbee default. GE sells a special USB-serial cable which allows you to use XCTU directly with the internal S2C XBee module.

Actual code ZIP

This file contains the files which must be merged with the DIA 2.x.x code.

GE_Files_Dia_2_1.zip

File list

```
vento_readme.txt
REM These YML samples are the same, just two names
cfg\ventostat_ge.yml
cfg\dia.yml
```

```
REM The GE core files including the support for the Min/Max/Average
src\devices\vendors\GE_Energy\__init__.py
src\devices\vendors\GE_Energy\ut_minmaxavg.py
src\devices\vendors\GE_Energy\ventostat.py

REM My own 'DeviceBase' since I find the Digi DIA one to unhelpful & not very
object oriented
src\samples\annotated_sample.py
src\devices\vendors\robust\__init__.py
src\devices\vendors\robust\robust_base.py
src\devices\vendors\robust\robust_xbee.py
src\devices\vendors\robust\robust_xserial.py
src\devices\vendors\robust\prodid.py
src\devices\vendors\robust\xserial_util.py
src\devices\vendors\robust\avail_base.py
src\devices\vendors\robust\parse_duration.py
src\devices\vendors\robust\sleep_aids.py diff\src\devices\vendors\robust
```

Example YML fragment

This shows some of the power of my own device base. It creates a 'description' channel, plus as configured below I get 1 reading every 5 minutes synchronized to the clock, so on the hour, then at 5, 10, 15 (etc) minutes after the hour. Does that clean & fancy timing matter? Does seeing the value in DegF matter? Not in the cosmic sense, but it's what I want!

These settings are better described in the vento_readme.txt file.

```
  - name: Z1CO2
    driver: devices.vendors.GE_Energy.ventostat:Ventostat
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:52:94:b2!"
        dev_poll_rate_sec: '5 min'
        dev_poll_cleanly_min: True
        degf: True
        add_statistics: True
        dev_desc: "Living Room Zone CO2 sensor"
```

Example Channel_Dump

- The channels basic to the Ventostat class: 'co2', 'humidity', 'temperature', 'error', 'version'
- The optional Ventostat statistic channels enabled: 'co2_stats', 'hum_stats', 'tmp_stats' (the format is like "CO2, min=917, avg=926, max=1026")
- The channels from my own device & Xbee Base: 'availability', 'online', 'description' ('availability' and 'online' relate to XBee health and the percentage of lost heartbeat messages - in my Xbee base, all routerss send IO data by default every 60 seconds as a heartbeat of mesh health.)

These channels are better described in the vento_readme.txt file.

```
Channel                   Value              Unit     Timestamp
------------------------- ------------------ -------- -------------------
availability              99.8               %        2012-07-19 19:07:24
co2                       947                PPM      2012-07-19 19:11:00
co2_stats                 CO2, min=917, avg=          2012-07-19 19:11:00
description               Co2 sensor by Desk          1970-01-01 00:00:00
error                     False                       2012-07-19 19:02:25
hum_stats                 HUM, min=53.4, avg          2012-07-19 19:11:00
humidity                  53.4               RH       2012-07-19 19:11:00
```

```
online                   True                        2012-07-19 19:02:44
temperature              74.3               F        2012-07-19 19:11:00
tmp_stats                TMP, min=73.9, avg          2012-07-19 19:11:00
version                  104, 2011/08/12 11          2012-07-19 19:02:25
```

Motion Detection with XBee

How to cub-class a DIA driver.

This page covers how to leverage an existing DIA driver without modifying the original file.

Motion detector example

In this example, I wish to connect a common Motion-Detector/Glass-Break device with relay contacts ('dry-contacts') via ZigBee to Device Cloud and DIA.

The motion-detector sensor requires 12vdc to run, so I used an older XBee DIO Adapter which runs on 9-30vdc. With a 12vdc supply, the entire setup can be powered including directly driving 12vdc relays and 12vdc low-voltage garden-style lighting.

Technically, the existing DIA DIO driver could be used to read the sensor contacts and drive the lamp output, however the motion detection event is very short-lived. It may be true for only a few seconds, and turning on a lamp for a few seconds is not what is required. Instead, a timer is required to turn the lamp on for a few minutes. In theory one could use the DIA transforms to do this, but at present they are stateless, so cannot function as timers.

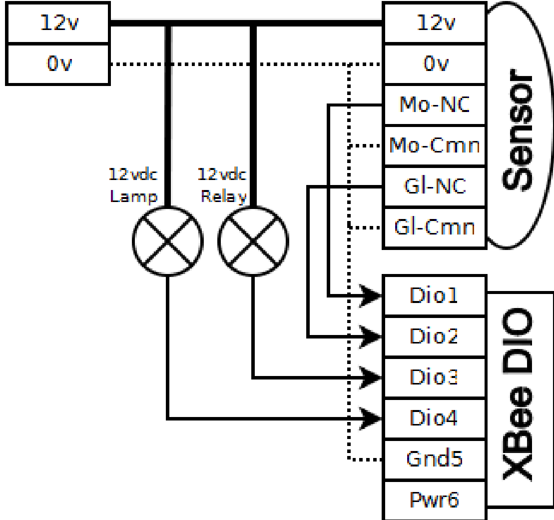
So one has two choices:

- Clone and rename the existing DIA DIO driver, and modify it directly.
- Create a new subclass of the existing DIA DIO driver which adds the new functionality.

In this demo example I chose to subclass. The new driver allows the existing DIO driver to manage incoming data and update the standard DIO data channels. Then it reads the input channels and adjusts the output channels as required.

Ideally, the time the lamp is held on should be a setting, which complicates the subclass slightly. That is left for a future TODO - including the support for the glass-break input and the 12vdc relay to drive brighter AC lights. If all goes well, my final design will have a programmable XBee to manage the light and relay control locally within the XBee adapter itself.

Hardware example



Wiring the system is quite strait forward. The items required:

- XBee Digital I/O Adapter preferably with 9-30vdc supply, but if you have a 3-6 vdc model, buy a PCB to step-down 12 vdc to 5 vdc. For example, ebay seller electronics-salon sells a nice assembled model for us$10 which I have used on several projects.
- Any 12 vdc supply large enough to power everything - anything larger than 1.5 Amp should be okay. As is, the 10-watt halogen garden lamp (at about 0.8 A) is the most power-hungry device.
- Motion Sensor, such as Bravo BV-500GB by DSC, sold by smarthome.com
- 12 vdc Low-Voltage Lamp, such as the Malibu brand in the image, purchased from Home Depo. Once the halogen lamp fails, they are easy to replace with LED versions from superbrightleds.com
- 12 vdc Power Relay with AC-rated contacts, such as Omron LY1F-DC12 from digikey.com which is rated at 15 Amp @ 110 vac. It requires 75m A @ 12 vdc to operate.

Notes

- The 12 vdc power ground link to the XBee DIO Adapter is optional if the XBee DIO Adapter is powered by the 12 vdc supply, but it is required if the XBee DIO Adapter is powered by (for example) the 5vdc wall-wart Digi will supply if you buy the 3-6 vdc XBee DIO Adapter
- The four DIO terminals of the XBee DIO Adapter when used as outputs can only sink current, so pull the 12 vdc low. That is why both the lamp and relay are tied to 12 vdc, with the XBee DIO Adapter acting as ground.
- The power-output (t erminal #6) of the XBee DIO Adapter can only supply 50 mA @ 12 vdc, which is not enough for either the landscape lamp or the power relay. However, it would be enough for direct LED lights or a smaller reed-relay which could power a larger lamp or relay. Moving an 'output' to this terminal would free up one of the XBee adapter IO for use as another input, such as a tamper contact, or perhaps even a sensor to detect if the lamp is really on and emitting light.

- The reason for having both a dim 12 vdc lamp and bright 120 vac lamp (via 12 vdc relay) is I plan to use the dim 12vdc lamp more as a night-light then as security. So based on time of day, sunlight level, and whether the security level is 'we are home' or 'we are out', either or neither lamp may be lit due to motion.
- XBee DIO Adapter DIP Switch settings: 2, 3, and 4 are on.
- For inputs, only terminals 1 & 2 have internal pull-up. To use terminals 3 & 4 as input, you may need to use resisters (for example 10K ohm) to pull the inputs up to 12vdc when floating/open.

YML code

This is just the fragment required for the custom DIO driver. We desire the DIO adapter to fresh the channel status every 30 seconds, plus the channels will be updated any time they change. The four IO are assigned like this:

- Channel 1 input is the Motion-Detect NC/normally-closed contact - it opens when motion is seen.
- Channel 2 input is the Glass-Break NC/normally-closed contact - it opens when the distinctive sound of break glass is heard. This input is for future use, and is ignored now.
- Channel 3 output is reserved for a 12 vdc relay which can drive 120 vac lamps, which would be appropriate for yard or full-room lights.
- Channel 4 output directly drives a 12 vdc garden-style lamp, which is used for demo purposes and night-lighting.

```
- name: motion
  driver: devices.experimental.xbee_motion:XBeeMotion
  settings:
      xbee_device_manager: xbee_device_manager
      extended_address: "00:13:a2:00:40:33:4e:a9!"
      sample_rate_ms: 30000
      sleep: False
      power: Off
      channel1_dir: "In"
      channel2_dir: "In"
      channel3_dir: "Out"
      channel4_dir: "Out"
```

Source code

Here is the entire custom file. Since it uses the normal XBee DIO driver, there is little function required. Basically, this driver allows the XBee DIO driver to process the incoming data, then compares the Motion-Detect input to the current light state.

```
import traceback
import time

from devices.device_base import DeviceBase
from devices.xbee.xbee_devices.xbee_base import XBeeBase
from settings.settings_base import SettingsBase, Setting

from devices.xbee.xbee_devices.xbee_dio import *

class XBeeMotion(XBeeDIO):
```

```
    OFF_DELAY = 30

    def __init__(self, name, core_services):

        ## Initialize the base class
        XBeeDIO.__init__(self, name, core_services)

        # zero means off, else holds time.time turned on
        self.__light_on = 0

        return

    ## Locally defined functions:
    def sample_indication(self, buf, addr, force=False):

        ## allow base class to process data message
        XBeeDIO.sample_indication(self, buf, addr, force)

        # then do our reaction to the status
        now = time.time()

        motion = self.property_get( 'channel1_input').value
        if motion:
            if self.__light_on == 0:
                print 'Motion Seen, turning light on'
                self.set_output(Sample(now, True, "On"), 3)

            # else: print 'Motion Seen, light already on'

            # in all cases, bump light_on time to now
            self.__light_on = now

      else:
            # else no motion
            if self.__light_on != 0:
                # then light is on
                if (now - self.__light_on) > self.OFF_DELAY:
                    print 'No Motion, turning light off'
                    self.set_output(Sample(now, False, "Off"), 3)
                    self.__light_on = 0

                # else: print 'No Motion Seen, light is on, should stay on'

      return
```

Unsupported Third Party Dia Code

The category includes Python code and drivers which might work (or might not!) It is a place to share ideas and things which others might benefit from and contribute to.

DIA Device - alarm clock

Alarm clock device

The Alarm Clock device is a low-speed general resource which can help other devices accomplish simple timed actions. It is designed to work with minutes or hours. **Users who needed timed behavior faster than once per minute should use their own thread and timer logic**.

At present it only offers:

- the ability to trigger a transform (or publish a set) every:
  - 'minute' = once per minute, when seconds = 0
  - 'hour' = once per hour, when minutes & seconds = 0
  - 'six_hour' = once per 6 hours, so at 00:05:00, 06:05:00, 12:05:00 and 18:05:00
  - 'day' = once per day, so at 00:00:00 / midnight
- It can print the line "{dev_name}: time is now 2009-05-31 10:47:00" at any of the above time intervals.

TODO

- Add a cron-like ability for other devices to request publish/sets at times such at 3:27am each Tuesday.
- Consider adding a second 'helper' thread to run complex/long jobs. For example STMP email service might take longer than 1 minute to return control to the alarm_clock thread.

Connections - inputs

There are currently no block inputs.

Connections - outputs

These are the block outputs.

Name	Type	Description
minute	tuple	Published once per minute (will always be True)..
hour	tuple	Published once per hour on the hour (will always be True).
six_hour	tuple	Published once per 6 hours, at 5 min after the hour (will always be True).
day	tuple	Published once per day, at 10 minutes after midnight (will always be True).

Notes

- The tuple sent is of form (time(), tuple(localtime(time())).
- It is converted to a tuple since the same object reference is passed to all subscribers.
- An actual example is: (1244544360.0, (2009, 6, 9, 10, 46, 0, 1, 160, -1))
- Your device might MISS sets/publishes - for example another task might cause the alarm clock to be busy LONGER than 1 minute. So you device should examine the TIME value in the set/publish to determine how many seconds have really occurred since the last event.

Settings

These are the block settings which affect operation; some should be saved to NVRAM while others are dynamic.

Name	Type	Default	Lifespan	Description
tick_rate	int	60	NVRAM	seconds between ticks - change only for debug purposes!
printf	str	'minute'	NVRAM	if and when alarm_clock prints a 'time is now' line in trace, in set ['none','minute','hour','six_hour','day']

Notes

Changing tick_rate does NOT affect the minutes etc, as those are based the real time clock; this only affects how often the thread wakes. Setting it to 15 means the thread wakes up 4 times for even 'minute' event. Setting it to 300 means 4 of 5 minute events are missed.

The Python code

I place this in my src\devices directory. However, it has few dependencies and could be placed anywhere (it affects the YML form).

YML example

This example shows the alarm clock device being named 'tick_tock', plus a second device which subscribes to tick_tock.minute, which allows that device to run once per minute without requiring a thread of it's own. *The HourMeterBlock is NOT part of DIA - it is a custom user-defined device.*

```
- name: tick_tock
  driver: devices.alarm_clock_device:AlarmClockDevice
  settings:
      printf: minute
- name: motor_01_hours
  driver: devices.blocks.counter_device:HourMeterBlock
  settings:
      active_true: True
      input_source: motor_01.channel1_input
      tick_source: tick_tock.minute
```

Sample output

This shows the once-per-minute printf with AIO adapters sleeping for 2 minutes at a time.

```
tick_tock: time is now 2009-06-10 09:23:00

XBeeAIO(smart01): AD0=0000 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank03): AD0=0501 AD1=0000 AD2=0000 AD3=0000 bat=okaytick_tock: time is
now 2009-06-10 09:24:00
XBeeAIO(itank01): AD0=0153 AD1=0000 AD2=0000 AD3=0000 bat=okay

XBeeAIO(ssi01): AD0=0087 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank02): AD0=0502 AD1=0000 AD2=0000 AD3=0000 bat=okay
tick_tock: time is now 2009-06-10 09:25:00
XBeeAIO(smart01): AD0=0000 AD1=0000 AD2=0000 AD3=0000 bat=okay
XBeeAIO(itank03): AD0=0501 AD1=0000 AD2=0000 AD3=0000 bat=okay
```

Python Code

2009jun09_alarm_clock_device.zip

Dec-2009 Enhancements - Events

In the Dec 2009 version I have added the ability to create time-based boolean event channels. So for example, you could create a channel named 'xmas' which is True from 6PM until 10PM, and False the rest of each day. Needless to say, you could subscribe to such a channel from a Digi SmartPlug, and use it to turn On/Off your outdoor Christmas lights.

New Details

This new (Dec 2009) version allows an optional 'action_list' section, following which you can add a number of events. Each event creates a NEW output channel on the AlarmClockDevice - so in the example below the new channel named **tick_tock.xmas** becomes available. In this example, the channel tick_tock.xmas will be True during the period from about 18:00 (6pm) until 23:00 (11pm).

The 'per' setting can also be set to 'hourly', in which case the 'on'/'off' should be minutes of the hour, so an output can be true at some arbitrary minutes-after-the-hour. Note that since one of my goals is to NOT load the CPU for unnecessary accuracy, it is possible the event triggers up to a minute late. So the outdoor lights might turn on at 6:01pm instead of 6:00pm.

My main TODO item is add the notion of which days of the week - if you look into the code you see that the 'do-daily' code is 0x007F, so 7 true bits. The design goal would be to enable causing the event to fire only once per week (say on Wednesday at 3am), or only have it occur on weekdays or weekends.

Example YML for Christmas lights

```
devices:

  - name: tick_tock
    driver: devices.experimental.alarm_clock_device:AlarmClockDevice
    settings:
        tick_rate: 60
        printf: minute
        action_list:
            - event: { 'nam':'xmas', 'per':'daily', 'on':'18:00', 'off':'23:00' }
  - name: xbee_device_manager
    driver: devices.xbee.xbee_device_manager.xbee_device_
manager:XBeeDeviceManager
  - name: my_lights
    driver: devices.xbee.xbee_devices.xbee_rpm:XBeeRPM
    settings:
        xbee_device_manager: "xbee_device_manager"
        extended_address: "00:13:a2:00:40:48:5a:65!"
        sample_rate_ms: 5000
        default_state: Off
        power_on_source: tick_tock.xmas
```

Python code

2009jun09_alarm_clock_device.zip

DIA device - Runtime Totalizer

Runtime Totalizer device

Many systems benefit from tracking how long things run (how long they are on or active or True). This example DIA device logs the time period during which boolean (True/False) samples are True. The totals are saved to an ASCII text file which can be read via the web interface, plus is reloaded when the gateway reboots.

This device is also a nice example of "trade-off". It only totalizes the samples at fixed periods (for example once per minute), and it assumes the input has been in its current state for the entire past minute. It also only saves the NVRAM state at a slow rate, for example once per 15 minutes.

Could you totalize in milliseconds? Yes, of course you could. Is it useful? Probably not. If your goal is to understand how many kilowatt-hours a room light was on per month, then tracking in minutes verse milliseconds is not likely to be statistically interesting. Could you save the totals to FLASH every second? Yes, of course you could, but then your Digi gateway's FLASH might fail within the year. Assuming the gateway only reboots due to power outages once every few months, then a few dozen minutes a season are not statistically interesting.

Fortunately, the Runetime Totalizer device allows you to control the rates. The Python code included does NOT use it's own thread. Instead, it subscribes to an external DIA sample which produces period refreshes of data. This could be any IO sensor which produces data periodically, or the DIA Device - alarm clock.

Configuration and settings

The Setting for the **RunTimeDevice** consist of a paired list of **channel** and **name**. Channel is the input sample to monitor with type Boolean, and Name is the totalized value output with type float. The channel/name list must be balanced, as otherwise the DIA does not treat the YML data correctly.

The **RunTimeDevice** is passive - it has no thread. It requires an external Sample refresh via the DIA's powerful publish-subscribe subsystem to "Push" it into action. Thus these two SPECIAL channels exist (see the YML example below).

- **time_source** is any periodic sample which causes the RunTimeDevice to evaluate and totalize the other channels. In the example below it is the Minute sample from the Alarm Clock Device. The type of data sample is not important; it is the act of update/refresh which causes the RunTimeDevice to process one cycle, and it uses the change in real-time since the last cycle to update the totals.
- **publish_source** is any periodic sample which causes the RunTimeDevice to refresh/update it's own collection of Samples out, plus saves the data to the NVRAM (flash) of the Digi gateway.

These two special-settings are handled this way because the author could not get DIA to handle the YML list of undefined size correctly if the settings did not consist COMPLETELY of these pairs. This might change in the future

Example YML File for a DIO Adapter with 4 inputs

This example defines six (6) channels, two of which are special cases:

- **time_source** causes the totals to be calculated once per minute, but they are not published nor saved to NVRAM/flash at this time.

- **publish_source** causes the totals to be published and saved to NVRAM/flash once per 15 minutes
- **lights, pc_stuff, chargers, light_active** are the four totalizers, which track the on (power-up) time of four external power relays.

```
- name: my_cube
  driver: devices.experimental.runtime_device:RunTimeDevice
  settings:
    - channel: tick_tock.minute
      name: time_source
    - channel: tick_tock.15_min
      name: publish_source
    - channel: relays.relay1_status
      name: lights
    - channel: relays.relay2_status
      name: pc_stuff
    - channel: relays.relay3_status
      name: chargers
    - channel: relays.relay4_status
      name: light_active
```

Example ASCII Text file saved by RunTimeDevice

The text file named **RunTotals.txt** is saved in the WEB/Python/ directory. You can access it from the web interface in the same place in which you see Python files. Literally this is a string representation of a Python dictionary, so order is NOT predictable. Technically you can edit this file carefully and then reboot the DIA system to "reload" changed values.

```
{'lights': 54784, 'pc_stuff': 250684, 'light_active': 101462, '_total_time':
1440243, 'chargers': 0}
```

Download info

Files in the runtime_device.zip

- **readme_runtime_device.txt** is a summary similar to this Wiki page.
- **src\devices\experimental\ia_trace.py** is a simple trace facility which allows increasing or decreasing the chattiness of the print output from DIA - the class RunTimeDevice() inherits from it.
- **src\devices\experimental\string_file.py** is routines to save and reload a single Python object saved as "string", and reload with "eval(string)" - it is a simplistic pickler.
- **src\devices\experimental\runtime_device.py** is the DIA device driver to be referenced in your YML file.

Python code

2009oct12_runtime_device.zip

DIA device - sample rate reducer filter

Sample rate reduction/filter device

Consider a DIA system with 10 tanks and sleeping level sensors which wake once per hour to upload new data to X4 and DIA. Do you really need to push 240 data samples per day over your cost-sensitive cellular link into the Device Cloud? What if on the average day, only 1 or 2 of those tank levels even change?

The DIA 'Filter Device' allows you to add very simple intelligence around the forwarding of new data. In this tank example, it would allow the DIA to apply the logic "**update Device Cloud only when the tank level changes by at least 1%, plus never update more often than once per hour, but always update at least once per day**." This could reduce the total data samples moved over cellular to only 15 or 20 samples per day - while still allowing the central host to see tank level changes within an hour of the change.

The filter_device acts as a simple, time-aware filter (or transform) which applies process knowledge to reduce the frequency of sample updates. In the above example, the DIA configuration would start with ten (10) sensor device objects which blindly produce new data samples every hour - so 240 changes per day. These might have names like ['TNK01in', 'TNK02in' ...] and so on, which means 'TNK01.channel1_value' might be the level in the first tank.

Ten (10) new filter_device objects would be created to selectively copy or mirror the output of those sensor device objects, having names like ['tank01', 'tank02' ...] and so on. They might apply a logic with the Python dictionary: { 'delta':1.0, 'atleast':'24 hr', 'atmost':'59 min' } They might also rename their 'input' to become output as 'level'. So they create new channels such as 'tank01.level' which change at most 24 times per day, but at least 1 time per day. If idigi_db uploads the channels from the filter_ device instead of from the sensors, then you will have less cellular traffic.

Filter conditions / configuration

By default each filter_device supports at most 4 channels - this is hard-coded within the filter_ device.py since it creates database items EVEN for channels which are NOT configured. The syntax of the filter_device is closer to Python to reduce the system overhead in created such 'mirror channels' on a large DIA system.

The filter_device is configured with:

- A tag name ['tag1', 'tag2', ... 'tagN'] with a source channel name such as template.counter or m3_01.temperature. By default, the outgoing filter_device channel has the same name as the source. Duplicate names causes exceptions to be thrown.
- An optional rename ['rename1', 'rename2', ... 'renameN'] which can over-ride the source channel name. For example, instead of a name like 'distance', you might want 'level'.
- A filter condition dictionary ['filter1', 'filter2', ... 'filterN'] which defines the list of conditions to suppress or cause the output channel to be updated.

Filter conditions

- All channels will produce an initial sample even if the conditions are not met.
- An empty dictionary such as "filter1: {}" causes the data to always propagate, otherwise by default the data is NOT propagated if conditions exist.
- The "**atleast**" clause over-rides all others and produces a new data sample AT LEAST as often as configured time in SECONDS. Examples are { 'atleast':'24 hr'} or { 'atleast':86400}, where the string '24 hr' or '1 day' would be internally converted to 86,400 seconds.
- The "**atmost**" clause also over-rides others and suppresses new data samples to happen no more that the configured time in SECONDS. Using filter conditions to track 'events' might result in data loss if 'atmost' suppresses samples. Examples are { 'atmost':'15 min'} or { 'atmost':900}
- The "**delta**" clause produces a new data sample if the value changes up or down by at least the value configured. The value depends on the units, so for a level measured in inches the delta-value would be inches. Examples are { 'delta':0.5} or { 'delta':25}
- The "**below**" clause produces a new data sample if the value is less-than (below) the value configured. There is no hysteresis, so an 'atmost' clause may be required to slow down new data samples.
- The "**above**" clause produces a new data sample if the value is greater-than (above) the value configured. There is no hysteresis, so an 'atmost' clause may be required to slow down new data samples.
- The "**equal**" clause produces a new data sample if the value is the same as the value configured. Be warned that sensor inputs treated as floats will rare 'equal' a constant such as 2.5 or 100. (It is a TODO item to offer a solution for this)
- The "**round**" clause is processed ONLY if the sample is being produced, and it forces the sample to become a float. It is primarily of use when data uploads are handles as ASCII text. For example, it makes no sense to upload a temperature as "21.094400000000007" if the accuracy is +/- 1 degree. Adding the condition { 'round':1 } would round the float to 21.1000000, which normally will be ASCII encoded as only "21.1"

Example YML file for TemplateDevice

This example produces two new channels with the following characteristics:

- **m3_01.counter** only produces a new sample (SETs a new output) if the data sample is greater than 6. However it produces atleast 1 sample every 5 minutes (300 seconds) and never produces more than 1 new sample every 1 minute and 10 seconds (70 seconds). This channel inherited its name from the source channel template.counter
- **m3_01.other** produces a new sample (SETs a new output) after the source channel has changed at least by a value of 3 (example: 1, 4, 7 etc). It has no time restrictions.

```
devices:

  - name: template
    driver: devices.template_device:TemplateDevice
    settings:
        count_init: 0
        update_rate: 1.0

  - name: m3_01a
    driver: devices.filter_device:FilterBlockDevice
    settings:
        tag1: template.counter
        filter1: { 'above':6, 'atleast':'5 min', 'atmost':'1 min 10 sec' }
        tag2: template.counter
        rename2: 'other'
        filter2: { 'delta':3 }
```

Example YML file for Massa M3 ultrasonic sensor

The Massa M3 battery powered device wakes on schedule - for example once per hour - and pushes new data into the Device Cloud/DIA channel database. However, this does NOT mean that a new tank level has been seen. Also the Massa M3 driver produces about 12 channels, all which are 'refreshed' with a new sample at this rate. So just uploading all changed channels wastes a lot of cellular bandwidth.

This example produces four new channels with the following characteristics:

- **m3_01.distance** produces a new sample if the level changed by at least 0.5 inch, but produces atleast 1 sample every 3 hours and never more than 1 new sample per hour.
- **m3_01.temperature** produces a new sample if the temperature changed by at least 2 degrees, but produces atleast 1 sample every day and never more than 1 per 2 hours.
- **m3_01.target_strength** only produces a new sample if the ultra-sonic signal quality is less than 50%, never more than 1 per hour.
- **m3_01.battery** produces a 1 new sample every day.

```
devices:

  - name: m3_01
    driver: devices.vendors.massa.massa_m3:MassaM3
    settings:
        xbee_device_manager: xbee_device_manager
        extended_address: "00:13:a2:00:40:4b:ae:b0!"
        poll_rate_sec: 3600
        sample_rate_sec: 3600

  - name: m3_01a
    driver: devices.filter_device:FilterBlockDevice
    settings:
        tag1: m3_01.distance
        filter1: { 'delta':0.5, 'atleast':'3 hr', 'atmost':'1 hr' }
        tag2: m3_01.temperature
        filter2: { 'delta':2.0, 'atleast':'24 hr', 'atmost':'2 hr' }
        tag3: m3_01.target_strength
        filter3: { 'below':50, 'atmost':'1 hr' }
        tag4: m3_01.battery
        filter4: { 'atleast':'24 hr' }
```

Download info

Files in the filter_device.zip

- **readme_filter_device.txt** is a summary similar to this Wiki page.
- **src\common\helpers\parse_duration.py** is a Python routine to convert strings like '24 hr' into msec or sec. It supports the tags ['ms','sec','min','hr','day']
- **src\common\helpers_test_parse_duration.py** is a Python regression test routine.
- **src\devices\filter_device.py** is the DIA device driver to be referenced in your YML file.

Python code

2009oct09_filter_device.zip

TODO

- The tag interface is cumbersome and wasteful - having tags named tag1/filter1 and tag2/filter2 is not ideal. A method should be added to accept them all as tag/filter and be handled properly.
- Since 'equal' doesn't work with floats, adding a 'nearly' tag to match when a float is within some tolerance of a target would be good.
- Enable inverting boolean conditions, so a sample of 'True' can be forwarded as 'False'.

DIA event uploader

Uploading events or discrete data samples to DIA

Comparing iDigi_DB to iDigi_Upload

The stock iDigi upload presentation (named iDigi_DB) assumes you only want a sampling (or subset or snap-shot) of data samples handled by DIA. For example, if 20 tanks send new level reading to Dia once per minute, this amounts to 1200 samples per hour. Yet the Device Cloud host may not desire any more than 20 samples per hour - just one per tank per hour. Therefore the stock iDigi_DB presentation is configured with an **interval** setting (for example: 3600 seconds or 1 hour), then all configured channels are sampled and uploaded to Device Cloud once per hour.

This design does not support uploading real time events. For example, if a door sensor is monitored, then Device Cloud will only see the door status on the hour (the sample interval). Device Cloud will not record that the door was opened three times during the hour.

iDigi_Upload is a custom adaptation of the original idigi_db.py file, modified to save a copy of every new channel sample for upload. These are saved in a cache and only cleared after successful upload.

Description	iDigi_DB	iDigi_Upload
Are discrete events uploaded to Device Cloud?	No - only the I/O status at the sample interval	Yes - every new sample is uploaded
If upload fails, are samples saved and retried?	No - new samples will be taken at the next sample interval	Yes - copies are made of every sample and only deleted after successful upload (unless cache grows beyond configured limit)
If a channel doesn't change, is it uploaded every interval?	Yes - all channels are sampled at each sample interval	No - copies are made only when sample change
Is the upload size limited per interval?	Yes - at most one sample of every channels is sent	No - every changed sample is uploaded
Is memory usage limited?	Yes - existing samples are used to create upload	No - a duplicate copy of every changed sample is held until upload

New settings

iDigi_Upload supports the original iDigi_DB settings, plus adds two new settings:

cache_size

Is the maximum number of bytes of flash to use buffering for upload. It defaults as 0, which disables this drastic changes of behavior.

- When **cache_size** = 0, idigi_upload holds all data in memory only. Restarting the gateway will discard your data.

- When **cache_size** is set > 0, then idigi_upload queues a deep-copy of every new sample first in memory, then saves this to flash approximately once a minute. You will see files such as 'data_cache.txt' appear and disappear from your Python directory. These are cleared only after upload or when the caches fills, then in FIFO design the oldest data is lost. NOTE that the design keeps a backup when deleting the cache, so the actual size of flash required will be twice the configured amount.

clean_minute_interval

Is an alternative to the base **interval** setting. It accepts only minutes in the set (0, 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60)! It causes upload ON THE HOUR, then minute intervals in a predictable pattern. So setting 10 cause uploads at 00:00:00, 00:10:00, 00:20:00 and so on.

- Setting **clean_minute_interval** to zero (0) causes the base interval setting to be used.
- Setting **clean_minute_interval** to non-zero over-rides the interval setting.

Download Info

Python Code

This version requires Dia 1.4.14 or newer. The files includes are:

- src\common\helpers\sleep_aids.py
- src\samples\annotated_sample.py
- src\presentations\idigi_upload__init__.py
- src\presentations\idigi_upload\idigi_upload.py
- src\presentations\idigi_upload\string_file.py
- 2012feb24_idigi_upload.zip

Twitter DIA example

Overview

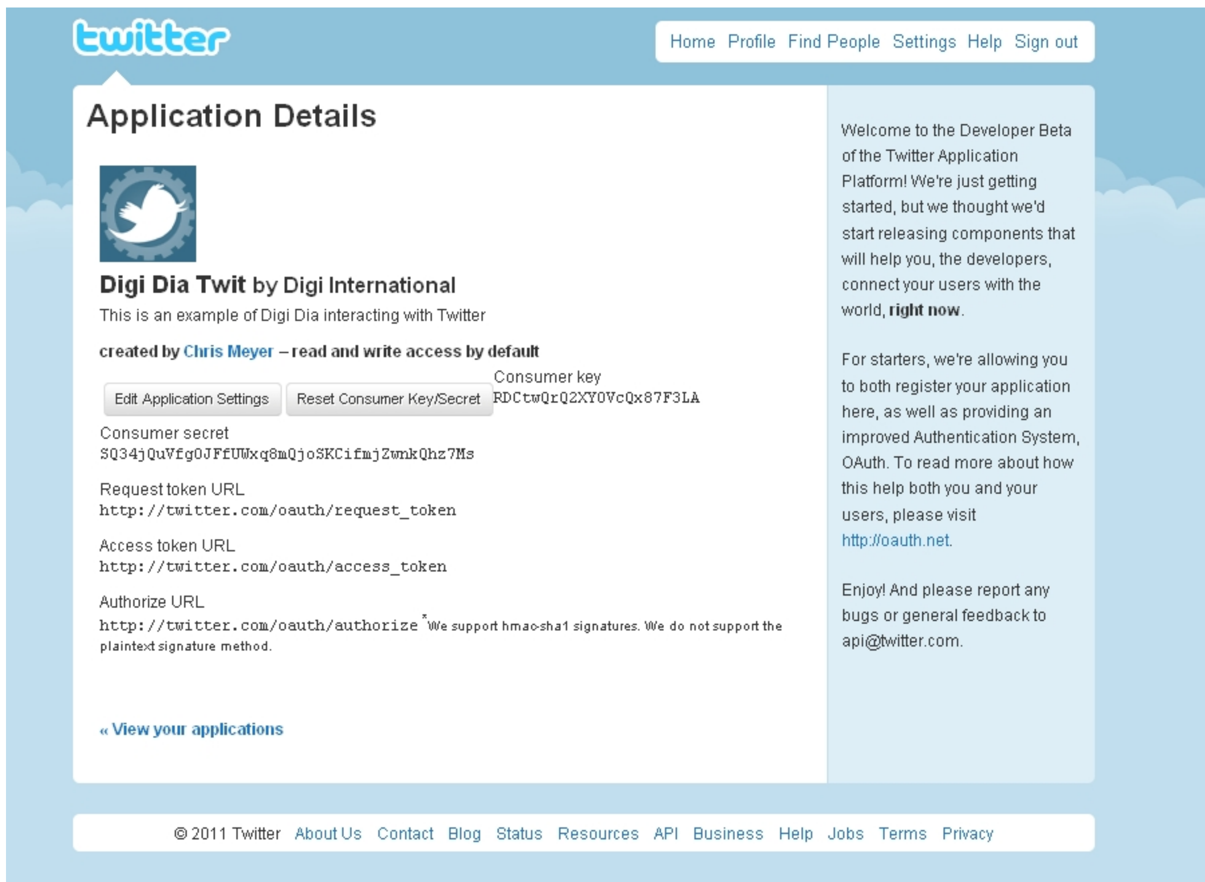
This simple DIA example interacts with the Twitter - API.

Prerequisites

You should register at www.etherios.com/devicecloud and download DIA. After that, read the Getting Started guide and begin looking at the simple drivers (found under ../src/devices in the DIA source code tree). You should have a basic working knowledge of DIA(how to start / stop & use basic drivers / presentations) before proceeding.

This demo requires the tweepy Python library. Tweepy can be found here: https://github.com/joshthecoder/tweepy.

Register a new twitter application by going to http://twitter.com/apps/new. When you submit the form, you will be presented with a page that shows your consumer key and consumer secret. Take note of these.



Download the following archive Media:Twitter_Dia.zip, unzipping its contents within the src/presentations directory of the DIA. Add the following to your yml config file (by default this will be the *cfg/dia.yml*) and change the settings accordingly, or remove the optional settings.

```
  presentations:
    - name: twitter
      driver: presentations.twitter.updater:Updater
      settings:
         interval: 60
         status_template: "uploaded from the iDigi Dia: %s"
         sample_template: "%s=%s"
         channels: [template.counter, template.adder_reg1]
```

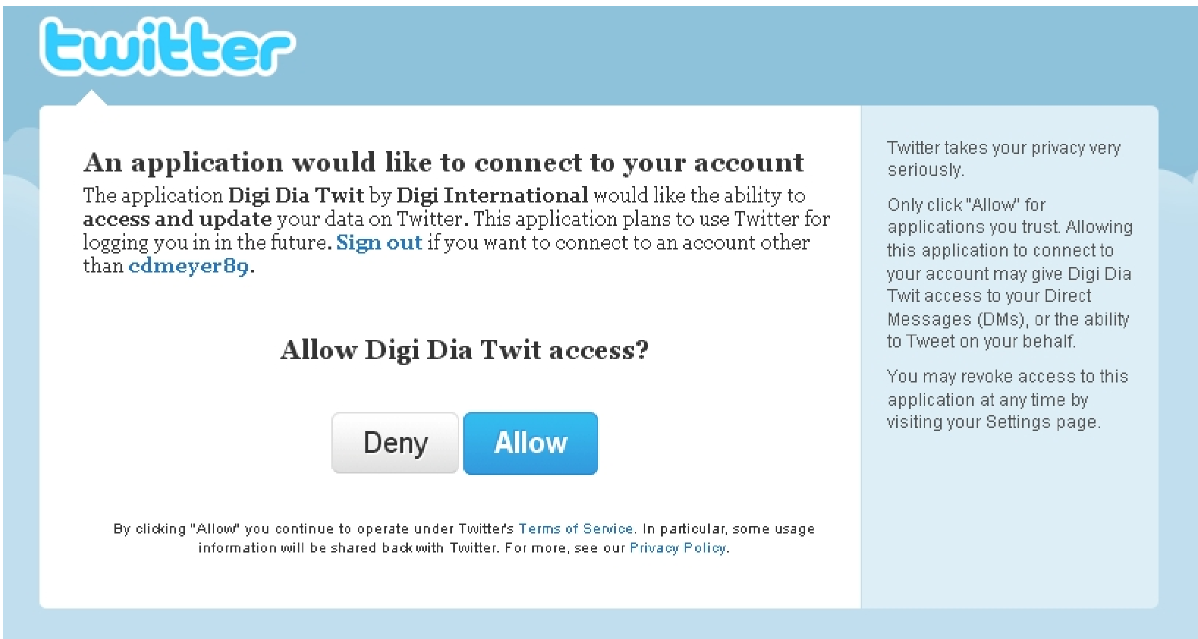
Here is a breakdown of the different options:

```
interval optional
  How often to "tweet". If this setting does not exist it will default to 180
seconds
status_template optional
  Format of the tweet, with a %s where to put the sample data.  If left blank
this will default to "iDigi Dia Update: %s"
sample_template optional
  Format of the samples, two %ss required, first for the channel name, second for
sample value.  defaults to "%s:%s"
channels optional
  List of channels the module is subscribed to. defaults to all channels
```

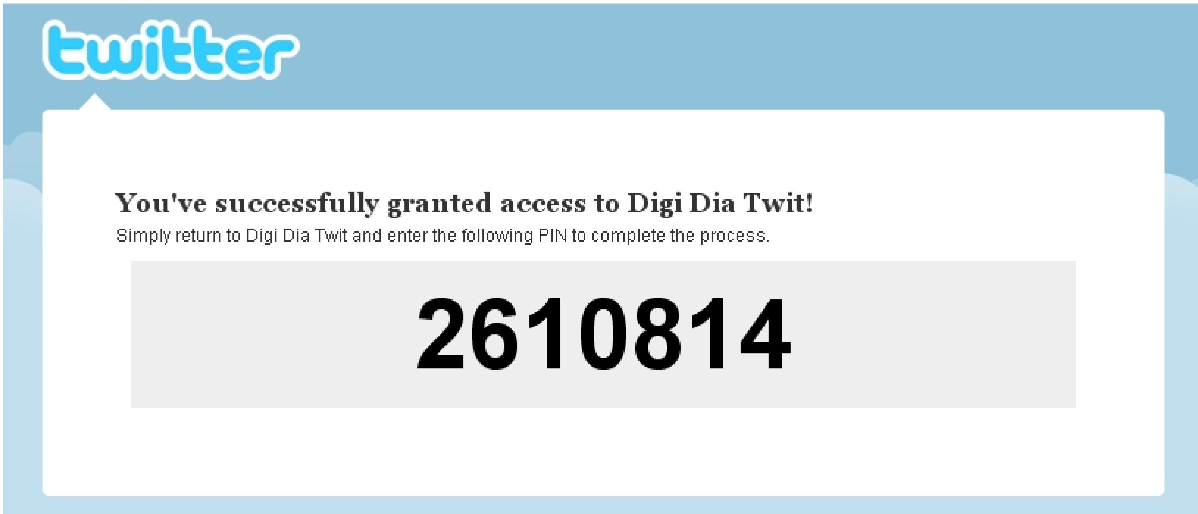
Re-compile DIA incorporating the above elements under presentations of the cfg/dia.yml file. You can do this via the command line (Python make.py dia.yml) or using the Digi IDE for DIA.

Run the DIA When it's running, the first thing you will be asked for is your consumer key. Enter the consumer key that was shown when you registered the twitter app. You will then be asked to enter your consumer secret. Do the same as before.

Next, you will see a prompt that says P*lease verify URL*: followed by a long URL. Type that URL into your browser and search for it. You will see a twitter page that asks whether you want to allow the application to connect to your twitter account. Click **Allow**.



You will then be presented with a PIN. This is the PIN you must enter into the command prompt.



Once you enter the PIN, DIA will begin tweeting to your account!

Hint

```
  If DIA fails with an error message "No module named gopherlib" followed by
"global name 'twitter' is not defined"
  The reason could be that the module gopherlib is excluded in the DIA build
process.
  Please adjust the file tools\digi_build_zip.py and comment out the gopherlib
line like:

  # Other things that can get in the way, and don't seem likely
```

```
    #"gopherlib",
    #"ftplib","pydoc",
```

Web Access

These pages explain how to use web servers, browsers, HTTP/HTML and port 80 with Digi Products.

Module: digiweb

Provides an interface to allow the Digi web server to call into Python and allow Python code to handle web page requests.

The module contains a single type named Callback that is initialized with a callback function as follows:

Summary

digiweb.Callback(cb) -> handle

Returns a handle to be used for deregistration later. The callback will be handled as long as the callback handle exists. If the handle goes out of scope or is deallocated, the callback will be unregistered.

Callback functions should be of the form: > cb(type, path, headers, args) -> None | (type, data)

parameters

type

HTTP request type. Can be 'HEAD', 'GET' or 'POST'.

path

URL path component.

headers

A dictionary containing header information from the client. Contains host, agent and referer strings

args

Request Type specific

GET

URL-decoded form arguments in a dictonary

POST

Body of POST request in a single string

For either request type, If there is nothing provided in the request, None will be specified instead.

return values

- **None** if this path or request type cannot be handled by the callback routine.
- 2-tuple
  - type: For a successful request, determines the MIME type the server will report. Choose from TextHtml, TextPlain or TextXml.
  - data: Body of response content.

Limitations & suggestions

- Remember, call back functions will be executed in another thread. Shared data will need to be protected from race conditions due to concurrent access.

- Do not try to use common page names like "index.htm", for while they may NOT exist within the Digi product, the Digi web server (as do all web servers) treat certain default pages as magic.
- If you are creating a page name for users to remember, you may not want to require any extension at all, plus make them case insensitive. Telling users to use "192.168.1.1/status" is far easier than remembering if it is "status.htm" or "status.html" or "Status.HTML".
- To add an image to your page, use this example HTML tag . The images must be uploaded to your Python directory by the same web ui page as for the Python code. Be very careful of the case, as 'Image.jpg' is not the same as 'image.jpg' or 'Image.JPG'. Also, your flash space is highly limited so keep image small and highly compressed. JPG at lower than normal compression seem the best. BMP are too large. PNG may be 'functionally too rich', so larger than required. GIF also tend to be larger than JPG unless you only have a few colors (line drawings etc).

Examples

The following code will serve any unrecognized path in the system with information on the request for one minute then exit.

```
import digiweb

template = """
<html><head><title>Request info</title></head>
<body>
%s request for path '%s'
<hr>
Headers: %s
<hr>
Args: %s
</body>
</html>
"""

def pageinfo(type, path, headers, args):
    return (digiweb.TextHtml, template % (type, path, headers, args))

if __name__ == "__main__":
    import time
    hnd = digiweb.Callback(pageinfo)
    time.sleep(60)
```

This example, like the one above will display request information, but it will make args be a dict object instead of a str during a POST request so it can be treated the same way as a GET request.

```
import digiweb
import cgi

template = """
<html><head><title>Request info</title></head>
<body>
%s request for path '%s'
```

```
<hr>
Headers: %s
<hr>
Args: %s
</body>
</html>
"""

def pageinfo(type, path, headers, args):
    tmp = {}
    if type=="POST":
      tmp = cgi.parse_qs(args)
      args=tmp
      for key in args:
        args[key] = args[key][0]
      return (digiweb.TextHtml, template % (type, path, headers, args))

if __name__ == "__main__":
    import time
    hnd = digiweb.Callback(pageinfo)
    time.sleep(60)
```

Availability

This feature is available as of firmware revision 2.8.0.

Python inside HTML

Allows you to be able to embed Python within HTML documents, similiar to mod_Python or PHP.

Summary

By taking advantage of both the Module: digiweb and a slightly modified version of PythonInsideHTML.zip from the BSD licensed project Karrigell Python web server to make it a stand alone library. It is possible to run embed Python within a HTML document that can be executed at run time.

Inside HTML

Syntax

By enclosing Python statements within a <% %> tag the Python interpreter will execute said statements. In the following example a "stored_time" variable will be created and will save the time on the local scope.

```
<% import time %>
<% stored_time = time.strftime("%d:%m:%y",time.localtime(time.time())) %>
```

If enclosed with <%= %> it will evaluate the statement and replace the tag with the result of the executed statement. In the following example the HTML created will contain the day:month:year from the devices internal clock.

```
<% import time %>
<%= time.strftime("%d:%m:%y",time.localtime(time.time())) %>
```

Indentation

Declared Indentation

A file is converted into Python code, which must be indented according to Python rules ; whereas in normal HTML indentation is used only for readability.

So beware if you mix Python and HTML :

```
1 <% for i in range(10): %>
2 <%= i %>*<%= i %> : <b> <%= i*i %> </b>
```

This will work because after a loop or a condition the following HTML is automatically indented by PIH.

To decrement indentation, use <% end %> :

```
1 <% for i in range(10): %>
2 <%= i %>*<%= i %> : <b> <%= i*i %> </b>
3 <% end %>
4 <h3>done</h3>
```

in this example, "done" will be written after the for loop is finished.

Another example for an if... else... :

```
1 <% if i: %>
2   output someting
3 <% end %>
4 <% else: %>
5   output someting else
6 <% end %>
7 <h3>done</h3>
```

(Don't forget the last <% end %> otherwise "done" would have the same indentation as line 5) But this :

```
1 <% for i in range(10):
2    data= '%s * %s' %(i,i) %>
3    <b> <%= i*i %> </b>
4 <h3>done</h3>
```

Won't work, because after the print statement on line 2 indentation goes back to 0 (it begins with plain HTML).

The <INDENT> Tag

If you have complex code where Python and HTML are mixed, embed it between the tags <indent> and </indent> :

```
1  <indent>
2  <% for i in range(10):
3    data= '%s * %s' %(i,i) %>
4    <b> <%= i*i %> </b>
5  </indent>
6  <h3>Table</h3>
7  <table>
8    <tr>
9      <td>A cell</td>
10   </tr>
11 </table>
```

<indent> means : from now on, and until the matching </indent> tag, use the indentation in PIH source and leave it as it is to produce Python code In the above example, indentation is used until line 5 and ignored afterwards If the <indent> tag itself is indented, the following code is indented relatively to it :

```
1   <table border=1>
2     <tr>
3       <th>Number</th>
4       <th>Square</th>
5     </tr>
6     <indent>
7     <% for i in range(10): %>
8       <tr>
9       <td><%= i %></td>
10      <td><%= i**2 %></td>
11      </tr>
12    </indent>
13  </table>
```

In line 7, <% is aligned on <indent> so the Python code will not be indented.

Ending Script

If you want to exit the script before the end of the document, raise a

```
SCRIPT_END exception
  raise SCRIPT_END,message
```

Writing to HTML output

If you want to write to the HTML output without using the evaluating tags you can write directly to the Python code output via

```
py_code.write( "<H1> Heading </H1>" )
```

Working example from the Python

For this example we will create a generic handler for a filetype for the web server by using the Module:digiweb.

```
            # test_pih.py
import sys,time,digiweb
sys.path.append("WEB/Python/PythonInsideHTML.zip")
from PythonInsideHTML import PIH

def http_handler(type, path, headers, args):
 exec PIH("WEB/Python%s"%(path)).PythonCode()
 return (digiweb.TextHtml,py_code.getvalue())


hnd = digiweb.Callback(http_handler)
while (True):
  time.sleep(1000)
```

Now you can just upload a file ending in a via the Python file management section of the webui, then just navigate to http:*//device_address/filename*.

For example uploading the following (template.pih) will demonstrate displaying information about the HTTP request passed to the http_handler. The Python code generated by this script is run on the same scope as the http_handler function so has access to the arguments (type, path, headers, args).

```
<html><head><title>Request info</title></head>
<body>
<%= type %> request for path '<%= path %>'
<hr>
Headers:
<table border=1 >
<% for h in headers: %>
      <tr>
      <td> <%= h %> </td> <td> <%= headers[h] %> </td>
      </tr>
<% end %>
</table>
<hr>
Args: <%= args %>
<hr>
<hr>
</body>
</html>
```

Navigating your browser to "http://*device_address/template.pih*" will give you a page displaying information about the HTTP request.

```
GET request for path '/template.pih'
________________________________________
Headers:
host   device_address
referer
agent  Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.19 (KHTML,
like Gecko) Chrome/1.0.154.48 Safari/525.19
________________________________________
Args: None
________________________________________
```

Source

by importing the following module you can use these features: Media:PythonInsideHTML.zip

Related

Module: digiweb

Python interactive web console

Introduction

This application demonstrates using the Module: digiweb to interact with the Python interpreter.

Screen shot

Requirements

Digi Gateway with Python enabled

Code organization

There are two parts of the code, the server side Python and the client side javascript. First, the Python side has three primary functions:

1. start_service() -- The function initializes data and registers the digiweb callback
2. web_callback() -- The function that handles HTTP requests
3. key_press() -- The function updates the state of a interactive session by one key

Secondly the javascript client-side application that runs on the web browser. This has a input/event handler and output area. For the purposes of simplicity we will use the Prototype cross-browser javascript library.

Code Listing

Python Server Side

Start Service

This function will initialize input/output streams and register the HTTP callback. For this we will wrap the stdout for stderr, adding color.

```
class ErrWrapper:
        """ io wrapper to add coloring to stderr messages """
        def write(self,s):
                # forward the message to stdout, wrapped in a div tag that can
                #   can be modified via a CSS style sheet
                sys.stdout.write("<div class=err>"+s+"</div>")
```

We will now create a StringIO object for stdout and initialize our input buffer as a empty string and redirect the IO. We will set stdin as a StringIO object so we can also write to this stream.

```
def start_service():
        """initializes and registers the Python interactive console"""
        global cli_input
        global cli_output
        # input buffer
        cli_input = ""
        # Use a StringIO object for the stdout
        cli_output = StringIO()
        # update system to use internal io objects
        sys.stdin = StringIO()
        sys.stdout = cli_output
        # use our ErrWrapper for stderr
        sys.stderr = ErrWrapper()
        # initialize prompt
        cli_output.write( prompt )
        # keep reference on global scope so it doesn't leave scope
        global hnd
        hnd = digiweb.Callback(web_callback)
```

Web Callback

This function is called by the digiweb module whenever there is a HTTP request that is unhanded by the internal web server. We will add a handler for a "/update_key" that will take in one javascript keycode which will be passed in the k parameter (via GET request).

```
def web_callback(type, path, headers, args):
        """callback for digiweb, serves pyh files and handles io to Python
interactive console"""
        if path.find("update_key")>=0:
                key_press(int(args["k"]))
                return (digiweb.TextHtml,cli_output.getvalue())
        return None
```

Key Press

This function is the heart of the application. It will take in a single key stroke and update the state machine and IO accordingly. Executing any completed statements. If a input does not compile we

check the reason, if its because of a unexpected EOF error we go into indentation mode and only execute the compiled code and leave indentation mode if enter is hit twice.

```
            # New line HTML sequence
NL = "<br\>"

# state table keeps track of the data for the state machine
state_table = {}
state_table["indented"] = False # are we expecting more input and should be
indented prompt?
state_table["last"]=" "         # the last key pressed

#Enter unicode value
ENTER_KEY = 13

def key_press( keycode ):
        """ handles a single character and updates input/output accordingly """

        global cli_input # buffer to store whats to be executed next
        global cli_output # output from past commands and echo of keystrokes
        global state_table # stores state information
        global NL # new line
        global ENTER_KEY

        if not keycode == ENTER_KEY: # if its not "enter"
                        sys.stdin.write( "%c"%keycode ) # append to stdin for
input
                        cli_input+="%c"%keycode
                        cli_output.write( "%c"%keycode )
                        state_table["last"] = keycode


        else: # enter pressed
                sys.stdin.write( "%c"%keycode )
                cli_input = cli_input+"\n"
                # if hit enter twice then pop out of indentation mode
                if state_table["last"]== ENTER_KEY and state_table["indented"]:
state_table["indented"]=False
                state_table["last"] = keycode
                try:
                        compiled= compile(cli_input, "webui-input", "single") #
check if the input works
                except:
                        ex = format_exc()
                        if ex.find("unexpected EOF")>=0 and not cli_input=="\n":
# expecting more input, go into indentation mode
                                cli_output.write(NL+ prompt_indented )
                                state_table["indented"] = True
                        else:
                                state_table["indented"] = False # formatting or
syntax error
                                sys.stderr.write( NL+format_exc())
                                cli_output.write( NL+ prompt )
                                cli_input = ""
                else:
                        if not state_table["indented"]: # dont execute if in
indentation mode until enter hit twice
                                cli_input = "" # remove old input from buffer
                                try:
```

```
                                        cli_output.write(NL)
                                        exec(compiled,globals(),globals()) # run
on global scope
                                except:
                                        sys.stderr.write( "EXEC ERROR:
Exception:" + str(sys.exc_info()[0]) + NL )
                                        sys.stderr.write( format_exc() + NL)
                                cli_output.write(NL+ prompt )
                        else:
                                cli_output.write( NL+prompt_indented )
```

Javascript Client

Input and Output

The input we will use a little trick, this is because in some browsers the onkeydown event can not be registered to anything but input fields. We will create a div for the output and add a onclick event listener that will give focus to a hidden text area to handle keyboard input.

```
        <pre id="output" class="terminal" onclick="$('real_input').focus()">
</pre>

        <div id="intput_wrapper">
                <textarea onkeypress="keypress_handler(event)" id="real_input"
style="border:0px;height:1px;width:1px;"/>
        </div>
```

We will then create the javascript handler for keyboard input.

```
function keypress_handler(e){
        var keynum = e.which || e.keyCode;
        new Ajax.Request('/update_key', {
                method: 'get',parameters: {k:keynum},                   // send
the unicode keycode in the k parameter
                onSuccess: function(transport) {
                        $('output').innerHTML=transport.responseText;     //
update output with response
                        $('output').scrollTop = $('output').scrollHeight; //
scroll to bottom
                }
        });
```

Source code

The complete source of this sample is included here, this includes a number of features including backspace/history support along with a HTML file that will copy the look and feel of the devices web page and add functionality to buffer input and clear the output : Pyic.zip take the two files within this zip file and uploading them using the Python file management in the webui. Run the pyic.py and point a browser to: http://device_ip/FS/WEB/Python/pyic.html.

RCI request

Remote Command Interface (RCI) is a method for remote clients to control, configure, and gather statistics from Digi Connect devices. RCI is a stateless, request/response protocol. RCI uses XML and HTTP to exchange data between clients and Digi devices.

RCI over HTTP

RCI requests are sent to the device using an URI of UE/rci. For example, if the Digi Device's IP address is 192.168.1.1, then RCI requests are sent to http://192.168.1.1/UE/rci.

RCI requests are sent as an HTTP POST with the XML request of the form specified in this document. Note, due to space limitations on the device, the largest request that can be processed is 32KB. If a request is larger than this, it must be split into multiple RCI requests. RCI replies from the device are not subject to this limit.

Security is handled in the usual HTTP mechanism. The username and password must be passed to the device in the header of each HTTP request.

RCI over serial

RCI requests can also be sent over the serial port. This is useful in scenarios where a master processor is connected to the Digi Device through a serial port. This allows the master processor to configure the Digi Device as part of its configuration process, so that a separate manual configuration step for the Digi Device is eliminated. You must enable 'RCI over Serial' in either the Web Interface or the Command Line Interface before the Digi Device will accept RCI requests and return replies. The RCI over Serial option is available only on the primary port. RCI over Serial uses the DSR (Data Set Ready) serial signal. Verify that the serial port is not configured for autoconnect, modem emulation, or any other application which is dependent on DSR state changes. Note: When the Digi Device sees its DSR raised, it will set the serial port settings to 9600 baud, 8 data bits, no parity, and 1 stop bit. When DSR is lowered, the Digi Device will restore the previous serial settings.

Configure using the Command Line Interface (CLI)

1. Access the CLI using telnet or rlogin and the module's IP address. Ex:

```
telnet 192.168.1.2 -or-
rlogin 192.168.1.2
```

2. At the command prompt type:

```
 #> set rciserial state=on
```

Configure using the web user interface

1. Access the web interface by entering the module's IP address in a browser's URL window.
2. Choose Serial Ports from the Configuration menu.
3. If the device has more than one port, select Port 1.
4. If a port profile has not been selected, select Custom and click Apply.
5. Select Advanced Serial Settings.
6. Select Enable RCI over Serial (DSR) and click Apply.

RCI request/reply

An RCI XML document is identified by the XML elements rci_request and rci_reply. An RCI request specifies the XML element “rci_request” optionally with a version number. The version should match the version of RCI the client expects. The current RCI version is 1.1. If a version is not specified, the RCI version of the device is used to form the reply. Not specifying a version can cause problems when communicating with devices at different RCI versions, if the client code is not written in a version independent way. Therefore, it is highly recommended to always supply the version of RCI in requests, unless the client code has been designed to be version independent. Example of a request element:

```
<rci_request version="1.1">
```

The device will respond to requests with the element "rci_reply" along with the version number as an attribute. Example reply:

```
<rci_reply version="1.1">
```

rci_reply errors

Errors that occur at the request level will result in an error element as a sub-element of the <rci_ reply>. Errors and warnings are explained below <rci_reply> errors: Error ID Description

1. Request not valid XML
2. Request not recognized
3. Unknown command

Command

The command section of the protocol indicates the action requested (or action performed in replies).

Commands are specified as sub-elements to <rci_request> and <rci_reply>.

This example requests all configuration settings:

```
    <rci_request version="1.1"> <!—Identifies the protocol and whether this is a
request or a response --
        <query_setting/> <!— request config of device -->
    </rci_request>
```

This example requests the configuration information for just boot settings and serial settings.

```
    <rci_request version="1.1">
        <query_setting>
            <boot/>
            <serial/>
        </query_setting>
   </rci_request>
```

Supported commands

COMMAND	REQUEST DESCRIPTION	RESPONSE DESCRIPTION
query_setting	Request for device settings. May contain setting group elements to subset query (only setting group subset supported. Subsetting below this level not supported).	Returns requested config settings. Requests specifying no settings groups (eg. <query_setting/>) return all settings.
set_setting	Set settings specified in setting element. Settings data required.	Empty setting groups in reply indicate success. Errors returned as specified below.
query_state	Request current device state such as statistics and status. Sub-element may be supplied to subset results.	Returns requested state. Requests specifying no groups (eg. <query_state/> return all state.
set_factory_default	Sets device settings to factory defaults. Same semantics as set_setting.	Same semantics as set_ setting.
reboot	Reboots device immediately.	None
do_command	see RCI do command	see RCI do command

Errors and warnings

Response documents may contain an element as a child of the command or data element that indicates the result of the request. More than one error or warnings may be present. Error and Warning elements:

error	An error occurred.	Attribute id: A numeric id specified by the parent element (the command or the data element). An error element id="0" is equivalent to no error.	Children Elements **name** — **desc** desc — Optional - Text description of the error. hint — Optional - Used to indicate to the client the source of the error. This will typically be set to the field name that the error.
warning	Command executed, but a warning was issued.		

Example:

```
<serial_setting>
   <error id="3">
       <hint>baud</hint>
       <desc>Value out of valid range.</desc>
   </error>
    </serial_setting>
```

Errors are required to have an id. <hint> and <desc> are optional and more than one are allowed.

Notes

RCI XML must be well-formed XML

The device parses incoming RCI requests in a sequential manner. Each XML element is parsed and acted upon as it arrives. This is not ideal behavior, but is necessary because of the inherent resource limitations of a device. Ideally, the entire XML request would be read into memory, validated, parsed and acted upon only after validation.

XML structure errors may be found after actions have been taken. For instance:

```
<rci_request version=”1.0”>
    <set_factory_default/>
</rci_requestBADENDTAG>
```

This request will result in an XML parse error, but since the parse error occurs after the set_factory_defaults, the device will be set to factory defaults. Therefore, it is highly recommended that RCI requests be validated with an XML parser before being sent to the device. Using any standard parsers, such as the XML parsing in the Java SDK, to form RCI requests accomplishes this.

XML structure characters must not be sent as character data

Care must be taken to avoid accidental badly formed XML in RCI requests because of including XML structure characters, such as "<", as user entered data. Any field that accepts character data must be checked to ensure that "<" and ">" are not present (fields such as the email body of an alarm are common places this can happen). It is recommended that all instances of "<" and ">" in character data be converted to "<" and ">", which is the standard XML representation(entities) of these characters.

To use RCI to Query DIA device/channel Information

Reading device/channel information by direct HTTP to a DIA device requires a different do_command set. See Simple RCI by HTTP for working code examples.

References

https://www.digi.com/search/results?q=900005www.digi.com/search/results?q=900005696 9

RCI do command

Python

You can add callbacks to unhandled do_commands target via the rci Python module.

File System

The file_system commands are accessed via the do_command of an rci request.

ls

attributes: dir

Reports all files in a given directory

dir: the path in which to list available files

Example:

```
<rci_request version="1.1">
  <do_command target="file_system">
    <ls dir="/WEB/Python"/>
  </do_command>
</rci_request>
```

returns

```
<rci_reply version="1.1">
  <do_command target="file_system">
    <ls dir="/WEB/Python">
      <file name="Python.zip" size="144321"/>
      <file name="digi_daq.zip" size="458980"/>
      <file name="digi_daq.yml" size="5270"/>
      <file name="digi_daq.py" size="6387"/>
      <file name="zigbee.py" size="1147"/>
    </ls>
  </do_command>
</rci_reply>
```

get_file

attributes: name

returns the base 64 encoded raw data from a file in the file system denoted by the name attribute

name: name of the file including path

Example:

```
<rci_request version="1.1">
    <do_command target="file_system">
        <get_file name="/WEB/Python/Python.zip"/>
    </do_command>
</rci_request>
```

returns

```
<rci_reply version="1.1">
    <do_command target="file_system">
      <get_file name="/WEB/Python/Python.zip">
         <data>UEs...KYmwaR</data>
      </get_file>
    </do_command>
</rci_reply>
```

put_file

attributes: name

uploads a base 64 encoded file onto the device

name: path and destination filename for the file

Example:

```
<put_file name="/WEB/destination.txt">
  <data>BASE64DATA</data>
</put_file>
```

rm

attributes: name

removes a given file

Example:

```
<rm name="/WEB/Python/somefile.py"/>
```

Zigbee

The zigbee rci command interacts with a xbee module.

ZigBee discover (<discover/>)

Optional attributes: start, size, clear

Returns back a list of discovered nodes, with the first indexed node being the node in the gateway.

- start: says the rci should return the nodes whose index is >= start. For some reason, if start > 1, the Gateway will return this list from cache, and not perform an actual discovery.
- size: Determines number of nodes to return
- option: If this is set to "clear", it forces a clearing of the device’s cache, and will always perform a discover to get fresh results

Example:

```
<do_command target="zigbee">
  <discover start="1" size="10" option="clear"/>
</do_command>
```

ZigBee query setting (<query_setting/>)

Optional attributes: addr

Returns back a detailed list of settings for a given radio

addr: 64 bit address of the node to retrieve settings for. If omitted, defaults to gateway node

Example:

```
<do_command target="zigbee">
  <query_setting addr="00:13:a2:00:40:34:0c:88!"/>
</do_command>
```

ZigBee query state (<query_state/>)

This is identical to query_setting, except it returns back different fields.

ZigBee set setting (<set_setting/>)

Optional attributes: addr

Basically the reverse of query_setting, so you can set settings for a particular node

addr: 64 bit address of node to set settings for. If omitted, defaults to gateway node

Example:

```
<do_command target="zigbee">
  <set_setting addr="00:13:a2:00:40:34:0c:88!">
      <radio>
          <field1>value1</field1>
          …
          <fieldn>valuen</fieldn>
      </radio>
  </set_setting>
</do_command>
```

ZigBee firmware update (<fw_update/>)

Required attribute: file

Updates the firmware of the radio in the gateway

file: Path to a firmware file which must already exist on the gateway

Example:

```
<do_command target="zigbee">
  <fw_update file="/WEB/firmware_file"/>
</do_command>
```

Simple RCI by HTTP

Reading DIA channels via web commands

Digi provides many detailed documents explaining web services and remote RCI calls, but most provide too much detail or partial examples. They assume you already know how to move the requests, so just want the core syntax.

Simple Python for use on a PC

Below is a simple script which allows using XML to query data in real-time from a gateway running DIA.

Sample YML

What this DIA system does isn't important. The RCI calls below will read or write the properties from the device named 'level', which are named **level.alert** and **level.config** respectively. The **RCIHandler** must be enabled, plus the setting **target_name** must match what you place into your RCI calls.

```
devices:
  - name: count
    driver: devices.template_device:TemplateDevice
    settings:
        update_rate: 10

  - name: level
    driver: devices.experimental.alert_output:AlertOutput
    settings:
        source: 'count.adder_total'
        rising: 12.0
        falling: 11.5

presentations:
  - name: rci_handler
    driver: presentations.rci.rci_handler:RCIHandler
    settings:
        target_name: idigi_dia
```

Writing the RCI request

Detailed information on the RCI commands accepted by the DIA RCI presentation can be found in the DIA user documentation.

Those RCI commands, such as <channel_dump />, <channel_set name="..." value="..." />, and <logger_set name="..." /> are wrapped in the following syntax. Notice that the string **target="idigi_dia"** matches the target name in the YML file.

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
      </do_command>
</rci_request>
```

In DIA versions 2.2.0.1 and below, the RCI handler can only accept a single tag in the <do_command> block. A workaround is to wrap multiple commands in an arbitrary wrapper tag like the following:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
`                <channel_get name="level.alert"/>
                <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

The only thing to remember with this is that the <blob> tag will wrap the response code in a similar fashion.

For DIA versions **newer than 2.2.0.1**, the RCI handler does not need a wrapper tag, so the below syntax would work:

```
<rci_request version="1.1">
       <do_command target="idigi_dia">
            <blob>
                 <channel_get name="level.alert"/>
                 <channel_get name="level.config"/>
            </blob>
      </do_command>
</rci_request>
```

However, this will generate an XML parse error with DIA versions 2.2.0.1 and below.

Source Code

Below is an actual script written and used under Python 2.4.3 on a Windows 7 PC.

```
            # Simple PC example to query the DIA device

import httplib, urllib

msg_dump = \
"""<rci_request version="1.1">
       <do_command target="idigi_dia">
            <channel_dump/>
       </do_command>
   </rci_request>"""

msg_get = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_get name="level.alert"/>
           </do_command>
           <do_command target="idigi_dia">
               <channel_get name="level.config"/>
           </do_command>
       </rci_request>"""

# notice that the VALUE this channel takes is complex - a list of 3 values.
# That is defined by the DRIVER involved.  Most channels will take a
# simple True/False, integer, floating point or string value.
# examine the channel_get response to discover what to return channel_set
msg_set = \
    """<rci_request version="1.1">
           <do_command target="idigi_dia">
               <channel_set name="level.config" value="[10,9,True]"/>
```

```
            </do_command>
        </rci_request>"""

if __name__ == '__main__':

    msg = msg_get

    conn = httplib.HTTPConnection("192.168.196.204:80")
    conn.request("POST", "/UE/rci", msg)

    response = conn.getresponse()
    print response.status, response.reason
    data = response.read()
    print data
    conn.close()
```

Sample Output

```
C:\py\dia\work>Python test_rci.py
200 OK
<rci_reply version="1.1"><do_command target="idigi_dia"><channel_get
name="level.alert"
value="False" units="alert" timestamp="Mon Jul 12 11:08:46 2010"></channel_
get></do_command>
<do_command target="idigi_dia"><channel_get name="level.config"
value="[12.000000,11.500000,False,'count.adder_total']"
units="" timestamp="Mon Jul 12 11:08:46 2010"></channel_get></do_command></rci_
reply>
```

See Also

RCI request` : This is geared more to using RCI to read/write values in the Digi hardware and not in DIA.

Look up the RCI handler module in the DIA HTML documentation, or see the doc strings directly in the DIA source file "src\presentations\rci\rci_handler.py". It supports the commands including, but not limited to:

- <channel_dump/>
- <channel_get name="..."/>
- <channel_refresh name="..."/>
- <channel_set name="..." value="..."/>
- <channel_info name="..."/>
- Plus there is a series of logger commands

Web Auto Refresh

How to make a web page auto-refresh

People often ask "How to enable active-content on a Digi Python-enabled gateway?", and the answer is with a huge investment and serious hard work. Many Digi customers accomplish this NOT with the gateway, but by hosting the web page on a PC which enables use of industry-standard server tools. The PC server fetches data from the gateway and pushes it up actively to the end user.

However, do you really need active-content? Are you really browsing dynamic multi-user databases which update in complex ways all of the time?

Keep It Simple

In reality, you probably just want the page to refresh itself from time to time. If so, you should first consider this good, old-fashioned HTML code:

```
<META HTTP-EQUIV="refresh" CONTENT="15">
```

Adding this one line to the page your gateway feeds back to the browser causes the browser to refresh (or reload) the page in 15 seconds. Your gateway code is still as sweetly simple as before, but the user display auto-updates.

Using JavaScript

This can also be accomplished using JavaScript. This can give an added benefit of only reloading a portion of the page that contains the data, reducing the bandwidth and load on the device. There are a number of ajax libraries that simplify this process.

- Prototype is a framework that aims to make web development easier. It provides a periodical updater that can update a portion of a page while increasing the time between refreshes if the data is not changing.
- JQuery does not directly support this function like prototype but there is currently a plugin here.

If a heavyweight framework is too much a repeated refresh can be accomplished with a combination of setTimeout and XMLHttpRequest. There is a setInterval function in javascript but it is recommended not to use it, as it can cause the same refresh occurring before the previous one has completed.

```
            /* function updates the "output" element with the result from the
"/update" url then calls itself again in 4 seconds
*/
function updater(){
  var request = createXMLHttpRequest();
  // add time to GET request to prevent caching issues
  request.open("GET", "/update?" + (new Date()).getTime(), true);
  request.onreadystatechange = function() {
    if (request.readyState == 4) {
        document.getElementById("output").innerText = request.responseText ;
        setTimeout("updater()",4000);
    }
```

```
  }
  request.send(null);
}

/* cross browser way of creating XMLHttpRequests */
function createXMLHttpRequest() {
    if (typeof XMLHttpRequest != "undefined") {
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined") {
        return new ActiveXObject("Microsoft.XMLHTTP");
    } else {
        throw new Error("XMLHttpRequest not supported");
    }
}
```

Cost as in Real Dollars!

Now before you run off and make every page auto-refresh every one (1) second, just b**e aware that someone might actually want to see your web pages over cellular or satellite** ... and leaving such an auto-refresh page open over a week-end could cost them thousands of $$$. Plus once a second is likely overkill because very few real world systems actually change every second. Therefore you'd be wiser to refresh at most every 15 second or even once a minute, then assume a user who cares to see faster refreshes for a short time during a test can manually refresh faster.

One solution is to have a FIRST copy of the page NOT auto-fresh, and then have TWO buttons on that page. One manually refreshes the page (just like the browser refresh button), while the other button is labeled "Start Auto-Refresh". This second button moves to a second page that looks the first, but includes the auto-fresh every 15 seconds. The second page has a single button to return to the manual-refresh page.

This doesn't fully solve the cost issue, but at least the user had to actively “select” to auto-refresh. Thus the boss cannot say *I didn't know it would cost so much to leave that page open* when the bill comes!

Working with 802.15.4

This category contains information about IEEE 802.15.4 Protocol and Python samples.

Channels, Zigbee

Channel detail

802.15.4 and Zigbee break the 2.4Ghz band into 16 channels as shown below.

Decimal	Hex	Frequency	SC mask	WiFi Conflict	Comments
11	0x0B	2.405GHz	0x0001	Overlaps Ch 1	Newer XBee only
12	0x0C	2.410GHz	0x0002	Overlaps Ch 1	
13	0x0D	2.415GHz	0x0004	Overlaps Ch 1	
14	0x0E	2.420GHz	0x0008	Overlaps Ch 1	
15	0x0F	2.425GHz	0x0010	Overlaps Ch 6	
16	0x10	2.430GHz	0x0020	Overlaps Ch 6	
17	0x11	2.435GHz	0x0040	Overlaps Ch 6	
18	0x12	2.440GHz	0x0080	Overlaps Ch 6	
19	0x13	2.445GHz	0x0100	Overlaps Ch 6	
20	0x14	2.450GHz	0x0200	Overlaps Ch 11	
21	0x15	2.455GHz	0x0400	Overlaps Ch 11	
22	0x16	2.460GHz	0x0800	Overlaps Ch 11	
23	0x17	2.465GHz	0x1000	Overlaps Ch 11	
24	0x18	2.470GHz	0x2000	Overlaps Ch 11	Newer XBee only
25	0x19	2.475GHz	0x4000	No Conflict	Newer XBee only
26	0x1A	2.480GHz	0x8000	No Conflict	Newer non-PRO XBee only

Things to note:

- WiFi bands vary by country and region. The three WiFi bands above are common worldwide, but for example, Japan allows enough extra channels to permit a fourth WiFi band starting at

channel 14. Therefore the "No Conflict" statements above would not be true in Japan.

- Sometimes Xbee documentation uses hex as "17" instead of "0x17" to mean a channel like 23. Be careful about the context!
- Digi Xbee starts on the first channel in the SC mask and stops on the first channel which grants it association. Depending on the age of the firmware on your Xbee ZB module it might start on channels 0x0B, 0x0C, or 0xD (11, 12, or 13)
- Once a coordinator selects a channel, it does not move regardless of error rate (i.e. not frequency agile), so ZigBee does not automatically hop to a clear channel - it does this to avoid confusing potentially sleeping devices. A user or external software must manually force the coordinator to move via setting changes, plus the routers will NOT follow unless various non-default settings have been enabled.
- More than one 802.15.4/Zigbee coordinator can settle on a single channel.
  - For example you could have 5 distinct WPAN, and all 5 Digi gateways decide they want to use the default channel 13/0x0D despite the fact that no WPAN uses any of the other 15 channels.
  - So in a crowded 802.15.4/Zigbee environment, it is helpful to change your SC to force your WPAN to less-used channels.
- Not all Zigbee radios support all 16. It depends on the age and power rating.
  - Older Digi XBee supported only 12 to 23, so default scan_channel/SC is 0x1FFE
  - Newer Digi XBee non-PRO (S2/S2B/S2C) support all channels, so scan_channel/SC can be 0xFFFF.
  - Newer Digi XBee PRO (S2) support channels 11 to 24, so scan_channel/SC can be 0x3FFF.
  - Newer Digi XBee PRO (S2B/S2C) support channels 11 to 25, so scan_channel/SC can be 0x7FFF.
- A new-hardware coordinator picking channels 11, 24, 25, or 26 won't be accessible by older hardware, so consider force the SC to 0x1FFE or narrower. You will have no clear 'feedback' that this is occurring

Channel Mask Examples

The following table shows four common channel masks, plus one single channel example, to demonstrate how binary bitmasks are linked to individual channels. If there is a 1 in the channel column, that channel is available for use, and will be scanned, when that mask is configured.

Remember - it is ONLY the mask in the coordinator which affects the selected channel. Assuming you used fixed PAN Id, then you have the most flexible design by putting the reduced/restricted mask in the coordinator alone, and leaving the other devices to check all channels.

The binary ones and zeros are then converted to hexadecimal to create the mask to configure. Using the last example, forcing channel 25: 0100,0000,0000,0000 in binary is 0x4000 in hexadecimal.

Channel/Mask	Notes	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11
0xFFFF	All channels, may get low-power channel 26	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x1FFE	Best for mixed old/new Xbee networks, very safe	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0
0x3FFF	Newer Digi XBee PRO (S2)	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x7FFF	Digi XBee PRO (S2B/S2C)	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x4000	Example of single channel (Channel 25 only)	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tips

Control SC within the coordinator

For maximum compatibility, it is advisable to restrict the number of channels the coordinator can use, and keep routers and end nodes as wide open as possible. The coordinator sets the channel the network will use, so it is best to make sure it will set a channel that all the other devices are configured to use. As some older chips are limited to which channels they can use it is important that the coordinator does not select a channel some chips cannot use.

- Coordinators should be set for 0x1ffe
- Routers and End nodes should be configured for as wide a range as possible. Newer low-power XBee should be set for 0xffff. All XBee should be AT LEAST 0x1ffe.

Helping your manufacturing / test people

If your own products contain Digi XBee, then careful channel control of local semi-permanent XBee networks will help a great deal. For example, you put sleeping XBee modules into a battery powered sensor and require manufacturing to test each unit as part of final quality-assurance. Assuming you also have a few test networks in engineering and tech support, your manufacturing people will find the final QA test frustrating, as the units-under-test may not join the manufacturing coordinator.

The simple solution is to leverage the knowledge that a default Digi XBee starts at the lowest enabled channel and works upwards looking for a mesh to join. This offers a handy solution:

- First, make sure all semi-permanent gateway\coordinators have SC set to one or a few of the higher channels. For example, if engineering and tech support use SC=0x1F00, then their mesh will never use the lower 8 ZigBee channels.
- Second, manufacturing should insure that their QA test gateway\coordinators are only in those lower 8 channels. This can be done by manually limiting SC to 0x00FE, or manually confirming that the functioning coordinators have selected a channel between 11 and 18 (hex 0x0B to 0x12).

If you do the above, then any new sensor powered for the final test will find the QA coordinators, not those used by engineering and tech support.

A more active way to help with ad-hoc testing is for engineering and tech support people (who understand the technology) to manually turn their joining on or off as required. For example, an engineer with 1 gateway and 3 XBee 485 adapters properly joined and running, no longer requires 'new joining'. So if they turn all joining of (set NJ=0 in the gateway AND in the 3 XBee 485 adapters) then systems set up by QA or company sales people preparing demos will never join or be affected by the semi-permanent engineering system. Should one of the XBee 485 loss association, or the engineer wish to join new adapters, they can temporarily enable joining.

Avoiding your Office/Home WiFi

In general, the small infrequent ZigBee packets are not affected by WiFi, and one could say that the destructive 'bullets' ZigBee punches in your WiFi bandwidth are treated as common back ground noise and worked around.

However, since you can control which WiFi channels your ZigBee may interfere with, and since you have over a dozen channels to select, it is smart to setup your ZigBee to *NOT* interfere with your home or office WiFi. You would enforce this avoidance by setting SC in your gateway\coordinators only.

Most consumer WiFi devices default to use 'channel 6' (so WiFi channels 6 to 10), therefore your home or small business WiFi is probably using channel 6. If your PC/notebook/tablet/smart-phone allows you to do a 'WiFi site survey', scanning all WiFi channels for raw traffic, then you can intelligently put your ZigBee either in the clearest bands, or those used by your neighbors!

Desired Outcome	SC to use
To Avoid WiFi Ch 1	0x7FF0
To Avoid WiFi Ch 6	0x7E0F
To Avoid WiFi Ch 11	0x41FF
To Avoid all WiFi	0x4000
To Force Use of WiFi Ch 1	0x000F
To Force Use of WiFi Ch 6	0x01F0
To Force Use of WiFi Ch 11	0x3E00
To Force Use of non-WiFi	0x4000

Be Aware of Wireless Audio or Phones

Successful use of ZigBee may require careful avoidance of competing NON-WIFI devices including:

- Wireless mic/headsets in a restaurant drive-through
- Wireless phone systems
- Wireless audio (music) headphones
- Wireless phone head sets.

The serious risk with these competing 2.4GHz devices is that many use as much raw bandwidth as possible to maximize audio quality. So for example, a ZigBee mesh in a fast food restaurant might work great when installed at 6AM, but start to see a 60-70% error rate when the drive-through audio system is operational and busy.

Ideally, review the technical documentation for the competing technology. Although it is NOT WiFi, it likely defines its frequency usage in WiFi terms, and it likely includes a way to assign it Wifi ranges 1, 6, or 11. Since most consumer grade WiFi devices default to use channel 6, it is safe to assume these non-WiFi devices likely default to either WiFi channel 1 or 11 - meaning exactly where you may wish to place your ZigBee.

If you do not have access to the product documentation, review the 'XBee Range Test' procedures on Digi's tech support website, then run a few range tests when the competing 2.4Ghz technology is being heavily used. If you do not detect any significant error rate, then either your selected XBee frequency is unaffected by the competing technology, or the competing technology is gracefully sharing the radio waves. The need to possibly change your XBee channel should be very obvious after very little testing.

Xbee Command to Device Cloud Cross Reference

Xbee experts and documentation generally use AT parameters to explain configuration parameters. Device Cloud uses verbose descriptions. The list below provides a cross-reference.

UPDATE: *Device Cloud now shows both the verbose command and the AT command in the Xbee properties screen, so this cross-refence is no longer necessary. Do still be aware that Device Cloud operates in decimal, rather than hex, in most, but not all, fields.*

Note Most of the parameters on Device Cloud use decimal, while most direct Xbee operations are in hex. Keep this in mind.

A1 - End device association

A2 - Coordinator association

AR - Aggregation route notification

BD - Serial interface data rate

BH - Broadcast radius

CA - CCA threshold

CC - Command sequence character

CE - Coordinator enable

CH - Operating channel

CI - Cluster identifier

CT - Command mode timeout

D0 - AD0/DIO0 configuration

D1 - AD1/DIO1 configuration

D2 - AD2/DIO2 configuration

D3 - AD3/DIO3 configuration

D4 - AD4/DIO4 configuration

D5 - DIO5/Assoc configuration

D6 - DIO6 configuration

D7 - DIO7 configuration

D8 - DIO8/SleepRQ configuration

D9 - DIO9/ON_SLEEP configuration

DE - Destination endpoint

DP - Disassociated cyclic sleep period

EE - Encryption enable

EO - Encryption options

FT - Flow control threshold

GT - Guard times

HP - Hopping sequence

IA - I/O input address

IC - DIO change detect

ID - Extended PAN identifier

ID - PAN identifier

IF - I/O sample from sleep rate

II - Initial PAN identifier

IR - I/O sample rate

IT - I/O samples before transmit

JN - Join notification

JV - Join verification

KY - Link encryption key

LT - Associate LED blink time

M0 - PWM0 output level

M1 - PWM1 output level

MM - MAC mode

MR - Mesh network retries

MT - Broadcast retries

MY - Network address

NB - Serial interface parity

NK - network_key

NH - Maximum hops

NJ - Node join time

NN - Network delay slots

NI - node_id

NT - Node discovery timeout

NW - Network watchdog timeout

P0 - DIO10/PWM0 configuration

P0 - PWM0 configuration

P1 - DIO11/PWM1 configuration

P1 - PWM1 configuration
P2 - DIO12/CD configuration
P3 - DIO13/DOUT configuration
PL - Transmit power level
PM - Power mode
PO - Polling rate
PR - Pull-up resistor enable
PT - PWM output timeout
RN - Random delay slots
RO - Packetization timeout
RP - RSSI PWM timer
RR - MAC retries
RR - XBee retries
SB - Stop bits
SC - Scan channels
SD - Scan duration
SE - Source endpoint
SM - Sleep mode
SN - Peripheral sleep count
SO - Sleep options
SP - Cyclic sleep period
ST - Time before sleep
SW - Sleep early wakeup
T0 - D0 output timeout
T1 - D1 output timeout
T2 - D2 output timeout
T3 - D3 output timeout
T4 - D4 output timeout
T5 - D5 output timeout
T6 - D6 output timeout
T7 - D7 output timeout
V+ - Supply voltage high threshold
WH - Wake host delay
ZA - ZigBee addressing enable
ZS - ZigBee stack profile

Xig

The XBee Internet Gateway ("XIG") is an application written for Digi's ConnectPort series of XBee-to-IP gateways. The XBee Internet Gateway gives any device the ability to connect seamlessly to the Internet by mirroring the interactions humans have with web browsers. Any device with an XBee radio can send a web URL to the XIG and receive back the contents of that web page. All the tricky technical aspects of web connections are all handled for you behind the scenes.

This simple service gives your prototype or device a simple yet completely flexible pathway to any web service that you can imagine, including posting sensor values, scraping Facebook or commanding your robotic kitten army.

XIG offeres a myriad of other interesting and useful communications services to your XBee network. For complete documentation and setup instructions please visit:

http://code.google.com/p/xig/

See also:

- http://code.google.com/p/xig/wiki/UserDocumentation
- http://www.faludi.com/projects/xbee-internet-gateway/
- http://www.digi.com/products/wireless-routers-gateways/gateways/

XIG is brought to you by an open-source team by makers Robert Faludi, Jordan Husney and Ted Hayes with valuable support from a community of commercial and educational users.

Working with DigiMesh

These pages covers DigiMesh specific details.

DIA difference between ZigBee and DigiMesh

Mesh setup

Although not related to DIA, people familiar with ZigBee setup will find DigiMesh a new experience.

Unlike ZigBee where new nodes can sometimes merely be powered up and joined to the correct network, DigiMesh works more like the older ZNet design where you will require either:

- Use XCTU and an XBIB board to manually configure the required Network setting
- Use a 'commissioning node' in an XBIB board with default settings to locate the new factory-fresh DigiMesh device, then using the 'Remote Configuration' setting, push the required Network settings into the new device.

See this Wiki page for a summary of the required parameters: Quick guide to DigiMesh setup.

Understanding sleeping

There is a basic difference between ZigBee and DigiMesh.

- ZigBee has powered, non-sleeping devices called routers (or parents). The coordinator is just a router with an extra role.
  - Routers 'mesh', forming the resilient mesh-topology hyped by ZigBee advocates.
  - Routers discover routes (paths) and adjust routes as old peers go offline and new peers come online.
  - Routers repeat broadcasts
  - In general routers don't say much - unless asked to move data.
- ZigBee also has sleeping end-devices (or children).
  - End-Devices do not mesh. They locate ONE specific router/parent to belong to.
  - When End-Devices wake up, they start polling their parent 10-times per second, sending data or looking for data the parent has buffered.
  - End-Devices do NOT repeat broadcasts
  - However, end-devices are rather chatty as long as they are awake!

DigiMesh

- All devices in DigiMesh are routers, forming the mesh
- DigiMesh devices can sleep, or be fully awake.
- All nodes need to understand the wake/sleep cycle, and only talk during wake cycles.

DigiMesh in DIA

What this means to DIA (and YML files) is that any ZigBee node can have any desired data sample rate. You can have 20 temperature sensors, all with random sample rates between 1 second and 1 day. The fully awake routers can send data at any time, and the sleeping end-devices can wake anytime, trusting that their parent is awake, ready and waiting to handle their data.

In contrast, under DigiMesh all data samples need to be an limited integral multiple of the entire mesh's sleep/wake cycle.

DIA/DigiMesh sleep coordinator

Within DIA, the Digi gateway acts as the designated sleep coordinator. This means DIA sets the sleep/wake times within the XBee within the gateway, and the gateway imposes the sleep/wake cycle over the entire mesh.

This is a DIA limitation, not a DigiMesh one. As of this release, DIA does not support nominated sleep coordinators, which is more appropriate to a peer-to-peer system. Most DIA systems are gateway-centric, assuming the gateway is critical and the devices have little or no function without the gateway, and therefore their sleep-coordinator.

DIA/DigiMesh sleep mode

To disable all sleeping, set the DigiMeshDeviceManager's **sleep_time** setting to zero. The SM value in all nodes is forced to 0x00.

To enable sleeping, set the DigiMeshDeviceManager's **sleep_time** and **wake_time** settings to a value higher than 10msec - for example 20000 and 10000 respectively, which means the mesh sleeps for 20 seconds, then is awake for 10 seconds. The SM value in all nodes is forced to either 0x07 or 0x08, which means 'always awake but sleep aware' and 'sleep'.

This 20/10 design means for 20 seconds none of the nodes transmit data - even the fully awake nodes (SM=0x07). Every device waits until the mesh wakes, then they all try to send messages as required, including forwarding (and 'meshing') as required. Of course the wake_time needs to be long enough to allow all the required messages to move, including passing across multiple hops.

This is a DIA limitation, not a DigiMesh one. DigiMesh includes some richer asynchronous sleep-modes, which allow nodes to wake up during the sleep-period and talk, however this implies the sleeping node is within 'ear shot' of a fully awake node since no other sleeping node will be awake to help route the message.

DIA/DigiMesh data sample rates

At this point, all nodes can use external logic to decide when (during any particular wake-time) that they wish to send data. If the node has its own processor, it can format a message and send it during the beginning portion of the wake cycle, hopefully having time to receive any responses. The DIA framework on the gateway can also send out polls. The DigiMeshDeviceManager will queue up any pending outgoing messages from DIA drivers, then sending them out when the mesh wakes.

Support for the Digi XBee's IC/IR commands is also supported. By default, a node will send data each time it awakes. This requires IR=0xFFFF, IF=0x01 and at least 1 of the XBee IO pins be configured as an input.

The DigiMeshDeviceManager's **set_if** setting can be set to True or False. If set_if=False, then the DIA does NOT interfere with the IC/IR/IF settings within the remote nodes. DIA assumes that the user has configured the remote devices as required.

If set_if=True, then the DIA uses the various sample_rate setting of drivers to calculate the correct IR/IF values. For example, if the device wishes a once per minute data sample and the mesh is waking every 30 seconds (for example, our 20/10 sleep_time/wake_time settings), the setting IR=0xFFFF and IF=0x02 means the XBee will send a data sample every second wake cycle. The DIA will round down, so if the driver sample_rate is once per 75 seconds with a wake-cycle of 30 seconds (20/10 sleep/wake), then DIA is forced to select IF=0x02, which will give you data every 60 seconds.

DIA Releases

DIA release 2.0.x (May 2012)

Note This version is not released yet.

DigiMesh support

One of the largest changes is the introduction of two different XBee manager devices, so your YML file should now use either of these:

```
    driver: devices.xbee.xbee_device_manager.zigbee_device_
manager:ZigBeeDeviceManager
```

Or

```
    driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
```

Although ZigBee users should migrate to call the ZigBeeDeviceManager explicitly, YML files including the old XBeeDeviceManager will run the ZigBeeDeviceManager instead.

DigiMesh sleep design

The DIA DigiMeshDeviceManager assumes 1 of 2 designs:

- Non-sleeping, so all nodes run as SM=0
- Sleeping, with the gateway acting as the preferred Sleep Coordinator. Nodes will be set to:
  - Gateway/Sleep-Coordinator: SM=7, SO=0x05
  - Non-Sleeping Node: SM=7, SO=0x02
  - Sleeping Node: SM=8, SO=0x02

Support for sleep-coordinator 'Nomination' and the asynchronous/pin-sleep modes of DigiMesh are not yet supported.

DigiMesh YML changes

Here is an YML usage for the DigiMeshDeviceManager:

```
  - name: xbman
    driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
    settings:
        dh_dl_force: True
        dh_dl_refresh_min: 300
        sleep_time: 9000
        wake_time: 1000
        set_if: True
```

The **dh_dl_force and dh_dl_refresh_min** settings apply to both ZigBee and DigiMesh systems and will be explained in another section of this document.

The three DigiMesh specific settings are:

- **sleep_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes wake up every 2 seconds. Set this to 0 to disable sleeping.
- **wake_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes to stay awake for 2 seconds. If sleep_time = 0, this setting is ignored.
- **set_if** is a boolean which defines if DIA will adjust the IR/IF settings in remote nodes. It defaults to False, which assumes the user manually handles data messages. If set to True, then DIA makes a best-effort to map individual driver 'sample_rate' settings to XBee IF intervals.

For example, if a XBee DM AIO adapter has a sample_rate_ms setting of 60000 (once a minute). The default DigiMeshDeviceManager settings of sleep_time = 2 seconds and wake_time = 2 seconds means a sleep-cycle of 4 seconds total. So DIA will set the XBee DM AIO adapter with IR=0xFFFF and IF=0x0F, which means the AIO adapter will send in an IO data sample when it wakes within the 15th sleep-cycle (ie: 60-sec / 4-sec). DIA rounds down if the rates do not factor cleanly.

Internal driver changes to support DigiMesh

Users wishing their custom driver code to run under DigiMesh will need to make some changes to their drivers.

First, drivers must not directly set SM, SO, or IR - they must submit the required settings using the XBee Manager. For example:

```
        xbee_sleep_cfg = self._xbee_manager.get_sleep_block(
                                    self._extended_address,
                                    sleep=False,
                                    sleep_rate_ms=sample_rate,
                                    awake_time_ms=0)
```

The XBee Manager (ZigBee or DigiMesh) uses the 'sleep' parameter to select the correct SM/SO settings. The 'sleep_rate_ms' and 'awake_time_ms' parameters are used to set the setting SN/SP/ST for ZigBee and IR/IF for DigiMesh.

For clearer examples, look at the DIA drivers for the Digi Adapters.

Second, your driver should no longer set the DH or DL settings directly. The support for these have been moved to the XBee Manager since the values required may be different for different XBee technology.

Destination address (DH/DL) support

In this release of DIA, drivers should no longer directly set the DH/DL of XBee nodes. This function has moved to the XBeeDeviceManager classes to support different behavior required by different Xbee technologies.

Two new settings have been added which can be used with both ZigBee and DigiMesh systems:

- dh_dl_force is a string and can be:
  - None or False, which means do NOT change DH/DL in any node
  - True, which means do what is most appropriate for the XBee technology. For ZigBee, this forces the default aggregator address of '00:00:00:00:00:00:00:00!' into all DH/DL. For DigiMesh, this forces the gateway's SH/SL address into all DH/DL.

- Coordinator, which means use the gateway or coordinator SH/SL address for all nodes DH/DL
- The string of an exact address such as '00:13:a2:00:40:32:d9:51!' which is to be used.

■ dh_dl_refresh_min is a string which defines when the DH/DL setting is affected.
- None, which means never change DH/DL at all
- Once, which means DH/DL will be set when nodes are configured, and also broadcast once.
- Config, which means DH/DL will only be set when nodes are configured, and not broadcast.
- A time with tag such as '5 min' or '1 day' which defines a repeated broadcast interval. if no tag is applied, then a number is assumed to be in minutes.

Rapid Reboot Detection

Many Digi gateways include an option to reboot if an auto-started Python script exits. This is a wise precaution against an expected error occurring months after reboot, but can make the unit unreachable by Device Cloud if the error occurs repeatedly at the start of the script.

For example, if the main Python script has a simple typo or a ZIP file was accidentally truncated during download, then the auto-started Python script may exit instantly, forcing a reboot within seconds of the last reboot and startup. A gateway locked in such a reboot-cycle will never be connected to Device Cloud long enough to allow fixing the problem.

This release of DIA addresses this within the main dia.py file. By default this feature is disabled, so users wishing this protection must manually create (or seed) a text file named nospin.txt in the Python area of the gateway.

The file name is case-sensitive, so prevent allowing Windows to rename your file Nospin.txt!

The file can be empty or contain a line of text. Once active, the file will be rewritten by dia.py upon every reboot with a timestamp, and if dia.py detects that the last 10 reboots have occurred in less than 20 minutes, then dia.py will sleep for 10 minutes before trying to start.

If you enable this feature on a gateway without time service (so it always boots as 1-Jan-1970), then after 10 reboots the gateway will always delay starting for 10 minutes. This may be undesirable, but not as undesirable as becoming unreachable from Device Cloud.

Users writing their own auto-start code should consider copying this same function to their own applications.

Other Internal Driver Changes

Older DIA drivers tended to make very poor use of Python or object oriented paradigms. For example, much of the code within most DIA Xbee drivers is duplicated in all of the peers - literally cut-and-pasted. This is a clear violation of Object Programming concepts which assume that if all derived classes will need the same code fragment, then it should be handled by the base class, not repeated within all derived classes.

Tracer Module

Besides the previous tracer-levels of ('debug', 'info', 'warning', 'error', 'critical'), two new lower levels are defined BELOW debug.

tracer.calls

These should be used for simple debug statements showing a routine was called. This was added because some programmers pepper too many 'debug' statements such as:

```
def calculate_average(self, a, b):
    # this should now be self._tracer.calls() instead
    self._tracer.debug("calculate_average")
```

These tend to pollute the debug trace with considerable low-grade information. Defining a new sub-debug level named 'calls' allows the user to enable/disable these simple, low-grade debug lines upon demand.

tracer.xbee

For example, these are used by the DigiMesh manager to announce all of the mesh-wake and mesh-sleep messages received from the gateway XBee. This information is generally used only when debugging the xbee manager, or sleep/performance problems with drivers.

tracer lines now return a boolean value, True if active

This should be used in situations where the tracer parameters are costly to format. In the example below, the parameter is a time-expensive operation - a STRING conversion of a list of hundreds of integers. Python must evaluate all parameters whether the tracer level is true or not, which can cost a huge time penalty even when tracing is not enabled. Using an if-then statement greatly reduces the performance hit.

```
    if self._tracer.debug():
        # my_list is converted to a string ONLY when debug is True
        self._tracer.debug("list data:%s", str(my_list))
```

class DeviceBase

The DeviceBase now creates three variables which can be used by derived classes:

- **self._name** = name as passed in by def __init__(self, name, core_services, settings, properties)
- **self._core** = core_services as passed in by def __init__(self, name, core_services, settings, properties)
- **self._tracer** = get_tracer(name)

In past DIA versions, most derived class duplicated the effort to create and manage these as self.__name, self.__core, and self.__tracer. These cause no harm, but waste resources and prevent derived classes from being created from derived classes. When porting drivers to the new DIA, you are encouraged to use DeviceBase's copies of these variables.

The DeviceBase also now has a setting named 'trace', which defaults to , *which means use global trace level in self._tracer. It can hold any valid tracer-level for the Tracer module, such as 'debug', 'info', and so on. This is used when a user has for example 20 devices, but wants only 1 of them to output tracer.debug lines, and the other 19 to output tracer.info lines.*

class XBeeBase

The XBeeBase now creates and manages two variables which can be used by derived classes:

- **self._xbee_manager**, which is set by the code:

  ```
      # Fetch the XBee Manager name from the Settings Manager:
      dm = self._core.get_service("device_driver_manager")
      self._xbee_manager = dm.instance_get(
              SettingsBase.get_setting(self, "xbee_device_manager"))
  ```

- **self._extended_address**, which is set by the code:

```
                    # Get the extended address of the device:
self._extended_address = SettingsBase.get_setting(self, "extended_
address")
```

This also means the XBeeBase class manages the settings named "xbee_device_manager" and "extended_address". Derived classes should attempt to create or manage these settings. When porting drivers to the new DIA, you are encouraged to use XBeeBase's copies of these variables and settings.

The XBeeBase class registers a callback named self.running_indication() which derived classes can overload as desired. The default base class routine looks like this:

```
    def running_indication(self):
        """
        Indicate that we have completed config and are running
        """
        self._tracer.info("Configuration is Complete. Running indication.")
        return
```

DigiMesh products

DigiMesh XBee technology

Digi produces several families of DigiMesh XBee module. The most important feature in determining support within XBee adapters and gateways is the foot-print of the Xbee.

- Most existing Digi Adapters and Gateways support only the 20-pin through-hole footprint, for which only DigiMesh at 2.4 Ghz and 900 Mhz is available.
- Newer XBee families have moved to a SMT format and include 865 Mhz, 868 Mhz and 900 Mhz.

Existing DigiMesh gateways

- Digi ConnectPort X2
  - ConnectPort X2 - Industrial with DM 2.4 GHz to Ethernet (metal case, extra memory)
  - ConnectPort X2 - Industrial with DM 900 MHz to Ethernet (metal case, extra memory)
  - Note: as of March 2012 there are no X2 Commercial models with DigiMesh (lower cost, plastic case, less memory)
  - Note: as of March 2012 there are no X2 models with Wifi and DigiMesh

- Digi ConnectPort X4
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & cellular (GSM/Edge)
  - ConnectPort X4 - DigiMesh 2.4 GHz to Ethernet & Wi-Fi
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet
  - ConnectPort X4 - DigiMesh 900 MHz to Ethernet & cellular (GSM/Edge)
  - Note: as of March 2012 there are no X4H or X4 IA models with DigiMesh
  - Note: as of March 2012 there are no X4 CDMA cellular models with DigiMesh
  - Note: as of March 2012 there is no SMT support within the X4, so no 865 Mhz or 868 Mhz DigiMesh support
  - Note: as of March 2012 there are no X5 vehicle models with DigiMesh

Existing DigiMesh adapters

- XBee-PRO DigiMesh 2.4 Range Extender
- Note: as of March 2012 there is no DigiMesh 900 Mhz Range Extender
- XBee-PRO DigiMesh Adapters
  - XBee-PRO DigiMesh 2.4 GHz, RS-232 adapter
  - XBee-PRO DigiMesh 2.4 GHz, RS-485 adapter
  - XBee-PRO DigiMesh 2.4 GHz, Digital IO adapter
  - XBee-PRO DigiMesh 2.4 GHz, Analog Input adapter
  - XBee-PRO DigiMesh 2.4 GHz, USB adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-232 adapter
  - XBee-PRO DigiMesh 900 Mhz, RS-485 adapter
  - XBee-PRO DigiMesh 900 Mhz, Digital IO adapter
  - XBee-PRO DigiMesh 900 Mhz, Analog Input adapter
  - XBee-PRO DigiMesh 900 Mhz, USB adapter
  - Note: as of March 2012 there is no DigiMesh Smart Plug
  - Note: as of March 2012 there is no DigiMesh Light/Temperature (LT/LTH) Sensor
  - Note: as of March 2012 there is no DigiMesh XStick
  - Note: as of March 2012 there are no DigiMesh WatchPort Adapters

Can Digi ZigBee/XBee products be converted to DigiMesh?

Officially, No.

However, one can remove the 20-pin ZigBee XBee and replace it with a 20-pin DigiMesh XBee. This works fine with the RS-232/485/USB adapters, but may not work with adapters expecting analog signals because the ZigBee Xbee and DigiMesh XBee use different reference voltages. Your software will need to compensate for this, as well as the fact that there are some minor AT/DDO settings differences between Zigbee-XBee and DigiMesh-Xbee.

DigiMesh support in DIA

DigiMesh support added to DIA 2.0 (May 2012)

The next release of DIA will add expand XBee support to include DigiMesh for firmware 8062 or higher. Support is limited to DigiMesh XBee technologies which are supported by Digi gateways (see Existing DigiMesh gateways).

Supported DIA drivers

Support is limited to adapters released by Digi. see Existing DigiMesh gateways.

Instead of the traditional XBeeDeviceManager, you should add a DigiMeshDeviceManager:

```
driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
```

Supported DIA drivers

Support is limited to adapters released by Digi. see Existing DigiMesh adapters

- Both DigiMesh 900 Mhz and 2.4 Ghz can be seated in the XBIB adapter, which is supported by the xbee_xbib driver, such as:

  ```
  driver: devices.xbee.xbee_devices.xbee_xbib:XBeeXBIB
  ```

- Both DigiMesh 900 Mhz and 2.4 Ghz have RS-232 and RS-485 adapters, which are supported by the xbee_serial driver and derivatives, such as:

  ```
  driver: devices.xbee.xbee_devices.xbee_serial_
  terminal:XBeeSerialTerminal
  ```

- Both DigiMesh 900 Mhz and 2.4 Ghz have digital adapters, which are supported by the xbee_dio driver, such as:

  ```
  driver: devices.xbee.xbee_devices.xbee_dio:XBeeDIO
  ```

- Both DigiMesh 900 Mhz and 2.4 Ghz have analog adapters, which are supported by the xbee_aio driver, such as:

driver: devices.xbee.xbee_devices.xbee_aio:XBeeAIO

- Only DigiMesh 2.4 Ghz has a wall router (called a 'Range Extender'), which is supported by the xbee_xbr driver, such as:

  ```
  driver: devices.xbee.xbee_devices.xbee_xbr:XBeeXBR
  ```

Unsupported DIA drivers

The following products are not supported by Digi.

- Smart Plug
- Light/Temperature (LT/LTH) Sensor
- WatchPort Sensor Adapters

Quick guide to DigiMesh setup

For those familiar with ZigBee

The method to setup a DigiMesh network is quite different from setting up ZigBee. In ZigBee, you can leave the PAN ID zero/0 (meaning wild-card, match any), plus ZigBee scans all supported Channels, Zigbee. If you only have 1 local ZigBee coordinator, ZigBee's automated search-and-find method is great; if you have dozens (or even hundreds) of local ZigBee coordinators, then it can be nearly impossible to accomplish.

In contrast, DigiMesh requires a new node to have the exact correct setting to join. This either means you manually configure the XBee in an XBIB board from a PC, or make use of a special 'commissioning node' on the PC with default settings.

Minimum Digi mesh configuration

Network Id (setting: ID)

There is no wild-card/match-any Network ID value within DigiMesh - you must manually assign the designed value. DigiMesh XBee come with the default of 0x7FFF. You can use ID=0x7FFF, but will have problems if a neighbor also installs a DigiMesh system with ID=0x7FFF. Multiple DigiMesh systems with the same ID can be on the same channel/frequency, but then you cannot predict which 'mesh' a new node will join. Once joined, the nodes will tend to remain with the same group, but various power-outage scenarios could cause nodes to seek a new system.

The recommended DigiMesh design: select your own ID, which is not 0x7FFF

Channel/Hopping Sequence (setting: CH or HP or CM)

DigiMesh channel behavior depends on the technology. Regardless, you must insure all of your XBee nodes have compatible settings.

DigiMesh 2.4 GHz - setting is CH

DM24 uses 802.15.4 as a base, so does NOT hop channels nor does it include 'frequency agility', which means it does NOT scan multiple frequencies. You select the channel/frequency to use (the same Channels, Zigbee, and your DigiMesh system uses that channel/frequency forever.

Changing channel/frequencies will be a manual process - perhaps your gateway can one by one manually change the CH setting in each node, but after setting the node vanishes and the gateway has no ability to 'see' that the change worked until all nodes have been forced off the old channel and the gateway also moves. Using an XBIB on a PC/Notebook might also be helpful - allow the gateway to change the channel/frequency first, then use XCTU from the PC to one-by-one push the nodes to the new channel/frequency, where they should show up on the gateway.

DigiMesh 2.4Ghz does not have the HP or CM settings.

DigiMesh 900 Mhz - setting is HP

DM09 supports hopping in several patterns (see the DM09 user guide). The HP setting defines the pattern, so all of your XBee devices need the same HP setting. Using 'frequency agility' or 'changing patterns' is not directly relevant here since the device are always hopping between frequencies, a few bad/noisy channels won't break the system.

DigiMesh 900 Mhz does not have the CH or CM settings.

DigiMesh 868/865 Mhz - setting is HP and CM

DigiMesh 868/865 Mhz do not support frequency hopping, but they do scan multiple channels and support 'frequency agility', which allows the system to gracefully move to a new channel/frequency. The CM setting holds a bit-mask of channels - much like the 'SC' setting in ZigBee. CM defines which channels/frequencies the nodes will examine for a network with the correct ID and HP value.

The HP value (unlike in DM09) is a preamble value, which can be thought of as an extended Network Id (setting ID). It allows more than the 15-bit limit of the normal DigiMesh Network ID.

DigiMesh 868/865 Mhz does not have the CH setting.

Sleep Mode (setting: SM)

DigiMesh devices need compatible sleep modes - for example, you should not mix devices with no sleep behavior (SM=0) with nodes supporting sleep (SM=7 or SM=8). Since the DigiMesh XBee default to SM=0, you should force in the correct new mode at the same time your set ID and the channel settings.

Set the settings in one-shot

If you use a PC with an XBIB and DM node in default to commission your new device, then you would want to do the following:

- Set the correct ID (but do not apply yet)
- Set the correct CH or HP or CM/HP (but do not apply yet)
- Set the correct SM (but do not apply yet)
- WR (write) these, applying at the same time.

With XCTU, this means you need to set all the values once before hitting the <WRITE> button.

With control from your own software, you'd issue the four packets assuming the first 3 are not applied directly. The fourth WR command both saves the settings and causes them to be applied, which causes the DM node to disconnect from your view. Accidentally setting and applying only 1 or 2 of the 3 will cause the DM node to vanish from your current DM network, but not move to the desired one - you will need to restore that node to factory defaults to recover.

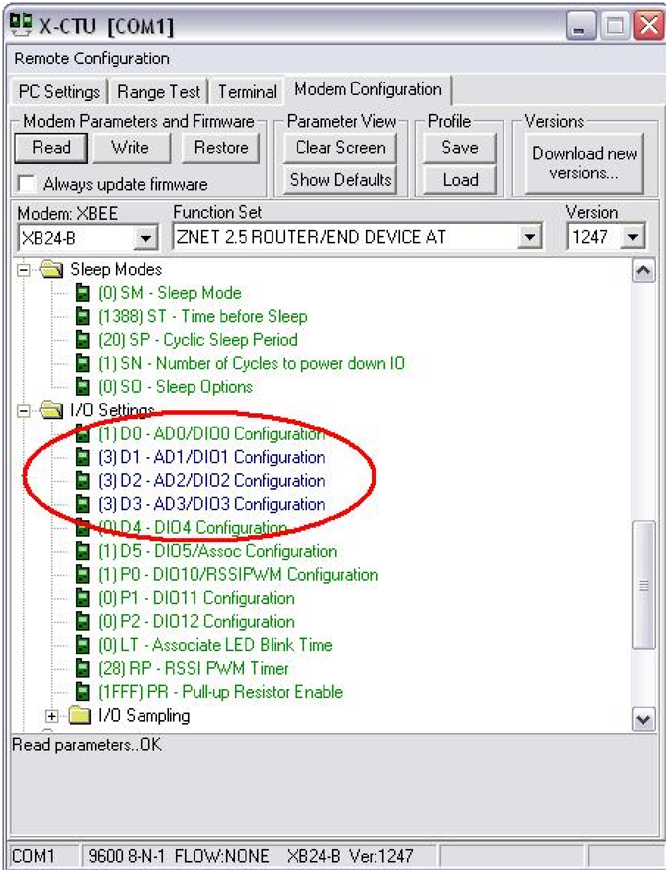
Remote XBee management with XCTU

XCTU allows you to directly edit the setting within any XBee in a mesh/network.

What you need

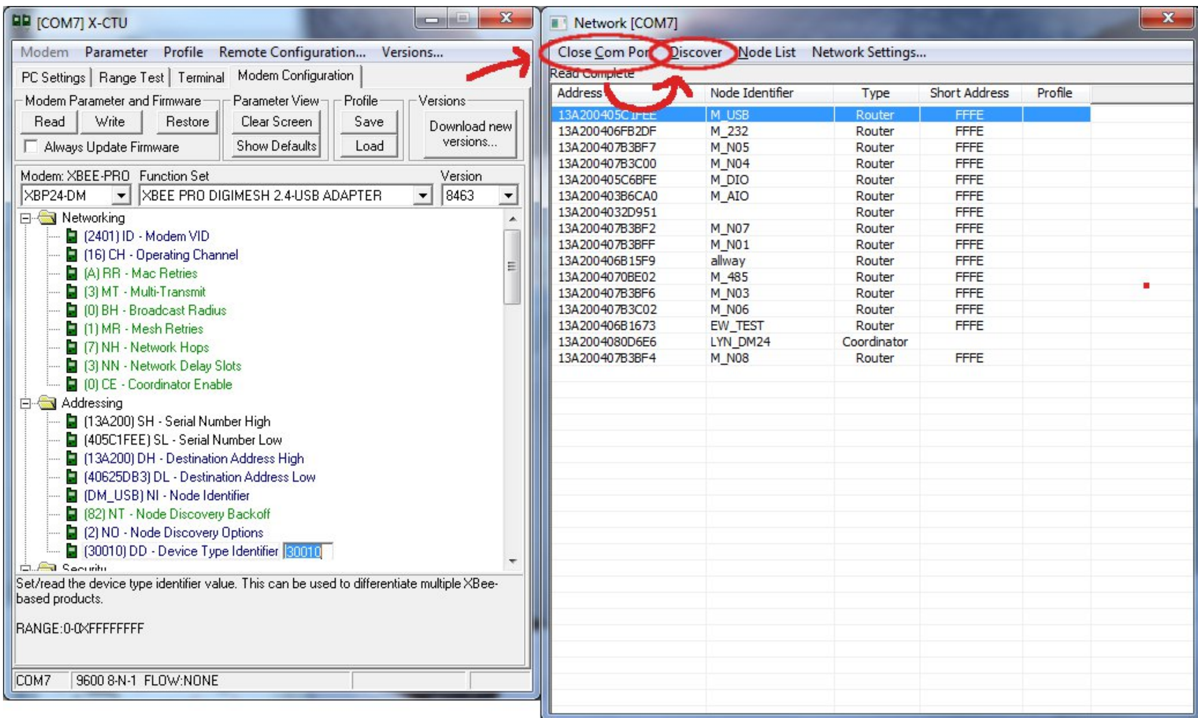
- Download the latest version of XCTU from Digi's support site: X-CTU Software.
- A computer with USB port which can run XCTU. *Win XP or Win 7 are the easiest to use, but XCTU may also be run under Windows on a MAC or under Linux Wine.*
- A USB-based XBee carrier, including one of:
  - XBIB board with suitable XBee
  - XBee USB Dongle Adapter
  - XStick

Activating remote configuration



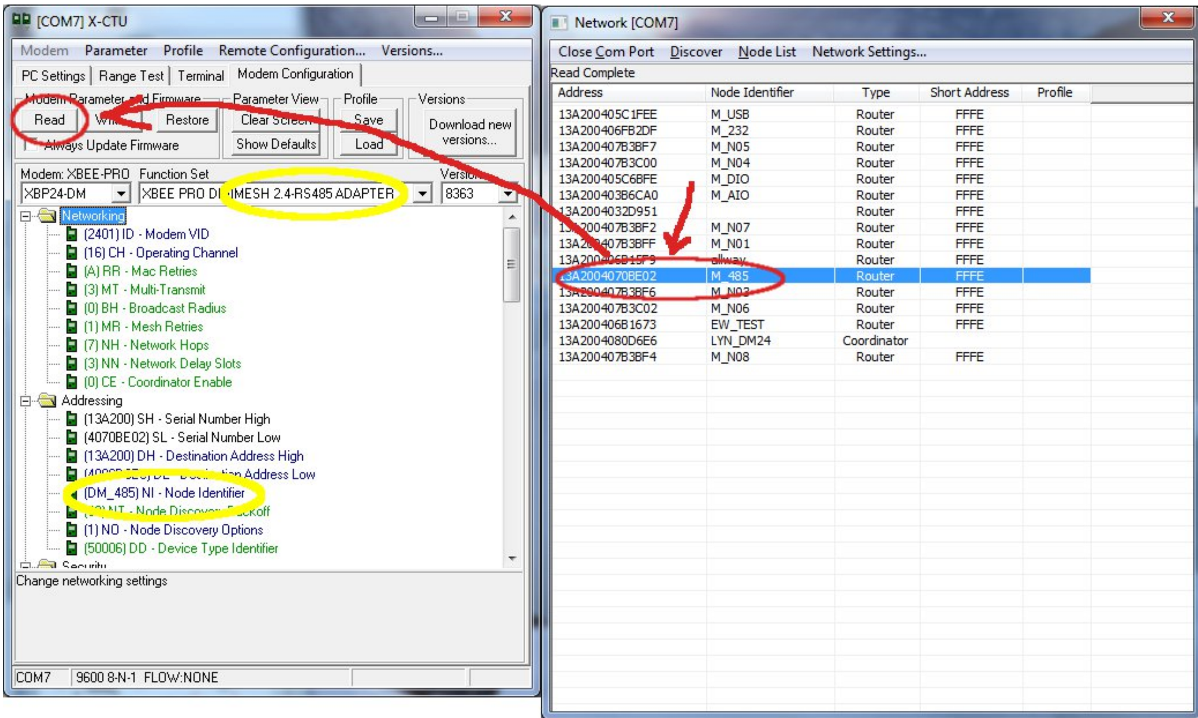
- Run XCTU
- Make sure the firmware on the computer-attached XBee supports API mode.
  - For ZigBee, use API Router with no enabled sleeping
  - For DigiMesh, enable API with the AP=1 and set Sleep-Mode (SM) to 0 or 7, as required
- Set any settings (such as Network/PAN ID, channel, etc) so that the computer-attached XBee joins your target network/mesh.
- From the **Modem Configuration** tab, click the **Remote Configuration** link and a new window will open.

Doing Network Discovery



- The first menu link will say **Open Com Port** - click this, which activates the Remote Configuration and grants it control over the serial/USB port. After clicking, the menu link changes to say **Close Com Port**.
- Click **Discover**, and you will see a list of all awake XBee nodes.
- Trying to edit sleeping node by this method is tricky - they must be awake during discovery, and also awake when you wish to <**Read**> or <**Write**> the Xbee settings.

Configure a Remote XBee



- You now select the XBee node you wish to read or change.
- Click <**Read**> on the main XCTU window, and XCTU will use Remote AT commands to fetch all of that XBee's settings.

Voila - you now can edit the remote XBee as if local. In the screen shot to the left you can see that the XBee being edited is now an RS-485 adapter, not the USB Dongle. Of course if you change critical settings (Network ID or encryption keys), then the XBee node will leave your mesh and you will lose contact with it.

ZigBee firmware can be updated OTA in this mode - DigiMesh firmware cannot be.

Exiting XCTU

It is critical that your first click the **Close Com Port** link, then close the second window before existing XCTU. If not, on occasion the comm port will remained locked and you will need to reboot your computer to recover.

Sleep settings within DigiMesh

Quick and easy DigiMesh sleep settings

Digimesh has a wide variety of sleep options, however most beginners should start with a very simple design.

Assume you have the following equipment:

- One Digi XBIB USB development board with a DigiMesh XBee (or Digi gateway with Digimesh). We will treat this as the network master which never sleeps - of course it COULD sleep, but that's not part of a **Quick and Easy** setup!
- One or more other devices with DigiMesh XBee which we'll set up to sleep.

The first step is to configure all the XBee nodes to be on the same mesh. By default, for DigiMesh 2.4 Ghz will all share ID=0x7FFF, CH=0x0C, SM=0x00. By default, they should all connect and be on the same mesh. See this page for an overview of the basic 'identity' settings: Quick guide to DigiMesh setup.

First Pass - enabling a simple mesh

Setting up the Coordinator/Master

Digimesh is not like ZigBee and really has no 'coordinator'. However, it can have a Sleep Coordinator, which is the node which defines the sleeping parameters for the entire mesh, plus sends out a synchronizing packet which allows nodes to detect if they are drifting away from the mesh (waking too soon or too late).

So in the XBIB board (or gateway), we will set: SM = 0x07 (means awake, but sleep aware) SO = 0x05 (0x01 means act as the sole designated Sleep Coordinator, 0x04 adds to send a wake/sleep message out the serial port. Set SO=0x01 to suppress these wake/sleep messages) SP = 0x00C8 (or 200. This is sleep time in 10s of msec, so 200 = 2 seconds of sleep) ST = 0x07D0 (or 2000. This is wake time in msec, so 2000 = 2 seconds of sleep)

At this point, the 'coordinator' will be always awake, and try to force all other DigiMesh nodes to confirm to the mesh with 2 seconds of sleep and 2 seconds or being awake.

Setting up your sleeping nodes

On your other nodes, we will set: SM = 0x08 (means sleep) SO = 0x02 (0x02 means never act as Sleep Coordinator) SP and ST = don't care

With some firmwares, you may need to power cycle the Digimesh Xbee to speed up any change between sleep modes.

At this point, you should see the sleeping devices sleeping for 2 seconds, then being awake for 2 seconds. You host/PC can talk to the sleeping nodes only when they are awake - which is why we wanted the SO=0x04, so that your code receives a message when the mesh wakes.

Enabling I/O Pin Sampling

Suppose you wish to see the XBee I/O pins when it wakes. That is quite easy. First, understand that your wake cycle is every 4 seconds - we sleep 2 seconds, then are awake for 2 seconds. 2 + 2 = 4

Set the following: IR = 0xFFFF (means 65,535 msec *BETWEEN* samples, assuming first is sent immediately upon wake. You want this number larger than ST so that only 1 sample is sent!) WH = 0x64 (means delay 100msec after wake before sending data. This is optional, but often your external circuitry needs to 'power up and stabilize' before you can read the IO pins) IF = 0x01 (means send data every '1-th' wake cycle, or every 4 seconds in this set-up. If you wanted data every 60 seconds, this would be 60/4 or 15th wake cycle. Dx = ?? (In order to enable data IO samples, at least one IO pin needs to be 2, 3, 4, or 5.)

Xbee Command to Device Cloud Cross Reference

Xbee experts and documentation generally use AT parameters to explain configuration parameters. Device Cloud uses verbose descriptions. The list below provides a cross-reference.

UPDATE: *Device Cloud now shows both the verbose command and the AT command in the Xbee properties screen, so this cross-refence is no longer necessary. Do still be aware that Device Cloud operates in decimal, rather than hex, in most, but not all, fields.*

Note Most of the parameters on Device Cloud use decimal, while most direct Xbee operations are in hex. Keep this in mind.

A1 - End device association

A2 - Coordinator association

AR - Aggregation route notification

BD - Serial interface data rate

BH - Broadcast radius

CA - CCA threshold

CC - Command sequence character

CE - Coordinator enable

CH - Operating channel

CI - Cluster identifier

CT - Command mode timeout

D0 - AD0/DIO0 configuration

D1 - AD1/DIO1 configuration

D2 - AD2/DIO2 configuration

D3 - AD3/DIO3 configuration

D4 - AD4/DIO4 configuration

D5 - DIO5/Assoc configuration

D6 - DIO6 configuration

D7 - DIO7 configuration

D8 - DIO8/SleepRQ configuration

D9 - DIO9/ON_SLEEP configuration

DE - Destination endpoint

DP - Disassociated cyclic sleep period

EE - Encryption enable

EO - Encryption options

FT - Flow control threshold

GT - Guard times

HP - Hopping sequence

IA - I/O input address

IC - DIO change detect

ID - Extended PAN identifier

ID - PAN identifier

IF - I/O sample from sleep rate

II - Initial PAN identifier

IR - I/O sample rate

IT - I/O samples before transmit

JN - Join notification

JV - Join verification

KY - Link encryption key

LT - Associate LED blink time

M0 - PWM0 output level

M1 - PWM1 output level

MM - MAC mode

MR - Mesh network retries

MT - Broadcast retries

MY - Network address

NB - Serial interface parity

NK - network_key

NH - Maximum hops

NJ - Node join time

NN - Network delay slots

NI - node_id

NT - Node discovery timeout

NW - Network watchdog timeout

P0 - DIO10/PWM0 configuration

P0 - PWM0 configuration

P1 - DIO11/PWM1 configuration

P1 - PWM1 configuration

P2 - DIO12/CD configuration

P3 - DIO13/DOUT configuration

PL - Transmit power level

PM - Power mode

PO - Polling rate

PR - Pull-up resistor enable

PT - PWM output timeout

RN - Random delay slots

RO - Packetization timeout

RP - RSSI PWM timer

RR - MAC retries

RR - XBee retries

SB - Stop bits

SC - Scan channels

SD - Scan duration

SE - Source endpoint

SM - Sleep mode

SN - Peripheral sleep count

SO - Sleep options

SP - Cyclic sleep period

ST - Time before sleep

SW - Sleep early wakeup

T0 - D0 output timeout

T1 - D1 output timeout

T2 - D2 output timeout

T3 - D3 output timeout

T4 - D4 output timeout

T5 - D5 output timeout

T6 - D6 output timeout

T7 - D7 output timeout

V+ - Supply voltage high threshold

WH - Wake host delay

ZA - ZigBee addressing enable

ZS - ZigBee stack profile

Working with Zigbee

These pages cover Python or mesh specific details:

Binding multiple zigbee sockets

Binding multiple sockets to the Zigbee interface

One of the common issues a developer will run into when developing for a Digi Gateway product is the behavior of the socket when multiple Python applications attempt to bind the Zigbee interface. Unlike TCP or UDP sockets where it's common to have multiple instances, Zigbee sockets support only a single instance. If an attempt to bind to the mesh socket twice, the Python interpreter will throw an exception:

```
from socket import *

s = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)
t = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)

s.bind(('', 0, 0, 0))
t.bind(('', 0, 0, 0))
Traceback (most recent call last):
  File "<console>", line 1, in ?
  File "<string>", line 1, in bind
error: (22, 'Invalid argument')
```

If multiple running applications do need to have access to the mesh socket, a locking mechanism of some sort would have to be used, similarly to if both applications shared a serial port. To summarize, a bound zigbee socket is a unique and limited resource on our Digi Gateway devices.

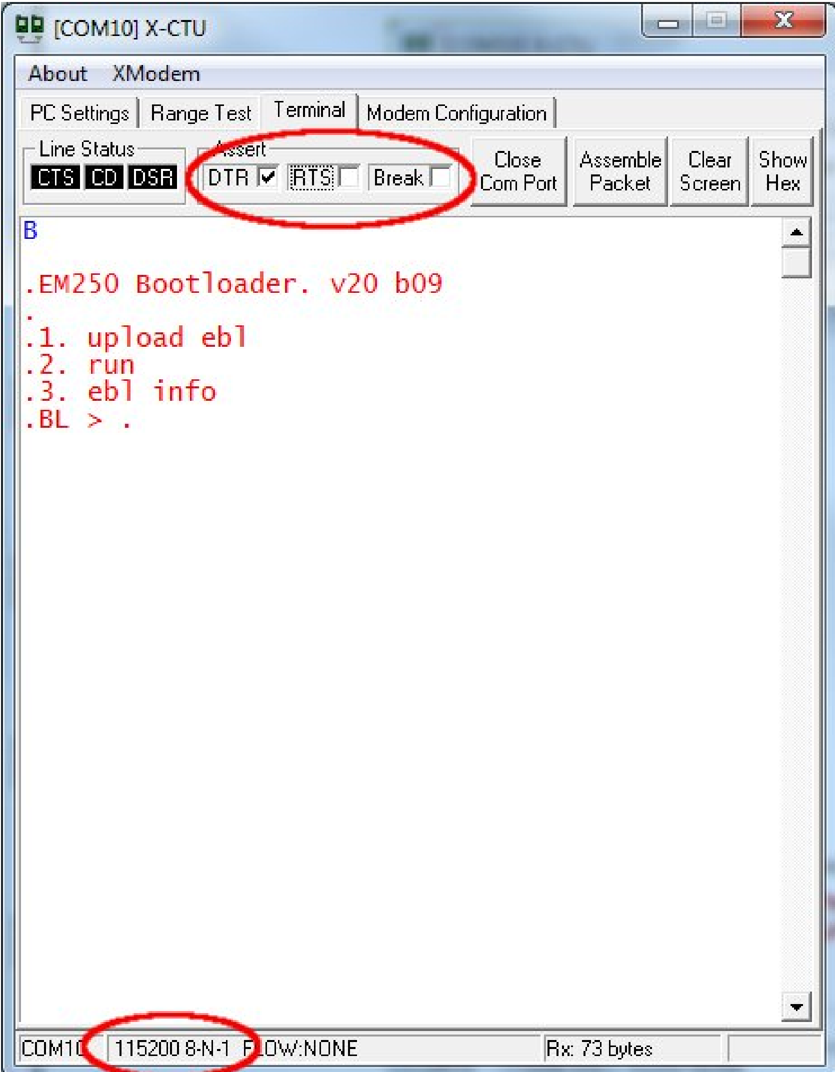
Bootloader to force XBee reflash

Using XCTU to invoke the Bootloader

Sometimes you have trouble talking to an XBee - perhaps the baud rate is unknown, or the firmware within it disables the serial port. If you work with the XBee AIO or DIO adapters, you will probably need to use this procedure.

You can help XCTU reflash the XBee by manually activating the XBee Bootloader.

- Open an X-CTU Terminal Window.
- Change Baud to **115200**.
- Assert/check **DTR**, De-assert/uncheck **RTS**, Assert/check **Break**.
- Click the **/RESET** button on the XBee development board (example: XBIB).
- De-assert/uncheck **Break**.
- Click the **Terminal Window**, then Type a **B** and press **Enter**. The B must be all CAPS).
- The XBee should return the Bootloader Menu that looks something like this:



- Without resetting or power cycling the XBee go to the **Modem Config Tab**.
- Check **Always Update Firmware** and select the firmware you wish to load./
- Click **Write**.

If this procedure does not work, then perhaps your XBee (or development board) is damaged.

Channels, Zigbee

Channel detail

802.15.4 and Zigbee break the 2.4Ghz band into 16 channels as shown below.

Decimal	Hex	Frequency	SC mask	WiFi Conflict	Comments
11	0x0B	2.405GHz	0x0001	Overlaps Ch 1	Newer XBee only
12	0x0C	2.410GHz	0x0002	Overlaps Ch 1	
13	0x0D	2.415GHz	0x0004	Overlaps Ch 1	
14	0x0E	2.420GHz	0x0008	Overlaps Ch 1	
15	0x0F	2.425GHz	0x0010	Overlaps Ch 6	
16	0x10	2.430GHz	0x0020	Overlaps Ch 6	
17	0x11	2.435GHz	0x0040	Overlaps Ch 6	
18	0x12	2.440GHz	0x0080	Overlaps Ch 6	
19	0x13	2.445GHz	0x0100	Overlaps Ch 6	
20	0x14	2.450GHz	0x0200	Overlaps Ch 11	
21	0x15	2.455GHz	0x0400	Overlaps Ch 11	
22	0x16	2.460GHz	0x0800	Overlaps Ch 11	
23	0x17	2.465GHz	0x1000	Overlaps Ch 11	
24	0x18	2.470GHz	0x2000	Overlaps Ch 11	Newer XBee only
25	0x19	2.475GHz	0x4000	No Conflict	Newer XBee only
26	0x1A	2.480GHz	0x8000	No Conflict	Newer non-PRO XBee only

Things to note:

- WiFi bands vary by country and region. The three WiFi bands above are common worldwide, but for example, Japan allows enough extra channels to permit a fourth WiFi band starting at channel 14. Therefore the "No Conflict" statements above would not be true in Japan.
- Sometimes Xbee documentation uses hex as "17" instead of "0x17" to mean a channel like 23. Be careful about the context!
- Digi Xbee starts on the first channel in the SC mask and stops on the first channel which grants it association. Depending on the age of the firmware on your Xbee ZB module it might start on channels 0x0B, 0x0C, or 0xD (11, 12, or 13)

- Once a coordinator selects a channel, it does not move regardless of error rate (i.e. not frequency agile), so ZigBee does not automatically hop to a clear channel - it does this to avoid confusing potentially sleeping devices. A user or external software must manually force the coordinator to move via setting changes, plus the routers will NOT follow unless various non-default settings have been enabled.
- More than one 802.15.4/Zigbee coordinator can settle on a single channel.
  - For example you could have 5 distinct WPAN, and all 5 Digi gateways decide they want to use the default channel 13/0x0D despite the fact that no WPAN uses any of the other 15 channels.
  - So in a crowded 802.15.4/Zigbee environment, it is helpful to change your SC to force your WPAN to less-used channels.
- Not all Zigbee radios support all 16. It depends on the age and power rating.
  - Older Digi XBee supported only 12 to 23, so default scan_channel/SC is 0x1FFE
  - Newer Digi XBee non-PRO (S2/S2B/S2C) support all channels, so scan_channel/SC can be 0xFFFF.
  - Newer Digi XBee PRO (S2) support channels 11 to 24, so scan_channel/SC can be 0x3FFF.
  - Newer Digi XBee PRO (S2B/S2C) support channels 11 to 25, so scan_channel/SC can be 0x7FFF.
- A new-hardware coordinator picking channels 11, 24, 25, or 26 won't be accessible by older hardware, so consider force the SC to 0x1FFE or narrower. You will have no clear 'feedback' that this is occurring

Channel Mask Examples

The following table shows four common channel masks, plus one single channel example, to demonstrate how binary bitmasks are linked to individual channels. If there is a 1 in the channel column, that channel is available for use, and will be scanned, when that mask is configured.

Remember - it is ONLY the mask in the coordinator which affects the selected channel. Assuming you used fixed PAN Id, then you have the most flexible design by putting the reduced/restricted mask in the coordinator alone, and leaving the other devices to check all channels.

The binary ones and zeros are then converted to hexadecimal to create the mask to configure. Using the last example, forcing channel 25: 0100,0000,0000,0000 in binary is 0x4000 in hexadecimal.

Channel/Mask	Notes	26	25	24	23		22	21	20	19		18	17	16	15		14	13	12	11
0xFFFF	All channels, may get low-power channel 26	1	1	1	1		1	1	1	1		1	1	1	1		1	1	1	1
0x1FFE	Best for mixed old/new Xbee networks, very safe	0	0	0	1		1	1	1	1		1	1	1	1		1	1	1	0
0x3FFF	Newer Digi XBee PRO (S2)	0	0	1	1		1	1	1	1		1	1	1	1		1	1	1	1
0x7FFF	Digi XBee PRO (S2B/S2C)	0	1	1	1		1	1	1	1		1	1	1	1		1	1	1	1
0x4000	Example of single channel (Channel 25 only)	0	1	0	0		0	0	0	0		0	0	0	0		0	0	0	0

Tips

Control SC within the coordinator

For maximum compatibility, it is advisable to restrict the number of channels the coordinator can use, and keep routers and end nodes as wide open as possible. The coordinator sets the channel the network will use, so it is best to make sure it will set a channel that all the other devices are configured to use. As some older chips are limited to which channels they can use it is important that the coordinator does not select a channel some chips cannot use.

- Coordinators should be set for 0x1ffe
- Routers and End nodes should be configured for as wide a range as possible. Newer low-power XBee should be set for 0xffff. All XBee should be AT LEAST 0x1ffe.

Helping your manufacturing / test people

If your own products contain Digi XBee, then careful channel control of local semi-permanent XBee networks will help a great deal. For example, you put sleeping XBee modules into a battery powered sensor and require manufacturing to test each unit as part of final quality-assurance. Assuming you also have a few test networks in engineering and tech support, your manufacturing people will find the final QA test frustrating, as the units-under-test may not join the manufacturing coordinator.

The simple solution is to leverage the knowledge that a default Digi XBee starts at the lowest enabled channel and works upwards looking for a mesh to join. This offers a handy solution:

- First, make sure all semi-permanent gateway\coordinators have SC set to one or a few of the higher channels. For example, if engineering and tech support use SC=0x1F00, then their mesh will never use the lower 8 ZigBee channels.
- Second, manufacturing should insure that their QA test gateway\coordinators are only in those lower 8 channels. This can be done by manually limiting SC to 0x00FE, or manually confirming that the functioning coordinators have selected a channel between 11 and 18 (hex 0x0B to 0x12).

If you do the above, then any new sensor powered for the final test will find the QA coordinators, not those used by engineering and tech support.

A more active way to help with ad-hoc testing is for engineering and tech support people (who understand the technology) to manually turn their joining on or off as required. For example, an engineer with 1 gateway and 3 XBee 485 adapters properly joined and running, no longer requires 'new joining'. So if they turn all joining of (set NJ=0 in the gateway AND in the 3 XBee 485 adapters) then systems set up by QA or company sales people preparing demos will never join or be affected by the semi-permanent engineering system. Should one of the XBee 485 loss association, or the engineer wish to join new adapters, they can temporarily enable joining.

Avoiding your Office/Home WiFi

In general, the small infrequent ZigBee packets are not affected by WiFi, and one could say that the destructive 'bullets' ZigBee punches in your WiFi bandwidth are treated as common back ground noise and worked around.

However, since you can control which WiFi channels your ZigBee may interfere with, and since you have over a dozen channels to select, it is smart to setup your ZigBee to *NOT* interfere with your home or office WiFi. You would enforce this avoidance by setting SC in your gateway\coordinators only.

Most consumer WiFi devices default to use 'channel 6' (so WiFi channels 6 to 10), therefore your home or small business WiFi is probably using channel 6. If your PC/notebook/tablet/smart-phone allows you to do a 'WiFi site survey', scanning all WiFi channels for raw traffic, then you can intelligently put your ZigBee either in the clearest bands, or those used by your neighbors!

Desired Outcome	SC to use
To Avoid WiFi Ch 1	0x7FF0
To Avoid WiFi Ch 6	0x7E0F
To Avoid WiFi Ch 11	0x41FF
To Avoid all WiFi	0x4000
To Force Use of WiFi Ch 1	0x000F
To Force Use of WiFi Ch 6	0x01F0
To Force Use of WiFi Ch 11	0x3E00
To Force Use of non-WiFi	0x4000

Be Aware of Wireless Audio or Phones

Successful use of ZigBee may require careful avoidance of competing NON-WIFI devices including:

- Wireless mic/headsets in a restaurant drive-through
- Wireless phone systems
- Wireless audio (music) headphones
- Wireless phone head sets.

The serious risk with these competing 2.4GHz devices is that many use as much raw bandwidth as possible to maximize audio quality. So for example, a ZigBee mesh in a fast food restaurant might work great when installed at 6AM, but start to see a 60-70% error rate when the drive-through audio system is operational and busy.

Ideally, review the technical documentation for the competing technology. Although it is NOT WiFi, it likely defines its frequency usage in WiFi terms, and it likely includes a way to assign it Wifi ranges 1, 6, or 11. Since most consumer grade WiFi devices default to use channel 6, it is safe to assume these non-WiFi devices likely default to either WiFi channel 1 or 11 - meaning exactly where you may wish to place your ZigBee.

If you do not have access to the product documentation, review the 'XBee Range Test' procedures on Digi's tech support website, then run a few range tests when the competing 2.4Ghz technology is being heavily used. If you do not detect any significant error rate, then either your selected XBee frequency is unaffected by the competing technology, or the competing technology is gracefully sharing the radio waves. The need to possibly change your XBee channel should be very obvious after very little testing.

Configuring Digi XBee modules

How to configure Xbee modules from the gateway

Of course one can use the web Interface, however if you have dozens of modules to configure you'll probably goof up and forget a parameter or two using the web UI. So you might find using the gateway's CLI or telnet/ssh interface easier - plus if you create your text lines in a text-editor then you have automatic documentation!

Obtain your node list

First, have your end devices associated - if not, that's a different topic.

Use the **disp xbee ref** command to refresh and display your nodes, then cut-and-paste this into your text editor. Here is a simple example with only two nodes running Zigbee 2007 firmware:

```
#> disp xbee ref
XBee network device list
PAN ID:           0x68a3 - 0x0000000000001105
Channel:          0x0d (2415 MHz)
Gateway address:  00:13:a2:00:40:0a:12:41!
Node ID      Network Extended address         Product type
------------ ------- ----------------------- ------------
COORDINATOR
QUBILYN      [0000]! 00:13:a2:00:40:0a:12:41! X2 Gateway
ROUTERS
END NODES
[500f]! 00:13:a2:00:40:3e:15:2d! RS-232 Adapter
[a109]! 00:13:a2:00:40:3e:15:18! RS-232 Adapter
```

While you are at it, you can dump a HELP list of the supported commands. Note that the '?' won't show on your display and the list scrolls out without pressing the Enter key:

```
#> set xbee ?

Configure xbee network.

syntax: set xbee [options...] [device_settings...]

options:

state=(off|on)              {Enable XBee gateway}
address=(id|address)        {Specify device}
<CC>[[=]param]              {Run AT command on device:
<CC> is 2 character upper case command
param is <decimal>, 0x<hex>, or "string"}

device_settings:

    aggregation=(0-255)         {AR, Aggregation route notification, x 10 sec}
    assoc_led=(0-65535)         {LT, Associate LED blink time, msec}
    broadcast_hops=(0-10)       {BH, Broadcast radius}
    command_char=(char)         {CC, Command sequence character}
    cluster_id=(0x0-0xffff)     {CI, Cluster identifier}
    command_timeout=(2-655)     {CT, Command mode timeout, x 100 msec}
    dest_addr=(address)         {DH, Destination address}
    dest_endpoint=(0x1-0xf0)    {DE, Destination endpoint}
```

```
dio0_config=(0-5)           {D0, AD0/DIO0 configuration}
dio1_config=(0-5)           {D1, AD1/DIO1 configuration}
dio2_config=(0-5)           {D2, AD2/DIO2 configuration}
dio3_config=(0-5)           {D3, AD3/DIO3 configuration}
dio4_config=(0-5)           {D4, AD4/DIO4 configuration}
dio5_config=(0-5)           {D5, DIO5/Assoc configuration}
dio6_config=(0-5)           {D6, DIO6 configuration}
dio7_config=(0-7)           {D7, DIO7 configuration}
dio10_config=(0-5)          {P0, DIO10/PWM0 configuration}
dio11_config=(0-5)          {P1, DIO11/PWM1 configuration}
dio12_config=(0-5)          {P2, DIO12/CD configuration}
dio_detect=(0x0-0xffff)     {IC, DIO change detect, bitfield}
discover_timeout=(32-255)   {NT, Node discovery timeout, x 100 msec}
encrypt_enable=(0-1)        {EE, Encryption enable}
encrypt_options=(0x0-0x7)   {EO, Encryption options, bitfield}
ext_pan_id=(0-8 bytes)      {ID, Extended PAN identifier}
guard_times=(2-3300)        {GT, Guard times, msec}
initial_pan_id=(0x0-0xffff) {II, Initial PAN identifier}
join_time=(0-255)           {NJ, Node join time, sec}
join_notification=(0-2)     {JN, Join notification}
join_verification=(0-1)     {JV, Join verification}
link_key=(0-16 bytes)       {KY, Link encryption key}
max_hops=(1-255)            {NH, Maximum hops, x 50 msec}
network_key=(0-16 bytes)    {NK, Network encryption key}
node_id=(0-20 chars)        {NI, Node identifier}
packet_timeout=(0-255)      {RO, Packetization timeout, chars}
power_level=(0-4)           {PL, Transmit power level}
power_mode=(0-1)            {PM, Power mode}
pullup_enable=(0x0-0x7fff)  {PR, Pull-up resistor enable, bitfield}
rssi_timer=(0-255)          {RP, RSSI PWM timer, x 100 msec}
sample_rate=(0-65535)       {IR, I/O sample rate, msec}
scan_channels=(0x1-0xffff)  {SC, Scan channels, bitfield}
scan_duration=(0-7)         {SD, Scan duration, exponent}
serial_parity=(0-4)         {NB, Serial interface parity}
serial_rate=(0-230400)      {BD, Serial interface data rate}
sleep_count=(1-65535)       {SN, Peripheral sleep count}
sleep_mode=(0-5)            {SM, Sleep mode}
sleep_options=(0x0-0xff)    {SO, Sleep options, bitfield}
sleep_period=(32-2800)      {SP, Cyclic sleep period, x 10 msec}
sleep_time=(1-65535)        {ST, Time before sleep, msec}
source_endpoint=(0x1-0xf0)  {SE, Source endpoint}
stack_profile=(0-2)         {ZS, ZigBee stack profile}
supply_threshold=(0-1200)   {V+, Supply voltage high threshold, mvolts}
```

Create your configuration

Now you can create our configuration. In this example we'll hard-code the following parameters:

1. ext_pan_id= the Extended PAN ID - you only want the end-devices to join a specific PAN
2. node_id= the Node Identifier - you give each end-device a user-friendly name, the tank tag in this case
3. dest_addr= the return address - this tells the end-device to always send data to the gateway
4. power_level= the Power Level - you crank down the power since all devices in the same room

Here are the lines created. Save this for your documentation.

```
set xbee address=00:13:a2:00:40:3e:15:2d! ext_pan_id=0x1105 node_id=TK1002R
set xbee address=00:13:a2:00:40:3e:15:2d! dest_addr=00:13:a2:00:40:0a:12:41!
power_level=2

set xbee address=00:13:a2:00:40:3e:15:18! ext_pan_id=0x1105 node_id=TK1001U
set xbee address=00:13:a2:00:40:3e:15:18! dest_addr=00:13:a2:00:40:0a:12:41!
power_level=2
```

Load your Configuration

Now you can use cut-and-paste to send your configuration to your end-devices. In theory you can paste the entire file at one time, however you get better feedback by doing it in small groups - one device or a few devices at a time. This way you can see if any settings fail.

Special Commands

What if your required setting or command isn't supported by set xbee? If you look at the HELP, you have the <CC>[[=]param] option. Below is an example of using the "NI" command, with the required "AC" and "WR" to mimic the node_id= command. So it is important to remember that the NI and node_id are NOT aliases for each other. The node_id command sets the id, then saves the value in NVRAM, while the NI command just pokes a temporary new identifier in RAM. So you need to understand the full use of the AT commands for this to work.

```
set xbee address=00:13:a2:00:40:3e:15:2d! NI=TK1002R AC WR
```

Create your own display mesh command

Create your own 'disp mesh' command

A realistic Python application using the get_node_list() function.

Users of the Digi CLI (telnet/command line) are familiar with the **disp mesh** command. It shows the same basic information as the web interface mesh networking page. However, suppose you wish to see other information listed? Suppose you wish to see it sorted in extended MAC address or node identifier order? Suppose you wish to see a list containing only end-devices which are temperature sensors and include the current temperature?

You can use the getnodelist() function to create your own custom tool, which creates your own dream-list. The fully functional Python program linked below runs on any Digi ConnectPort X gateway and shows a list of nodes sorted by either node identifier, extended HW MAC addres, or product type. This sample application was created because the author tests large Zigbee systems with up to 50 devices and needs to quickly confirm if all nodes are active - and the various sort orders rapidly allows detecting if any nodes are missing.

Routines used; things you can learn

The program my_disp_mesh.py does the following:

- Uses **sys.argv[]** to specify the desired sort order from the command line.
- Uses **zigbee.getnodelist()** to obtain a list of associated nodes.
- The device_type information as part of the getnodelist response is converted to a string such as "XB24-ZB:XBee232" for a Digi Xbee RS-232 adapter running on a ZB network.
- A list of dictionary items are sorted in various ways based on dictionary keys using an inline lamda function - (for example **nodes.sort(key=lambda x: x['name']**) to sort the list of dictionaries on the key 'name'.
- The results are printed in a table form.

Sample output

```
#> Python my_disp_mesh.py help
These arguments can be used with my_disp_mesh.py:
 name    = sorts node list on Node Id
 mac     = sorts node list on HW MAC Address
 short   = sorts node list on 16 bit Address
 type    = sorts node list on Product Type
 refresh = do a mesh refresh before creating your list

#> Python my_disp_mesh.py mac

My DISP MESH - sorted on HW MAC Address
GW: [00:13:a2:00:40:3e:1c:80]! [0000]! n:TankMom XB24-ZB:X4

01: [00:13:a2:00:40:34:16:14]! [a8d2]! n:DEBI_4 t:XB24-ZB:XBee232
02: [00:13:a2:00:40:3e:15:18]! [d756]! n:ANNA_1 t:XB24-ZB:XBee232
03: [00:13:a2:00:40:3e:15:2d]! [a865]! n:BELA_2 t:XB24-ZB:XBee232
04: [00:13:a2:00:40:4a:70:7e]! [6789]! n:CALI_3 t:XB24-ZB:XBee232
```

```
05: [00:13:a2:00:40:52:29:d7]! [458a]! n:FANI_6 t:XB24-ZB:XBee232PH
06: [00:13:a2:00:40:52:29:f9]! [1234]! n:ELSA_5 t:XB24-ZB:XBee232PH

#> Python my_disp_mesh.py name

My_Disp_Mesh - sorted on Node Name + HW MAC Address
GW: [00:13:a2:00:40:3e:1c:80]! [0000]! n:TankMom XB24-ZB:X4

01: [00:13:a2:00:40:3e:15:18]! [d756]! n:ANNA_1 t:XB24-ZB:XBee232
02: [00:13:a2:00:40:3e:15:2d]! [a865]! n:BELA_2 t:XB24-ZB:XBee232
03: [00:13:a2:00:40:4a:70:7e]! [6789]! n:CALI_3 t:XB24-ZB:XBee232
04: [00:13:a2:00:40:34:16:14]! [a8d2]! n:DEBI_4 t:XB24-ZB:XBee232
05: [00:13:a2:00:40:52:29:f9]! [1234]! n:ELSA_5 t:XB24-ZB:XBee232PH
06: [00:13:a2:00:40:52:29:d7]! [458a]! n:FANI_6 t:XB24-ZB:XBee232PH
```

Download the Python code

This code only runs on a Digi ConnectPort X gateway:Python program "My_disp_mesh.zip" in ZIP form.

DIA difference between ZigBee and DigiMesh

Mesh setup

Although not related to DIA, people familiar with ZigBee setup will find DigiMesh a new experience.

Unlike ZigBee where new nodes can sometimes merely be powered up and joined to the correct network, DigiMesh works more like the older ZNet design where you will require either:

- Use XCTU and an XBIB board to manually configure the required Network setting
- Use a 'commissioning node' in an XBIB board with default settings to locate the new factory-fresh DigiMesh device, then using the 'Remote Configuration' setting, push the required Network settings into the new device.

See this Wiki page for a summary of the required parameters: Quick guide to DigiMesh setup.

Understanding sleeping

There is a basic difference between ZigBee and DigiMesh.

- ZigBee has powered, non-sleeping devices called routers (or parents). The coordinator is just a router with an extra role.
  - Routers 'mesh', forming the resilient mesh-topology hyped by ZigBee advocates.
  - Routers discover routes (paths) and adjust routes as old peers go offline and new peers come online.
  - Routers repeat broadcasts
  - In general routers don't say much - unless asked to move data.
- ZigBee also has sleeping end-devices (or children).
  - End-Devices do not mesh. They locate ONE specific router/parent to belong to.
  - When End-Devices wake up, they start polling their parent 10-times per second, sending data or looking for data the parent has buffered.
  - End-Devices do NOT repeat broadcasts
  - However, end-devices are rather chatty as long as they are awake!

DigiMesh

- All devices in DigiMesh are routers, forming the mesh
- DigiMesh devices can sleep, or be fully awake.
- All nodes need to understand the wake/sleep cycle, and only talk during wake cycles.

DigiMesh in DIA

What this means to DIA (and YML files) is that any ZigBee node can have any desired data sample rate. You can have 20 temperature sensors, all with random sample rates between 1 second and 1 day. The fully awake routers can send data at any time, and the sleeping end-devices can wake anytime, trusting that their parent is awake, ready and waiting to handle their data.

In contrast, under DigiMesh all data samples need to be an limited integral multiple of the entire mesh's sleep/wake cycle.

DIA/DigiMesh sleep coordinator

Within DIA, the Digi gateway acts as the designated sleep coordinator. This means DIA sets the sleep/wake times within the XBee within the gateway, and the gateway imposes the sleep/wake cycle over the entire mesh.

This is a DIA limitation, not a DigiMesh one. As of this release, DIA does not support nominated sleep coordinators, which is more appropriate to a peer-to-peer system. Most DIA systems are gateway-centric, assuming the gateway is critical and the devices have little or no function without the gateway, and therefore their sleep-coordinator.

DIA/DigiMesh sleep mode

To disable all sleeping, set the DigiMeshDeviceManager's **sleep_time** setting to zero. The SM value in all nodes is forced to 0x00.

To enable sleeping, set the DigiMeshDeviceManager's **sleep_time** and **wake_time** settings to a value higher than 10msec - for example 20000 and 10000 respectively, which means the mesh sleeps for 20 seconds, then is awake for 10 seconds. The SM value in all nodes is forced to either 0x07 or 0x08, which means 'always awake but sleep aware' and 'sleep'.

This 20/10 design means for 20 seconds none of the nodes transmit data - even the fully awake nodes (SM=0x07). Every device waits until the mesh wakes, then they all try to send messages as required, including forwarding (and 'meshing') as required. Of course the wake_time needs to be long enough to allow all the required messages to move, including passing across multiple hops.

This is a DIA limitation, not a DigiMesh one. DigiMesh includes some richer asynchronous sleep-modes, which allow nodes to wake up during the sleep-period and talk, however this implies the sleeping node is within 'ear shot' of a fully awake node since no other sleeping node will be awake to help route the message.

DIA/DigiMesh data sample rates

At this point, all nodes can use external logic to decide when (during any particular wake-time) that they wish to send data. If the node has its own processor, it can format a message and send it during the beginning portion of the wake cycle, hopefully having time to receive any responses. The DIA framework on the gateway can also send out polls. The DigiMeshDeviceManager will queue up any pending outgoing messages from DIA drivers, then sending them out when the mesh wakes.

Support for the Digi XBee's IC/IR commands is also supported. By default, a node will send data each time it awakes. This requires IR=0xFFFF, IF=0x01 and at least 1 of the XBee IO pins be configured as an input.

The DigiMeshDeviceManager's **set_if** setting can be set to True or False. If set_if=False, then the DIA does NOT interfere with the IC/IR/IF settings within the remote nodes. DIA assumes that the user has configured the remote devices as required.

If set_if=True, then the DIA uses the various sample_rate setting of drivers to calculate the correct IR/IF values. For example, if the device wishes a once per minute data sample and the mesh is waking every 30 seconds (for example, our 20/10 sleep_time/wake_time settings), the setting IR=0xFFFF and IF=0x02 means the XBee will send a data sample every second wake cycle. The DIA will round down, so if the driver sample_rate is once per 75 seconds with a wake-cycle of 30 seconds (20/10 sleep/wake), then DIA is forced to select IF=0x02, which will give you data every 60 seconds.

DIA Releases

DIA release 2.0.x (May 2012)

Note This version is not released yet.

DigiMesh support

One of the largest changes is the introduction of two different XBee manager devices, so your YML file should now use either of these:

```
    driver: devices.xbee.xbee_device_manager.zigbee_device_
manager:ZigBeeDeviceManager
```

Or

```
    driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
```

Although ZigBee users should migrate to call the ZigBeeDeviceManager explicitly, YML files including the old XBeeDeviceManager will run the ZigBeeDeviceManager instead.

DigiMesh sleep design

The DIA DigiMeshDeviceManager assumes 1 of 2 designs:

- Non-sleeping, so all nodes run as SM=0
- Sleeping, with the gateway acting as the preferred Sleep Coordinator. Nodes will be set to:
  - Gateway/Sleep-Coordinator: SM=7, SO=0x05
  - Non-Sleeping Node: SM=7, SO=0x02
  - Sleeping Node: SM=8, SO=0x02

Support for sleep-coordinator 'Nomination' and the asynchronous/pin-sleep modes of DigiMesh are not yet supported.

DigiMesh YML changes

Here is an YML usage for the DigiMeshDeviceManager:

```
  - name: xbman
    driver: devices.xbee.xbee_device_manager.digimesh_device_
manager:DigiMeshDeviceManager
    settings:
        dh_dl_force: True
        dh_dl_refresh_min: 300
        sleep_time: 9000
        wake_time: 1000
        set_if: True
```

The **dh_dl_force and dh_dl_refresh_min** settings apply to both ZigBee and DigiMesh systems and will be explained in another section of this document.

The three DigiMesh specific settings are:

- **sleep_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes wake up every 2 seconds. Set this to 0 to disable sleeping.
- **wake_time** is an integer which defaults to 2000 (2 seconds), which means the sleep coordinator instructs all nodes to stay awake for 2 seconds. If sleep_time = 0, this setting is ignored.
- **set_if** is a boolean which defines if DIA will adjust the IR/IF settings in remote nodes. It defaults to False, which assumes the user manually handles data messages. If set to True, then DIA makes a best-effort to map individual driver 'sample_rate' settings to XBee IF intervals.

For example, if a XBee DM AIO adapter has a sample_rate_ms setting of 60000 (once a minute). The default DigiMeshDeviceManager settings of sleep_time = 2 seconds and wake_time = 2 seconds means a sleep-cycle of 4 seconds total. So DIA will set the XBee DM AIO adapter with IR=0xFFFF and IF=0x0F, which means the AIO adapter will send in an IO data sample when it wakes within the 15th sleep-cycle (ie: 60-sec / 4-sec). DIA rounds down if the rates do not factor cleanly.

Internal driver changes to support DigiMesh

Users wishing their custom driver code to run under DigiMesh will need to make some changes to their drivers.

First, drivers must not directly set SM, SO, or IR - they must submit the required settings using the XBee Manager. For example:

```
        xbee_sleep_cfg = self._xbee_manager.get_sleep_block(
                                     self._extended_address,
                                     sleep=False,
                                     sleep_rate_ms=sample_rate,
                                     awake_time_ms=0)
```

The XBee Manager (ZigBee or DigiMesh) uses the 'sleep' parameter to select the correct SM/SO settings. The 'sleep_rate_ms' and 'awake_time_ms' parameters are used to set the setting SN/SP/ST for ZigBee and IR/IF for DigiMesh.

For clearer examples, look at the DIA drivers for the Digi Adapters.

Second, your driver should no longer set the DH or DL settings directly. The support for these have been moved to the XBee Manager since the values required may be different for different XBee technology.

Destination address (DH/DL) support

In this release of DIA, drivers should no longer directly set the DH/DL of XBee nodes. This function has moved to the XBeeDeviceManager classes to support different behavior required by different Xbee technologies.

Two new settings have been added which can be used with both ZigBee and DigiMesh systems:

- dh_dl_force is a string and can be:
  - None or False, which means do NOT change DH/DL in any node
  - True, which means do what is most appropriate for the XBee technology. For ZigBee, this forces the default aggregator address of '00:00:00:00:00:00:00:00!' into all DH/DL. For DigiMesh, this forces the gateway's SH/SL address into all DH/DL.

- Coordinator, which means use the gateway or coordinator SH/SL address for all nodes DH/DL
- The string of an exact address such as '00:13:a2:00:40:32:d9:51!' which is to be used.

- dh_dl_refresh_min is a string which defines when the DH/DL setting is affected.
  - None, which means never change DH/DL at all
  - Once, which means DH/DL will be set when nodes are configured, and also broadcast once.
  - Config, which means DH/DL will only be set when nodes are configured, and not broadcast.
  - A time with tag such as '5 min' or '1 day' which defines a repeated broadcast interval. if no tag is applied, then a number is assumed to be in minutes.

Rapid Reboot Detection

Many Digi gateways include an option to reboot if an auto-started Python script exits. This is a wise precaution against an expected error occurring months after reboot, but can make the unit unreachable by Device Cloud if the error occurs repeatedly at the start of the script.

For example, if the main Python script has a simple typo or a ZIP file was accidentally truncated during download, then the auto-started Python script may exit instantly, forcing a reboot within seconds of the last reboot and startup. A gateway locked in such a reboot-cycle will never be connected to Device Cloud long enough to allow fixing the problem.

This release of DIA addresses this within the main dia.py file. By default this feature is disabled, so users wishing this protection must manually create (or seed) a text file named nospin.txt in the Python area of the gateway.

The file name is case-sensitive, so prevent allowing Windows to rename your file Nospin.txt!

The file can be empty or contain a line of text. Once active, the file will be rewritten by dia.py upon every reboot with a timestamp, and if dia.py detects that the last 10 reboots have occurred in less than 20 minutes, then dia.py will sleep for 10 minutes before trying to start.

If you enable this feature on a gateway without time service (so it always boots as 1-Jan-1970), then after 10 reboots the gateway will always delay starting for 10 minutes. This may be undesirable, but not as undesirable as becoming unreachable from Device Cloud.

Users writing their own auto-start code should consider copying this same function to their own applications.

Other Internal Driver Changes

Older DIA drivers tended to make very poor use of Python or object oriented paradigms. For example, much of the code within most DIA Xbee drivers is duplicated in all of the peers - literally cut-and-pasted. This is a clear violation of Object Programming concepts which assume that if all derived classes will need the same code fragment, then it should be handled by the base class, not repeated within all derived classes.

Tracer Module

Besides the previous tracer-levels of ('debug', 'info', 'warning', 'error', 'critical'), two new lower levels are defined BELOW debug.

tracer.calls

These should be used for simple debug statements showing a routine was called. This was added because some programmers pepper too many 'debug' statements such as:

```
def calculate_average(self, a, b):
    # this should now be self._tracer.calls() instead
    self._tracer.debug("calculate_average")
```

These tend to pollute the debug trace with considerable low-grade information. Defining a new sub-debug level named 'calls' allows the user to enable/disable these simple, low-grade debug lines upon demand.

tracer.xbee

For example, these are used by the DigiMesh manager to announce all of the mesh-wake and mesh-sleep messages received from the gateway XBee. This information is generally used only when debugging the xbee manager, or sleep/performance problems with drivers.

tracer lines now return a boolean value, True if active

This should be used in situations where the tracer parameters are costly to format. In the example below, the parameter is a time-expensive operation - a STRING conversion of a list of hundreds of integers. Python must evaluate all parameters whether the tracer level is true or not, which can cost a huge time penalty even when tracing is not enabled. Using an if-then statement greatly reduces the performance hit.

```
    if self._tracer.debug():
        # my_list is converted to a string ONLY when debug is True
        self._tracer.debug("list data:%s", str(my_list))
```

class DeviceBase

The DeviceBase now creates three variables which can be used by derived classes:

- **self._name** = name as passed in by def __init__(self, name, core_services, settings, properties)
- **self._core** = core_services as passed in by def __init__(self, name, core_services, settings, properties)
- **self._tracer** = get_tracer(name)

In past DIA versions, most derived class duplicated the effort to create and manage these as self.__name, self.__core, and self.__tracer. These cause no harm, but waste resources and prevent derived classes from being created from derived classes. When porting drivers to the new DIA, you are encouraged to use DeviceBase's copies of these variables.

The DeviceBase also now has a setting named 'trace', which defaults to , *which means use global trace level in self._tracer. It can hold any valid tracer-level for the Tracer module, such as 'debug', 'info', and so on. This is used when a user has for example 20 devices, but wants only 1 of them to output tracer.debug lines, and the other 19 to output tracer.info lines.*

class XBeeBase

The XBeeBase now creates and manages two variables which can be used by derived classes:

- **self._xbee_manager**, which is set by the code:

```
    # Fetch the XBee Manager name from the Settings Manager:
    dm = self._core.get_service("device_driver_manager")
    self._xbee_manager = dm.instance_get(
            SettingsBase.get_setting(self, "xbee_device_manager"))
```

- **self._extended_address**, which is set by the code:

```
                # Get the extended address of the device:
self._extended_address = SettingsBase.get_setting(self, "extended_
address")
```

This also means the XBeeBase class manages the settings named "xbee_device_manager" and "extended_address". Derived classes should attempt to create or manage these settings. When porting drivers to the new DIA, you are encouraged to use XBeeBase's copies of these variables and settings.

The XBeeBase class registers a callback named self.running_indication() which derived classes can overload as desired. The default base class routine looks like this:

```
    def running_indication(self):
        """
        Indicate that we have completed config and are running
        """
        self._tracer.info("Configuration is Complete. Running indication.")
        return
```

Determine MTU

How to determine the maximum payload size (MTU) on a Mesh network.

Currently none of the Digi mesh systems automatically support the fragmentation or reassembly of packets larger than the Maximum Transmission Unit or MTU. Thus, the actual maximum packet size is an important fact for your program to know. Unfortunately it varies based upon underlying protocols and might be 72, 75, 84, 100 or even more than 200 bytes. Plus you need to take into account any 'options' enabled which consume part of this payload space, such as source routing or encryption headers as these reduce the data payload remaining for actual data transfer.

AT mode

In AT mode this is hidden from you - if you stream 200 bytes into an XBee serial port, it will automatically break this into as many packets as required. For example, if the MTU is 66, then it will most likely be sent as 3 packets of 66 bytes and a 4th packet of 2 bytes. If the MTU was 84 bytes, then it will most likely be sent as 2 packets of 84 bytes and a 3rd packet of 32 bytes.

API mode

In API mode you must directly honor the MTU - if you send an API frame with 84 data bytes into an XBee module set up for ZB-2007 and encryption, you will receive an error response that the frame is too long.

On newer Digi firmware there is a AT command "NP" which returns the maximum number bytes of RF payload per packet. Unfortunately, not all firmware supports this command, plus enabling some stack options causes the "NP" response to be invalid.

An indirect solution

If you have a Python program talking to an XBee coordinator set to API mode, and at the remote end you have serial devices using AT mode to return responses, then there is an indirect method to determine the MTU. As example, consider sending Modbus/RTU requests to a remote device which returns Modbus/RTU responses. Reading 100 registers of data sends an 8-byte request and receives a 205-byte response. The indirect method relies upon the fact that the remote XBee in AT mode will break the 205 byte response up into several packets, many set to the actual MTU in effect.

To use this method, the sender (your Python program) starts with a safe MTU limit for sending which is lower than all known MTU - 50 might be a good starting point. Thus to send a 100 byte Modbus request, your code would need to break it into two 50-byte packets to be submitted to the Zigbee UDP-like SendTo() socket call. The trick for this indirect method is to monitor the size of all responses in the Zigbee UDP-like RecvFrom() socket call. In all normal cases, the maximum size of any one packet received from the RecvFrom() will match the MTU for packets sent by SendTo. So this indirect method is a smart learning process over time; for example, the first 8 responses might have sizes of 8, 25, 66, 10, 84, 84, 32 and 52 bytes. In this situation, it is generally safe to assume the largest value seen - 84 bytes - is the current MTU given the firmware, protocol and options in use at this time.

Note You MUST provide a manual override to disable this auto-adjust of MTU - for example, the use of 'source routing' is known to cause RX/responses to be a different size than TX/transmissions. This override could be a fixed reduction - for example, have the TX MTU always set to 10 bytes less than the maximum bytes received. Alternatively this override could be a fixed TX MTU - for example, force the TX MTU to be 66 bytes and disable any auto-adjust.

ZigBee PRO/2007 fragmentation support

Xbee which are running in API mode and have firmware 2x6x (such as 0x2164 in your coordinator and 0x2364 in your remotes) can send up to 255 bytes within a single API frame using the ZigBee protocol 'fragmentation' support (see manual 9000976_D or never for the ZB XBee).

XBee running in API mode can send or receive fragmented messages. **XBee running in AT-transparent mode can ONLY receive fragmented packets**, but will always break outgoing packets into single-RF chunks.

So a Digi ConnectPort X4 gateway with an XBee coordinator running 0x2164 firmware can send Modbus/RTU serial requests to remove XBee 232/485 Adapters running with firmware 0x2264 (AT Router) firmware. The remote serial adapters will gracefully send out the Modbus/RTU request without time-gaps which might confuse Modbus slaves. However, any Modbus/RTU response larger than the XBee max (72, 75 or 84 bytes) will be sent as multiple non-fragmented packets. Fortunately the IA/Modbus bridge code within the X4 can reassemble these as required.

Differences between API frame 0x10 and 0x11

Sending API frames

If your serial device sends a payload using API frame 0x10 (**Transmit Request**), then these defaults are assumed:

- The endpoints are 0xE8 and 0xE8
- The cluster id is 0x0011
- The profile id is 0xC105

Sending the API frame 0x11 (**Explicit Addressing Command Frame**) allows your device to override these defaults. If you do not wish to override them, then there is *NO* advantage to using API frame 0x11 over 0x10. The same information moves over the RF channel regardless.

The Digi gateways always sends API frames as 0x11.

Receiving API frames

Although the API documentation might lead you to believe your device receives API frames 0x90 when the peer uses 0x10, and 0x91 when the peer uses 0x11, this is not true. What your device receives is defined by the AT Command "AO" (API Output).

- Setting AO=0 means your device receives API frames 0x90 (**Receive Packet**) regardless of whether the peer sent the frame to its local Xbee via API frame 0x10 or 0x11.
- Setting AO=1 means your device receives API frames 0x91 (**Explicit Rx Indicator**) regardless of whether the peer sent the frame to its local Xbee via API frame 0x10 or 0x11.

The Digi gateways always receives API frames as 0x91, and will see the default endpoints, cluster and profile id if the remote node sent the data via the 0x10 API frame.

Error messages

Common error messages

Many error messages are misleading, as the messages are from secondary causes. Document such messages and the results here:

Bad local file header

```
#> py dia.py
Determining platform type...Digi Python environment found.
Traceback (most recent call last):
  File "<string>", line 161, in ?
  File "<string>", line 82, in main
zipimport.ZipImportError: bad local file header in WEB/Python/dia.zip
```

The most likely cause is that the zip file has already been opened, and you are trying to open a 'new version' a second time. You need to reboot after uploading a new copy of the ZIP file.

For example, if **dia.py is already running**, then you'll always get this error. Check the 'connections' or 'who' list. Perhaps you left it running before, or have it set to auto-run.

A secondary cause might be a bad ZIP image - HTTP/web upload is not totally reliable. At times large files timeout/abort during upload and you may end up with only 320K of a 350K file. So confirm the file size of the ZIP is as expected, and/or upload dia.zip again.

Error workaround

As a workaround for this error without needing to reboot, one can clear zipimport's cached file headers. As part of one's startup script (before zip files are loaded onto sys.path call the following):

```
def clear_zipimport_cache():
    """Clear out cached entries from _zip_directory_cache"""
    import sys, zipimport
    syspath_backup = list(sys.path)
    zipimport._zip_diretory_cache.clear()
    # load back items onto sys.path
    sys.path = syspath_backup
```

Exception while uploading

Literally, the message each time DIA tried to upload data was the following:

```
iDigi_DB(idigi_db1): exception while uploading: (-6, 'The name does not resolve
for the supplied parameters. Neither nodename  nor servname were supplied. At
least one of these must be supplied.')
```

The solution (or problem) was that the DNS IP addresses within the CPX4 were NOT set properly, thus the **remote management server name (sd1-na.idigi.com)** could not be resolved, thus Device Cloud was NOT connected. Other symptoms:

- Device Cloud (or connectware manager) listed the device as disconnected.
- The web UI connections page did not list the "connectware tcp" entry

Socket.error

Socket is already open

```
Traceback <most recent call last>:
File "<string>", line 23, in ?
File "<string>", line 1, in bind
socket.error: <22, 'invalid argument'>
```

Although this error could mean a badly formed IP or port value, it also can mean that some other Python script is running and holding that specific IP+port combination open. Navigate to your ConnectPort's web UI, click on Applications->Python->Auto-start Settings, uncheck whichever script may be set to autorun, click Apply and reboot your gateway.

Syntax error (in YML file)

A very common cause is the use of tab characters. YML requires spaces only, so either use a text editor which replaces tabs with spaces, or manually make sure only spaces are use.

ValueError: failed to parse request (in RCI call)

One cause for this is 'fancy quotes' - if you cut and paste the request from a PDF or other web document, at times the quotations used are the fancy 'angled' quotes. Thus the string <rci_request version="1.1"> is really seen as <rci_request version=ö1.1ö>, which causes the RCI call to fail.

List index out of range error

A node which was part of the "index" is no longer available

```
#> Python EmbeddedKitService.py
Starting up...
Ready for incoming requests!
Discovering nodes...
Exception in thread WPANSerialEndpoint:
Traceback <most recent call last>:
  File "WEB/Python/Python.zip/threading.py", line 442, in __bootstrap
  File "WEB/Python/EmbeddedKitManager.py", line 199, in run
    nodes = self.get_bindings_hash_list<>
  File "WEB/Python/EmbeddedKitManager.py", line 496, in get_bindings_hash_list
    hash_list = [self._bindings.bindings[k].to_hash<> \
  File "WEB/Python/EmbeddedKitManager.py", line 38, in to_hash
    return {
IndexError: list index out of range
-
```

This error typically means that a node which was once associated with the parent (CP-X?) where the Python script is running is no longer available. Common causes: node has been configured for sleep parameters and is now asleep, node is powered off, node is in an unknown state. Power off parent and all associated nodes to clear out routing and neighbor tables, then power them back on and re-try the script when the nodes are available.

Hardcoding a fixed XBee PAN ID

Many customers wish to preload an XBee PAN ID to simplify installations where there may be multiple XBee networks in view. While XCTU can manually set a fixed PAN ID into an XBee, this value is lost if the XBee adapter is reset to 'factory', is reflashed to a new firmware, or the commissioning button is pressed four times. Once lost, the XBee may associate to the wrong gateway without manual intervention.

There is a simple way to force in a persistent PAN ID - this is especially useful for the XBee AIO and DIO adapters since XCTU cannot talk serially to them.

Save a profile for an "AT Router" firmware with your desired PAN ID added. The result is a text file - here is an example which sets the PAN Id to 9999

```
XB24-ZB_2270.mxi
80
0
251
2270
0
[A]ID=9999
```

The trick is to edit the file, changing the [A] to [D], so the .PRO file becomes:

```
XB24-ZB_2270.mxi
80
0
251
2270
0
D]ID=9999
```

Once you load this profile and reprogram the XBee to AT Router firmware, then the PAN ID will always be 9999 (or as you specified). This survives a reflash to other firmwares such as the XBee AIO firmware.

How to detect radio series in Python

How to detect the radio series in your Gateway product.

This sample is designed to be used in code that may have to run on multiple radio platforms (802.15.4, ZNet 2.5, ZB, etc.) that will make conditional statements based on the type of radio module present in the Gateway.

Code:

```
import zigbee
import struct

try:
  hw_version = zigbee.ddo_get_param(None, "HV")
  hw_version = struct.unpack("=H", hw_version)[0]
except Exception, e:
  print "Failed to retrieve hardware version from local radio"

if hw_version != None:
  if hw_version > 6400: #If the hardware version is greater then 6400, it must be
a series 2 radio
    print "Detected Series 2 radio in gateway"
  else:
    print "Detected Series 1 radio in gateway"
```

The key to this example is retrieving the Hardware version via the AT command 'HV' which is a unsigned 16 bit integer, and performing a comparison operator to decimal 6400.

Series 2 radio modules' hardware version are greater than 6400 for both PRO and regular versions.

Series 1 radio modules' hardware version are 6400 and less for both PRO and regular versions.

Note that the struct.unpack(...) call returns a tuple object, regardless of how many elements it unpacks from the input.

Availability:

This example requires the User to have a Gateway product with a mesh radio of some kind installed. On the gateway they must have the two libraries uploaded: zigbee and struct.

Joining Under Xbee ZigBee

Joining and Association under ZigBee

The Digi Xbee manuals include very detailed information of the ZigBee joining process, however it may contain too much jargon for most novices to understand. This page provides a slightly over-simplified discussion of the process.

Xbee Defaults

Gateway / Coordinator

- The coordinator does a simple scan of permitted channels (as defined in the SC setting), starting at lowest channel enabled, stopping at the first channel which appears usable. It does NOT care if another Digi gateway (or ZigBee mesh) is already on that channel. If in the future, they channel becomes unusable, the coordinator will not move without user manual intervention. For more information on the SC setting and ZigBee channels, see Page: Channels, Zigbee.
- If II setting is default of 0xFFFF, the coordinator picks a random IEEE 802.15.4 16-bit 'Operating PAN ID' (shown as OI in Device Status). If you do any sniffing of the wireless, this is the PAN ID you will see (not the ZigBee extended PAn ID)
- If ID setting is default of 0, the coordinator also picks a random 64-bit 'Extended Pan ID' (shown as OP in Device Status). ZigBee adds this because 802.15.4's 16-bit OI was not considered unique enough considering there are potentially billions of ZigBee Smart-Energy meshes in a future world.
- Once the 'mesh' has been formed, the Digi Xbee Coordinator saves the details in flash (channel, OI, OP), to be reused forever. Normal power cycles and reset will not change the mesh details; review the Xbee user manual to see what does.
- The coordinator then waits for other ZigBee nodes asking to join, comparing a submitted criteria list of: 802.15.4, ZigBee-PRO, and Digi Xbee extensions.
- The coordinator only responds to association requests when the criteria list is a complete match. Therefore, without a setting change, a Digi ZigBee coordinator will not offer association to SmartEnergy or stock/third-party ZigBee nodes.
- Since the coordinator selects a random back off before the offer, if there are multiple coordinator on the same ZigBee channel, you cannot predict which will answer first.

Always-Awake Router

- The router using the SC setting to begin scanning for functional meshes, starting at lowest channel enabled, stopping at the first channel upon which it receives an offer.
- The router request to join is broadcast on the currently monitored channel, including the required criteria list of: 802.15.4, ZigBee-PRO, and Digi Xbee extensions.

- Since II and ID are default of "don't care", those are not used in the criteria list.
- The router accepts the FIRST offer, not the best offer.
- If no offer is received within a configured time-period, the router moves up to the next permitted channel (per SC setting) and tries again.
- Once the router has accepted an association, it saves the details in flash (channel, OI, OP), to be reused forever. Normal power cycles and reset will not change the mesh details; review the Xbee user manual to see what does.
- The associate (Assc) LED now blinks forever. Note that this does NOT mean a router with default settings is currently in a healthy mesh association. It only means that at one time in the past, it joined a mesh.

Sleeping End-Device

An End-Device (whether sleeping or fully awake) follows the Router logic, however since it requires a parent, it more aggressively seeks a new mesh if no parent is found in the old mesh as recorded in flash.

Settings Changes which Effect Joining

- Many systems force a predictable Extended PAN ID (so ID/OP) instead of allowing a random one to be selected. This is commonly the last 6 digits of the Ethernet MAC Address, which makes it predictable from the Device Cloud and during troubleshooting. Assuming this fixed PAN ID is also forced into routers or end-devices, it forces them to remain loyal to this mesh should they seek a new mesh to join. For example, take a food court scenario, where multiple Digi gateway may exist within competing businesses. One would not want a grill in the Chinese food stall to join the gateway in the taco stall because the gateway in the Chinese stall is powered off for a few days.
- Setting JV=1, NW=15, and IR=60000 in a router causes the router to attempt to relocate the gateway (on other channels for example) if the router has not been able to talk to the gateway for 15 minutes.

Large ZigBee Networks and Source Routing

Introduction

ZigBee Pro utilizes Ad hoc On-Demand Distance Vector ("AODV") for routing across the mesh network. In AODV, the network is silent until a connection is needed. At that point the network node that needs a connection broadcasts a request for connection. Other AODV nodes forward this message, and record the node that they heard it from, creating an explosion of temporary routes back to the needy node. When a node receives such a message and already has a route to the desired node, it sends a message backwards through a temporary route to the requesting node. The needy node then begins using the route that has the least number of hops through other nodes. Unused entries in the routing tables are recycled after a time. When a link fails, a routing error is passed back to a transmitting node, and the process repeats.

In applications where a device must transmit data to many remotes, AODV routing would require performing one route discovery for each destination device to establish a route. If there are more destination devices than there are routing table entries, established AODV routes would be overwritten with new routes, causing route discoveries to occur more regularly. This could result in larger packet delays and poor network performance. Utilizing many-to-one routing and source routing helps solve these problems.

It is generally recommended to use many-to-one routing when there are more than 40 nodes in a single PAN.

Many-to-one routing

In networks where many devices must send data to a central collector or gateway device, AODV mesh routing requires significant overhead. If every device in the network had to discover a route before it could send data to the data collector, the network could easily become inundated with broadcast route discovery messages.

Many-to-one routing is an optimization for these kinds of networks. Rather than require each device to do its own route discovery, a single many-to-one broadcast transmission is sent from the data collector to establish reverse routes on all devices.

The many-to-one broadcast is a route request message with the target discovery address set to the address of the data collector. Devices that receive this route request create a reverse many-to-one routing table entry to create a path back to the data collector. The ZigBee stack on a device uses historical link quality information about each neighbor to select a reliable neighbor for the reverse route.

When a device sends data to a data collector, and it finds a many-to-one route in its routing table, it will transmit the data without performing a route discovery. The many-to-one route request should be sent periodically to update and refresh the reverse routes in the network. Applications that require multiple data collectors can also use many-to-one routing. If more than one data collector device sends a many-to-one broadcast, devices will create one reverse routing table entry for each collector.

Source routing

In contrast to many-to-one routing that establishes routing paths from many devices to one data collector, source routing allows the collector to store and specify routes for many remotes. To use source routing on a network, many-to-one routes must first be established on the network from remote nodes to the central data collector.

Enabling many-to-one routing using an XBee-based central collector

Using many-to-one routing with an XBee-based central collector node is easy: the AR command is used to enable many-to-one broadcasting on a device. The AR command sets a time interval (measured in 10 second units) for sending the many to one broadcast transmission. Setting AR to 0xFF disables many-to-one broadcasting on the device. Setting AR to 0 will cause a device to immediately send a single a many-to-one broadcast.

Using Source Routing with XBee Serial API

In order to use source routing with the XBee Serial API, the following steps must be taken:

- To store source routes for remote nodes:
  - Remote nodes must first send a unicast transmission to the central collector
  - Upon receipt of a unicast, the XBee will emit a route record indicator frame (XBee API frame type **0xA1**)
  - The information from the route record frame must be interpreted and stored by your application for later use
- To transmit using a source route:
  - Configure the XBee with the source route using Create Source Route (XBee API frame type **0x21**)
  - Transmit request (XBee API frame types **0x10** or **0x11**)
  - Interpretation of the Transmit Status (XBee API frame **0x8B**)

When source routing is used, the 16-bit addresses in the source route are inserted into the RF payload space. This means the RF payload will be reduced by two bytes per intermediate hop. However, it is up to the user to account for the payload size reduction when using source routing. For example, if NP returns 84 bytes, and a source route must traverse 3 intermediate hops (3 16-bit addresses), the total number of bytes that can be sent in one RF packet is 78.

Using Source Routing with a ConnectPort Gateway

If many-to-one routing is enabled on the mesh network (i.e. the AR parameter is set to less than 0xFF on the gateway), the ConnectPort will automatically capture and automatically utilize source routes when transmitting to remote nodes. Simply address your transmissions as you would normally.

Because the handling of source routing is automatic when using a ConnectPort X gateway it can be difficult to determine what the maximum payload size should be. It is recommended to use the value returned from 'NP' less the maximum source routing overhead (10 hops, or 20 bytes).

Further Reading

- XBee / XBee-PRO ZB RF Modules manual (90000976_U) - S2, S2B ZB
- XBee / XBee-PRO ZB RF Modules manual (90002002_N) - S2C ZB

Module: xbee

Additional ZigBee functionality is provided by the zigbee module included on the Software and Documentation CD media included with your Python-enabled Digi product. Descriptions of supported methods and types follow.

Methods

ddo_get_param()

Purpose

Get a Digi Device Objects parameter value.

Syntax

```
ddo_get_param(addr_extended, id[, timeout, order=False]) . string
```

Description

- Get a DDO parameter id by using the 64-bit address given by *addr_extended*.
- Parameter addr_extended is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.
- Parameter *id* is a 2-byte string such as 'NI'. To make DDO parameter requests of remote radios, all radio module firmware versions must support this capability. For a description of valid *id* values, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).
- Optional parameter **timeout** is maximum time in seconds to wait for a response.
- Optional parameter **order** is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Return

To properly interpret the binary string returned from this function, please see the API manual for the radio module. It may be useful to use the i module to construct the type into a more useful data type.

An exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a try:/except: statement.

This call may block for many seconds if the remote node is sleeping or offline - therefore avoid application designs which repeatedly query nodes which might be offline. For example do not blindly attempt to read ten DDO parameters from the same node in a row; if the first ddo_get_param() call timeouts, then so should the remaining nine. Instead, when the first DDO call fails, record the node as offline and return at a future time to try reading the ten DDO parameters again.

ddo_set_param()

Purpose

Set a Digi Device Objects parameter value.

Syntax

```
ddo_set_param(addr_extended, id[, value, timeout, order=False, apply=True]) .
boolean
```

Description

Set a DDO parameter id by using the 64-bit address given by *addr_extended* and the given value value.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.

Parameter *id* is a 2-byte string such as 'NI'. To make DDO parameter requests of remote radios, all radio module firmware versions must support this capability. For a description of valid values for *id*, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).

Parameter *value* must be either a string or an integer. Do not submit any value when the parameter does NOT require a value - as for example as with the 'FR' command to reboot the Xbee device.

Optional parameter *timeout* is maximum time in seconds to wait for a response.

Optional parameter *order* is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Optional parameter *apply* is **True** to apply changes to node settings immediately. If *apply* is **False**, changes are queued in the node until a command with *apply* set to **True** or the AC command is sent to the node.

Note If *addr_extended* is a broadcast address (such as "00:00:00:00:00:00:FF:FF!") which will have no response, then you must set *timeout*=0 or ddo_set_param() will throw an exception and fail.

Return

A boolean True or False is returned if the remote node accepts or rejects the SET command. Otherwise an exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except** statement. See **ddo_get_param()** for more usage hints.

ddo_command()

Purpose

Execute a Digi Device Objects command.

Syntax

```
ddo_command(addr_extended, id[, param, timeout, order=False, apply=True]) .
string or None
```

Description

Execute a DDO command given by id by using the 64-bit address given by *addr_extended* and the optional parameter *param*.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!". If *addr_extended* is **None**, the request is performed on the local radio.

Parameter *id* is a 2-byte string such as 'NI'. To send DDO commands to remote radios, all radio module firmware versions must support this capability. For a description of valid values for *id*, see the XBee™ Series 2 OEM RF Modules Product Manual (part number 90000866_B).

Parameter *param* must be either a string or an integer. Do not submit any value when the command does NOT require a value - as for example as with the 'FR' command to reboot the Xbee device.

Optional parameter *timeout* is maximum time in seconds to wait for a response.

Optional parameter *order* is **True** to send this command in the same order relative to other commands and data transmissions. Concurrent commands to multiple nodes may be processed out of order. This option forces all previous commands to be sent to the local radio before this one, and all later commands to be sent after this one. Use of this option may significantly delay processing of commands.

Optional parameter *apply* is **True** to apply changes to node settings immediately. If *apply* is **False**, changes are queued in the node until a command with apply set to True or the AC command is sent to the node.

Return

A string is returned if the command produces a result. To properly interpret the binary string returned from this function, please see the API manual for the radio module. It may be useful to use the *struct* module to construct the type into a more useful data type.

An exception is thrown if the addressed node does not respond or is sleeping, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except statement**. See **ddo_get_param()** for more usage hints.

get_node_list()

Purpose

Perform a node discovery.

Syntax

```
get_node_list([refresh=True, clear=refresh, discover_digi=False, discover_
zigbee=False]) . (node, node, …, node)
```

Description

Perform a node discovery and return a tuple of nodes.

If the *refresh* parameter is set to **True**, this function will block and a fresh node discovery is performed. If no discovery methods are selected, a method appropriate for the network type will be used.

If *refresh* is set to **False**, this function returns a cached copy of the node discovery list. This cached version may include devices that were unable to be discovered within the discovery timeout imposed during a blocking call. If discovery methods are selected, newly discovered nodes will be added to the cached list.

If the *clear* parameter is set to **True**, this function will clear the cached list and perform a network discovery. If the *clear* parameter is set to **False**, this function will add newly discovered nodes to the

existing cached list. If the clear parameter not specified, this function will clear the cached list if a network discovery is being performed.

If the *discover_digi* parameter is set to **True** this function will block and network discovery of Digi nodes will be performed. This obtains extended information supported by Digi nodes.

If the *discover_zigbee* parameter is set to **True** this function will block and network discovery of ZigBee nodes will be performed. This obtains information from standard ZigBee nodes.

register_joining_device()

Purpose

Register a new node into the local trust center key table.

Syntax

```
register_joining_device(addr_extended, key)
```

Description

This method is available on a gateway running a Smart Energy profile trust center.

Register a new node with the 64-bit address given by *addr_extended*, and set its initial trust center link key to *key*.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!".

Parameter *key* is a binary string of up to 16 bytes. If key is less than 16 bytes, the upper bytes are padded with 0.

Return

An exception is thrown if a registration error occurs, or if the parameters are malformed. Therefore you must wrap any calls with a **try-except** statement.

unregister_joining_device()

Purpose

Unregister a node from the local trust center key table.

Syntax

```
unregister_joining_device(addr_extended)
```

Description

This method is available on a gateway running a Smart Energy profile trust center.

Remove the node with the 64-bit address given by *addr_extended* and its key from the local trust center key table.

Parameter *addr_extended* is a string formatted like "[00:13:a2:00:40:0a:07:8d]!".

Return

An exception is thrown if the given node is not registered, an error occurs, or if the parameter is malformed. Therefore you must wrap any calls with a **try-except** statement.

Classes

node

Name
node – a Python object returned from a node discovery

Attributes

Name	Type	Description	Example
type	string	node role/type in mesh	is in ['coordinator', 'router', 'end']
addr_ extended	string	64-bit extended hardware address	" [00:13:a2:00:40:0a:07:8d]!"
addr_short	string	16-bit network assigned address	"[49c3]!"
addr_parent	string	16-bit network parent address	"[fffe]!"
source_route	tuple	tuple of 16-bit network addresses in the source route from the node.First element is a neighbor of the node. Last element is a neighbor of the gateway node.Empty if the node is one hop away, or no source route has been received from the node.Present in gateway firmware version 2.15 and later.	("[1234]!", "[5678]!")
profile_id	int	node profile ID	0xC105 or 49413
manufacturer_ id	int	node manufacturer ID	0x101E or 4126
label	string	node’s string label (Setting 'NI')	"TK103U"
device_type	int	node’s device type (Setting 'DD').Upper 16 bits contain the module type. Lower 16 bits contain the product type.Will be 0 if the node does not support 'DD'	0x00030001

Methods

to_socket_addr()

Purpose
Transform a node into a socket address tuple

Syntax

```
to_socket_addr([endpoint,] [profile_id,] [cluster_id,] [use_short]) .
    (“address!”, endpoint, profile_id, cluster_id)
```

Desc ription

Transform this node into a socket address tuple, suitable for use with functions from the socket modules. If use_short is True, the short node address representation is used instead of the 64-bit extended address, which is used by default.

ZigBee Module Examples

Perform a Network Node Discovery

```
            #
# Perform a node discovery and print out# the list of discovered nodes to stdio.
#

# import the zigbee module into its own namespace:
import zigbee

# Perform a node discovery:
node_list = zigbee.getnodelist()

# Print the table:
print "%12s %12s %8s %24s" % \
    ("Label", "Type", "Short", "Extended")
print "%12s %12s %8s %24s" % \
    ("-" * 12, "-" * 12, "-" * 8, "-" * 24)

for node in node_list:
    print "%12s %12s %8s %12s" % \
        (node.label, node.type, \
            node.addr_short, node.addr_extended)
```

Use DDO to Read Temperature from XBee Sensor

```
            #
# Collect a sample from a known XBee Sensor adapter
# and parse it into a temperature.
#


# import zigbee and xbee_sensor modules:
import zigbee
import xbee_sensor


# configure known destination:
DESTINATION="[00:13:a2:00:40:0a:07:8d]!"


# Note: for clarity, the try: except: statements required to handle timeout is
not shown

# ensure sensor is powered from adapter:
zigbee.ddo_set_param(DESTINATION, 'D2', 5)
zigbee.ddo_set_param(DESTINATION, 'AC', '')


# get and parse sample:
```

```
sample = zigbee.ddo_get_param(DESTINATION, '1S')
xbee_temp = xbee_sensor.XBeeWatchportT()
xbee_temp.parse_sample(sample)
print "Temperature is: %f degrees Celsius" % (xbee_temp.temperature)
```

Create your own display-mesh command

This page includes a fully functional application using ddo_get_param() and getnodelist(): Create Your Own Custom Node List on a Digi ConnectPort X2/X4/X8 gateway

Monitoring a ZigBee Network

Introduction

This application demonstrates how ZigBee sockets can be used to sample values from a nodes on a mesh network and monitor the network for new nodes. In this demonstration, existing nodes and nodes that join the network later on are configured to sample their supply voltage every five seconds.

Requirements

- Digi Gateway with Python enabled
- XBee endpoint assosciated with gateway
- CLI access to the gateway

Code Organization

The code is organized into three primary functions:

1. main() -- The main function of the program which listens for data on the network
2. handle_join() -- The function called when a node joins the network
3. handle_sample() -- The function called when a sample is received from a node

Code Listing

Main Function

The main function reads from the created socket for any data on clusters 0x92 and 0x95. Data received on cluster 0x92 is interpereted as a sample, while data received on 0x95 is interpereted as a node joining the network.

```
            # Our main function called at program start
def main(args):
    # First we create our socket to listen
    # for nodes joining the network
    # The socket is configured to listen on the
    # data endpoint (0xE8) and to time out after
    # blocking for 1 second
    sd = socket.socket(socket.AF_ZIGBEE,
                       socket.SOCK_DGRAM,
                       socket.ZBS_PROT_TRANSPORT)
    sd.bind(('', 0xe8, 0, 0))
    sd.settimeout(1)

    # We wrap the rest of the program in a try-except
    # so that we always close the socket descriptor before
    # the program exits
    try:
        # Next, we perform a network discovery
        # to identify nodes already on the network
        nodes = zigbee.getnodelist(refresh=True)
```

```
        # We use filter to remove any coordinators from our list
        nodes = filter(lambda x: x.type != 'coordinator', nodes)

        # We use map to get the extended address of each node
        nodes = map(lambda x: x.addr_extended, nodes)

        # We create a dictionary to hold our sample data
        node_values = dict()

        # And finally we configure each node and
        # add them to the dictionary
        for node in nodes:
            handle_join(node, node_values)

            try:
                configure_node(node)
                node_values[node] = None

            # If we can't configure the node, don't add it to the dictionary
            except:
                pass

        # Now we loop to receive data from the network
        while True:
            try:
                # Update the screen
                print_data(node_values)

                # Receive data from the network
                packet, source = sd.recvfrom(72)

                # Check the source cluster to determine what to do with it
                # 0x92 -- IO Sample data, call handle_sample
                # 0x95 -- A node has joined the network, configure it
                # Others -- ignore
                if (source[3] == 0x92):
                    handle_sample(source[0], node_values, packet)
                elif (source[3] == 0x95):
                    handle_join(source[0], node_values)

        # Handle any thrown exceptions
        except socket.timeout:
            # Ignore the timeout exception
            pass

        # Any other exceptions are left unhandled so that
        # the exception will be printed to the screen when one occurs
    except:
        # Close the socket before we exit
        sd.close()
        raise
```

Join handler function

The join handler function checks to see if we've already seen the connecting node, and if not, adds it to the list of nodes that the program is monitoring. This is done so that the program does not attempt to reconfigure nodes if they've already been configured.

```
            # Method called when a node joins the network
def handle_join(node_addr, node_values):
    # First verify that the node isn't already
    # in our list
    if not node_values.has_key(node_addr):
        try:
            # Now configure the node and add it to the dictionary
            node_values[node_addr] = None
            configure_node(node_addr)

        # If we can't configure the node properly, ignore it.
        except:
            pass
```

Sample handler function

The sample handler function uses the included xis.py module to parse received sample packets for the supply value and then prints the value to the screen.

```
            # Method called when a sample is received from the network
def handle_sample(node_addr, node_values, packet):
    # The XBeeIOSample class is used to parse XBee sample packets
    sample = XBeeIOSample(packet)

    # We use get_analog_pin to extract the supply voltage from the packet
    supply = sample.get_analog_pin('SUPPLY')

    # Save the updated value in the dictionary
    node_values[node_addr] = supply
```

Source code

The complete source of this sample can be found here.

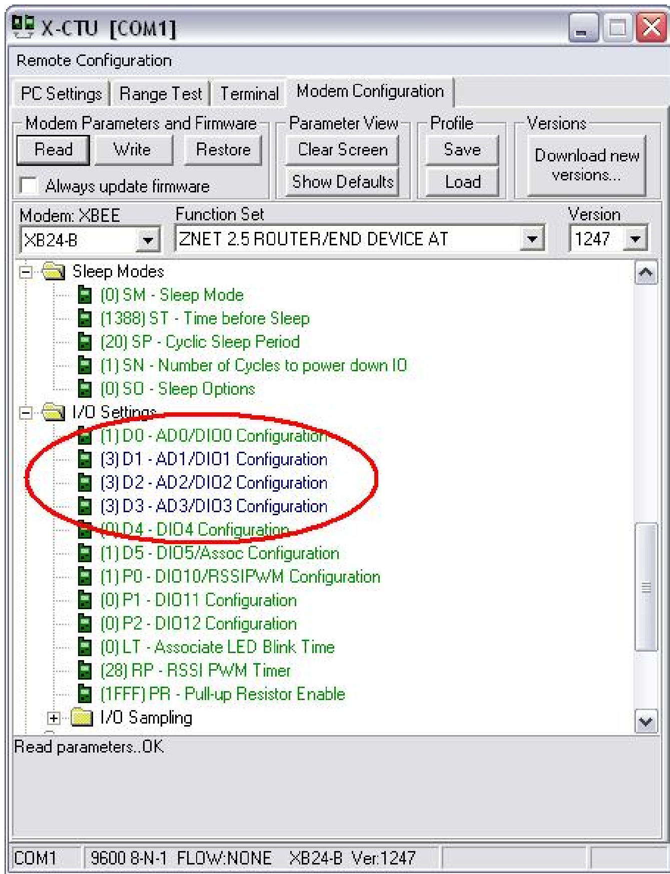
Remote XBee management with XCTU

XCTU allows you to directly edit the setting within any XBee in a mesh/network.

What you need

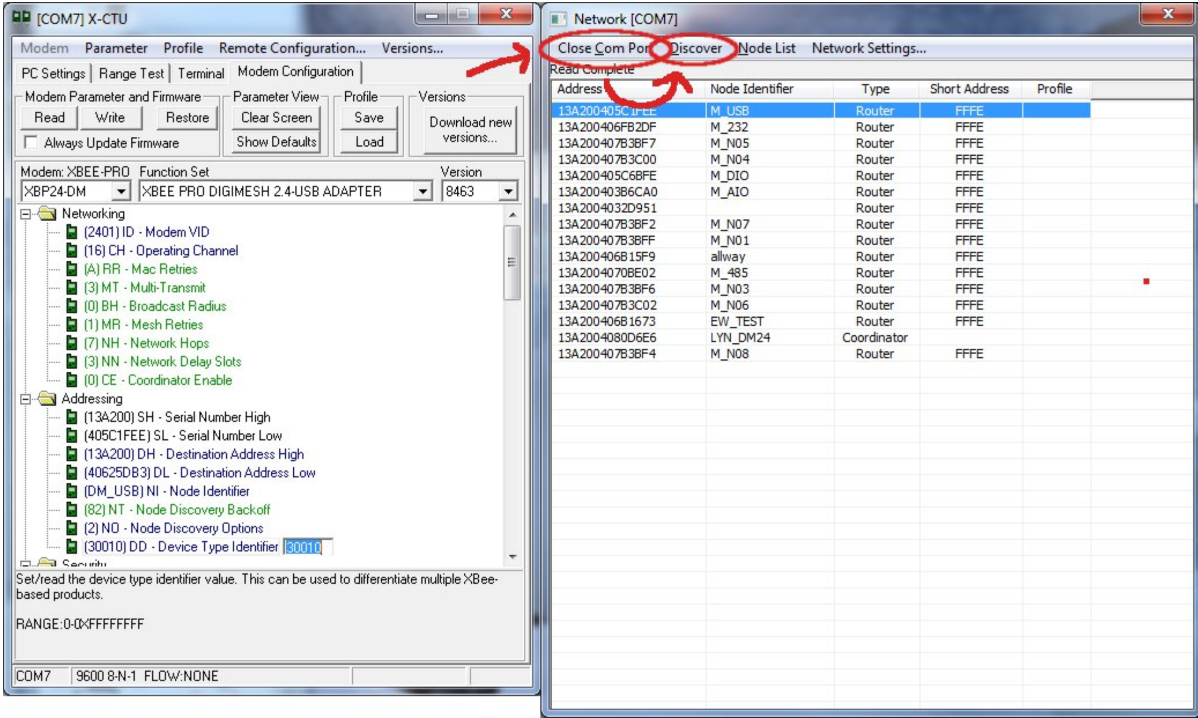
- Download the latest version of XCTU from Digi's support site: X-CTU Software.
- A computer with USB port which can run XCTU. *Win XP or Win 7 are the easiest to use, but XCTU may also be run under Windows on a MAC or under Linux Wine.*
- A USB-based XBee carrier, including one of:
  - XBIB board with suitable XBee
  - XBee USB Dongle Adapter
  - XStick

Activating remote configuration



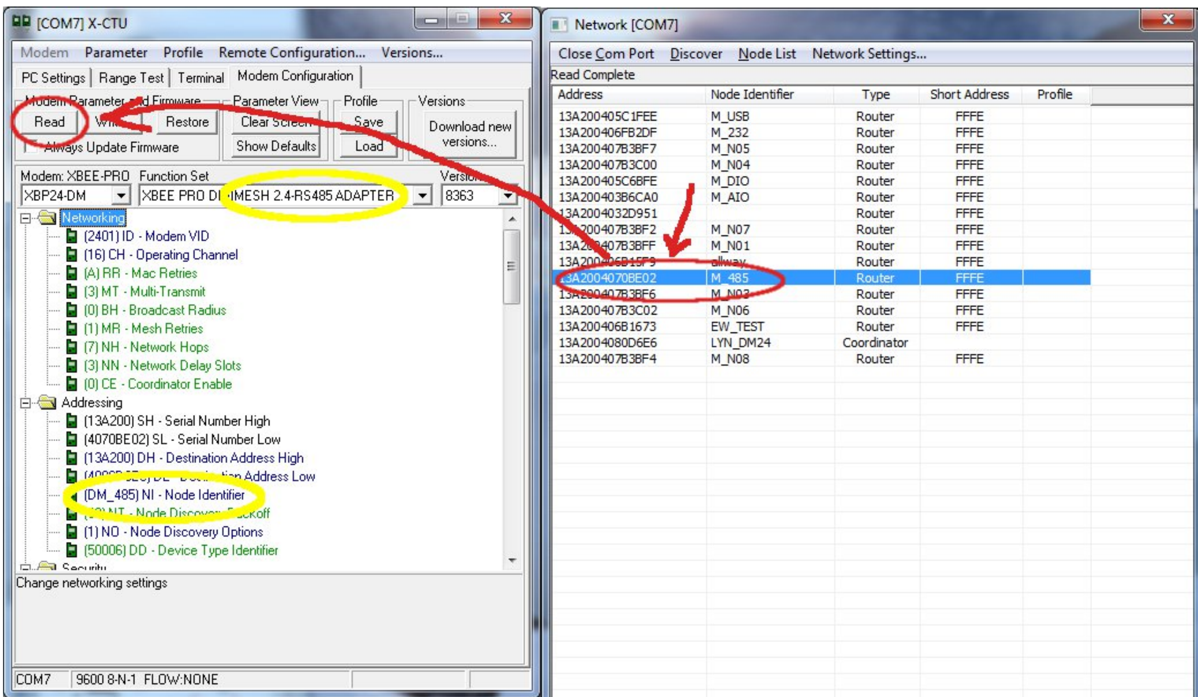
- Run XCTU
- Make sure the firmware on the computer-attached XBee supports API mode.
  - For ZigBee, use API Router with no enabled sleeping
  - For DigiMesh, enable API with the AP=1 and set Sleep-Mode (SM) to 0 or 7, as required
- Set any settings (such as Network/PAN ID, channel, etc) so that the computer-attached XBee joins your target network/mesh.
- From the **Modem Configuration** tab, click the **Remote Configuration** link and a new window will open.

Doing Network Discovery



- The first menu link will say **Open Com Port** - click this, which activates the Remote Configuration and grants it control over the serial/USB port. After clicking, the menu link changes to say **Close Com Port**.
- Click **Discover**, and you will see a list of all awake XBee nodes.
- Trying to edit sleeping node by this method is tricky - they must be awake during discovery, and also awake when you wish to <**Read**> or <**Write**> the Xbee settings.

Configure a Remote XBee



- You now select the XBee node you wish to read or change.
- Click <**Read**> on the main XCTU window, and XCTU will use Remote AT commands to fetch all of that XBee's settings.

Voila - you now can edit the remote XBee as if local. In the screen shot to the left you can see that the XBee being edited is now an RS-485 adapter, not the USB Dongle. Of course if you change critical settings (Network ID or encryption keys), then the XBee node will leave your mesh and you will lose contact with it.

ZigBee firmware can be updated OTA in this mode - DigiMesh firmware cannot be.

Exiting XCTU

It is critical that your first click the **Close Com Port** link, then close the second window before existing XCTU. If not, on occasion the comm port will remained locked and you will need to reboot your computer to recover.

Sending AT commands to the gateway

AT Commands and the Gateway

Many beginners study the XBee documentation, then decide using AT-mode is easier than API-mode.

However, your Python program can use neither AT nor API commands to talk to the XBee module within a Digi Gatewaysuch as the ConnectPort X4/X8. The gateway's XBee is always in API-mode, but it must be shared with the Web interface and other functions which browse the mesh.

AT commands to read/write settings in the gateway's XBee Module

To send AT parameter read/write commands to the local Xbee within the gateway, your python program uses the ddo_get_param() and ddo_set_param() functions with the extended MAC address set to None.

AT commands to read/write settings in remote XBee Modules

To send AT parameter read/write commands to remote Xbee, your python program uses the ddo_get_param() and ddo_set_param() functions with the extended MAC address set as required.

AT commands to forward serial data to remote serial devices

If you goal is sending serial data via the mesh to remote serial devices, then you use the Zigbee Extensions to the Python Socket module. Your Python program opens a UDP-like socket, then uses sendto() and recvfrom() functions to send and receive packets. There are simple examples of this on that web page.

Sending broadcast transmissions

Sending broadcast transmissions from the Gateway across the Mesh network.

This sample contains the means to send broadcast transmissions from the gateway to the Mesh network. It is intended to work for both 802.15.4 and Znet 2.5 radio modules.

*However, note that **sending broadcasts on a mesh such as ZNet, Zigbee or DigiMesh is not scalable**. If your network grows beyond a few nodes, then every broadcast can literally stop all communication for several seconds. Do not design an application treating the mesh as an RS-485 multi-drop - your design will fail to perform and you will need to design in unicast transmissions based on MAC address eventually.*

Example:

```
from socket import *
s = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)
s.bind(("", end_point, profile_id, cluster_id))
s.sendto(data, 0, ("[00:00:00:00:00:00:FF:FF]!", end_point, profile_id, cluster_
id))
```

The key to the above code is the hardware address specified in the sendto(...) command. On both 802.15.4 and Znet2.5 networks, the broadcast address is "[00:00:00:00:00:00:FF:FF]!".

Simple serial app quick index

Quick Index of relevant pages for an Xbee serial product

Overview

A customer has a simple sensor with an integrated XBee module, which is pin-sleep controlled by a PIC serially connected to the XBee. The sensor PIC wakes up periodically, takes a reading, and send out a string. It expects either an ACK, or an ACK plus a new configuration string in response. The vendor wishes the Digi gateway to actively open a TCP socket and push information to a remote server.

For such a customer, this wiki page gives a quick index of relevant pages.

Pages to review

The XBee module

Since the PIC actively configures and manages the XBee, running with the **API firmware** simplifies the design - trying to switch between API and AT mode is not worth the timing complexities. Some newer firmware also split API and AT into two different firmwares, so your PIC cannot switch on the fly.

Even though you pin-sleep the **XBee, the XBee's SN and SP settings must be valid** or the parent node hangs up on the sleeping end-device. For example, to sleep for one hour the setting SP=0x0AF0 and SN=0x0081 would be correct. This means the parent will buffer requests for the sleeping end-device for 28 seconds, and the expected wake-up time of the end-device is 3612 seconds, so about once per hour. If the end-device does NOT contact the parent within 3 hours (3 x SP x SN), then it will drop the sleeping Xbee from the network.

This Digi support page lists the most recent XBee product manuals.

Your sensor sending data

When the sensor wakes, it creates its data as a binary or ASCII string. Binary is best for the mesh as it packs more information into less space, however if the overall message is less than 60 bytes your design is safe on all normal Xbee - even if security and other options enabled. You do NOT need to include the MAC address of your XBee or any other slave information. The remote device receiving your data will know which MAC the data came from.

Send your data as API code 0x10, targeting MAC 00:00:00:00:00:00:00:00, network address 0xFFFE. On Znet and ZB this moves the message to the Digi gateway.

Do NOT use Broadcast. This creates a non-scalable solution which will fail to work correctly in the field. You have been warned. Broadcast and modern wireless do not mix.

Your sensor receiving the data

If you have set the XBee's AO setting set to zero (0), then **your PIC will see any data responses as API code 0x90**. The MAC address included will be the Digi gateway's address (or the source of the message). You should normally ignore this information. Although the XBee manuals indicate up to 72 bytes will be received, this varies by XBee technology. Common sizes are 72, 75, 84, 100, and over 200. Your PIC doesn't need to accept more data then desired, however it should gracefully handle seeing too much by either truncating or rejecting the message. The API frame length is used to calculate the actually bytes received.

Python on the Gateway talking to your sensor

For a simple single-function loop, your code would use the UDP-like **socket()** function. You bind on the AF_ZIGBEE socket, which makes your code owner of the entire mesh. Your code then uses recvfrom() to receive the messages from your sensor, and the actual MAC of the sensor is in the **addr** return value. Optionally, your code sends any response by sendto(), reusing the **addr** to unicast. It is that simple.

If you are using the 802.15.4 firmware, you would socket.bind on endpoint 0x00, not 232/0xE8.

See XBee extensions to the Python socket API for details regarding the use of the Python socket functions.

Be warned that this dedicates the Digi gateway for your hardware, so your customers cannot use other Zigbee products - unless you expand your application to handle those products. Two Python programs cannot share access to the mesh. If you expect your customers to mix vendors equipment on a single mesh, then you should look at the Digi DIA platform which allows configurable drivers to be mixed from different sources. See www.etherios.com/devicecloud.

If you desire your Python application to manage the XBee settings, you can use the get_ddo/set_ddo functions. However, if the Xbee sleeps for hours at a time, these will always fail after 16 to 28 seconds. Since the XBee remains awake for the SP setting each time your PIC wakes it, you can hook the need to read/write XBee setting to the socket recvfrom() above. Quickly **sending set_ddo/set_ddo commands would only succeed when the Xbee is awake**.

See Module: xbee for an explanation of how to use of the get_ddo/set_ddo functions.

Python talking outwards or upstream

This is normal TCP or UDP sockets functions. You can find simple client examples on the public internet.

This page Handling socket error and Keepalive explains common error handling, which many web examples ignore.

Robustness issues

Error handling

Make extensive use of try/except at a low level. Avoid putting one big try/except at the highest level, plus always print something if the except is unexpected. A common beginner mistake is put a few very high level try/excepts which hide even common typos at lower levels.

Watch Dog

You can enable a simple watch dog to hard-reset the gateway. If you sensors sleep for hours at a time, you probably want the reset time to be quite long - 3 hours perhaps (or 10,800 seconds). You do not want the gateway rebooting every 5 minutes - especially if the problem is missing sensors.

See Module: digiwdog for an explanation of use of the watch dog functions.

Memory management

Make sure your application **manually forces garbage collection at least once per day**. While Python garbage collection is 'automatic', the algorithm used can be complex and not optimized for small embedded systems. If your code send out a report once or twice per day, calling gc.collect() after the outgoing client socket closes is ideal.

See Python garbage collection for an explanation of Python garbage Collection.

Local diagnostic web pages

Even if the totality of your application is 1) collect data, 2) forward daily as HTTP or email, adding a few status and diagnostic web pages is a valuable addition. The Digi Web Interface normally consumes port 80, so you do not want to create your own web server. Instead, see Module: digiweb, which allows your code to register a callback with the web ui. This would allow a user (or you) to open a web page like http://192.168.0.10/status and pull up a user-friendly web page. You need to understand how to manually build a raw page, but the effort will be worth it.

See Module: digiweb for an explanation of the Digi Web UI callback to pass unknown URL to your Python program.

TCP to Zigbee dynamic name mapping

Introduction:

This sample is a simplistic design to allow TCP traffic to a Digi Gateway product be routed to the Gateway's Mesh network and vice versa using a naming system defined by the user.

Requirements:

- Digi Gateway product
- XBee Endpoint device associated with the Gateway (Control of device via serial recommended)
- TCP access to the Digi Gateway product
- Standard Python.zip

Overview:

The following guide will describe the application in several steps, with the completed application listed at the end. The goal of this application is to demonstrate how to form a mapping between the 64 bit hardware addresses and the user defined name of the node, the enforcement of the naming scheme for the client connection, and the queuing of data for the respective interfaces.

Code walkthrough

Declarations

```
##################################################################################
##################
# Import statements

##################################################################################
##################

import socket
import select
import struct
import zigbee
import errno
import table   #User defined file


##################################################################################
##################
# declarations

##################################################################################
##################

MAX_TCP_PACKET_SIZE = 8192  #Maximum size of tcp packet we will receive or push
at once
MAX_ZIG_PACKET_SIZE = 100   #Maximum size of zigbee packet we will read or send
```

```
at once

tcp_port = 20000            #TCP Port the application sends and receives on
quit_port = 30000           #TCP Port if connected to throws a keyboard exception

end_point  = 0x00           #address information that we will bind and send to
profile_id = 0x0000
cluster_id = 0x00

zig_addr_name_dict = table.table  #The dictionary of zigbee 64 bit hardware
address mapped to names
#This could be generated or typed in by the user

zig_queue = []                    #queue of data we send out the zigbee socket
tcp_queue = []                    #queue of data we send out the TCP socket
```

The first section *Import statements* declares the libraries we intend to use. *socket*, *select*, *struct*, *zigbee*, and *errno* are all libraries you should have encountered before. The *table* library is a user defined library intended to be used only in this application. It's purpose is to define the 64 bit hardware address to node name mapping scheme.

The second section 'declarations *declares several constants* that are used *throughout the application*.

MAX_TCP_PACKET_SIZE

Describes the size of the maximum read and write we will perform on the TCP client socket.

MAX_ZIG_PACKET_SIZE

Describes the size of the maximum read and write we will perform on the Zigbee socket.

tcp_port

Defines which port the client socket will be expected upon.

quit_port

Defines which port the application will terminate on if we see a connection. Note: this is not necessary but helpful for sample purposes.

end_point, profile_id, cluster_id

Declare the address information we will use to bind and send when interacting with the mesh network.

zig_addr_name_dict

The dictionary object that provides the 64 bit hardware address and node name look up. This information is retrieved from the user defined table module.

zig_queue

The list object that acts as a queue for all information that will be sent out the zigbee socket.

tcp_queue

The list object that acts as a queue for all information that will be sent out the TCP client socket.

User defined procedures

```
################################################################################
##################
# cleanUp - removes the client socket from the read/write lists, closes and sets
to None

################################################################################
##################
```

```
def cleanUp(client_sock):
  try:
    read_list.remove(client_sock)
    write_list.remove(client_sock)
    client_sock.close()
    client_sock = None
  except Exception, e:
    print e
```

This code defines a procedure that inputs a socket, removes the socket from the read_list and write_list, closes it and sets it to value None. This is necessary for our application because we must be capable of having the client TCP connection close unexpectedly without exiting our application.

Initialization

```
#################################################################################
##################
# Init the dictionary, declare the sockets

#################################################################################
##################

# We provide reversal lookup to this dictionary. So we can go from the name -> 64
bit addr or
# 64 bit addr -> name.

for item in zig_addr_name_dict.keys():
  zig_addr_name_dict[zig_addr_name_dict[item]] = item
  #Reversal lookup now available!

# Declare the sockets
listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_sock.bind(("", tcp_port))
listen_sock.listen(1)

zig_sock = socket.socket(socket.AF_ZIGBEE, socket.SOCK_DGRAM, socket.ZBS_PROT_
TRANSPORT)
zig_sock.bind(('', end_point, profile_id, cluster_id))
zig_sock.setblocking(0)

quit_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
quit_sock.bind(("", quit_port))
quit_sock.listen(1)

client_sock = None

read_list = [listen_sock, zig_sock, quit_sock]
write_list = [zig_sock]
```

Here we provide a reversal lookup between the address keys and name values in the zig_addr_name_dict. For example previously we could retrieve the name via:

```
name = zig_addr_name_dict[addr]
```

Now we can retrieve the address via:

```
addr = zig_addr_name_dict[name]
```

This works provided that the address and name values cannot be the same. This is highly unlikely unless a user defines the name of a node to be the 64 bit address of an existing node.

Also defined here are the applications 3 static sockets. The listen socket which is the one which clients will connect to, the zigbee socket which we will communicate with the Mesh network with and the quit socket, which we use to terminate the application by request.

We also define the client socket as **None**, though we intend to make it a socket object later.

For use in the upcoming select call, we create a list consisting of all the sockets we expect to receive from, and a list of sockets we expect to write to.

Main Loop

```
##################################################################################
##################
# Main loop

##################################################################################
##################

print "Entering main loop"

while 1:

  rl, wl, el = select.select(read_list, write_list, [])
```

We are now entering the main loop of the application, where we will stay until termination.

We now perform a select call on the sockets we listed in the last step. We now have to define the behavior of each of those sockets if they appear in the appropriate list. The next few steps will cover that topic.

Zigbee socket in the read list

```

##################################################################################
##################
    # Zigbee present in read list, we have new data

##################################################################################
##################

    if zig_sock in rl:
      try:
        data, addr = zig_sock.recvfrom(MAX_ZIG_PACKET_SIZE)
        print "Read %d bytes from address: %s" %(len(data), addr)
      except Exception, e:
        print e
      else:
        try:
          name = zig_addr_name_dict[addr[0]] ## Get the name from the dictionary
        except KeyError, e:                  ## If that address doesn't have a
name
          print e                            ## print and go no further here
        else:
          tcp_queue.append("%s:%s" %(name, data)) ## Append the 'name:data'
format to the queue
```

If the Zigbee socket is in the read list, it means we have incoming date from the mesh network. We retrieve the data and address of that packet and perform a name lookup on the address. If we can perform that lookup successfully, we append the data to the TCP queue in the 'name:data' format.

Note that the address information is a tuple object structured like (Hardware_address, end_point, profile_id, cluster_id). We only need the hardware address portion to perform the lookup. While performing the name lookup, we can potentially receive a KeyError exception. This is because we could receive data from the mesh network that we don't have a 64 bit hardware address specified for.

Zigbee socket in the write list

```
#################################################################################
##################
    # Zigbee present in write list AND we have data to write

#################################################################################
##################

    if (zig_sock in wl) and (len(zig_queue) != 0):
      name, data = zig_queue[0].split(":", 1)    ## Retrieve the 'name:data'
datoms
      name = name.strip()                        ## Strip excess unprintable
characters
      try:
        addr = zig_addr_name_dict[name]          ## Get the addr dervied from the
name
      except KeyError, e:
        #An node with an undefined address-&gt;Name mapping has contacted us
        print e

      else:
        if len(data) &gt; MAX_ZIG_PACKET_SIZE:
          segment = len(data)
        else:
          segment = MAX_ZIG_PACKET_SIZE

        try:
          sent_data = zig_sock.sendto(data[:segment], 0, (addr, end_point,
profile_id, cluster_id))
          data = data[sent_data:]
          if len(data) == 0:                     ## If all data has been sent,
pop it
            zig_queue.pop(0)
          else:                                  ## Otherwise store the remaining
data
            zig_queue[0] = "%s:%s" %(name, data)
        except Exception, e:
          print e
```

If the zigbee socket is present in the write list and we have data to send. The latter part of the condition is critical to avoid errant sends. All the data in the zig_queue is structured in the 'name:data' format. This allows us to make a few assumptions with our code. We copy the name and data from the first element of the queue, making sure to split only on the first ':' to avoid splitting the data incorrectly.

We use the name to retrieve the hardware address of the node we will send to. We then send the maximum amount of data we can and use the return value of the sendto(...) function to determine

how much was actually sent. If we determine that all the data was sent, we pop the first element out of the queue, otherwise we return the remaining data to the queue in the 'name:data' format.

Note that we never send the name portion of the data, yet we must keep it in case we do not completely send, so when the next time the select call returns the zigbee socket in the write list, we can perform the same operation on it without making adjustments that we have processed it once before. In short, we want to keep the data in a single state until we have finished it.

Listen socket in read list.

```
#################################################################################
##################
    # listening socket in read list, means we have a client!

#################################################################################
##################

    if listen_sock in rl:
      client_sock, addr = listen_sock.accept()
      client_sock.settimeout(0)                 ## disable blocking
      tcp_queue = []                            ## Remove all current data in
queues
      zig_queue = []
      read_list.append(client_sock)             ## put the socket into a list so
the select will cover
      write_list.append(client_sock)            ## it when we next get back
there.
```

The listen socket we defined is now in the read list. We accept the socket, set the new socket to non-blocking IO, clear our data queues to avoid old data, and append the client socket to the read and write list the select call uses.

The cleanUp function defined earlier will only be called on the client socket. It's intended to be used when the client connection errors out or disconnects.

Client Socket in the read list

```
#################################################################################
##################
    # client socket in read list, we have data

#################################################################################
##################

    if client_sock in rl:

      try:
        data = client_sock.recv(MAX_TCP_PACKET_SIZE)

    except socket.error, e:           #We have a socket exception
      if (e.args[0] == errno.EAGAIN): #If it's a blocking related exception
        pass                          #come back again next select call
      else:                           #If it's NOT a block related exception
        print e                       #Clean it up
        cleanUp(client_sock)
      except Exception, e:
        print e
```

```
        cleanUp(client_sock)
      else:
        print "Read %s bytes from client" %len(data)
        if len(data) == 0:              ## If 0 bytes read, we clean up the
connection
          cleanUp(client_sock)

        pack =  data.split(":", 1)      ## split it by the first ':'
        if len(pack) != 2:              ## If the split item doesn't have 2 parts
          tcp_queue.append("ERROR: Invalid format, must follow 'NAME:DATA'
format")
        else:                           ## Send back a message saying they didn't
follow the format
          name = pack[0].strip()
          tail = pack[1]

          ## If we don't have an address associated with the key
          ## Send back a mesage saying it's unknown

          ## or if the item does not have anything after the ':'
          ## send back a message saying you sent no data

          ## if passing the above, queue up the message

          if zig_addr_name_dict.has_key(name) == False:
            tcp_queue.append("ERROR: %s is an unknown name" %name)
          elif len(tail) == 0:
            tcp_queue.append("ERROR: Cannot send messages of 0 bytes in length")
          else:
            zig_queue.append(data)
```

We now process any incoming data from the TCP client socket. In this situation it means we have incoming data. While receiving the data, we watch for the EAGAIN socket exception. This is an indicator that this operation would block, and we'll come back next select call. All other exceptions on the socket will cause us to disconnect the client_socket by call ing the cleanUp(...) function on it. We will also call the cleanUp(...) function if we receive 0 bytes in data.

Next we split the incoming data once by ':' to determine if it's in the proper format. If that is not the case, we queue up a response to the client, stating the error and the correct format to use.

We then check to see if the 'name' portion of the data maps to a hardware address we know about. If it doesn't we queue up a response to the client, stating the error.

If the data passes all the criteria, we can then append it to the zigbee queue.

Client socket in the write list

```
################################################################################
##################
    # client socket in write list AND we have data to write

################################################################################
##################

    if (client_sock in wl) and (len(tcp_queue) != 0):
      name = tcp_queue[0].split(":", 1)[0]  ## Retrieve the name from the item
      data = tcp_queue[0]                   ## Make a copy of the complete item
```

```
      if len(data) &gt; MAX_TCP_PACKET_SIZE:
        segment = len(data)
      else:
        segment = MAX_TCP_PACKET_SIZE

      try:
        sent_data = client_sock.send(data[:segment])
      except socket.error, e:           #We have a socket exception
        if (e.args[0] == errno.EAGAIN): #If it's a blocking related exception
          pass                          #come back again next select call
        else:                           #If it's NOT a block related exception
          print e                       #Clean it up
          cleanUp(client_sock)
      except Exception, e:
        print e
        cleanUp(client_sock)
      else:
        data = data[sent_data:]         ## Make the copy shorter by the amount we
sent

        if len(data) == 0:              ## If that's all the data, pop the item
from the queue
          tcp_queue.pop(0)
        elif sent_data &lt; len(name) + 1: ## If we only sent enough to cover the
'name:' portion
          pass                          ## Don't store the changes, we didn't do
anything
        else:
          tcp_queue[0] = "%s:%s" %(name, data)  ## Otherwise store the remaining
chunk.
```

If the client socket is in a writable state and we have data to write, we attempt to send the data. Note that when we send the data we keep the 'name' portion of the data. This is to help identify who the response is coming from, as it is possible that the client code have many outstanding requests at once. With that said, this may not be the response they are looking for with that name.

While performing the send, if we have the EAGAIN exception, we come back next select call. Any other exception will cause us to assume the client socket is now bad and call the cleanUp(...) function on it.

A key part in us sending this data is that when the sent the data, we sent more than the 'name' + ':'. If we didn't, we must preserve the structure of the data, and keep the data in the same structure it was before the select call. If we sent all the data, we pop the element out of the queue. If we sent more then 'name' + ':', we return the remaining in the 'name:data' format.

Quit socket in read list

```
##################################################################################
##################
    # quit socket - convience to quit from the app

##################################################################################
##################

    if quit_sock in rl:
      raise KeyboardInterrupt("Quitting the application")
```

If the quit socket is in the read list, we throw a KeyboardInterrupt and cause us to exit the main loop.

The address and name table in table.py

```
table = {
     "[00:13:a2:00:40:01:4c:e1]!": "Node_1",
     "[00:13:a2:00:40:01:51:a3]!": "Node_2",
     "[00:13:a2:00:40:01:e9:20]!": "Node_3"
}
```

The table that we use to provide the address to name lookup. We keep it separate from the rest of the application because we may want to have it generated. The only limitation is that the address must be a valid 64 bit address in the above format, and the name of the node must be smaller than MAX_TCP_PACKET_SIZE - 1. This mapping also has the potential to send broadcast transmissions by having a user input an entry: "[00:00:00:00:00:00:FF:FF]!": "Broadcast".

Known limitations and notes

The code above has a few known limitations. The first being we do not keep the data in the same format when it arrives from the client all the way to the mesh nodes. This is because of the limitation of the mesh network's maximum transport size. In 802.15.4, the maximum packet size is 100 bytes, in Znet2.5, it's 72 bytes. ZB architecture depends on the options, but is generally in the same range.

If we were to maintain the data in the same format throughout the application, we could run into a issue where packets being sent to mesh nodes that are mapped to a large name would suffer throughput. In addition, the mesh nodes should be able to assume that all packets received are intended for them. Broadcast transmissions (which could be defined in the name table) could also be taken into account because the addressing scheme of the packets would indicate that it was a broadcast being sent rather then a directed packet.

In the case of the client sending multiple requests to the same node, it is possible the responses to those requests would return in a different order then when we sent them. A potential solution would be to add a sequence number as part of the data to indicate the response, however both the TCP client and the mesh nodes would have to respect that structure, and that is beyond the capabilities of this application.

The source is located here: Tcp_zig_dynamic_mapping.zip.

Understanding XBee EndPoints

The way Digi implimented the XBee/ZigBee end-points in the XBee extensions to the Python socket API implies that they function like TCP or UDP source and destination port numbers. However unlike TCP or UDP which treats each src/dst port pair as unique conversations, XBee/ZigBee assigns each distinct 'destination end-point' to a single application (or handler) for message delivery. The handler is free to manually assign a meaning to the 'source end-point' in received packets, but it is NOT likely unique.

For example, a Modbus/TCP server can bind on TCP port 502. When five clients connect, each uses a destination port of 502 and a unique 'source port' (per IP address pair). Five 'sockets' may be created, and each of the five server-tasks spawned is blissfully unaware that other clients are active.

In contrast, an XBee server can bind on an end-point such as 0xE8, but five remote devices sending to destination end-point 0xE8 will appear to use the same 'socket'. Additionally, they all may claim a source end-point of 0xE8. The Xbee server code must manually process and distribute messages based on extended address or other criteria.

Which XBee technologies support end-points?

You can use the upper word of the DD setting to distinguish XBee which support end-points and those which do not.

Upper DD Word	Name	End-Point Support	Description
0x0000	Unspecified	Unknown	This means the XBee is mis-configured
0x0001	802.15.4	No	Non-mesh star/peer configured nodes on 2.4GHz
0x0002	ZNet 2.5	Yes	Digi's older pre-ZigBee firmware
0x0003	Digi Zigbee	Yes	Digi's firmware supporting Zigbee 2007
0x0004	Digi Mesh 900	Yes	Digi's proprietary mesh on 900Mhz
0x0005	Digi Mesh 2.4	Yes	Digi's proprietary mesh on 2.4Ghz
0x0006	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 868Mhz
0x0007	Digi Point-to-MultiPoint 868	No	Non-mesh star configured nodes on 900Mhz

Notes:

- Use the DD value in the Digi gateway's XBee - you cannot trust that the remote XBee has a valid DD setting!
- Even if the XBee does NOT support end-points, Python may allow you to bind on a specific end-point like 0xE8. However, you will never receive any responses on 0xE8 because all responses appear to be targeted at end-point 0x00.

Managing end-points in XBee

Under XBee AT-transparent mode

So a concrete example, sending a packet with (src=0xE9 dst=0xE8) will NOT automatically result in return-responses being received as (src=0xE8 dst=0xE9) - you need to explictely change the AT settings in the remote XBee to return serial data to destination end-point 0xE9.

Therefore, to move your Python script bind end-point away from 0xE8 (the default), then do the following:

- Your Python script binds on an alterative end-point such as 0xE9, 0x01, or 0x41.
- Manually set all required remote XBee with an AT setting DE=0xE9 (or as required). You can do this by:
  - Under XCTU, relevant Xbee firmwares (such as ZigBee Router AT 0x2264) will have a closed/collapsed sub-option named "ZigBee Addressing" under Adressing. Click on it to open and you'll see the DE/SE/CI options.
  - Newer 2.9.x Digi gateway firmwares expose the DE command under the advanced settings for remote XBee nodes.
  - Use standard "remote AT command" API frames or ddo_set_param() calls to force DE to the desired value.

Note that changing the XBee's SE setting does NOT allow moving serial encapsulation away from 0xE8 - it only changes the source end-point cliamed by AT-Transparent responses. The remote XBee in AT-transparent mode will only forward data out the serial port which is targetted at 0xE8.

Under XBee API mode

When running XBee firmware in API mode, then most source/destination end-points are passed through transparently. It is up to the serial-attached CPU to parse and assign meaning to the end-points. Therefore, an external CPU might gracefully and automatically return responses to requests with (src=0xE9 dst=0xE8) as (src=0xE8 dst=0xE9).

End-point numbers to avoid

Although no exact list is available, you should avoid using these end-points:

End-Point	Description
0x00	ZigBee Device Object end-point - reserved by ZigBee stack
0xDC to 0xEE	Reserved for Digi use
0xEF to 0xF0	Reserved for other vendor use
0xF1 to 0xFE	Reserved by ZigBee Alliance / for other uses
0xFF	Digi treats as 'any end-point' (a wild-card) during Python socket operations

End-points 0x01 to 0xDB should be free to use. However, the rules for picking end-point numbers is much like that for TCP/UDP port numbers - be flexible and ready to change if required. For example, you might use TCP port 2101 to tunnel data to a Digi device server. It works, but TCP port 2101 is officially assigned for use as "RTCM-SC104", which is a Radio Technical Commission for Maritime

Services standard used to move GPS data via TCP/IP. Microsoft also overloads this port for use with RPC-based MQIS and Active-Directory Lookups. The same may be true of XBee end-points - even if you reuse a preassigned end-point it is only a problem if you require some other tool or function with a conflicting use.

The examples above used 0xE9, which Digi claims is reserved, so might be used for some other purpose in the future. If you select end-point 0x77, some other vendor might create a product requiring a Python script binding on end-point 0x77. So design your system to allow the end-point number selected to be easily changed as required.

EndPoints in the Modbus/IA Engine

You can use the Digi Modbus/IA engine to route Ethernet-based Modbus/TCP requests to remote Modbus/RTU serial devices via XBee RS-232 adapter and XBee RS-485 adapter. Configuring the XBee module is covered in Modbus Example Serial Adapter.

However, the default Modbus/IA behavior is to use ZigBee endpoint 0xe8, which will conflict with most sample Python programs and the Device Cloud/DIA framework. This conflict is because only 1 task can bind on (or register to receive) incoming XBee packets on endpoint 0xe8.

Therefore you should move the Modbus/IA traffic away from endpoint 0xe8 and use another, such as 0xe9. This is easier than trying to change the behavior of a Python application in an unknown number of places:

- In the remote XBee adapter, set DE as explained above to E9 (or to your desired value)
- In the IA Modbus unit id/slave address mapping table, add the new Xbee endpoint after the '!', so MAC looks like as 00:13:a2:00:40:30:de:cd!E9. Note that this does NOT change the 'destination' endpoint in the outgoing Modbus request - this remains 0xe8 for XBee module reasons. Instead, it causes the Modbus/IA engine to bind on incoming endpoint 0xE9 instead of 0xE8.

Utility script to get/set AT commands on local/remote zigbee node

Introduction: A Utility script that gets or sets AT commands on local or associated remote zigbee nodes

This utility script is designed for an easy to use mechanism to query and set a small amount of AT commands on a local or associated remote zigbee node.

Requirements:

- A Digi Gateway product
- Command line access to the Digi Gateway product
- Optional: An associated remote zigbee node

Declarations

```
"""  This utility script is designed to execute AT commands for local or
associated remote nodes.

It can both 'get' or 'set' the AT commnad, which is determined by the command
line arguments
passed to it.  See print_usage() for more details on how to use.

In the application we import a 'hidden' library called _zigbee.  Normally the
user would import
zigbee (no leading underscore) which in turn would import the _zigbee file,
however this allows
us fewer requirements when running, as we don't have to have the probable, but
not guaranteed
zigbee library to run.

The other two libraries used, sys and binascii.  Sys is used for the argv values,
binascii is
used to interpret the results from the ddo_get_param commands.  The reason why
binascii is used
instead of struct, is that struct would require knowledge of the command and its
return value.
"""

import _zigbee as zigbee
import sys
import binascii



################################################################################
#########
# Variable declaration

################################################################################
#########
```

```
val    = None         # Value to set
addr   = None         # Address of the mesh node
id     = None         # AT command to execute
dowhat = 'get'        # Flag to determine if we are setting or getting
```

At the start of the script we create a docstring that describes the purpose of the script. We import 3 libraries, 2 are from the standard Python.zip, and the final one is a hidden script that we import directly instead of using the 'zigbee.py' wrapper file. We do this to remove the requirement of having zigbee.py present in our filesystem.

We then declare 4 variables that will be used later.

- val: The value that the AT command will be set to.
- addr: The 64 bit address of the zigbee node. Expected format is 11:22:33:44:55:66:77:88
- id: The 2 character AT command code.
- dowhat: A flag to signal whether we are getting or setting.

Procedure definitions

```
#################################################################################
#########
# print usage()

#################################################################################
#########

def print_usage():
  print "Python %s -a address -i ID [-v Value] [-th]" %sys.argv[0]
  print ""
  print "Required:"
  print "  -a address"
  print "  -i ID"
  print ""
  print "Optional:"
  print "  -v Value"
  print ""
  print "Example:"
  print "Python %s -a 11:22:33:44:55:66:77:88 -i NI -v 'Node Identifier'"
%sys.argv[0]
  print ""
  print "Options:"
  print "  -a    64 bit address of the zigbee adapter you want to communicate
with"
  print "  -i    ID of the variable you want to interact with"
  print "  -v    Indicates what you want the value to be"
  print "NOTE:  To set a blank value, use -v ''"
  print ""
  sys.exit(0)
```

We define the *print_usage()* procedure, where if the user gives no, or invalid command line arguments, we print the usage of the script. It is important to be very verbose here.

```
#################################################################################
#########
# validMeshAddress

```

```
################################################################################
#########

def validMeshAddress(addr):
  """  validMeshAddress validates that the given input matches the same format a
64 bit
  mesh address is in.

  It takes one argument:
    addr - mesh address, expected format is '11:22:33:44:55:66:77:88'

  It returns a boolean on whether or not it passed the criteria of after being
split by
  the semicolon, it has 8 elements, and each element is in the '0-F' range.

  """

  split_addr = addr.split(':')

  if len(split_addr) != 8:
    return False

  for char in split_addr:
    try:
      char = int(char, 16)
    except:
      return False
```

We define the *validMeshAddress* procedure here. We use this to determine if the input matches the format of the 64 bit hardware address a zigbee node might have. This does not guarantee that the mesh address is communicable, or exists. Just that it is in the proper format to feed into the zigbee.ddo_get_param(...) or zigbee.ddo_set_param(...) functions.

Parsing and validating command line arguments

```
################################################################################
#########
# Parse Args

################################################################################
#########

if len(sys.argv) &lt; 2:
  print_usage()

try:
  i = 1
  while i &lt; len(sys.argv):

    if sys.argv[i] == '-a':
      i+=1
      addr = sys.argv[i]

    elif sys.argv[i] == '-v':
      i+=1
      dowhat = 'set'
      val = sys.argv[i]
```

```
    elif sys.argv[i] == '-i':
      i+=1
      id = sys.argv[i]

    else:
      print "Unrecognized parameter: %s" %sys.argv[i]
      print_usage()

    i+=1

except:
  raise ValueError("Invalid set of parameters given: %s" %sys.argv)
```

The above shows how we are processing the command line arguments in 'sys.argv' list. We don't use the standard 'optparse' library for two reasons. First it doesn't not exist in the standard Python.zip. Second, it increases the size of the code base one must understand to use and modify the script.

```
################################################################################
#########
# Verify args

################################################################################
#########

if validMeshAddress(addr) == False:
  raise ValueError("Invalid mesh address given: %s" %addr)
else:
  addr = '[%s]!' %addr

if len(id) != 2:
  raise ValueError("Invalid ID given, ID must be 2 characters")
```

We validate the two parameters that are not flags. We verify that the address given is a possible mesh address, and we verify that the id given is of 2 characters in length.

Main procedure

```
################################################################################
#########
# Main section

################################################################################
#########

if dowhat == 'set':

  try:
    val = int(val)
  except:
    pass

  try:
    zigbee.ddo_set_param(addr, id, val)
  except:
    print "Could not set parameter: %s to value: %s at address: %s" %(id, val, 
addr)
    print "Command failed!"
    sys.exit(0)
```

```
  else:
    try:
      data = zigbee.ddo_get_param(addr, id)
      print "Set parameter: %s to Value: %s at Address: %s" %(id, data, addr)
    except:
      print "Failed to retrieve ID after setting it"
```

We define the process if the user decided to 'set' the AT command. First we attempt to make the 'val' variable a integer. If it fails, we don't mind. We perform a zigbee.ddo_set_param(...) using the supplied address, id, and value. We catch any exceptions, and print the details and result of the command.

```
elif dowhat == 'get':

  try:
    data = zigbee.ddo_get_param(addr, id)
  except:
    print "Could not get parameter: %s At address: %s" %(id, addr)
    print "command failed!"
  else:
    print "Parameter: %s is Value: [%s] at Address: %s" %(id, data, addr)
    try:
      print "Parameter in hexlify form is: %s" %(int(binascii.hexlify(data), 16))
    except:
      print "Failed to interpret into Hex form."
```

We define the process if the user specified to 'get' the AT command. We perform a zigbee.ddo_get_param(...) with the supplied id and address. And we attempt to print it out in two formats: First the raw mode, which will work for strings and second the hexlify mode, which will print out the information as a decimal number.

If the returned value from the command is a complex byte structure, these print statements may appear as gibberish.

Notes and source

The above script is designed as a utility to quickly query and set parameters on various associated nodes from your gateway. It is not designed to replace the functionality of the WebUI or CLI interface to those nodes.

Query_param.zip

Utility to set dest addr in all associated nodes

Utility to set dest_addr (DH/DL) in all associated nodes

A realistic Python application using the ddo_get_param(), ddo_set_param(), and get_node_list() functions.

Users who send serially encapsulated data to RS-232/485 adpaters running AT-mode firmware need to ensure the DH/DL (destination address) registers of each associated node are correct. This is tedious to do by hand - especially if testing requires the same collection of remote RS-232/485 to be routinely moved between different gateways.

The fully functional Python program linked below runs on any Digi ConnectPort X gateway. It obtains a list of associated nodes and makes sure the DH/DL registers of each are set to the gateway running the program. The application can also be MODIFIED to be a more general-purpose configuration refresh tool - for example the code could be updated to ensure the baud rate and RS-232 control signals are properly configured.

Routines used; Things you can learn

The program **dest_addr_to_me.py** does the following:

- Uses **zigbee.getnodelist()** to obtain a list of associated nodes.
- Uses **zigbee.ddo_get_param()** to read parameters from each node.
- Uses **zigbee.ddo_set_param()** to write parameter to each node.
- Detects the Xbee RS-232 PH (Power Harvesting) adapters and limits the ddo_get_param ()/ddo_set_param() calls to a slow enough rate to NOT deplete the charge of the super-cap, which causes the adapter to go offline
- After running, all DH/DL parameters will be as desired. Users can manually set an alternative address in the Python program if they do not wish to use the gateway's address.

Sample output

```
#> Python dest_addr_to_me.py

Will confirm that Gateway/Coordinator address is every node's dest_addr
  FYI: Coordinator is ZB_2007, with address=[00:13:a2:00:40:3e:1c:80]!

Checking node BELA_2=[00:13:a2:00:40:3e:15:2d]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:3e:15:2d]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:3e:15:2d]!,WR) returned True

Checking node FANI_6=[00:13:a2:00:40:52:29:d7]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:52:29:d7]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:52:29:d7]!,WR) returned True

Checking node ANNA_1=[00:13:a2:00:40:3e:15:18]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:3e:15:18]!,DL,1077812352) returned True
```

```
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:3e:15:18]!,WR) returned True

Checking node DEBI_4=[00:13:a2:00:40:34:16:14]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:34:16:14]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:34:16:14]!,WR) returned True

Checking node ELSA_5=[00:13:a2:00:40:52:29:f9]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:52:29:f9]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:52:29:f9]!,WR) returned True

Checking node CALI_3=[00:13:a2:00:40:4a:70:7e]! settings
- Parameter DL is not as expected; desire 0x403E1C80 but saw 0x40341824
    ddo_set_param([00:13:a2:00:40:4a:70:7e]!,DL,1077812352) returned True
- Node changed, need to save settings
    ddo_set_param([00:13:a2:00:40:4a:70:7e]!,WR) returned True

#> Python dest_addr_to_me.py

Will confirm that Gateway/Coordinator address is every node's dest_addr
    FYI: Coordinator is ZB_2007, with address=[00:13:a2:00:40:3e:1c:80]!

Checking node BELA_2=[00:13:a2:00:40:3e:15:2d]! settings
- Node has desired settings already

Checking node FANI_6=[00:13:a2:00:40:52:29:d7]! settings
- Node has desired settings already

Checking node ANNA_1=[00:13:a2:00:40:3e:15:18]! settings
- Node has desired settings already

Checking node DEBI_4=[00:13:a2:00:40:34:16:14]! settings
- Node has desired settings already

Checking node ELSA_5=[00:13:a2:00:40:52:29:f9]! settings
- Node has desired settings already

Checking node CALI_3=[00:13:a2:00:40:4a:70:7e]! settings
- Node has desired settings already

#>
```

Download the Python code

This code only runs on a Digi ConnectPort X gateway: Python program "Dest_addr_to_me.zip.py" in ZIP form

Xbee Command to Device Cloud Cross Reference

Xbee experts and documentation generally use AT parameters to explain configuration parameters. Device Cloud uses verbose descriptions. The list below provides a cross-reference.

UPDATE: *Device Cloud now shows both the verbose command and the AT command in the Xbee properties screen, so this cross-refence is no longer necessary. Do still be aware that Device Cloud operates in decimal, rather than hex, in most, but not all, fields.*

Note Most of the parameters on Device Cloud use decimal, while most direct Xbee operations are in hex. Keep this in mind.

A1 - End device association

A2 - Coordinator association

AR - Aggregation route notification

BD - Serial interface data rate

BH - Broadcast radius

CA - CCA threshold

CC - Command sequence character

CE - Coordinator enable

CH - Operating channel

CI - Cluster identifier

CT - Command mode timeout

D0 - AD0/DIO0 configuration

D1 - AD1/DIO1 configuration

D2 - AD2/DIO2 configuration

D3 - AD3/DIO3 configuration

D4 - AD4/DIO4 configuration

D5 - DIO5/Assoc configuration

D6 - DIO6 configuration

D7 - DIO7 configuration

D8 - DIO8/SleepRQ configuration

D9 - DIO9/ON_SLEEP configuration

DE - Destination endpoint

DP - Disassociated cyclic sleep period

EE - Encryption enable

EO - Encryption options

FT - Flow control threshold

GT - Guard times

HP - Hopping sequence

IA - I/O input address

IC - DIO change detect

ID - Extended PAN identifier

ID - PAN identifier

IF - I/O sample from sleep rate

II - Initial PAN identifier
IR - I/O sample rate
IT - I/O samples before transmit
JN - Join notification
JV - Join verification
KY - Link encryption key
LT - Associate LED blink time
M0 - PWM0 output level
M1 - PWM1 output level
MM - MAC mode
MR - Mesh network retries
MT - Broadcast retries
MY - Network address
NB - Serial interface parity
NK - network_key
NH - Maximum hops
NJ - Node join time
NN - Network delay slots
NI - node_id
NT - Node discovery timeout
NW - Network watchdog timeout
P0 - DIO10/PWM0 configuration
P0 - PWM0 configuration
P1 - DIO11/PWM1 configuration
P1 - PWM1 configuration
P2 - DIO12/CD configuration
P3 - DIO13/DOUT configuration
PL - Transmit power level
PM - Power mode
PO - Polling rate
PR - Pull-up resistor enable
PT - PWM output timeout
RN - Random delay slots
RO - Packetization timeout
RP - RSSI PWM timer
RR - MAC retries
RR - XBee retries
SB - Stop bits
SC - Scan channels
SD - Scan duration
SE - Source endpoint
SM - Sleep mode

SN - Peripheral sleep count

SO - Sleep options

SP - Cyclic sleep period

ST - Time before sleep

SW - Sleep early wakeup

T0 - D0 output timeout

T1 - D1 output timeout

T2 - D2 output timeout

T3 - D3 output timeout

T4 - D4 output timeout

T5 - D5 output timeout

T6 - D6 output timeout

T7 - D7 output timeout

V+ - Supply voltage high threshold

WH - Wake host delay

ZA - ZigBee addressing enable

ZS - ZigBee stack profile

XBee extensions to the Python socket API

Digi has extended the standard Python sockets interface to abstract XBee/ZigBee Mesh networking technology. This has been accomplished by defining a new address family, **AF_XBEE**, and by defining new protocol constants **XBS_PROT_APS and XBS_PROT_TRANSPORT** for use with standards-compliant ZigBee XBee modules and proprietary ZigBee-like DigiMesh modules, respectively.

In firmware version 3.2.7.6 and later (NOTE: Xbee Gateway only), the **XBS_PROT_DDO** protocol may be used to send and receive asynchronous AT commands using the sockets interface.

ZigBee transmits data in distinct frames, referred to by the ZigBee protocol specification as “APS Data” or “Application Data Units” (APDU). Frame transmission is connectionless: frames may be lost or be received in a different order from which they were sent by the transmitter. These behaviors match the **SOCK_DGRAM** socket type, socket datagram service.

Creating a socket and binding a socket to an endpoint using the *socket()* and *bind()* methods of the Python sockets module allows for easy communication on a Mesh network. The XBee Sockets extension for Python has been authored semantically to operate under the principle of least surprise: methods behave as much as possible as they do for any other network type. The *socket()* call creates a socket, *sendto()* and *recvfrom()* are used to send and receive datagrams, *select()* is used to wait on socket descriptor activity, and so on.

To get started writing Python code that takes advantage of the XBee extensions to the Python sockets API, see the ZigBee Sockets Examples later in this guide.

For detailed information on what functions have been extended to be used with XBee, see the following function reference:

close()

Purpose

Close the socket.

Syntax

```
close(self) . None
```

Description

Close the XBee socket. The socket cannot be used after this call.

If the socket was bound to an application endpoint, that application endpoint becomes available for other sockets to use.

getsockopt()

Purpose

Get socket options

Syntax

```
getsockopt(self, level, optname) . integer or string
```

Description

Get an XBee socket option.

level

level specifies which level the socket option is to be read from. The following levels are specified:

SOL_SOCKET	Regular socket options, such as **SO_LINGER**
XBS_SOL_ENDPOINT	Get options for the endpoint bound to the socket object.
XBS_SOL_EP	Alias for **XBS_SOL_ENDPOINT**

optname

optname is expected to be an integer constant from the socket module. The following optname values are specific to the given level:

XBS_SOL_ENDPOINT / XBS_SOL_EP

XBS_SO_EP_FRAMES_TX	Get the number of frames transmitted from this endpoint.
XBS_SO_EP_FRAMES_RX	Get the number of frames received at this endpoint.
XBS_SO_EP_FRAMES_TX_ERR	Get the number of frames that could not be transmitted due to an error.
XBS_SO_EP_FRAMES_RX_ERR	Get the number of received frames dropped due to an exhaustion of internal buffers.
XBS_SO_EP_BYTES_TX	Get the number of bytes transmitted from this endpoint.
XBS_SO_EP_BYTES_RX	Get the number of bytes received at this endpoint.
XBS_SO_EP_BYTES_TX_ERR	Get the number of bytes that could not be transmitted due to an error.
XBS_SO_EP_BYTES_RX_ERR_UTRUNC	Get the number of bytes dropped because the user buffer passed torecvfrom()was not large enough to contain the entire packet.
XBS_SO_EP_BYTES_RX_ERR_NOBUF	Get the number of received bytes dropped due to an exhaustion of internal buffers.
XBS_SO_EP_BCAST_RADIUS	Get the value of the XBS_SO_EP_BCAST_ RADIUS option. This value specifies the number of hops for a broadcast transmission. It is ignored if the frame is not a broadcast transmission. A value of 0 specifies the maximum number of hops (NH) is used.

XBS_SO_EP_SYNC_TX	Get the value of the **XBS_SO_EP_SYNC_TX** flag. A value of 0 indicates messages sent by sendto() are queued and delivered asynchronously. If an error occurs, it is not reported. A value of 1 indicates sendto() waits until the message is acknowledged by the receiver. If an error occurs, an exception is thrown. Use of this value may reduce message throughput.
XBS_SO_EP_TX_STATUS	Get the value of the **XBS_SO_EP_TX_STATUS** flag. A value of 0 indicates that transmit status frames will not be made available by a call to recvfrom() on the socket. A value of 1 indicates that a call to recvfrom() may include XBee Transmit Status information (generated by a previous sendto() call) allowing a program to determine asynchronously whether or not a prior transmission was successfully received by the remote recipient. See "Notes on Transmit Status", below.

recvfrom()

Purpose

Receive a message from a socket.

Syntax

```
recvfrom(self, buflen [,flags]) . (data, addr)
```

Description

Receive up to buflen bytes of a datagram from an endpoint bound on the socket. If the datagram contained more bytes than buflen, the extra bytes are silently discarded (like with UDP/IP). In practice, set buflen to 255 for most XBee technologies.

This function returns the data from the socket along with the sender’s address for the data in a tuple.

If receiving an AT command, *data* contains the results of a command sent by sendto(). It may be empty if the command returns no results. Command status is always returned in *addr*.

See Notes on XBee Address Tuples below for the format of *addr*.

The only flag supported is **MSG_DONTWAI**T to force a single socket transaction to be non-blocking.

sendto()

Purpose

Send a message to a socket.

Syntax

```
sendto(data[, flags], addr) . count
```

Description

Send a datagram specifying the destination address.

If sending an AT command, data contains the command paramters. It may be empty if the command takes no parameters. The command results and status are received at a later time by calling recvfrom ().

See Notes on XBee Address Tuples below for the format of *addr*.

The only flag supported is **MSG_DONTWAIT** to force a single socket transaction to be non-blocking.

setsockopt()

Purpose

Set socket options.

Syntax

```
setsockopt(self, level, optname, value) . integer or string
```

Description

Set a XBee socket option.

level specifies the level to which the socket option is to be applied. The following levels are specified:

SOL_SOCKET	Regular socket options.
XBS_SOL_ENDPOINT	Mesh extension options.

optname is expected to be an integer constant from the socket module. The following optname values are specific to the given level:

XBS_SOL_ENDPOINT / XBS_SOL_EP

XBS_SO_EP_BCAST_RADIUS	Set the value of the XBS_SO_EP_BCAST_RADIUS option. This value specifies the number of hops for a broadcast transmission. It is ignored if the frame is not a broadcast transmission. A value of 0 specifies the maximum number of hops (NH) is used.
XBS_SO_EP_SYNC_TX	Set the value of theXBS_SO_EP_SYNC_TXflag. A value of 0 indicates messages sent by sendto() are queued and delivered asynchronously. If an error occurs, it is not reported. A value of 1 indicates sendto() waits until the message is acknowledged by the receiver. If an error occurs, an exception is thrown. Use of this value may reduce message throughput.

XBS_SO_EP_TX_STATUS	Set the value of theXBS_SO_EP_TX_STATUSflag. A value of 0 indicates that transmit status frames will not be made available by a call to recvfrom() on the socket. A value of 1 indicates that a call to recvfrom() may include XBee Transmit Status information (generated by a previous sendto() call) allowing a program to determine asynchronously whether or not a prior transmission was successfully received by the remote recipient. See "Notes on Transmit Status", below.

Value can be either an integer or a string depending on the argument requirements of the socket option named by optname.

socket()

Purpose

Create a XBee endpoint for communication.

Syntax

```
socket(AF_XBEE [, type [, proto]]]) . socket object
```

Description

Open a XBee socket of the given type. At present the type must be **SOCK_DGRAM**.

proto can be either **XBS_PROT_TRANSPORT** for the proprietary mesh transport or **XBS_PROT_APS** for the ZigBee standards compliant APS transport. The transport type depends on which Mesh radio and Mesh radio firmware is loaded in the gateway device.

In gateway firmware version 3.2.7.6 and later, *proto* can be **XBS_PROT_DDO** to send and receive asynchronous AT commands.

If the transport protocol specified is unusable with the installed radio, **EINVAL** will be returned.

Notes on XBee address tuples

Address tuples for data frames

This applies to data frames sent and received using the **XBS_PROT_APS or XBS_PROT_TRANSPORT** protocol.

The general format for an XBee address is: (address_string, endpoint, profile_id, cluster_id, options_bitmask, transmission_id)

address_string must be of the form "[nn:nn:nn:nn:nn:nn:nn:nn]!" for 64-bit addresses or "[nnnn]!" for 16-bit addresses.

endpoint is an 8-bit unsigned integer

profile_id is a 16-bit unsigned integer

cluster_id is a 16-bit unsigned integer

options_bitmask is an 8-bit unsigned integer

transmission_id is an 8-bit unsigned integer

Values for *options_bitmask* mean the following on a frame received from a call to recvfrom():

XBS_OPT_RX_ACK	Received as unicast (should be renamed, as it does not indicate acknowledgement).
XBS_OPT_RX_BCADDR	Received via broadcast.
XBS_OPT_RX_BCPAN	Received on broadcast PAN.
XBS_OPT_RX_APSSEC	Received using APS end-to-end security.

Note that certain receive options are only valid when using certain XBee modules and firmware combinations (e.g. such as the XBee Series 2 Smart Energy firmware).

Values for *options_bitmask* mean the following on a frame transmitted from a call to sendto():

XBS_OPT_TX_NOACK	Disable end-to-end acknowledgement.
XBS_OPT_TX_NOREPEAT	Do not repeat this packet.
XBS_OPT_TX_BCPAN	Send to broadcast PAN.
XBS_OPT_TX_TRACERT	Invoke traceroute.
XBS_OPT_TX_PURGE	Purge if delayed by duty cycle.
XBS_OPT_TX_APSSEC	Transmit using APS end-to-end security.

Values for *transmission_id* must be greater than zero if the XBS_SO_EP_TX_STATUS option is used to track an individual transmission status.

Address tuples for AT commands

This applies to AT commands sent and received using the **XBS_PROT_DDO** protocol.

The general format for an XBee address is: (address_string, command, options_bitmask, transmission_id, status)

address_string must be of the form "[nn:nn:nn:nn:nn:nn:nn:nn]!" for 64-bit addresses. 16-bit addresses are not allowed.

command is a 2-character AT command

options_bitmask is an 8-bit unsigned integer

transmission_id is an 8-bit unsigned integer

status is an 8-bit unsigned integer

Values for *options_bitmask* mean the following on a frame transmitted by a call to sendto():

XBS_OPT_DDO_APPLY	Apply changes to node settings immediately. If not set, changes are queued until a command with XBS_OPT_DDO_APPLY set or the AC command is sent to the node.
XBS_OPT_DDO_ORDER	Send this command in the same order relative to other commands and data transmissions. If not set, commands and data sent to other nodes may be reordered for efficiency. Commands and data sent to a single node are always sent in order.

The *options_bitmask* field is present but reserved in the address tuple returned by a call to recvfrom().

The *transmission_id* field may be used to match a transmitted command with the received results. The value in *transmission_id* supplied to sendto() will be returned in the address tuple returned by recvfrom ().

Values for *status* mean the following:

XBS_STAT_OK	Command completed successfully.
XBS_STAT_ERROR	Command processing error.
XBS_STAT_BADCMD	Command is invalid or not supported.
XBS_STAT_BADPARAM	Command parameter is invalid or out of range.
XBS_STAT_TXFAIL	Transmission error to remote node.

The *status* field is present in the address tuple returned by recvfrom(). It is not used by sendto().

Notes on Transmit Status

If a user sets the **XBS_SO_EP_TX_STATUS** flag using a call to setsockopt() with a *level* of **XBE_SOL_ENDPOINT** subsequent calls to recvfrom() will produce information pertaining to the transmit status of previous XBee transmissions created with prior calls to the sendto() function. Here is an example of enabling transmit status frames using a call on an XBee socket object called *xbee_sd*:

```
            # Enable transmission status reports on calls to recvfrom()
xbee_sd.setsockopt(XBS_SOL_EP, XBS_SO_EP_TX_STATUS, 1)
```

In order to correlate a specific transmission with a transmit status frame received via the recvfrom() call one must use the *transmit_id* of the address tuple when addressing a frame in a call to sendto(). For example:

```
            # Send hello world to remote XBee using transmit ID of 0x42
xbee_sd.sendto("Hello, World!", 0, ('[00:13:a2:00:40:0a:07:a5]!', 0xe8, 0xc105,
0x11, 0, 0x42))
```

Subsequent calls to recvfrom() may then receive a transmit status frame and additional logic may detect one of several types of transmit status frames:

```
            # Receive message, detect if transmit status frame:
buf, addr = xbee_sd.recvfrom(84)
cluster_id = addr[3]
xmit_id = addr[5]

if cluster_id == 0x89:
    # X-API transmit status frame, TX_STATUS 3rd byte:
    tx_status = ord(buf[2])
elif cluster_id == 0x8b:
    # X-API ZigBee transmit status frame, TX_STATUS 6th byte:
    tx_status = ord(buf[5])
elif cluster_id == 0:
    # XBee driver status indication:
    tx_status = struct.unpack("i", buf)[0]
else:
        # must be a regular data frame:
```

```
        handleRegularData(buf, addr)

if tx_status == 0:
    # Transmission successful!
    accountForGoodTransmission(xmit_id)
```

Transmit status is only available for sockets using the **XBS_PROT_APS or XBS_PROT_TRANSPORT** protocol.

For reference on the possible values of the X-API transmit status frames, please refer to the XBee OEM Module Manual.

XBee Sockets Examples

Send “Hello, World!”

```
            #
# This example sends "Hello, World!" using the Digi
# proprietary mesh transport to a fixed node address.
#


# include the sockets module into the namespace:
import xbee
from socket import *


# The Format of the tuple is:
# (address_string, endpoint, profile_id, cluster_id)
#
# The values for the endpoint, profile_id, and
# cluster_id given below are the values used to write
# to the serial port on an Ember-based XBee module.
# For 802.15.4 use 0,0,0
DESTINATION=("00:0d:6f:00:00:06:89:29!", \
0xe8, 0xc105, 0x11)

# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)


# Bind to endpoint 0xe8 (232) for ZB/DigiMesh, but 0x00 for 802.15.4
sd.bind(("", 0xe8, 0, 0))


# Send "Hello, World!" to the destination node, endpoint,
# using the profile_id and cluster_id specified in
# DESTINATION:
sd.sendto("Hello, World!", 0, DESTINATION)
```

socket.bind() Notes

If the bind fails with socket.error:(22, 'Invalid argument'), then double check that another running Python script has not already bound to the end-point you selected - for example, check the Python auto-start settings. Only one Python script can bind to a specific end-point, such as the 0xE8 (232) in this example.

The last 2 parameters are critical for the sendto(), but ignored by the bind(). The sd.bind() will return any packets received on end-point 0xE8 regardless of cluster or profile codes.

bind() is not needed or allowed on sockets using the XBS_PROT_DDO protocol.

Reading and writing

```
            #
# This example binds to application endpoint 0xe8,
# receives a single frames at this endpoint and then
# sends the frame's payload back to the originator
# using the radio's proprietary mesh transport.
#

# include the sockets module into the namespace:
import xbee
from socket import *

# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)

# Bind to endpoint 0xe8 (232) for ZB/DigiMesh, but 0x00 for 802.15.4
sd.bind(("", 0xe8, 0, 0))

# Block until a single frame is received, up to 255 bytes:
payload, src_addr = sd.recvfrom(255)

# Send the payload back to the source we received it from:
sd.sendto(payload, 0, src_addr)
```

Note The maximum size messages returned by recvfrom() varies based on technology. At times it is 72 or 75 bytes, or it might be 84, 100, or even 220 bytes. Keep in mind this socket layer mimics UDP, so reading 72 bytes from the socket when 75 bytes are waiting causes the 3 orphaned bytes to be discarded.

AT commands

```
            #
# This example sends an AT command and receives its result using XBS_PROT_DDO.
# While this uses blocking I/O, non-blocking I/O may also be used for AT
commands.
#

# include the sockets module into the namespace:
import xbee
from socket import *

# Create the socket, datagram mode, DDO protcol.
# bind() is not needed.
sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_DDO)

# Send a VR command to get a node's firmware version.
# The VR command takes no parameters.
sd.sendto('', 0, (address_string, 'VR', 0, 1))

# Block until the response is received, up to 255 bytes:
result, src_addr = sd.recvfrom(255)

# Print command results and status:
print result.encode('hex'), src_addr[4]
```

Non-blocking I/O

```
            # This example gives a simple demonstration of how
# to set and use ZigBee sockets configured for
# non-blocking I/O with select. This application
# echoes packets back to the originator.
#
# The socket is marked for reading only if
# the payload buffer is empty; if the buffer is
# non-empty then the socket is marked for writing.
# Select is used to arbitrate when a socket is
# ready to be read or written.
#
# This example could be easily extended to operate
# on multiple sockets.
#


# Include the socket and select modules:
import xbee
from socket import *
from select import *


# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_XBEE, SOCK_DGRAM, XBS_PROT_TRANSPORT)
# Bind to endpoint 0xe8 (232) for ZB/DigiMesh, but 0x00 for 802.15.4
sd.bind(("", 0xe8, 0, 0))
# Configure the socket for non-blocking operation:
sd.setblocking(0)


try:
    # Initialize state variables:
    payload = ""
    src_addr = ()

    # Forever:

    while 1:
        # Reset the ready lists:
        rlist, wlist = ([], [])
        if len(payload) == 0:


            # If the payload buffer is empty,
            # add socket to read list:
            rlist = [sd]

        else:
            # Otherwise, add the socket to the
            # write list:
            wlist = [sd]


        # Block on select:
        rlist, wlist, xlist = select(rlist, wlist, [])
```

```
        # Is the socket readable?
        if sd in rlist:
            # Receive from the socket:
            payload, src_addr = sd.recvfrom(72)
            # If the packet was "quit", then quit:
            if payload == "quit":
                raise Exception, "quit received"

        # Is the socket writable?
        if sd in wlist:
            # Send to the socket:
            count = sd.sendto(payload, 0, src_addr)
            # Slice off count bytes from the buffer,
            # useful for if this was a partial write:
            payload = payload[count:]

except Exception, e:
    # upon an exception, close the socket:
    sd.close()
```

XBee sleeping problems

XBee Sleeping FAQ

ZigBee

Data from Sleeping End-Device is Lost

Question: *I set my ZigBee end-device to sleep and wake every 10 seconds. It should send me data (for example the IR or IC-driven I/O pin data), but nothing arrives at the gateway/destination.*

Question: *My micro wakes the Xbee running as ZigBee end-device, but the data I send never arrives at the gateway/destination.*

Answer: One of the biggest problems beginners suffer when trying to use ZigBee sleeping end-devices is a failure to properly configure the non-sleeping parent (the router or coordinator). While at first this seems a needless complexity, it has to do with the limited resources within the parent. Literally, the parent maintains a collection of free memory (a scarce resource) to both buffer messages for the end-device to fetch upon waking, and also to guarantee that the sleeping device can hand-off a message to the parent the instant it wakes.

Now consider the scenario where 5 end-devices are sleeping, waking once each hour. The parent needs to allocate 5 separate sets of this scarce memory resource just in case all 5 wake up at the same time. Suppose 1 of the 5 fails, and is replaced. A new sixth end-device joins the parent, and now the parent has 6 separate sets of memory buffers - one set which will never be used again.

Therefore the parent (the router or coordinator) includes a Child Poll Timeout - search the XBee ZB module reference for that term. The parent overloads the meaning of the SN and SP settings, using the value (3 * SP * SN) to define the Child Poll Timeout. In our once-per-hour example, if the parent has SN * SP set to one hour, then three hours after the broken/missing device fails to have polled its parent for work, the parent frees up the resource allocated to that device. If the device later connects and tries to poll the parent for data, the parent rejects the child. If the child was trying to send data to the parent, the data is discarded ... sounds like your problem, right?

The child of course renegotiates, and if the parent has spare memory resources, then the child is again allocated a memory resource. However, unless special code within the child retries the data-send, then the data has been lost. The standard XBee 'IS' data-send is not retried - it is lost. If your micro woke the XBee and sent a data packet, then in AT mode that data has been lost without warning. In API mode, the TX Status packet should return an error (probably 0x22 = Not Joined to network). Your firmware will need to understand means your last data packet was lost, so it should be resent. Your code does NOT need to fixed the 'Not Joined' error. The XBee does that itself.

Example SN/SP setting for the router/parent

Below are some example values for SN/Sp to use in your 'parents'.

Expected Sleep Time	SN	SP	Actual End-Device Timeout
default (minimum)	0x01	0x20	less than 1 second
maximum	0xFFFF	0xAF0	over 60 days
5 seconds	0x01	0x1F4	15 seconds

Expected Sleep Time	SN	SP	Actual End-Device Timeout
28 seconds	0x01	0xAF0	84 seconds
1 minute	0x03	0x7D0	3 minutes
5 minute	0x0F	0x7D0	15 minutes
15 minute	0x2D	0x7D0	45 minutes
1 hours	0xB4	0x7D0	3 hours
3 hours	0x21C	0x7D0	9 hours
12 hours	0x870	0x7D0	36 hours
24 hours	0x10E0	0x7D0	3 days

Notes:

- The values need to be copied to all of our routers, including your coordinator.
- This means if you have 20 sleeping end-devices, some waking once per 5 minutes and others waking once per hour, you will need to set SN/SP values suitable for 1 hour in ALL routers, for you cannot control which router your end-devices will connect to, all must expect the worst case 1 hour of no contact.
- if your router/coordinator is 'full', meaning the value NC=0 (Number of Remaining Children), then it will take the 3 * SN * SP time before a faulty device which has been replaced will be recognized. So in our 1 hour example, if a coordinator has NC=0 (which as-of fw 2xA7 means 10 children are attached), then a new eleventh replacement device will be ignored for 3 hours.

Xig

The XBee Internet Gateway ("XIG") is an application written for Digi's ConnectPort series of XBee-to-IP gateways. The XBee Internet Gateway gives any device the ability to connect seamlessly to the Internet by mirroring the interactions humans have with web browsers. Any device with an XBee radio can send a web URL to the XIG and receive back the contents of that web page. All the tricky technical aspects of web connections are all handled for you behind the scenes.

This simple service gives your prototype or device a simple yet completely flexible pathway to any web service that you can imagine, including posting sensor values, scraping Facebook or commanding your robotic kitten army.

XIG offeres a myriad of other interesting and useful communications services to your XBee network. For complete documentation and setup instructions please visit:

http://code.google.com/p/xig/

See also:

- http://code.google.com/p/xig/wiki/UserDocumentation
- http://www.faludi.com/projects/xbee-internet-gateway/
- http://www.digi.com/products/wireless-routers-gateways/gateways/

XIG is brought to you by an open-source team by makers Robert Faludi, Jordan Husney and Ted Hayes with valuable support from a community of commercial and educational users.

XBee Zigbee

This category covers use of sample applications for XBee, Zigbee products.

XBee bootloader menu

Program to bypass bootloader menu

XBee bypass bootloader menu (Xbee program) This sample application skips and bypasses freescale MCU & directly access radio.

Test files

This sample program contains several files. Program is in file "main.c".

Bypass bootloader menu Test Sample Application

The bootloader menu Test sample application can be found here: Bypass_bootloader_menu.zip.

Basic usage

```
  APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250.
```

Sample of bootloader bypass menu:

```
#include <xbee_config.h>
/* #include <types.h>
#include <xbee/platform.h>*/


void main(void)
{
/*       sys_hw_init();
        sys_xbee_init();
        sys_app_banner(); */
        //APP_CAUSE_BYPASS_MODE passes the local UART data directly to the EM250
        sys_reset(APP_CAUSE_BYPASS_MODE);
}
```

XBeeComm

Program to communicate XBeeComm through Com port with Windows 7

XBeeComm Sample (For Windows 7 PC) This application can communicate with the Xbee module from the PC through the COM port. Using the different buttons on the form we can get the different Xbee parameters like pan.

How does it work?

Using this C sharp application , we can communicate with the Xbee module from the PC. Using the different buttons on the form we can get the different Xbee parameters like pan id, version of the software.

Test files

This sample program contains nine files.

XBeeComm Test Sample Application

The XBeeComm Test sample application can be found here: XbeeComm.zip.

Basic usage

Compile, load and run program using Windows Tools.

Sample Program.cs file:

```
#using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace app2
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Categories

Reading from ZipFiles on the Gateway

Reading from a zip file on a Gateway product

This article describes how to read individual files from a zip file, such as text, that are uploaded to a Python enabled device.

Getting started

The library to use to read a single file from a zip file, such as the Python.zip, on a Python enabled device is the 'zipfile' library. The standard Python.zip currently available (12/17/08) doesn't have it included by default, but the file can be uploaded directory to your device from the Python2.4 library installed on a PC.

Source code

```
#> Python

>>> import zipfile
>>> pzip = zipfile.ZipFile("WEB/Python/Python.zip")
>>> pzip.read("VERSIONS.txt")
'Version 40002643_B created April 29, 2008\n\n- Added codecs and basic encoding
support\n\nVersion 40002643_A\n\n- Initi
al revision tracked version of Python.zip\n'>>>
```

Above we have an idle session started on a ConnectPort X8 that has the standard Python.zip file uploaded, as well as the zipfile.py from the Python libs directory. We import the zipfile, and create a new instance of the zipfile object using the intended zipfile's path as the input, and read a specific file from the archive to the console. In this case we used the 'VERSIONS.txt' file that is included in the Python.zip.

EmbeddedLinux - time sample

Time program for embedded Linux

EmbeddedLinux Time Sample (For Digi EmbeddedLinux 5.x modules) This example Demonstrates how to get time with the Digi EmbeddedLinux modules.

Test Files

This sample program contains one file. The main function is in main.c.

EmbeddedLinux Time Program Test Sample Application

The EmbeddedLinux Time Test sample application can be found here: Time_del_sample.zip.

Basic Usage

```
  Compile, load and run application.
```

Sample of main.c file:

```
/*
*  main.c
*
*  Created on: Jan 9, 2012
*      Author: athomas
*/


#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

int main()
{
        struct timeval tv;
        while(1)
        {
                if(gettimeofday(&tv,NULL) != 0)
                        perror("Gettime Error\n");
                printf("gettimeofday() Seconds : %ld , Microseconds :
%ld\n",tv.tv_sec,tv.tv_usec);
                sleep(1);
        }
        return 0

}
```

EmbeddedLinux - UDP server-client

Time program for EmbeddedLinux

EmbeddedLinux UDP Server-Client (For Digi EmbeddedLinux 5.x modules) This example Demonstrates UDP sockets.

Test Files

This sample program contains three files, Makefile, Client.c and Serv.c.

EmbeddedLinux UDP Server-Client Sample Application

The EmbeddedLinux UDP Server-Client Sample application can be found here: UDP_Server-Client.zip.

Basic Usage

```
  Compile, load and run application.
```

Sample of client.c file:

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
```

```
int main(int argc, char**argv)
{
        if(argc != 3)
        {
                printf("Enter Servr IP & UDP port as command line arguments, eg:
./client x.x.x.x 5001\n");
                exit(-1);
        }
        int serv_des;
        struct sockaddr_in serv_addr;
        char buff[20];
        socklen_t sock_len;

        printf("UDP Client\n");
        serv_addr.sin_family = AF_INET;

        if(inet_aton(argv[1],&(serv_addr.sin_addr)) == 0)
        {
                printf("Wrong IP format\n");
                exit(-1);
        }

        serv_addr.sin_port = htons(atoi(argv[2]));
        memset(&(serv_addr.sin_zero),'\0',8);
        sock_len = sizeof(struct sockaddr);

        if( (serv_des = socket(PF_INET,SOCK_DGRAM,0) ) < 0){
                perror("Server Descriptor Creation Failed\n");
        }


        if(sendto(serv_des,"I Am UDP Client\0",16,0,(struct sockaddr*)&serv_
addr,sock_len) <0){
                perror("Send To Failed\n");
        }



        if(recvfrom(serv_des,buff,16,0,(struct sockaddr*)&serv_addr,&sock_len) <0
){
                perror("Receive From Failed\n");
        }

        printf("Server Says : %s\n",buff);


        return 0;
}
```

IDigi Easy Demo Details - Support for iDigi Easy Demo

Delete this text and replace it with your own content.

The Basic Framework of the main function in your first Programmable XBee Application

What is happening in the main() function of my Programmable XBee Application?

After you have selected your first example application from the Programmable XBee Tutorial, you will notice a few functions already populated in the main() procedure call of your application, followed by the typical "run forever" for-loop found in most processor/MCU based code.

If you have started your tutorial with an example from the XBee samples, then you will notice that your main() function will look something like the following:

```
void main(void)
{
 (1)sys_hw_init();
 (2)sys_xbee_init();
 (3)sys_app_banner();

    for (;;) {
        /*
 (4)     * Nothing to do...
         * Everything is done in the periodic task
         */
        sys_watchdog_reset();
        sys_xbee_tick();
    }
}
```

While at the top-most level, there are only a few lines of code present in the main function itself (besides the for-loop), there is quite a bit of code being executed within these few statements. That being said, it's always good to know exactly what is happening underneath these function calls in order to give you a better idea as to how and when structures are set up and interfaces initialized on the Freescale side of the equation, as well as when and how the various XBee radio parameters are initialized and how they work together with the Freescale CPU to provide you with an interface to create a programmable XBee application.

For your first sample application, it's not necessary that you understand all of the details and inner-workings of these functions, but it is a good idea to know what is already being set up for you in these functions so that you have a better idea of what your options are should you decide to add additional functionality to the main() function itself.

Let's start with the first line in the main function, which is where the majority of the functionality and setup for your Programmable XBee application takes place.

Step 1: sys_hw_init()

This function is by far the "busiest" function, when it comes to setting up your application, primarily because it is initializing the hardware and software features of the onboard Freescale CPU, in addition to setting up some of the core requirements for the communciation between the CPU and XBee radio itself.

The first function call which is invoked from sys_hw_init() is *sys_init()*. This function first calls *sys_gpio_init()*, which primarily initializes basic system GPIOs, one of those being the XBee RESET pin.

The next function call which is made is *sys_clocks_init()*, which is used to initialize the Freescale system clock.

Finally, sys_init() makes a call to *sys_radio_reset()*, which sets the XBee's RESET pin low for 250ns and at the end of initializing is released, so that it can be raised and lowered later on as needed.

This is the last configuration step that is made by sys_init(). From here, sys_hw_init() continues by configuring the real time clock in the Freescale CPU by calling *rtc_config()*. This function serves to enable the RTC by enabling the interrupt and setting the frequency of each clock tick to 4ms. The RTC can be used later in your programmable XBee application in different ways including watchdog operations as well as making calls to functions such as delay_ticks() for adding delays in your application.

The next call which is made by sys_init() is *sys_irqs_enable()*, enabling the CPU's IRQs in preperation for more upcoming CPU/radio configuration. Note: if the "ENABLE_SLEEP_RQ" option is set, sys_init() will at this time set the "sleep_rq" pin to low.

The next function which is called takes different steps beyond the scope of this Wiki article, depending on whether it is configuring an S2B or S2C module. In short, a call is made to *radio_switch_to_api_mode_and_autorate()* which uses a baud rate array structure and various calls in "uart.c" to determine the XBee radio's baud rate.

After this function returns, the sys_init() function then starts a process to initialize the XBee radio by first calling *xbee_dev_init()*. This function is responsible for setting up the structure needed to hold the various configuration options for the radio, but contrary to the name of the function, does not yet start communication to the XBee radio itself.

When xbee_dev_init() returns, program flow then continues on by calling xbee_cmd_init_device(), which starts the XBee initialization process. Based on the results of built-in macros and checking a configuration bit, this function determines whether or not the XBee device needs to be queried to fill in a special structure called "xbee_dev_t", containing certain XBee specific parameters, such as: HV, VR, SH, SL, GT, CT, CC, EO, AI, NP, MY and others, described in the XBee Firmware Library reference documenation. If this structure needs to be filled in with values, xbee_cmd_query_device() is then called. A callback routine is set in order to handle setting up the radio specific parameters, as they are returned from the results of calling commands to get radio configuration.

After this structure has been set up for the XBee device, sys_hw_init() then continues to call *radio_gpio_init()*, which is responsible for initializing radio functionality pins, such as enabling the association LED pin, commissioning pin, Sleep_RQ, and the Sleep pin, set up as an input to allow for waking up the XBee while sleeping (if configured for certain types of power management), and finally the RSSI_PWM pin. It then disables certain pins which are user configurable and can be controlled by the Freescale CPU. These pins are different, depending on whether the XBee is an S2B or S2C device (see "hwinit.c" for more information and details).

After the radio's GPIO pins have been initialized, a call is made to *sys_irqs_disable()* so that the configuration of system GPIO can begin. A call to *gpio_init()* is then made, which in turn calls *gpio_config()* for each pin. The number and range of pins configured will depend on whether the XBee is an S2B or S2C device. In addition, *gpio_config_irq()* is called to configure IRQ driven pins, *port_config()* is called to configure ports 0-4, and the ADC mechanism (Analog to Digital Converter) is also configured, thereby enabling all appropriate ADC channels for the radio.

From here, sys_hw_init() finishes up by enabling the UART, FLASH memory, Analog to Digital Converter, and any other hardware functionality and interfaces which the programmer has specified to use, such as the i2c and 1-wire interfaces, etc. At the end, sys_hw_init() finishes up by re-enabling the system IRQs which were disabled previously by making a call to *sys_irqs_enable()*.

Step 2: sys_xbee_init()

After the long initialization phase of the sys_hw_init() function, the XBee radio should now be in a state to be initialized further. This happens first by a call to *xbee_params_init()*, which initializes

parameters for the XBee, including ID, NI, NT, SC, SD, SM, SN, SP, ST, SO, WH, PO, DD and A0. A final call is made to *xbee_wpan_init()*, which performs the ultimate initilization and start of the WPAN layer of the XBee device. This is the layer which is responsible for enabling things such as endpoints and clusters, and other implementation needed by the ZigBee layers, including the ability to transmit basic data frames.

After this step, the driver attempts to query the XBee device by calling *xbee_device_tick()*, *sys_watchdog_reset()*, and finally *xbee_cmd_query_status()*, to verify the proper initialization previously performed in earlier steps (init functions).

Step 3: (Step 3): sys_app_banner()

After sys_xbee_init() is finished, a final call is made to sys_app_banner(), which prints out information primarily for debugging purposes such as the radio's 64-bit address, the application version string, and other XBee version information. In addition, hardware address and hardware version is also reported, as well as the PAN ID.

Step 4: The "for-loop"

From here, the Freescale CPU enters the typical "for-loop" which runs "forever" until power to the programmable XBee is turned off. This type of loop is typical for CPU/MCU devices, so that it runs in a continuous loop, while servicing the needs of the device (the XBee radio in the case of the programmable XBee).

Using an additional XBIB from a different kit

Can I use an X-BIB from a different kit in the Getting Started excercises?

If you’re following the Help documentation for the S2B Programmable XBee “Getting Started” topic, it’s a good idea to go through step #5, “Building an advanced application”.

For this step, you will need two programmable XBee modules, which are usually included in the kit. The kit does not come with two X-BIB boards for this exercise. However, you can use another X-BIB from a different kit you may already have in your possession. This is not necessary for programming the each of the XBee S2B modules for the exercise, but they are necessary for testing purposes when running the tutorial once you have finished. There are a couple of things worth noting.

Depending on the kit that you get your second X-BIB device from, there are some differences which when pointed out, make the sample exercise go a lot smoother.

First, if this is the USB connected X-BIB (much easier to use since it is self-powered through the USB connection and does not require an extra power source brick), you’ll find the buttons are not mapped out in the same way that is listed in the tutorial. The primary difference is in the DIO lines that the switches are connected to.

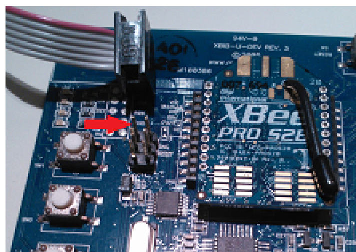
Switches and the Silk Screen Labels

If you follow the instructions in the manual and use the exact same code to program the second XBee S2B unit (not changing pin assignments), you will be using button number 2, which is labeled on the USB X-BIB as “SW2”. (Note: this is different than the X-BIB that usually comes with the programmable XBee kit, which is button 3 (“SW3”).

When following the sample found in the S2B Programmable XBee tutorial, you can keep both XBee S2B modules in each X-BIB, for programming. You will simply load the sample application and after programming the XBee S2B module using the X-BIB that is shipped with the Programmable XBee kit, you can then connect the debugger directly to your extra, USB connected X-BIB, with the second programmable XBee module installed in the socket.

Program Header may not indicate Pin 1

The second thing you may notice is that the silk screen for the debug header does not always indicate pin 1 with the typical diamond shape symbol or arrow. In order to successfully connect the programmer to this version of the USB connected X-BIB, you will align the red wire of the programmer with the first pin on the header, starting at the first pin on the top left, with the X-BIB board oriented so that the LED stack and USB connection are at the bottom, and the silk screen lettering on the board are facing correctly and not up-side down.



The debugger header is labeled “J2”, with the first pin being VCC and the second pin being ground (“gnd”).

Once the second XBee S2B module has been successfully programmed, you may then proceed with the testing of the “Knight Rider Blink” application.