Remote Command Interface (RCI)

Reference

Contents

RCI device implementer notes

Legacy RCI

RCI descriptors

RCI descriptor reference

Remote Command Interface (RCI) Specification

Remote Command Interface (RCI) provides a specification for remote configuration, control, and information exchange between an RCI client and RCI target. This guide provides RCI reference information.

About RCI

Remote Command Interface (RCI) provides a specification for remote configuration, control, and information exchange between an RCI client and RCI target. Typically, an RCI client is a web services client acting via Digi Remote Manager®, and an RCI target is a Digi device implementing the RCI specification.

RCI consists of a transport mechanism, such as the Remote Manager 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 devices
- Device file system operations (list, get, put, delete)
- Send requests and retrieve replies from dynamically registered agents such as Python programs
- Returns the device to factory default settings. The definition of factory defaults is determined by the device implementer.

RCI requests are sent to devices via Remote Manager and are wrapped in a server request, called SCI.

- For more information on SCI, see SCI (Server command interface).
- For details about non-Remote Manager RCI information, see Legacy RCI.

Here is an example of an RCI request and reply that queries a device for its system configuration:

```
<rci_request version=”1.1”>
  <query_setting>
    <system/>
  </query_setting>
</rci_request>

<rci_reply version=”1.1”>
  <query_setting>
   <system>
    <description>NewCos Device</description>
    <contact>Joe Thompson</contact>
    <location>Building 30-2</location>
   </system>
  </query_setting>
</rci_reply>
```

RCI protocol

RCI is made up of two parts:

- XML documents containing requests and replies
- Transport layer over which requests and replies are exchanged between the server and device

Request and reply XML documents

RCI exchanges data between clients and devices using XML documents. All RCI XML documents are well-formed XML. To reduce the complexity of XML parsing on devices with limited capabilities, a limited set of XML parsing requirements are enforced in an RCI-capable device:

- RCI requires only XML version 1.0.
- RCI uses ISO-8859-1 encoding. XML 1.0 requires XML parsers to be able to parse UTF-8. This requirement is not enforced in XML parsers used for RCI in devices. If any RCI requests are sent to Remote Manager in a different encoding than ISO-8859-1, Remote Manager transforms the request to ISO-8859-1 (other transports behavior with non-ISO-8859-1 is not defined).
- Not all predefined entities need be supported. Only the following are required:
  - &
  - <
  - >
  - "
  - '
- Entity declarations are not supported.
- Only simple StringData attribute types are required (see XML specification, section 3.3 at http://www.w3.org/TR/REC-xml/#attdecls).

Transport layer

The Transport layer is a mechanism that handles communication between a server and a device. The Transport layer specifies the initialization process, the sending and responding mechanism, the closing mechanism, any error recovery mechanism needed, as well as security. The Transport layer for Remote Manager is EDP. EDP is described in the EDP Specification.

For other RCI transports, see Legacy RCI.

Terminology

This document uses the Extensible Markup Language (XML) language and definitions. For more information, go to http://www.w3.org/TR/REC-xml/

Commonly used terms include:

- **Element tag:** An XML element tag is dest in this example: <dest index="23">
- **Attribute:** An XML attribute is the index="23" in this example: <dest index="23">, where index is the name of the attribute, and "23" is the value of the attribute. Note the double-quote characters are required.
- **RCI requests and responses:** An RCI client is the originator of an RCI request. A client sends a request to a device and a device responds to a client.

Related documentation

- EDP Specification—Describes the Remote Manager device-to-server protocol
- ADDP Specification—Local device discovery and configuration

- Digi Remote Manager Programmer Guide
- Digi Remote Manager User Guide
- XBee RF module user guides

RCI document structure

An RCI XML document is identified by the root XML elements: <rci_request> or <rci_reply>.

<rci_request>

An RCI request specifies the XML element tag <rci_request> with an optional version number. The version number should match the version of RCI the client expects.

The current RCI version number is 1.1. If a version number is not specified, the RCI version number of the device is used to form the reply.

Here is an example request element:

```
<rci_request version="1.1">
```

A client forms an RCI request by building an XML document with <rci_request> as the root element. The content of the request is a supported command with command specific content. For command descriptions, see RCI commands .

<rci_reply>

The device parses the RCI request, performs the requested action, and forms a reply.

An RCI reply specifies the element tag <rci_reply> along with the version number as an attribute. For example:

```
<rci_reply version="1.1">
```

An RCI reply is an XML document that is structured in the same way as the RCI request. If the request was successful, the RCI reply contains no <error> elements (see RCI errors and warnings). The response document echoes the same structure as the request document, even when the request is a command that does not return data. This is required so that a client can confirm the command was executed successfully; it is also a means for the client to match up sub-command completion in a compound request.

The following example demonstrates this request/reply symmetric relationship. The <info> element is data being added by the device in response to the request. This addition of <info> shows the symmetry is not exact, but rather at the container level, where the <c> element is considered a container.

```
<rci_request version="1.1">
  <a>
    <b>
      <c/>
    </b>
  </a>
</rci_request>
```

```
<rci_reply version="1.1">
  <a>
    <b>
      <c>
        <info>returned</info>
    </b>
  </a>
<rci_reply>
```

RCI commands

The command section of the protocol indicates the action requested or the action performed in replies.

Supported commands

The following table summarizes the required commands in RCI.

Devices may support additional commands. These custom commands will be reflected in a device's RCI descriptor.

Command	Request description	Reply description
<query_descriptor>	Request device capabilities.	RCI descriptor.
<query_setting>	Request for device settings.	Returns requested settings.
<set_setting>	Set settings specified in setting element.	Empty setting groups returned as confirmation of set. See RCI errors and warnings.
<set_state>	Set the device state.	Same semantics as set_setting.
<query_state>	Request current device state, such as statistics and status. Sub-element may be supplied to subset results.	Returns requested state.
<set_factory_default>	Sets device settings to factory defaults. Same semantics as set_setting.	Same semantics as set_setting.
<reboot>	Reboots device immediately.	Confirm reboot command.
<do_command>	Send a request to a subsystem specified by the target element.	Response from subsystem.

Compound commands

An RCI request can contain more than one command. The device replies with its command responses concatenated together in one reply in the order in which they appear in the request. Here is an example of a compound command:

```
<rci_request version="1.1">
  <command_1><command_specific_info/></command_1>
  <command_2><command_specific_info/></command_2>
  <command_3><command_specific_info/></command_3>
</rci_request>
```

```
<rci_reply version="1.1">
  <command_1><command_specific_response_1/></command_1>
  <command_2><command_specific_response_2/></command_2>
  <command_3><command_specific_response_3/></command_3>
</rci_reply>
```

Command rules

While the contents and structure of RCI commands vary, there are some rules that are common to all commands.

Specifying commands

Commands are specified as a child element to <rci_request> and <rci_reply>.

This example requests all configuration settings:

```
<rci_request version=”1.1”> <!—Identifies protocol & request vs reply-->
   <query_setting/>        <!— “query_setting” command -->
</rci_request>
```

This example requests the configuration information for system and serial settings only. The valid children of <query_setting> are determined by the command.

```
<rci_request version=”1.1”>
  <query_setting>
    <system/>
    <serial/>
  </query_setting>
</rci_request>
```

XML constraints

If an element has a child element, it cannot also have character data as a child.

This following structure is allowed:

```
<command_1>
  <data>character data ok</data>
</command_1>
```

But this structure is not allowed:

```
<command_1> illegal character data
  <data>character data ok</data>
</command_1>
```

Data collections

When there is more than one set of data in a command, an attribute is used to uniquely identify an item. Use either of the following data structures:

- An array-like entry, using the attribute index.
- A hash map-like entry, using the attribute name.

Arrays

Arrays start at index 1 and continue to a stated maximum, as listed in the RCI descriptor. If there is a natural mapping of an array, it should be documented in the RCI descriptor.

For example, if a device has two serial ports, then the index maps to a serial port. Serial ports are selected as follows:

```
<serial index=”1”/>
<serial index=”2”/>
```

If a data element uses index as an attribute, as specified in its RCI descriptor, it can be assumed that the data element is an array.

If a data element is an array type and the index attribute is not specified, index=”1” is implied on operations that act on an instance, such as a <set_setting>.

Dictionaries

Dictionaries identify data instances by the attribute name. If there is a list of allowed names for a data element, that list of names is specified in the RCI descriptor.

Dictionary data type example:

```
<interface name=”eth0”>
  <ip>1.1.1.1</ip>
</interface>

<interface name=”wln0”>
  <ip>2.2.2.2</ip>
</interface>
```

If a data element declares an attribute named “name," the data element is a dictionary.

The name attribute is required for dictionary data. If a data element is a dictionary and a name attribute is not specified on an instance operation, such as a <set_setting>, an error is returned.

RCI command reference

<query_descriptor>

Retrieves the RCI descriptor from a device. The RCI descriptor describes all RCI commands supported on a device, as well as all children and contents of those commands. For more information on the RCI descriptors, see About RCI descriptors.

Although recommended, support for the <query_descriptor> command is optional on a device. However, RCI descriptors are required for a device managed by Remote Manager. If the <query_descriptor> command is not supported on a device, a device maker must arrange to manually push the RCI descriptor up to the server.

Attributes

None.

<query_setting>

Requests configuration parameters from a device. A configuration is split into three separate sources:

- **current:** Current settings running in the device. The concept of current can vary device by device and setting group by setting group. For instance, the current settings for a serial port are the initial settings of the port (the application opening the port can change the serial port settings via IOCTLs once it has opened the port—this temporal change happens outside of the configuration mechanism). Once the port is open, setting the current settings for serial changes the initial settings, not the actual running settings of the open port.

  When current configuration settings differ from running values, we recommend exposing the current values using a <query_state> command group.
- **stored:** Value used on the next device reboot.
- **default:** Value present following a <set_factory_default> command.

Sending a <query_setting> without children returns all configuration information for a device, and the complete device configuration is returned. You can use a full <query_setting> reply as a backup of the full configuration of a device.

The settings returned by <query_setting> are arranged in a set of groups called setting groups which logically group associated configuration settings. The children of <query_setting> are setting groups.

The children of settings groups are field value pairs which make up the actual configuration. Values are typed and are declared in the RCI descriptor. See About RCI descriptors for a list of available types.

Optionally, field value pairs can be grouped together when appropriate. These groupings are called a list. Alternatively, if an element X inside a setting group has child elements, then element X is a list. Lists should not be nested more than one deep.

Attributes

The following attributes are optional but recommended in device implementations. If a device does not support an attribute, it must behave as if the attribute is ignored and not return an error solely because of an unknown attribute.

Supported attributes must be declared in the RCI descriptor. Devices can also add other attributes specific to their device. All attributes must be declared in the RCI descriptor.

source

The source attribute lets a user request the settings from a particular source. Supported source values:

- **"current"**: Current running settings.
- **"stored"**: Configuration stored persistently. This is the configuration that will be used by the device if it is rebooted.
- **"defaults"**: Default configuration of the device. This is the configuration that will be used if <set_factory_default> is issued.

The default is **"current."**

compare_to

The compare_to attribute works with the source attribute to return only the differences from the compare_to settings and the settings source specified in source. For example, to return only the settings that are different from defaults, issue this request:

```
<rci_request version="1.1">
  <query_setting source="current" compare_to="defaults"/>
</rci_request>
```

Supported compare_to values include:

- **"none"**: No difference requested. Return all values as specified in source.
- **"current"**: The current running settings.
- **"stored"**: The configuration stored persistently. This is the configuration used by the device if it is rebooted.
- **"defaults"**: The default configuration of the device. This is the configuration used if <set_ factory_default> is issued. The default value is compare_to="none"

Example:

```
<rci_request version="1.1">
  <query_setting/>
</rci_request>

<rci_reply version="1.1"
 <query_setting>
  <serial index="1"> <!—setting group "serial", serial port# 1-->
   <baud>300</baud>   <!—field, value pair (baud, 300) -->
   <stop>2</stop>
  </serial>
   <serial index="2">
   <baud>1200</baud>
   <stop>1</stop>
  </serial>
  <routing>    <!—setting group "routing", index # 1 implied -->
   <in_ip>1.2.3.4</in_ip>
   <acl index="1">    <!—ACL is a list, array position #1 -->
    <allowed>tom, john</allowed>
    <deny>fred</fred>
  </acl>
```

```
    <acl index="2">
     <allow>bill</allow>
     <deny>betty</deny>
   </acl>
   </routing>
  </query_setting>
<rci_reply>
```

<set_setting>

The <set_setting> command is used to set device configuration. It follows the same structure as <query_setting>.

Attributes

The following attributes are optional but recommended. If a device does not support an attribute, it must behave as if the attribute is ignored. Supported attributes must be declared in the RCI descriptor. Devices can also add other attributes specific to their device. All attributes must be declared in the RCI descriptor.

action

Specifies when the <set_setting> should take place. Values include:

- **"immediate":** Set is applied to the current running settings. This is equivalent to the source="current" in <query_setting>.
- **"deferred":** Set is applied to NVRAM settings but not to current settings. This is equivalent to source="stored" in <query_setting>.

encrypt

Supported values:

- **"1":** Use encrypt type "1". Instructs the device to return sensitive information in encrypted form. The device determines which fields are deemed sensitive. In general, sensitive fields include passwords and keys. Any field that is returned encrypted is marked with an attribute encrypt="1" in the reply. The encrypt attribute is also declared in the descriptor for any field that may be returned this way. The actual encryption method is not specified. The caller treats the value as opaque. The only use of an encrypted value is as a backup of configuration that will later be set to the device.
- **"none":** Do not encrypt fields. If encrypt="none" is specified, it is recommended that sensitive fields are not returned at all.

The default is encrypt=**"none"**.

Required handling of read-only settings

A warning (not an error) must be generated if a setting marked access="read_only" is sent as the contents of the <set_setting> command. This allows an RCI client to use a full result of a <query_setting> in a <set_setting> command.

<query_state>

The <query_state> command requests and returns information about the current state of the device. Device state includes: statistics, device info, and transient events, such as GPS coordinates. The format of the groups returned by <query_state> is the same as for <query_setting>.

Attributes

None.

<zigbee> element

The <zigbee> element of the <query_state> command is used to return information about the XBee RF module installed in the gateway. For all other XBee nodes in the network, see <do_command target="zigbee">.

Example request and reply

```
<rci_request version="1.1">
 <query_state/>
</rci_request>

<rci_reply version="1.1">
<query_state>
 <zigbee>
   <gateway_addr>01:23:45:67:89:ab:cd:ef!</gateway_addr>
   <caps>0x1234</caps>
   <field>value</field>
   ...
 </zigbee>
 ...
</query_state>
</rci_reply>
```

gateway_addr

The 64-bit extended device address of the XBee RF module on a gateway.

caps

A bitmap of gateway capabilities. See also Calculations and terms used for testing the "xbee_type" conditional.

Capability	Bitmap specification
Advanced addressing; that is, the ability to send and receive to any endpoint and cluster ID on remote nodes. Without advanced addressing, only a single endpoint and cluster ID (for example a remote serial port) can be used on each node.	ZB_CAP_ADV_ ADDR=0x00000001
Ability to access XBee device objects.	ZB_CAP_ ZDO=0x00000002

Capability	Bitmap specification
Ability to access Digi device objects on remote nodes. DDO access to the gateway radio is always available.	ZB_CAP_REMOTE_ DDO=0x00000004
Ability to update firmware on the gateway radio.	ZB_CAP_GW_ FW=0x00000008
Ability to update firmware on remote nodes via the gateway.	ZB_CAP_REMOTE_ FW=0x00000010
Supports XBee mesh networking.	ZB_CAP_ZIGBEE =0x00000020
Supports XBee Pro/2007 mesh networking.	ZB_CAP_ ZBPRO=0x00000040
Supports DigiMesh networking.	ZB_CAP_ DIGIMESH=0x00000080
Supports parent/child relationship.	ZB_CAP_CHILDREN 0x00000100
Supports XBee Smart Energy profile.	ZB_CAP_ SE=0x00000200

field

Each state parameter of the gateway radio. These are the same values returned by the XBee <query_ state> command.

<set_state>

Sets the temporary running state of the device. Because <set_state> does not set device configuration, it is rarely used. To set device configuration, use the <set_setting> command. Examples of using the <set_state> command include: setting the current time on a device or setting the current output voltage of a GPIO pin.

The format of the groups returned by <query_state> is the same as for <query_setting>.

Attributes

None.

<set_factory_default>

WARNING! Executing the **set_factory_default** command can result in unintended side effects and its use must be considered dangerous. For example, Remote Manager server information will most likely not be the default information saved. Therefore, executing this command through Remote Manager causes the device to lose connection to the server and local action will be required to recover.

Returns the device to factory default settings. The definition of factory defaults is determined by the device implementer.If the source="defaults" attribute is supported on <query_setting>, executing <set_factory_default> must take an action that matches the settings returned by source="defaults".

A list of groups may optionally be provided as children to <set_factory_default>. If any groups are present, only those groups will be returned to the factory defaults. If a device does not support group level defaults, the device must detect that groups have been specified and must return an appropriate error indicating that the group-wide factory default is not supported and no action was taken. This must also be documented in the RCI descriptor by not specifying any child elements of <set_factory_default>.

Attributes

action

Modifies the behavior of the <set_factory_default> command in the specified manner. This attribute is only valid for full <set_factory_default> commands; that is, this attribute is ignored if any groups are specified as children of the command.

Values include:

- **"factory":** Command takes effect immediately and the device immediately reboots.
- **"revert":** Command resets stored configuration to defaults, but current settings are not changed. Changes take effect on the next boot.
- **"erase_user_flash":** All device configuration is reset to factory defaults in NVRAM. In addition, all user flash is erased. All files are erased including: Python programs, all customization files including custom defaults, and all XBee firmware files stored in the file system. Only on-board NVRAM is erased. USB drives are not erased.

The default is action=**"factory"**.

<reboot>

Reboots the device immediately after replying to the RCI request.

Attributes

None.

<do_command>

The <do_command> is a wrapper command. It passes its contents to the sub-system specified by the target attribute. It is commonly used for file system commands, XBee or ZigBee commands, and for passing commands to user-created targets, for example by dynamically registering a target in a Python program.

The content of the <do_command> is determined by the target. The only limitation is that it must be well-formed XML and must adhere to the limitation set forth in this specification. However, any data, including binary data, may be passed inside a <do_command> by encoding it in base-64.

All static targets supported on a device must be documented in the RCI descriptor.

If a device supports dynamically-registered targets, its descriptor will document a target value of "*":

```
<value value="*" desc="Dynamically registered target"/>
```

Attributes

target

Specifies the sub-system, entity, or program routine that receives the enclosed command. The target attribute is required and there is no default. To pass parameters to a Python program routine using the do_command, the Python program must be loaded in the device and you must specify the registered method name as the target, rather than the program name.

<do_command target="file_system">

The <do_command target="file_system"> variant is an interface to manipulate the file system on a device. There are several subcommands:

- **<ls>**: Lists information about the contents of a directory or file.
- **<get_file>**: Returns the contents of a file as a base-64-encoded block.
- **<put_file>**: Uploads the contents of a file as a base-64-encoded block.
- **<rm>**: Deletes a file.

If a device supports more than one volume, for RCI purposes, the volumes should be considered mounted to a root dir: "/" and listing of "/" returns all volumes available. Directories are specified with leading slashes ("/"). If a path has a trailing slash, the path must specify a directory. If no trailing slash is specified, the path may be a file or a directory. The Backward slash ("\") character is not allowed.

<do_command target="zigbee">

This <do_command> variant is an interface to interact with an XBee network. The device must contain an XBee RF module.

RCI errors and warnings

RCI responses may contain RCI errors or warnings to indicate that a requested command did not complete normally.

Message type	Description
<error>	An error occurred and the operation failed.
<warning>	An RCI command was executed, but a warning was issued. Changes were made, but not all requested changes were successful.

More than one error or warning may be present in a reply. The location of the <error> or <warning> tag implies the scope of the error. The parent element of <error> or <warning> is the source of the error/warning. For example, in this RCI code, a <set_setting> command experienced an error and the command failed to execute. The caller can assume that none of the requested <set_setting> changes occurred.

```
<rci_reply version="1.1">
  <set_setting>
```

```
    <error id="1">
      <desc>Operation unavailable</desc>
    </error>
  </set_setting>
</rci_reply>
```

The following example shows that an error occurred while setting the <serial> group. An error under <serial> implies that no changes were made to the <serial> group, even if baud is the only invalid field; that is, group level sets are treated atomically. Note also that the <system/> element was returned without an error, so the <system> set was successfully applied.

```
<rci_reply version="1.1">
 <set_setting>
   <serial>
    <error id="2">
      <desc>Invalid baud"</desc>
      <hint>baud</hint>
    </error>
   </serial>
   <system/>
 </set_setting>
</rci_reply>
```

This example shows the same situation, except a <warning> is returned instead of an error. This indicates to the caller that the <serial> group was changed as requested, but a problem was encountered and the changes may not be as expected. Again, the requested <system> set was successful.

```
<rci_reply version="1.1">
 <set_setting>
  <serial>
   <warning id="2">
     <desc>Invalid baud</desc>
     <hint>baud</hint>
   </warning>
  </serial>
  <system/>
 </set_setting>
</rci_reply>
```

This example shows a similar error, except that the <error> is more precisely placed as a child of the <baud> field. Note, however, that since the <baud> field encountered an error, the entire serial group does not get saved. Generally, if an error occurs inside a setting group, the entire group is not saved. The <set_setting> command is special in this way. Usually, if an <error> occurs as a child of a command (or any descendent) the command fails and changes do not take place.

```
<rci_reply version="1.1">
 <set_setting>
  <serial>
   <baud>
     <error id="2">
       <desc>Invalid baud</desc>
     </error>
   </baud>
  </serial>
  <system/>
 </set_setting>
</rci_reply>
```

RCI device implementer notes

The <query_setting> command and the compare_to and source attributes, if implemented, must handle internal_defaults as a valid value. internal_defaults should be treated identically to defaults for most implementers.

internal_defaults is reserved for server use and is not valid for use by general RCI clients.

Legacy RCI

The following RCI information concerns the original RCI implementation in Digi devices. It is included here for completeness only.

RCI over HTTP

The primary RCI transport is HTTP through the embedded web server. The web server provides the initialization, receiving and sending, and security.

RCI requests are sent to the device using an URI of UE/rci. For example, if the Digi device IP address is 192.168.1.1, then RCI requests are sent to the following:

```
http://192.168.1.1/UI/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. Any requests larger than 32 KB 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. See the samples shipped with devices for examples of RCI over HTTP RCI.

Standard HTTP errors will be returned for HTTP related problems. Common HTTP errors that should be handled by clients, for example:

413 – Buffer too large. Usually caused by sending a request larger than 32KB in size.

RCI over serial

RCI requests can be sent over the serial port, known as RCI over serial. This option is useful in scenarios where a master processor is connected to the Digi device through a serial port. It 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. The RCI over Serial option is available only on the primary port of the Digi device.

You must enable 'RCI over Serial' in either the Digi device's web or command line before the Digi device will accept RCI requests and return replies.

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 RCI over serial from the command-line interface (CLI)

1. Access the command-line interface using telnet or rlogin and the module's IP address. For example:

   ```
   telnet 192.168.1.2
   ```

   or

   ```
   rlogin 192.168.1.2
   ```

2. At the command prompt, type:

```
> set rciserial state=on
```

Configure RCI over serial from the web interface

1. Access the web interface by entering the module IP address in a browser 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 descriptors

About RCI descriptors

An ongoing problem consumers of RCI have is that although RCI specifies a format for requests and replies, it does not mandate a complete command set nor does it dictate what is allowed to be configured via RCI; for example, whether a device allows configuration of IP passthrough.

These limitations result in knowledge needing to be built into RCI clients about specific RCI targets, including firmware version-by-version differences, product-by-product differences, and so on. RCI also does not specify firmware version explicitly, so every client must know about every possible RCI target type and specific target variations.

Some observed problems with the current RCI implementation include:

- RCI clients must be manually kept up to date with changes to RCI targets. If RCI targets change frequently, the RCI client must change very frequently or will become out-of-date and progressively less useful. This is exactly what has happened with the legacy Remote Manager management interface with respect to configuring NDS based devices.
- Documentation must be manually generated that attempts to document the features of RCI targets. This is a very manual, intensive process, resulting in infrequent updates; as a result, documentation tends to be extremely out of date. This results in a significant mismatch between what users see in a device and what the documentation covers. Users must rely on the documentation since there is no other way for them to use RCI, resulting in a very poor user experience.
- Testability is lacking. Thorough testing of RCI is essentially not possible currently, since a tester would need to inspect all of RCI and settings implementation to figure out what has changed. Such testing is not practical.

To solve these problems, RCI descriptors are introduced into the RCI specification.

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 SKU. This allows descriptors to be exact for a given RCI target. RCI 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 must be complete enough such that a program iterating over 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 information 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 web interface (if it has one). The look and feel and content of a generic GUI will be different than the web UI.

In addition to the goals above, other goals of descriptors include:

- Keep the design as simple as possible. Do not attempt to capture the outlier cases.
- Make exceptions to rules obvious, for example, 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.

RCI descriptors are 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 descriptors are 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 Remote Manager by EDP device type and firmware level.

Note RCI descriptors are required for Remote Manager support.

Determining whether a device supports RCI descriptors

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 example, 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="RCI request">
    <attr name="version" desc="RCI version of request. Response returned in
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>
```

Information included in RCI descriptors

Using the previous example as a model, RCI 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 the element, descriptions of the attributes, and any valid values.
- Elements supported as children of the current element. These elements are either a full descriptor themselves, or 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 missing descriptors; that is, for any dscr_avail=”true” descriptors). For example, 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).

Notes:

- This descriptor contains no <descriptor> elements with dscr_avail=”yes” attributes. This means that there are no other descriptors 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, that is:

  ```
  (<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, as shown:

  ```
  <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=”true”, the client needs to make multiple requests to gather the entire descriptor recursively until no more dscr_avail=”true” attributes are encountered. The resulting set of documents can be formed into a single descriptor by replacing each instance of <descriptor element dscr_avail=”true”> with the full descriptor retrieved, or by adding the child descriptor as a child of elements that are not <descriptor>.

For example, consider the following set of retrieved descriptors. Both requests and replies are shown.

```
<rci_request version="1.1">
  <query_descriptor/>
</rci_request>
```

```
<rci_reply version="1.1">
  <query_descriptor>
    <descriptor element="rci_request" desc="RCI request">
      <descriptor element="x" dscr_avail="true"/>
      <descriptor element="y" dscr_avail="true"/>
      <error_descriptor id="1" desc="Unknown command"/>
    </descriptor>
  </query_descriptor>
</rci_reply>
```

```
<rci_request version="1.1">
  <query_descriptor>
    <x/>
  </query_descriptor>
</rci_request>

<rci_reply version="1.1">
  <query_descriptor>
    <descriptor element="x" desc="X command">
      <element name=”q” type=”string” desc=”q item”/>
      <error_descriptor id="1" desc="X error"/>
    </descriptor>
  </query_descriptor>
</rci_reply>
```

```
<rci_request version="1.1">
  <query_descriptor>
    <y/>
  </query_descriptor>
</rci_request>

<rci_reply version="1.1">
  <query_descriptor>
    <descriptor element="y" desc="Y command">
      <attr name=”option”>
        <value name=”z” dscr_avail=”true”/>
      </attr>
      <error_descriptor id="1" desc="Unknown Y command"/>
    </descriptor>
  </query_descriptor>
</rci_reply>
```

The replies can be combined together into a single descriptor, as shown:

```
<descriptor element="rci_request" desc="RCI request">
  <descriptor element="x" desc="X command">
   <element name=”q” type=”string” desc=”q item”/>
   <error_descriptor id="1" desc="X error"/>
  </descriptor>
 <descriptor element="y" desc="Y command">
   <attr name=”option”>
    <value value=”z”>
<descriptor element=”a” desc=”A group”>
   <element name=”b” type=”int” desc=”an int to configure”/>
    <error id=”1” desc=”A group error”/>
  </descriptor>
   </value>
   </attr>
```

```
    <error_descriptor id="1" desc="Unknown Y command"/>
</descriptor>
  <error_descriptor id="1" desc="Unknown command"/>
</descriptor>
```

Client requirements for parsing responses

As is the case with the rest of RCI, clients need to be prepared to parse responses that contain more information than the minimum requested. For example, 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 any of the following:

- The descriptor for do_command only
- The do_command and all of its sub-commands
- The do_command and all of its peer commands (set_setting, reboot, etc)
- The entire RCI descriptor
- 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_avail=”true” is returned. If a client requests a specific command descriptor and the device does not declare whether it supports the descriptor (with dscr_avail=”true”), it is recommended that the device return an error.

Validating settings

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 the full range of configuration available to them, even if some values are further restricted beyond what the descriptor mentions. When this occurs, 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 descriptor reference

Format for descriptions

All descriptions (desc=”Description text example”) in RCI descriptors can optionally be specified in two parts delimited by a colon, in this format:

```
“short description: extra information”
```

The short description must be at most 40 characters.

This description will be parsed by clients such that the primary and GUI-displayable label is the short description. Any extra information may be provided as a tool-tip, help text, or left as is for a single long description as determined by the client.

If a colon character is desired in the description in RCI descriptors, two colons “::” should be used. This notation will be replaced by clients as a single displayed colon.

The use of compound descriptions is optional for descriptions of commands and attributes of commands. It is required for all other descriptions when the total description length is 41 characters or more.

Encrypted fields

Some RCI implementations offer the ability to request that passwords and other sensitive fields be returned in an encrypted form. Encryption is requested by specifying the attribute encrypt=”1” on the <query_setting> element. In the reply, security-sensitive fields (as determined by the device) will be returned with encrypt=”1”. For example:

```
<simple_pw>
  <password encrypt=”1”>0x5511d15321</password>
</simple_pw>
```

The client must specify encrypt=”1” in the <set_setting> command when returning this value on a set to distinguish it from a clear text set:

```
<rci_request version=”1.1”>
  <set_setting>
    <simple_pw>
      <password encrypt=”1”>0x5511d15321</password>
    </simple_pw>
  </set_setting>
</rci_request>
```

Clients must treat the value returned as opaque. The only use of the value is to return it later. There is no way for the client to generate an encrypted value.

All fields that can be returned in an encrypted form can be found from the descriptor as parents of **<attr name=”encrypt” value=”1”>**.

The maximum length specified for these fields is the maximum length of the encrypted value. The maximum size of the unencrypted value can be calculated by the following formula for encrypt=”1”:

```
unencrypted_max_len = (max_len_in_descriptor – 2)/2 – 4
```

To determine whether a device supports the encrypt option, parse the query_setting descriptor for **<attr name=”encrypt” value=”1”>**.

<descriptor>

Purpose

This is the definition of a descriptor. <descriptor> contains metadata about the element named in the element attribute. <descriptor> also list all available attributes supported by that element as well as supported nested <descriptor> children.

Attributes

element

Name: element

Value: XML element name

Description: A valid XML element that should conform to standard RCI naming conventions. This is the name of the RCI element being described.

Use: Required

desc

Name: desc

Value: Description text

Description: A short description for this element, suitable for displaying in a generic GUI to represent this element.

Use: Required

dscr_avail

Name: dscr_avail

Value:

- “true”
- “false”

Description: If true, indicates that a full descriptor is available and only a placeholder descriptor is present. If this attribute is present, only the name is required in the placeholder descriptor. For example, the following usage is valid:

<descriptor element=”reboot” dscr_avail=”true”/>

Use: Optional; if omitted, default is “false.”

format

Name: format

Value: A format name defined elsewhere in the descriptor.

Description: See <format_define> for more information.

Use: Optional

access

Name: access

Value:

- “read_write” (default)
- “read_only”
- “write_only”

Description: Indicates whether a described element is readable, writeable, or both.

- “read_write” (default) means the described element can be read by query commands, and can be set by set commands.
- “read_only” indicates that the described element is not settable. The described element can be read by a query command, but it cannot be set. Note that if a read-only element appears in a set_setting command, a warning must be returned, not an error. This allows the result of a full query_setting to be sent as the contents of a set_setting.
- “write_only “indicates the field is settable but not readable. The usual use of this attribute is for passwords.

Unlike most attributes, the access attribute is inheritable. This means that if this attribute is set in a parent descriptor, all children take on the same attribute.

Use: Optional. Default is “read_write”.

Allowed children

<attr>

<element>

<descriptor>

<error_descriptor>

<format_define>

<conditional>

<attr>

Purpose

Describes allowed attributes, their types, descriptions, and allowed values. <attr> can also contain nested descriptors when the attribute value results in a command that is complex enough to need a descriptor.

The parent of <attr> is the element on which the described attribute is found. For example:

```
<descriptor element=”set_setting”>
  <attr name=”option” value=”reset” desc=”Reset option for
   configuration setting”/>
</descriptor>
```

Describes the following:

```
<set_setting option=”reset”/>
```

When attributes have a distinct set of values they may take on, those values are listed using the <value> element. For example:

```
<descriptor element=”query_setting”>
  <attr name="content" desc="Content of reply should be complete
   or only differences from custom defaults" default="full">
    <value name="full" desc="Complete results returned"/>
    <value name="custom_default_difference" desc="Return only
     differences from custom defaults"/>
  </attr>
</descriptor>
```

Note <value> is a child of the <attr> for which it is a value.

If an attribute has only one value option, the following shorthand can be used:

```
<descriptor element=”do_command”>
  <attr name=”target” value=”zigbee” desc=”ZigBee commands”/>
</descriptor>
```

Which is equivalent to:

```
<descriptor element=”do_command”>
  <attr name=”target”>
    <value name=”zigbee” desc=”ZigBee commands”/>
  </attr>
</descriptor>
```

Attributes

name

Name: name

Value: Any valid attribute name

Description: Defines the name of the attribute that is being described.

Use: Required

desc

Name: desc

Value: Description text

Description: A short description of the meaning of the attribute.

Use: Required

default

Name: default

Value: Valid value

Description: The value the attribute takes on if it is not explicitly set in the RCI request.

Use: Optional; if no attribute is declared, this attribute must be present in the RCI request.

type

Name: type

Value: A valid type name (see the Descriptor types section on page Descriptor types).

Description: Restricts the value of this attribute to this type only.

Use: Optional. If not specified, the type is assumed to "enum" in which case the valid values are listed as children of <attr>.

min

Name: min

Value: A valid value of the specified type.

Description: The minimum value the attribute can take on. Minimum values are type-specific. See the Descriptor types section on page Descriptor types.

Use: Optional; minimum defaults are type-specific.

max

Name: max

Value: A valid value of the specified type.

Description: The maximum value the attribute can take on. Maximum values are type-specific. See the Descriptor types section on page Descriptor types.

Use: Optional; maximum defaults are type-specific.

value

Name: value

Value: The one allowed value for this attribute.

Description: Used to identify an attribute with a specific value. If specified, no <value> children can be present. This is a shorthand way of specifying an attribute with a name and value in one line. See the example in the main <attr> section above.

Use: Optional

Allowed children

<descriptor>

<value>

<conditional>

<error_descriptor>

<element>

<value>

Purpose

Used to describe an allowed value or range of values. The value that <value> describes can be the value of an attribute (name=”value”) or the character object that is the child of an element, such as:

```
<serial>
  <baud>300</baud>
</serial>
```

An example descriptor for this RCI is:

```
<descriptor element="serial" desc="Serial port configuration">
  <element name="baud" desc="baud" type="enum">
    <value value="300"/>
  </element>
</descriptor>
```

Attributes

value

Name: value

Value: Valid value for the type.

Description: The value of the field being described.

Use: Optional; required if the type attribute is not specified.

type

Name: type

Value: A valid Type. See Descriptor types.

Description: Specifies the type of value being described.

Use: Required if the <value> attribute is not specified. If the <type> attribute is not present, the type is assumed “enum”, meaning this value is one enumerated value.

desc

Name: desc

Value: Description text.

Description: A short description for the value.

Use: Optional. If desc is omitted, the <value> attribute will be used by clients as the description. If the <value> attribute is omitted, no description will be available.

min

Name: min

Value: A valid value of the specified type.

Description: The minimum value the attribute can take on. Minimum values are type-specific. See Descriptor types.

Use: Optional; minimum defaults are type-specific.

max

Name: max

Value: A valid value of the specified type.

Description: The maximum value the attribute can take on. Maximums are type-specific. See Descriptor types.

Use: Optional; maximum defaults are type-specific.

dscr_avail

Name: dscr_avail

Value:

- “true”
- “false”

Description: Only valid for <value> element children of <attr>. Not valid for <value> children of <element>. If true, indicates that a full descriptor is available for this attribute value.

The following shows an example of dscr_avail:

```
<rci_reply version="1.1">
  <descriptor element="y" desc="Y command">
    <attr name=”option”>
      <value value=”z” dscr_avail=”true”/>
    </attr>
    <error_descriptor id="1" desc="Unknown Y command"/>
  </descriptor>
</rci_reply>
```

This example indicates that there is a descriptor available with option=”z” which can be retrieved with the following request:

```
<rci_request version="1.1">
  <y option=”z” type=”descriptor”/>
</rci_request>
```

Use: Optional; if omitted, default is “false.”

Allowed children

<descriptor>

<element>

Purpose

Describes valid fields in a descriptor. It describes an element under the descriptor that has a type and contains a value or values. Usually, <element> is used for fields in a group, for example:

```
<descriptor element=”serial”>
  <element name="baud" desc="baud" type="enum" default="9600">
    <value value="1200"/>
    <value value="9600"/>
    <value value="115200"/>
  </element>
</descriptor>
```

Which corresponds to the following RCI snippet:

```
<serial>
  <baud>9600</baud>
</serial>
```

The particulars of the value such as type, min, max, are specified as attributes. The value referred to for an <element> is found in RCI as the child characters of that element. That is, the value is located between the start and end tags. To illustrate, in the following example, 9600 is the value between the start and end tags <baud> and </baud>:

```
<baud>9600</baud>
```

Attributes

name

Name: name

Value: Any valid element name. The name should follow RCI element naming conventions. These conventions are demonstrated in existing element names and include:

- Use underscore to separate words.
- Use lowercase only.
- Keep element names as short as possible.
- Make names as descriptive as possible while keeping them short.

Description: Defines the name of the element that is being described.

Use: Required

type

Name: type

Value: A valid Type. See the Descriptor types section on page Descriptor types.

Description: The type of value of the element being described. An element value is defined as being the character value child of the element, for example: **<baud>9600</baud>**.

Use: Required

desc

Name: desc

Value: Description text.

Description: A short description for the value.

Use: Optional. If desc is omitted, the name attribute will be used by clients as the description. If the name is not a suitable human readable name, desc should be considered mandatory.

min

Name: min

Value: A valid value of the specified type.

Description: This is the minimum the value can take on. Minimums are type specific. See Descriptor types.

Use: Optional; defaults are type-specific.

max

Name: max

Value: A valid value of the specified type. This is the maximum the value can take on. Maximums are type-specific. See Descriptor types.

Use: Optional; defaults are type-specific.

format

Name: format

Value: A valid defined format; See <format_define> on page <format_define> for more information.

Description: Include the content of the format directly after the <element>.

Use: Optional

access

Name: access

Value:

- “read_write” (default)
- “read_only”
- “write_only”

Description: Indicates whether a described element is readable, writeable, or both.

- “read_write” (default) indicates that the field can be read by query commands and can be set by set commands.
- “read_only” indicates that the field is not settable. The field can be read via query command, but it cannot appear in a set command. An error will be returned if this is attempted.
- “write_only” indicates the field is settable but not readable. This is usually only used for passwords.
- Unlike most attributes, the access attribute is inheritable. This means that if an element is a child of another element (for example, a list type), and the parent element has access=”read_only”, all children of that element are ”read_only”.

Use: Optional; default is “read_write".

default

Name: default

Value: A valid value. The value must match the type defined for this element and must be in the range, if one is specified, or must be in the set of valid values for enum type.

Description: The default value of this element. This default is used by descriptor clients to show a reasonable initial value for this field. This default is particularly useful when the descriptor is being used to generate a request to a device without having first queried the device for its actual current configuration; that is, offline-configuration).

Use: The use of a default value is optional, not required. It may be omitted in some cases such as:

- When there is no default for a field in a device.
- When the default value for a field is calculated dynamically.

units

Name: units

Value: Displayable text.

Description: Text to be used as the units for this element; for example:

```
<element name=”time” desc=”Amount of time elapsed” units=”seconds”>
```

Use: Optional

Allowed children

<attr>

<value>

<error_descriptor>

<element>

<error_descriptor>

Purpose

Describes an error or warning that can be seen in an rci_reply. See RCI errors and warnings for details on errors and warnings. <error_descriptor> is a way for a target to inform a client about all possible errors that can be seen at a particular level in an RCI reply. The parent of the <error_descriptor> defines the scope in which the error is valid.

Once a device declares its errors this way, it need only return an ID in RCI replies (a hint may also be returned). The caller can then look up the ID in the descriptor to resolve the description.

Attributes

id

Name: id

Value: Integer ID.

Description: The error ID used to identify the error.

Use: Required

desc

Name: desc

Value: Description text.

Description: Description of the error.

Use: Optional

<error_group>

Purpose

The <error_group> element allows <error_descriptors> to be grouped by name, so that error IDs can be scoped to a group by name, rather than being scoped to the parent of <error_descriptor>.

The <error_group> is scoped to its parent. This means that the error definition is valid for the RCI level at which it is defined and all descendents.

Example: Define errors common to all settings

The following example shows a way to define errors common to all settings. Note the <error_group> appears as a child of <query_setting>, which implies the group name is valid when it appears anywhere as a child of <query_setting> or any descendent of <query_setting>.

```
<descriptor element="query_setting" desc="Retrieve device configuration">

<error_group name=”setting_errors”>
  <!—Errors common to all setting groups -->
  <error_descriptor id="1" desc="Internal error (load failed)" />
  <error_descriptor id="2" desc="Internal error (save failed)" />
  <error_descriptor id="3" desc="Field specified does not exist" />
  <error_descriptor id="4" desc="Invalid value" />
</error_group>

<descriptor element="profile" desc="Serial port profile">
  <attr name="index" desc="Serial port number" type="int32" min="1"
   max="1" />
  <element name="profile_type" desc="Serial port profile" type="enum"
   default="unassigned">
  <error_descriptor id="1" desc="Profile not compatible with network
   mode" />
</descriptor

</descriptor>
```

Example: Use a grouped error

Here is an example of using a grouped error:

```
<error from="setting_errors" id="3"/>
```

<format_define>

Purpose

Offers a convenient way to define a descriptor snippet that is used in multiple places. It can be thought of as equivalent to a #define/#include pair in C. Descriptors can declare a format with <format_define> and include that format wherever the format="" attribute is supported. When included this way, the contents defined under <format_define> are included as a child of the element where the format="" is declared: For example:

```
<format_define name="x_format">
  <element name="x" type="string"/>
</format_define>

<descriptor element="group" format="x_format">
  <element name="y" type ="string"/>
</descriptor>
```

Is logically equivalent to:

```
<descriptor element="group" format="x_format">
  <element name="x" type="string"/>
  <element name="y" type ="string"/>
</descriptor>
```

Note that there is currently no way to use defined formats anywhere but as the first child under elements supporting the format attribute. For example, there is currently no way to "include" a format in the middle of a descriptor's set of children.

<format_define> can be declared anywhere as a child of a <descriptor>.

The name given to a format with the name attribute is not scoped to any level in the descriptor and can be considered a global format name. This format name can be used anywhere in the descriptor tree. Only one definition of a format name is allowed unless the definition is an exact duplicate of a previous definition.

To make it easy to immediately recognize the name as a format name, it is recommended that format names be specified using the form "name_format".

Attributes

name

Name: name

Value: Format name. Recommended form: "name_format"

Description: The name given to this format definition. This name is used to refer to this format elsewhere in the descriptor. The name is globally scoped.

Use: Required

Allowed children:

See the allowed children for <descriptor> on page .

<conditional>

Purpose

Allows a way to mark part of a descriptor as conditional based on a test. If the test is met, the <conditional> and </conditional> tags are removed, and the contents of the conditional are considered part of the descriptor. If the test fails, everything between <conditional> and </conditional> including those tags are considered removed from the descriptor.

The only conditional type currently supported is “custom”. As is true with the custom type for the <descriptor> element, the “custom” type for conditional means that the process of testing the given conditional value must be known by the client outside of the definition of the descriptor. For example:

```
<conditional type="custom" name="xbee_type" value="0x8022">
<element type="0x_hex" name="channel" desc="Operating channel"
 alias="CH" min="0x0000000b" max="0x0000001a"/>
</conditional>
```

In this example, the conditional element is of type “custom”, which means that the client must know how to decode the “xbee_type” value. The name “xbee_type” is a way to name the particular custom name used. In this example, if 0x8022 matches the value determined at runtime for this XBee node, this element is included as part of the descriptor for that XBee node.

Attributes

type

Name: type

Value: Only “custom” is currently supported.

Description: The type of test to use to determine the state of the condition. “custom” means the method for determining the condition is not defined in the descriptor, and must be known by the client by means other than the descriptor.

Use: Required

name

Name: name

Value: Name of conditional custom test.

Description: This is a way to help indicate to the RCI consumer which specific conditional to use. The name has no meaning to the rest of the descriptor.

Use: Optional

value

Name: value

Value: The value used to test the condition. The meaning of the value is determined by the type.

Description: The value used to test the condition.

Use: Required.

Allowed children:

No restriction.

Descriptor types

Descriptor types are used to indicate the value type of an attribute or value. The type indicates to the descriptor user which restrictions to place on a value, and how to interpret a value received from RCI.

“none”

Type: “none”

Description: This type specifies that there is no data associated with this element.” none” is typically used when the attributes fully describe the field so there is no character data. For example, for RCI code with this format:

```
<file name="python.zip" size="100101"/>
```

The descriptor snippet for the above RCI code is:

```
<element name="file" type="none">
  <attr name="name"/>
  <attr name="size"/>
</element>
```

min: Minimum is not meaningful for this type. It should not be used, but if it is present it must be ignored.

max: Maximum is not meaningful for this type. It should not be used, but if it is present it must be ignored.

“string”

Type: “string”

Description: String value. For example: “George”

The “string” type should be used for single-line strings. If a string contains a line-feed and/or carriage-return, multiline_string should be used instead.

The GUI renders string values as a text entry control. If new-lines are present, they will get converted to another character, usually a space.

min: The minimum length of the string. Default is 0, meaning an empty string. An example of an empty string is:

```
<match_value></match_value>
```

max: The maximum length of a string. There is no default maximum for string type so max must always be specified.

“multiline_string”

Type: “multiline_string”

Description: A string value that may be more than one line long.

“multiline_string” will be rendered by the GUI as a text area that allows a user to form multiple lines by pressing Enter. If a field cannot or should not handle embedded multiple lines, use the “string” type.

For example, given this multiline string:

```
 “This is a string with multiple lines:
- Line Two
- Line Three
”
```

The corresponding RCI code is:

```
<example> This is a string with multiple lines:
- Line Two
- Line Three
</example>
```

Note the separate lines directly inserted in the text. Because of this, care must be taken to not change the formatting when using an XML parser to parse or generate RCI. Text format is preserved in XML, but some parsers give the option to “pretty print” XML that results in structure change.

min: The minimum length of the string. Default is 0, meaning an empty string. Example of an empty string:

```
<match_value></match_value>
```

max: The maximum length of a string. There is no default maximum for string type, a so maximum must always be specified.

“password”

Type: ” password”

Description: Functions much the same as a string descriptor type, except that the user should treat the value as a password. In particular, the usage should show “*” instead of echoing back characters as they are typed in.

In general, passwords are not returned in query results, and are only settable.

See Encrypted fields for information about the encrypt option.

min: See the discussion of minimums for “string”.

max: See the discussion of maximums for “string”.

“int32”

Type: “int32”

Description: 32-bit signed integer value.

min: Minimum valid number. The smallest min is MIN_INT (-2147483648). For backward compatibility, the default is 0.

max: Maximum valid number. Default is MAX_INT (2147483647).

“uint32”

Type: “uint32”

Description: 32-bit unsigned integer value.

min: Minimum valid number. Default is 0.

max: Maximum valid number. Default is MAX_UINT (4294967295).

“hex32”

Type: “hex32”

Description: Same as “uint32”, except the value is represented as hexadecimal instead of decimal.

min: Minimum valid number. Default is 0.

max: Maximum valid number. Default is MAX_UINT (FFFFFFFF).

“0x_hex32”

Type: “0x_hex32”

Description: Same as “hex32”, except the value is prefixed by “0x”.

min: Minimum valid number. Default is 0x0.

max: Maximum valid number. Default is MAX_UINT (0xFFFFFFFF).

“float”

Type: “float”

Description: A floating-point number. Equivalent to ANSI C float.

min: Minimum valid number.

max: Maximum valid number.

“enum”

Type: “enum”

Description: One of a defined set of values is allowed. The defined set of values is listed with <value> elements. For example:

```
<element name=”baud” type=”enum”>
  <value value=”300”/>
  <value value=”9600”/>
</element>
```

The values 300 and 9600 are the complete set of allowed baud rates.

min: Minimums are not meaningful for “enum”. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful for “enum”. They should not be used, but if they are present they must be ignored.

“enum_multi”

Type: “enum_multi”

Description: Similar to “enum”, except that the field may take on 0 or multiple values instead of one and only one.

For a value, 0, 1 or more than one value are allowed. When two or more values are specified, they are delimited by commas. For example, given this element:

```
<element name=”protocols” type=”enum_multi”>
  <value value=”tcp”/>
  <value value=”udp”/>
  <value value=”icmp”/>
</element>
```

The following are valid RCI snippets for this element:

```
<protocol>tcp</protocol>
<protocol>tcp,udp</protocol>
<protocol></protocol>
```

But the following is not allowed:

```
<protocol>tcp,tcp</protocol> <!— entries must be unique -->
```

min: Minimums are not meaningful for enum_multi. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful for enum_multi. They should not be used, but if they are present they must be ignored.

“on_off”

Type: “on_off”

Description: Identical to an enum type that has only two values: “on” and “off”.

min: Minimums are not meaningful for on_off. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful for on_off. They should not be used, but if they are present they must be ignored.

“boolean”

Type: ”boolean”

Description: Identical to an “enum” type that has only two values: “true” and “false”.

min: Minimums are not meaningful for “boolean”. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful for “boolean”. They should not be used, but if they are present they must be ignored.

“ipv4”

Type: “ipv4”

Description: Valid IPv4 address of the form: aaa.bbb.ccc.ddd.

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“fqdnv4”

Type:” fqdnv4”

Description: Valid IPv4 address of the form: aaa.bbb.ccc.ddd or a valid DNS name (name or name.domain.com).

min: The minimum length of the DNS name. Ignored for IPv4 addresses.

max: The maximum length of the DNS name. Ignored for IPv4 addresses

“fqdnv6”

Type: “fqdnv6”

Description: Valid IPv6 address, IPv4 address, or a valid DNS name (name or name.domain.com).

min: The minimum length of the DNS name. Ignored for IP addresses.

max: The maximum length of the DNS name. Ignored for IP addresses.

“multi”

Type: “multi”

Description: The type is a composite of types. The valid types are listed with <value> elements. To illustrate, this example allows all valid IPv4 addresses and a special value of “disabled” for the element <server_address>:

```
<element name=”server_address” type=”multi”>
  <value type=”ipv4”/>
  <value value=”disabled” desc=”Service disabled”>
</element>
```

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“list”

Type: ” list”

Description: Instead of a single value, this element contains a set of elements. Only valid for <element>. Not valid for <attr>. For example, given this descriptor:

```
<descriptor element=”udp_dest”>
  <element name="dest" desc="Destination" type="list">
    <attr name="index" desc="UPD destination list" type="int"
min="1" max="64"/>
    <element name="state" desc="Destination enabled/disabled"
type="on_off" default="off"/>
```

```
    <element name="desc" desc="Destination description"
type="string" default="" max="31"/>
  </element>
</descriptor>
```

The following is valid RCI based on the example descriptor:

```
<udp_dest>
  <dest index=”3”>
    <state>on</state>
    <desc>Server 1</desc>
  </dest>
  <dest index=7”>
    <state>on</state>
    <desc>Server 1</desc>
  </dest>
</udp_dest>
```

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“raw_data”

Type: “raw_data”

Description: Base-64 encoded data. The generic GUI will prompt users for a file upload/download for this type of value.

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“xbee_ext_addr”

Type: “xbee_ext_addr”

Description: A valid XBee 64-bit address of the form: 00:11:22:33:44:55:66:77!

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“file_name”

Type: “file_name”

Description: A fully qualified, valid path and file name on the device. For example, on a ConnectPort X8, a valid file name would be **/WEB/python/hello_world.py**.

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“mac_addr”

Type: “mac_addr”

Description: A valid MAC address of the form: 00:11:22:33:44:55.

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

“datetime”

Type: “datetime”

Description: A representation of date and time. This format follows the ISO 8601 standard for date and time representation. The format is:

```
YYYY-MM-DDTHH:MM:SStz
```

Where:

YYYY: Year

MM: Month

DD: Day

T: The separator between date and time

HH: Hours in 24-hour clock

MM: Minutes

SS: Seconds

tz: Time zone. Specified in either of the following formats:

[+,-]HHMM

or

Z for Coordinated Universal Time (UTC).

Examples:

2002-05-30T09:30:10-0600

2002-05-30T015:30:10Z

Notes:

- Time zone is optional but recommended on queries.
- If the time zone is missing, the time is assumed to be the local time of the response side of the RCI request-response; that is, the device.
- If “datetime” is used to set time, it is recommended to not include time zone.
- This “datetime” format is the same as the standard XML “datetime xs:datetime”, with the exception of the time-zone format. In xs:datetime, the time zone is represented as -/+HH:MM. This format omits the : character.

min: Minimums are not meaningful. They should not be used, but if they are present they must be ignored.

max: Maximums are not meaningful. They should not be used, but if they are present they must be ignored.

conditional custom name="xbee_type"

The descriptor for the <do_command target="zigbee"> uses a conditional custom name of "xbee_type". This section describes the calculations to use the "xbee_type" conditional.

The value returned in the conditional is a bit mask. To use this value, a comparison value is computed as specified below and bitwise ANDed with the bit mask. If the ANDed value equals the original, the condition is true.

For more information about the XBee terms and concepts used in this section, see the XBee RF Module Product Manual for the particular RF module model in the gateways.

Calculations and terms used for testing the "xbee_type" conditional

In the table and the steps on the next page, the items highlighted in blue are constants.

Term	Description
device_type	The node device type returned in the ZigBee discover list (the DD value). If the device_type value is not available (such as on Smart Energy nodes), the device_type tests below should be ignored.
caps I	The device's internal XBee capabilities bit flag value which is found in the zigbee_state state group, in the field caps.
FZ = 0x0001	Freescale ZigBee RF module type.
SE = 0x0100	XBee Smart Energy RF module type.
GW = 0x8000;	The XBee RF module is installed in a device.
MODULE_TYPE_MASK = 0xffff0000;	The mask used to pull the XBee RF module type from device type.
MS_MOD_SERIES1 = 0x00010000	802.15.4 or ZigBee (Freescale) RF module.
MS_MOD_ZB = 0x00030000	XBee ZB (Ember) RF module.
ZB_CAP_REMOTE_DDO = 0x00000004	Device supports remote DDO (Digi Data Object) commands. For additional discussion of this term and the other ZB_CAP terms below, see <query_state>.
ZB_CAP_ZIGBEE = 0x00000020	Device supports ZigBee mesh networking.
ZB_CAP_SE = 0x00000200	Device supports ZigBee Smart Energy profile.

Test "xbee_type" conditional

The steps to test the "xbee_type" conditional are as follows:

1. Build (unsigned int32) flags f, starting with f=0.
2. Strip lower 2 bytes from device type:

```
device_type &= MODULE_TYPE_MASK
```

3. Check whether the Digi device contains an XBee Freescale ZigBee RF module:

```
If (device_type & MS_MOD_SERIES1) && (caps & ZB_CAP_ZIGBEE) then f = FZ
```

4. Check whether the Digi device is a Smart Energy product:

```
if (device_type & MS_MOD_ZB) && (caps & ZB_CAP_SE) then f = SE
```

5. If the device neither contains an XBee Freescale ZigBee module nor is a Smart Energy product, set f to module flag:

```
f |= 1 << (device_type >> 16);
```

6. If an XBee module is in the device,

```
then f |= GW
```

7. If an XBee module is not in device:

```
 if !(caps & ZB_CAP_REMOTE_DDO)
```

then remote DDO is not supported, and therefore remote configuration is not allowed.

```
f = 0xFFFFFFFF
```

8. If all the bits of the flags are in the conditional value:

```
((cond_value & f) == f)
```

then condition is satisfied, and that element is valid for that node.