Wireless Vehicle Bus Adapter

Application Developer Guide

Revision history—90001930

Revision	Date	Description
B	December 2013	Updated RF exposure statement and Industry Canada certifications statement.
C	March 2014	Corrected errors in the summary table of WVA web services and clarified text in introductions to web services URI descriptions. Updated URI list to note these URIS as protected: **/ws/config/factory_default** **/ws/config/*settings_group*/*instance_specifier*** **/ws/files/*volume*/*path*/*filename*** **/ws/password** Corrected titles of several URIs. Updated URI **/ws/state/state_path/*element_name*** to have check boxes for **GET**, **XML**, and **JSON**.
D	November 2014	Updated to add URIs the vehicle/dtc URIs for fault codes and any new and changed features for WVA, including Wi-Fi Direct, additional Python programmability, file management features.
E	October 2017	Rebranded and edited the document.
F	June 2019	Updated Event Channel service information. See Assumptions regarding the Event Channel and Event Channel service configuration.

Trademarks and copyright

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

WVA Application Developer Guide

WVA components and interfaces

Managing web services

Web services URIs

Programming

WVA Application Developer Guide

This guide covers programmable aspects of the Wireless Vehicle Bus Adapter (WVA) and the web services, the primary programming interface for the WVA and vehicle data.

Programmable aspects of the Wireless Vehicle Bus Adapter

The WVA has several programmable aspects:

- Hardware, including LEDs, button, buzzer, real time clock, accelerometer
- Vehicle bus interface
- Device management, including firmware updates and rebooting the device
- Configuration settings

The WVA components and interfaces, Managing web services and Programming sections provide more details on these programmable aspects of the WVA.

Web services

The primary interface for handling vehicle data via the WVA is the web services. Web services are resources, organized in a RESTful interface, that provide the primary method for handling vehicle data from the WVA. The web services allow users and applications to access and manipulate units of data in the WVA system, particularly over a local network. The Managing web services section gives an overview of the web services and the Web services URIs section describes each resource.

Programming resources

Programming resources that are available for the WVA, or that you may want to locate or purchase, include the following:

- The WVA demo application and its source code, available from the **Documentation** section on the WVA support page.
- The WVA Android Library tutorial, available from the **Documentation** section on the WVA support page.
- SAE specifications for J1587/J1708 and J1939/CAN vehicle bus protocols. These protocols are available for purchase from the SAE International website.
- RESTful interface resources: The web services are based on RESTful interface principles, noted in RESTful interface principles in the web services. As needed, consult available primers and tutorials on RESTful interfaces.

- HTTP/HTTPS: Using the web services requires an understanding of HTTP/HTTPS and the common operations **GET**, **PUT**, **POST**, and **DELETE**. As needed, consult available primers and tutorials on HTTP/HTTPS. In this guide, see also the following topics about HTTP operation for the WVA and web services:
  - Leveraging HTTP operations
  - Leveraging HTTP security
  - Security features in the WVA

WVA components and interfaces

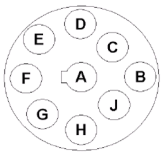
The Wireless Vehicle Bus Adapter (WVA) is a compact on-board hardware device measuring 2.33 in x 2.15 in (5.9 cm x 5.5 cm) that uses the J1708/J1939 protocols to provide vehicle data via standard Wi-Fi and web services. For more information on the WVA hardware and communications specifications, see the Specifications tab of the WVA product page.

This section provides an overview of the WVA device features and interfaces, and lists regulatory statements and certifications for this product.

Connector pinout

The connector on the WVA is a 9-pin Deutsch connector with the following pin orientation and pinout.

Pin locations



Pin signals

Pin	Signal
A	Power (-)
B	Power (+)
C	J1939/CAN 1 HI (+)
D	J1939/CAN 1 LO (-)
E	J1939/CAN Shield
F	J1587/J1708 (+)
G	J1587/J1708 (-)
H	J1939/CAN 2 HI (+)
J	J1939/CAN 2 LO (-)

Additional wiring and connection resources

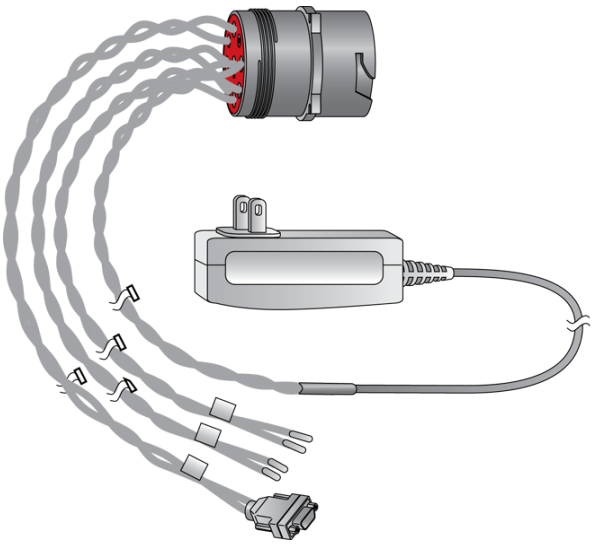
For additional wiring information, such as wire color and locations, refer to the wiring schematics for the vehicle or vehicles in which you are deploying the WVA.

For recommended connection methods, consult the vehicle manufacturer.

Available interfaces on the wiring harness

The wiring harness for the WVA is available for purchase separately from Digi. The Digi part number is **76000931**.

The wiring harness uses a 9-pin Deutsch connector. See Connector pinout for pin locations and signals.



The following interfaces are available on the wiring harness. Labels on the wires indicate their interface function. A DB-9 connector is provided, and wired for compatibility with the recommended CAN bus simulator.

- Power
  - Ground and 12 V power for powering the WVA
  - Ground and 12 V power for powering a CAN simulator (via the DB-9 connector)
- J1939/CAN+ and J1939/CAN- (via the DB-9 connector)
- A second J1939/CAN+ and J1939/CAN-
- J1587/J1708+ and J1587/J1708-

Recommended CAN simulator model

For the CAN simulator, Digi recommends the following model and part number, which you can order from the AU Group Electronics on-line store:

- Model: **Au SAE J1939 Simulator-Gen II 1.00A (Engine Basic Edition)**
- Part number: **SIMJ1939-001**

The CAN simulator connects directly to the WVA wiring harness.

You can also purchase cabling to connect the CAN simulator to a PC from AU Group Electronics.

Interfaces: firmware, software, and hardware

This section includes information about the WVA interfaces.

Firmware and software interfaces

Specification	Value
Configuration and management interface	Web user interface. See Configure the WVA section in the the *Wireless Vehicle Bus Adapter Getting Started Guide* for more information.
Programming interface	RESTful web services interface. See Managing web services.

Hardware interfaces

Specification	Value
Internal sensors	Three internal sensors are included: ■ **Vibration**: Wakes the WVA from sleep mode. ■ **Voltage**: Wakes the WVA from sleep mode. ■ **Accelerometer**: Available to be sampled by custom Python applications. See Power management for configuration information.
Button	The button on the device performs several functions: ■ In Wi-Fi Direct mode, the button can be configured to establish a Wi-Fi Direct connection. Press and hold the button for **10** seconds to reset the WVA to factory defaults. ■ Silences the buzzer/audible alarm. This is applicable only when the alarm is enabled through applications. ■ Wakes the WVA from sleeping. See Power management for configuration information.
LEDs	Two LEDs: ■ **Power**: Green when power is applied to unit. ■ **Application-defined**: Amber when enabled. This LED's use is controlled through the application programming interface (API). LEDs are controlled programmatically through the web services resource hw/leds/led_name.

Memory and development specifications

Specification	Value
Total memory	128 MB flash, 64 MB RAM.
Available user space	20 MB flash.

Regulatory and safety statements

The following regulatory and safety statements apply to the WVA.

RF exposure statement

This equipment complies with FCC radiation exposure limits set forth for an uncontrolled environment. This equipment should be installed and operated with a minimum distance of 20 cm between the radiator and persons. This transmitter must not be co-located or operating in conjunction with any other antenna or transmitter, except in accordance with FCC multi-transmitter product procedures.

FCC Part 15 Class B certifications and regulatory information (USA Only)

All Wireless Vehicle Bus Adapter products comply with the FCC Part 15 Class B standards cited in this section:

Radio Frequency Interface (RFI) (FCC 15.105)

This device has been tested and found to comply with the limits for Class B digital devices pursuant to Part 15 Subpart B, of the FCC rules. These limits are designed to provide reasonable protection against harmful interference in a residential environment. This equipment generates, uses, and can radiate radio frequency energy, and if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try and correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

Labeling requirements (FCC 15.19)

This device complies with Part 15 of FCC rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

If the FCC ID is not visible when installed inside another device, then the outside of the device into which the module is installed must also display a label referring to the enclosed module FCC ID.

Modifications (FCC 15.21)

Changes or modifications to this equipment not expressly approved by Digi may void the user's authority to operate this equipment.

Industry Canada (IC) certifications

This device complies with Industry Canada licence-exempt RSS standard(s). Operation is subject to the following two conditions:

(1) this device may not cause interference, and (2) this device must accept any interference, including interference that may cause undesired operation of the device.

Le présent appareil est conforme aux CNR d'Industrie Canada applicables aux appareils radio exempts de licence. L'exploitation est autorisée aux deux conditions suivantes:

(1) l'appareil ne doit pas produire de brouillage, et (2) l'utilisateur de l'appareil doit accepter tout brouillage radioélectrique subi, même si le brouillage est susceptible d'en compromettre le fonctionnement.

Safety statements

5.10 ignition of flammable atmospheres

Warnings for use of wireless devices

CAUTION! Observe all warning notices regarding use of wireless devices.

Potentially hazardous atmospheres

Observe restrictions on the use of radio devices in fuel depots, chemical plants, etc. and areas where the air contains chemicals or particles, such as grain, dust, or metal powders, and any other area where you would normally be advised to turn off your vehicle engine.

Safety in hospitals

Wireless devices transmit radio frequency energy and may affect medical electrical equipment. Switch off wireless devices wherever requested to do so in hospitals, clinics, or healthcare facilities. These requests are designed to prevent possible interference with sensitive medical equipment.

Pacemakers

Pacemaker manufacturers recommended that a minimum of 15 cm (6 inches) be maintained between a handheld wireless device and a pacemaker to avoid potential interference with the pacemaker. These recommendations are consistent with independent research and recommendations by Wireless Technology Research.

Persons with pacemakers:

- Should *always* keep the device more than 15 cm (6 inches) from their pacemaker when turned ON.
- Should not carry the device in a breast pocket.
- If you have any reason to suspect that the interference is taking place, turn OFF your device.

Certifications

The following certifications apply to WVA.

Automotive certifications

The WVA is certified for the following automotive certifications:

- ISO-10452
- ISO-10605
- SAE 1455
- ISO 7637-2, -3

International EMC (Electromagnetic Emissions/Immunity/Safety) standards

The WVA is certified for the following International EMC standards:

- Emissions: CE, FCC PART 15 (Class B), CISPR25, EN5502
- Immunity: EN55024, EN301, SAE J1113
- Safety: UL 60950-1, EN 60950-1, CSA 22.2 No. 60950

Environmental certifications

The WVA has been tested for the following environmental certifications:

- Automotive Environmental tests per SAE 1455.
- Resistant to the following chemicals, per SAE J1455 Section 4.4:
  - Gasoline
  - Fuel additives
  - Diesel fuel

NEMA certifications/IP rating

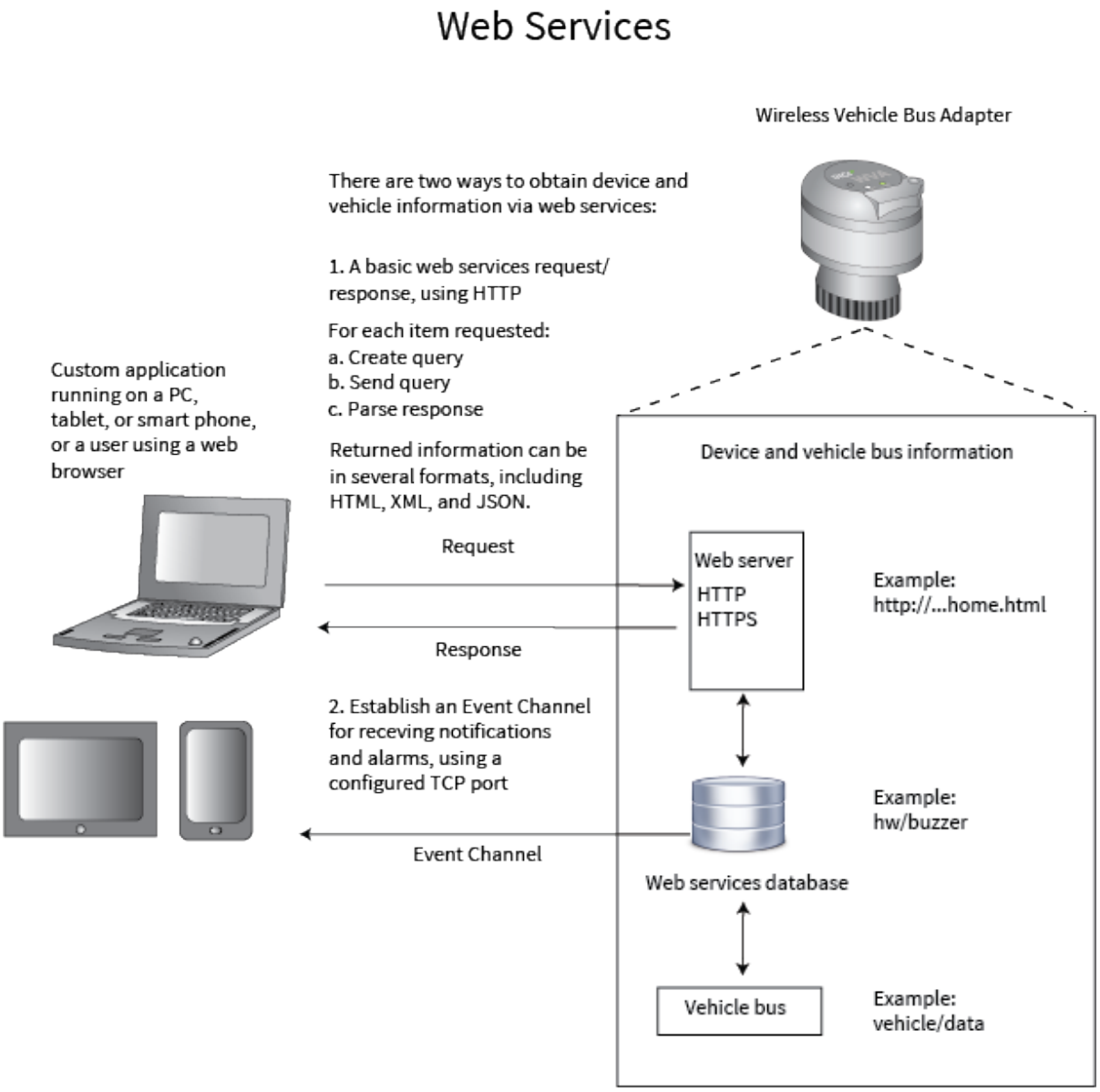
The WVA is certified with an IP rating of **IP54**.

Managing web services

These topics describe the WVA web services, define key terms, and show how to access and navigate the web services.

Data flow using the web services

The following figure shows how a custom application, or user using a web browser, can use the web services to obtain device and vehicle information.



Web services terms

The table below includes web services terms and concepts.

Term	Description
ECU	Engine control unit. ECUs are intelligent components on a vehicle bus that may asynchronously push data, or respond to vehicle bus protocol queries. For example, an engine or a transmission control device.
URI	Uniform resource identifier. In this document, a URI is written relative to the root of the web services tree in the device. The elements in this document use a URL in this format: https://<*device_ip*>/ws/<*uri*> where <*device_ip*> is replaced with the IP address of the WVA device, and <*uri*> is replaced with the resource identifier. See Index of web services resources for a list of the web services resource identifiers.
VIN	Vehicle identification number.

RESTful interface principles in the web services

The web services are a RESTful interface. Key principles used by the web services include the following.

Providing access to individual resources

A key principle in the web services system is providing access to individual resources. Manageable items in this system, including data elements and control endpoints, have unique identifiers and are arranged in a hierarchy. Using this system, you can locate any item in the web services system by its URI.

Granularity based on independence of resources

For many items that can be requested and returned by the web services system, reading, and in some cases writing, an element as an individual entity is a natural operation. For other items, dividing groups of objects into individual fields may be unreasonable, because the web services system may be incapable of manipulating the objects independently. For such items, the “resource” at the web services level is a collection of the related values. Where possible, the web services system makes available individual data elements.

Leveraging HTTP operations

The HTTP specification allows for a variety of useful operations, including **GET**, **PUT**, **POST**, and **DELETE**. The RESTful interface maps logical operations to HTTP request types. For example:

- Use **GET** to perform a “read” operation.
- Use **PUT** to perform a “write” operation.
- **POST** is reserved for control elements with an RPC-style interface, such as passing parameters and getting a response.
- Use **DELETE** to remove elements capable of being deleted.

Leveraging HTTP security

The only security specified in web services is HTTP-related access control. The web services provide access to URLs with the same level of security as any web page in the system.

If HTTP is enabled:

- There is no access-level security by default.
- There is encryption, but no other security by default.

If basic authentication is enabled for the WVA, the web services user must enter the WVA username and password to access the web services system. The web services user may be an application or a user at a browser.

- username: **admin**
- default password: **admin**

For more information on the different types and levels of security implemented in the WVA, see Security features in the WVA.

Support for multiple content types

Web tools support a variety of methods for expressing data. The web services interface is sensitive to the content type-related HTTP headers in order to determine the format of a request and the format to use for a response. The same information is conveyed, and only the data format may change between the content types.

To request a specific format, an HTTP request includes an **Accept** header, with a MIME-type matching the desired format. All successful responses include a **Content-Type** header that indicates the MIME type of the response payload. Available content types are:

- HyperText Markup Language (HTML): text/html
- Extensible Markup Language (XML): application/xml
- JavaScript Object Notation (JSON): application/json

See the Index of web services resources for indicators as to which types are applicable to which URIs.

WVA security and protected URIs

On the WVA, some URIs are password-protected. The Index of web services resources indicates password-protected URIs by an **x** in the **Protected URI** column. Protected URIs are intended to be used only by administrators to configure the device. Other URIs are available to an application for normal operations.

If basic authentication is enabled, **PUT** and **DELETE** operations applied to a protected URI require authentication. Unprotected URIs do not require authentication. A **GET** operation to any URI also does not require authentication.

On any non-WVA devices that implement web services, all methods on all URIs require authentication if enabled.

Generally, interface devices using web services, such as a smart phone or tablet, should not have access to the resources that require authentication.

Access and navigate the web services

There are two ways to access and navigate the web services:

- From a web browser. This use is intended as a means of familiarizing yourself with the web services structure and available resources.
- Through requests from an application. This is the most typical use of the web services.

Access web services from a web browser

1. In the address bar of the browser, enter **https://*<device address>*/ws**, where <*device address*> is replaced with the IP address of the WVA device. For example, the following command accesses the web services for the WVA at the default address in Infrastructure mode, 192.168.100.1: **https://192.168.100.1/ws**
2. A certificate management warning statement from the web browser appears. Click **Proceed Anyway**.
3. The web services interface appears.
4. Click the links to navigate among the subgroups of web services resources, or URIs. Note that several resources are password-protected. These resources display a login prompt. For the username and password, enter **admin**and **admin**. The complete list of protected resources is listed in the Protected URI column in the Index of web services resources.

Access web services from an application

To access the web services programmatically, see the examples in the Index of web services resources. The individual resource demonstrates the resources in a variety of content types.

See also the WVA demo application source code, available from the **Documentation** section on the WVA support page.

As noted in the discussion of WVA security and protected URIs in RESTful interface principles in the web services, some resources are password-protected and are intended to be used by administrators to configure the device.

Web services URIs

This section includes information on all of the web services URIs and describes the Event Channel.

Index of web services resources

The following table is a summary of the web services resources, listed by their URIs in alphabetical order. Click the URI name in the **Device Relative URL** column to jump to the detailed description of the URI.

In the URI name, strings in italics are placeholders to be replaced with appropriate values for the specific operation.

Protected URI: For URIs with an **x** in the **Protected URI** column, the URI is read-only to applications and read/write to administrators.

Note **config/ws_events** and **config/wireless/wlan0** are the only **settings_groups** that are not protected.

Device relative URL (web services path + element URI)	Action	Protected URI	Request methods				Content types		
			GET	PUT	POST	DELETE	HTML	XML	JSON
/ws	Get URI list of web services categories.		✓				✓	✓	✓
/ws/alarms	Get URI list of web service alarm records.		✓				✓	✓	✓
/ws/alarms/*short_name*	Manipulate a web service alarm record.		✓	✓		✓		✓	✓
/ws/config	Get URI list of configuration setting groups.		✓				✓	✓	✓
/ws/config/factory_default	Reset all settings groups to their factory defaults.	✓		✓					
/ws/config/*settings_group*	Manipulate a single settings record, for settings without an instance specifier.	✓	✓	✓				✓	✓
/ws/config/*settings_group* (with instances)	Get URI list of known instance specifiers, for settings that have them.		✓				✓	✓	✓
ws/config/*settings_group*/*instance_specifier*	Manipulate a single settings record for settings with an instance specifier.	✓	✓	✓				✓	✓

Device relative URL (web services path + element URI)	Action	Protected URI	Request methods				Content types		
			GET	PUT	POST	DELETE	HTML	XML	JSON
/ws/files/*volume/path* Where ***volume*** is either **userfs** or a USB flash drive name.	Manipulate a single settings record for settings with an instance specifier.		✓	✓				✓	✓
/ws/files/*volume/path*?type=dir	Create a new directory at the named path.			✓					
/ws/files/*volume/path/filename*	Manipulate the named file in the named path.	✓	✓	✓		✓			
/ws/hw	Get URI list of categories of manageable hardware interfaces.		✓				✓	✓	✓
/ws/hw/buttons	Get URI list of manageable buttons.		✓				✓	✓	✓
/ws/hw/buttons/*button_name*	Read the state of the named button.		✓					✓	✓
/ws/hw/buzzer	Manipulate the buzzer, also known as the audible alarm.		✓	✓				✓	✓
/ws/hw/fw_update	Initiate or test status of a firmware update.		✓	✓				✓	✓
/ws/hw/leds	Get URI list of manageable LEDs.		✓				✓	✓	✓
/ws/hw/*led_name*	Manipulate the named LED.		✓	✓				✓	✓
/ws/hw/reboot	Reboot the system.			✓					
/ws/hw/time	Manipulate system time.		✓	✓				✓	✓
/ws/password	Set the password for admin username.	✓		✓				✓	✓
/ws/state	Get URI list of state categories.		✓				✓	✓	✓

Device relative URL (web services path + element URI)	Action	Protected URI	Request methods				Content types		
			GET	PUT	POST	DELETE	HTML	XML	JSON
/ws/state/*state_path* (containing subgroups or instances)	Get URI list of subgroups or instances for the named state path.		✓				✓	✓	✓
/ws/state/*state_path* (containing elements)	Get URI list of elements for the named state path.		✓				✓	✓	✓
/ws/state/*state_path*/*element_name*	Query the system for a specific element of the named state group in the system.		✓					✓	✓
/ws/subscriptions	Get URI list of web service subscription records.								
/ws/subscriptions/*short_name*	Manipulate a web service subscription record.		✓				✓	✓	✓
/ws/vehicle	Get URI list of vehicle object categories.		✓				✓	✓	✓
/ws/vehicle/data	Get URI list of vehicle data elements.		✓				✓	✓	✓
/ws/vehicle/data/*element_name*	Get a specific vehicle data element.		✓					✓	✓
/ws/vehicle/dtc	Get URI list of DTC message types for which the system listens.	✓					✓	✓	✓
/ws/vehicle/dtc/can0_active	Get URI list of ECUs on first J1939/CAN bus for active DTCs.	✓					✓	✓	✓
/ws/vehicle/dtc/can0_active/*ecu_reference*	Get latest active DTC from an ECU on first J1939/CAN bus.	✓						✓	✓
/ws/vehicle/dtc/can0_inactive	Get URI list of ECUs on first J1939/CAN bus for inactive DTCs.	✓					✓	✓	✓
/ws/vehicle/dtc/can0_inactive/*ecu_reference*	Get latest active DTC from an ECU on first J1939/CAN bus.	✓						✓	✓

Device relative URL (web services path + element URI)	Action	Protected URI	Request methods				Content types		
			GET	PUT	POST	DELETE	HTML	XML	JSON
/ws/vehicle/dtc/can1_active	Get URI list of ECUs on second J1939/CAN bus for active DTCs.	✓					✓	✓	✓
/ws/vehicle/dtc/can1_active/*ecu_reference*	Get latest active DTC from an ECU on second J1939/CAN bus.	✓						✓	✓
/ws/vehicle/dtc/can1_inactive	Get URI list of ECUs on second J1939/CAN bus for inactive DTCs.	✓					✓	✓	✓
/ws/vehicle/dtc/can1_inactive/*ecu_reference*	Get latest inactive DTC from an ECU on second J1939/CAN bus.	✓						✓	✓
/ws/vehicle/ecus	Get URI list of detected vehicle ECUs.		✓						
/ws/vehicle/ecus/ecu_reference	Get URI list of descriptive items for an ECU.		✓				✓	✓	✓
/ws/vehicle/ecus/*ecu_reference*/*item*	Get a specific ECU information element.		✓					✓	✓

Device web services root (/)

The device web services root, /, gets a URI list of web services categories.

URI path

```
/
```

Supported request methods

GET

Queries the system for the list of categories of web service operations available. The data record returned is a list of URIs corresponding to the “children” of **/** in the web services data tree.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

```
<ws>
        <element>vehicle</element>
        <element>config</element>
        <element>hw</element>
</ws>
```

JSON

The result URI strings are collected in an array assigned to a **ws** field. For example:

```
{ “ws” : [ “vehicle”, “config”, “hw” ] }
```

Translated vehicle bus data

Devices with vehicle bus connectivity allow raw message manipulation at the bus level. Remote entities outside the vehicle bus device may communicate with the device in terms of vehicle data elements, as the abstraction of vehicle elements hides the underlying bus architecture.

Devices such as the Wireless Vehicle Bus Adapter (WVA) include vehicle bus abstraction software to translate vehicle bus messages into individual units of information. These units become resources that can be queried in the web services infrastructure.

This interface allows processing of two different types of information:

- Listing and querying vehicle bus data elements
- Listing and querying ECU information elements

Translated vehicle bus data URIs

- vehicle
- vehicle/data
- vehicle/data/element_name
- vehicle/dtc
- vehicle/dtc/can0_active
- vehicle/dtc/can0_active/ecu_reference
- vehicle/dtc/can0_inactive
- vehicle/dtc/can0_inactive/ecu_reference
- vehicle/dtc/can1_active
- vehicle/dtc/can1_active/ecu_reference
- vehicle/dtc/can1_inactive
- vehicle/dtc/can1_inactive/ecu_reference
- vehicle/ecus
- vehicle/ecus/ecu_reference
- vehicle/ecus/ecu_reference/ecu_info_item

vehicle

The **vehicle** URI gets a list of vehicle object categories.

URI path

```
vehicle
```

Supported request methods

GET

Queries the system for the list of categories of vehicle service information available to web services. The data record returned is a list of URIs corresponding to the “children” of **vehicle** in the web services data tree.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within a vehicle block. For example:

```
<vehicle>
         <element>vehicle/ecus</element>
         <element>vehicle/data</element>
</vehicle>
```

JSON

The result URI strings are collected in an array assigned to a vehicle field. For example:

```
{ “vehicle” : [ “vehicle/ecus”, "vehicle/data”]
```

vehicle/data

The **vehicle/data** URI gets a list of vehicle data elements.

URI path

```
vehicle/data
```

Supported request methods

GET

Queries the system for the list of vehicle data elements. The returned data record is a list of URIs corresponding to the “children” of **vehicle/data** in the web services data tree.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned in a **data** block. For example:

```
<data>
          <element>vehicle/data/item1</element>
          <element>vehicle/data/item2</element>
          :
          <element>vehicle/data/itemN</element>
</data>
```

JSON

The result URI strings are collected in an array assigned to a field corresponding to the named ECU. For example:

```
{ “data” : [ “vehicle/data/item1”,  “vehicle/data/item2”, ...
                  “vehicle/data/itemN” ] }
```

vehicle/data/*element_name*

The **vehicle/data/element_name** URI gets a specific vehicle data element.

URI path

```
vehicle/data/element_name
```

Supported request methods

GET

Queries the system for the record associated with the specific vehicle data element. An example of a queryable element is **EngineSpeed**.

Supported content types

Returned vehicle data element values may be expressed in a variety of categories, such as decimal or hexadecimal numbers or strings. The examples below demonstrate several types of returned values.

J1939 SPNs, or parameters, that are described as bit fields or enumerations are represented as ints. J1939 SPNs that can have floating point values are represented as floats. In general, variables that are strings of bytes are represented as binary data.

XML

The result is returned in a data content block, including a timestamp and a value (scalar, where numeric), and wrapped in a block named for the requested element. For example, here is the returned value for **EngineSpeed**:

```
<EngineSpeed>
          <timestamp>2012-12-09T05:15:32Z</timestamp>
          <value>2350.0</value>
</EngineSpeed>
```

The variable **vehicle/data/ParkingBrake** is an int variable, and its values are reported in decimal:

```
<ParkingBrake>
            <value>0</value>
            <timestamp>2013-11-07T16:54:12Z</timestamp>
</ParkingBrake>
```

The variable **vehicle/data/IgnitionSwitchStatus** is a binary value. Its values are reported in hexadecimal. In the following example, the returned value is **0x55**, which is **85** in decimal:

```
<IgnitionSwitchStatus>
            <value>55</value>
            <timestamp>2013-11-07T16:57:33Z</timestamp>
</IgnitionSwitchStatus>
```

JSON

The result is returned as an object with timestamp and value fields, assigned to a field corresponding to the requested value. Following are JSON examples of same values shown above.

```
{ "EngineSpeed" : { "timestamp" :  "2012-12-09T05:15:32Z",
   "value": "2350.0" } }
```

```
{ "ParkingBrake": {  "value": 0,
                  "timestamp": "2013-11-07T16:55:37Z" } }
 { "IgnitionSwitchStatus": {
                "value": "55",
                "timestamp": "2013-11-07T16:59:39Z" }
```

vehicle/dtc

The **vehicle\dtc** URI gets a list of DTC message types for which the system listens.

URI path

```
vehicle/dtc
```

Supported request methods

GET

Queries the system for the list of vehicle diagnostic trouble codes. The data record returned is a list of URIs corresponding to the "children" of **vehicle/dtc** in the web services data tree. These children represent classes of diagnostic messages on the various buses of the system.

Supported content types

HTML

The result URIs is turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs will each be wrapped in **element** tags, and returned in a **dtc** block. For example:

```
<dtc>
          <element>vehicle/dtc/can0_active</element>
          <element>vehicle/dtc/can0_inactive</element>
          <element>vehicle/dtc/can1_active</element>
          <element>vehicle/dtc/can1_inactive</element>
</dtc>
```

JSON

The result URI strings is collected in an array assigned to a **dtc** field. For example:

```
{ "dtc" : [ "vehicle/dtc/can0_active",
                "vehicle/dtc/can0_inactive",
                "vehicle/dtc/can1_active",
                "vehicle/dtc/can1_inactive"  ] }
```

vehicle/dtc/can0_active

The **vehicle/dtc/can0_active** URI gets a list of ECUs on the first J1939/CAN bus for active DTCs.

URI path

```
vehicle/dtc/can0_active
```

Supported request methods

GET

Query the system for the list of ECUs that the system knows exist on the primary J1939/CAN bus, since they might be providing active DTC messages (J1939 PGN 65226). The data record returned is a list of URIs corresponding to each ECU on a bus.

Each time the vehicle data subsystem restarts, it exposes a fixed list of ECUs. Once the subsystem begins running, it samples the bus for 60 seconds to determine whether the list of ECUs it has recorded matches the current running vehicle. If it does not match, the subsystem automatically stores an updated list and restarts the subsystem to use the new, fixed list.

Supported content types

HTML

The result URIs is turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs will each be wrapped in **element** tags, and returned within a **can0_active** block. For example:

```
<can0_active>
         <element>vehicle/dtc/can0_active/ecu0</element>
         <element>vehicle/dtc/can0_active/ecu3</element>
</can0_active>
```

JSON

The result URI strings is collected in an array assigned to a **can0_active** field. For example:

```
{ “can0_active” : [ “vehicle/dtc/can0_active/ecu0”,
                        “vehicle/dtc/can0_active/ecu3” ] }
```

vehicle/dtc/can0_active/ecu_reference

The **vehicle/dtc/can0_active/ecu_reference** URI gets the latest active DTC from an ECU on the first J1939/CAN bus.

URI path

```
vehicle/dtc/can0_active/ecu_reference
```

Supported request methods

GET

Queries the system for the most recent DTC report from the referenced ECU on the primary J1939/CAN bus, if any (J1939 PGN **65226**).

Note PGN **65226** messages are generally broadcast by ECUs once per second, so it is expected that these messages are updated regularly.
HTML status code **503 (Service Unavailable)** indicates that the referenced ECU is expected to be valid, but no PGN 65226 message has yet been received. HTML status code **404 (Not Found)** generally indicates that the system is not listening for trouble codes on the referenced ECU.

Supported content types

XML

The result, if any, is returned in a data content block, including a timestamp and a value. The value is a hexadecimal string representing the binary payload of the PGN **65226** message. The data block name matches the ECU reference. For example:

```
<ecu0>
          <timestamp>2012-12-09T05:15:32Z</timestamp>
          <value>00ff00000000ffff</value>
</ecu0>
```

JSON

The result, if any, is returned as an object with timestamp and value fields, with the value as a hexadecimal string representing the binary payload of the PGN **65226** message. The name of the object matches the ECU reference. For example:

```
{ “ecu0” : { “timestamp” :  “2012-12-09T05:15:32Z”,
                 “value” : “00ff00000000ffff”  } }
```

vehicle/dtc/can0_inactive

The **vehicle/dtc/can0_inactive** URI gets a list of ECUs on the first J1939/CAN bus for inactive DTCs.

URI path

```
vehicle/dtc/can0_inactive
```

Supported request methods

GET

Queries the system for the list of ECUs that the system knows exist on the primary J1939/CAN bus, since they might be providing inactive DTC messages (J1939 PGN **65227**). The record returned is a list of URIs corresponding to each ECU on a bus.

Note Each time the vehicle data subsystem restarts, it exposes a fixed list of ECUs. Once the subsystem begins running, it samples the bus for **60** seconds to determine whether the list of ECUs it has recorded matches the current running vehicle. If not, the subsystem automatically stores an updated list and restarts the subsystem to use the new fixed list.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned in a **can0_inactive** block. For example:

```
<can0_inactive>
         <element>vehicle/dtc/can0_inactive/ecu0</element>
         <element>vehicle/dtc/can0_inactive/ecu3</element>
</can0_inactive>
```

JSON

The result URI strings are collected in an array assigned to a **can0_inactive** field. For example:

```
{ “can0_inactive” : [ “vehicle/dtc/can0_inactive/ecu0”
                      “vehicle/dtc/can0_inactive/ecu3”  ] }
```

vehicle/dtc/can0_inactive/*ecu_reference*

The **vehicle/dtc/can0_inactive/*ecu_reference*** URI gets the latest active DTC from an ECU on the first J1939/CAN bus.

URI path

```
vehicle/dtc/can0_inactive/ecu_reference
```

Supported request methods

GET

Queries the system for the most recent inactive trouble code report from the referenced ECU on the primary J1939/CAN bus, if any (J1939 PGN **65227**).

Note PGN **65227** messages are polled no more frequently than once every ten seconds. HTML status code **503 (Service Unavailable)** indicates that the referenced ECU is expected to be valid, but no PGN 65227 message has yet been received. HTML status code **404 (Not Found)** generally indicates that the system is not listening for trouble codes on the referenced ECU.

Supported content types

XML

The result, if any, is returned in a data content block, including a timestamp and a value. The value is a hexadecimal string representing the binary payload of the PGN **65227** message. The data block name matches the ECU reference. For example:

```
<ecu0>
          <timestamp>2012-12-09T05:15:32Z</timestamp>
          <value>00ff00000000ffff</value>
</ecu0>
```

JSON

The result, if any, is returned as an object with timestamp and value fields, with the value as a hexadecimal string representing the binary payload of the PGN **65227** message. The name of the object matches the ECU reference. For example:

```
{ “ecu0” : { “timestamp” :  “2012-12-09T05:15:32Z”,
             “value” : “00ff00000000ffff” } }
```

vehicle/dtc/can1_active

The **vehicle/dtc/can1_active** URI gets a list of ECUs on the second J1939/CAN bus for active DTCs.

URI path

```
vehicle/dtc/can1_active
```

Supported request methods

GET

Query the system for the list of ECUs that the system knows exist on the secondary J1939/CAN bus, since they might be providing active DTC messages (J1939 PGN 65226). The data record returned is a list of URIs corresponding to each ECU on a bus.

Each time the vehicle data subsystem restarts, it exposes a fixed list of ECUs. Once the subsystem begins running, it samples the bus for 60 seconds to determine whether the list of ECUs it has recorded matches the current running vehicle. If it does not match, the subsystem automatically stores an updated list and restarts the subsystem to use the new, fixed list.

Supported content types

HTML

The result URIs is turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs will each be wrapped in **element** tags, and returned within a **can1_active** block. For example:

```
<can1_active>
         <element>vehicle/dtc/can1_active/ecu0</element>
         <element>vehicle/dtc/can1_active/ecu3</element>
</can1_active>
```

JSON

The result URI strings is collected in an array assigned to a **can1_active** field. For example:

```
{ “can1_active” : [ “vehicle/dtc/can1_active/ecu0”,
                        “vehicle/dtc/can1_active/ecu3” ] }
```

vehicle/dtc/can1_active/*ecu_reference*

The **vehicle/dtc/can1_active/*ecu_reference*** URI gets the latest active DTC from an ECU on the second J1939/CAN bus.

URI path

```
vehicle/dtc/can1_active/ecu_reference
```

Supported request methods

GET

Queries the system for the most recent DTC report from the referenced ECU on the secondary J1939/CAN bus, if any (J1939 PGN **65226**).

Note PGN **65226** messages are generally broadcast by ECUs once per second, so it is expected that these messages are updated regularly.
HTML status code **503 (Service Unavailable)** indicates that the referenced ECU is expected to be valid, but no PGN 65226 message has yet been received. HTML status code **404 (Not Found)** generally indicates that the system is not listening for trouble codes on the referenced ECU.

Supported content types

XML

The result, if any, is returned in a data content block, including a timestamp and a value. The value is a hexadecimal string representing the binary payload of the PGN **65226** message. The data block name matches the ECU reference. For example:

```
<ecu0>
          <timestamp>2012-12-09T05:15:32Z</timestamp>
          <value>00ff00000000ffff</value>
</ecu0>
```

JSON

The result, if any, is returned as an object with timestamp and value fields, with the value as a hexadecimal string representing the binary payload of the PGN **65226** message. The name of the object matches the ECU reference. For example:

```
{ “ecu0” : { “timestamp” :  “2012-12-09T05:15:32Z”,
                 “value” : “00ff00000000ffff”  } }
```

vehicle/dtc/can1_inactive

The **vehicle/dtc/can1_inactive** URI gets a list of ECUs on the second J1939/CAN bus for inactive DTCs.

URI path

```
vehicle/dtc/can1_inactive
```

Supported request methods

GET

Queries the system for the list of ECUs that the system knows exist on the secondary J1939/CAN bus, since they might be providing inactive DTC messages (J1939 PGN **65227**). The record returned is a list of URIs corresponding to each ECU on a bus.

Note Each time the vehicle data subsystem restarts, it exposes a fixed list of ECUs. Once the subsystem begins running, it samples the bus for **60** seconds to determine whether the list of ECUs it has recorded matches the current running vehicle. If not, the subsystem automatically stores an updated list and restarts the subsystem to use the new fixed list.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned in a **can1_inactive** block. For example:

```
<can1_inactive>
         <element>vehicle/dtc/can1_inactive/ecu0</element>
         <element>vehicle/dtc/can1_inactive/ecu3</element>
</can1_inactive>
```

JSON

The result URI strings are collected in an array assigned to a **can0_inactive** field. For example:

```
{ “can1_inactive” : [ “vehicle/dtc/can1_inactive/ecu0”
                      “vehicle/dtc/can1_inactive/ecu3”  ] }
```

vehicle/dtc/can1_inactive/*ecu_reference*

The **vehicle/dtc/can1_inactive/*ecu_reference*** URI gets the latest inactive DTC from an ECU on the second J1939/CAN bus.

URI path

```
vehicle/dtc/can1_inactive/ecu_reference
```

Supported request methods

GET

Queries the system for the most recent inactive trouble code report from the referenced ECU on the secondary J1939/CAN bus, if any (J1939 PGN **65227**).

Note PGN **65227** messages are polled no more frequently than once every ten seconds.
HTML status code **503 (Service Unavailable)** indicates that the referenced ECU is expected to be valid, but no PGN 65227 message has yet been received. HTML status code **404 (Not Found)** generally indicates that the system is not listening for trouble codes on the referenced ECU.

Supported content types

XML

The result, if any, is returned in a data content block, including a timestamp and a value. The value is a hexadecimal string representing the binary payload of the PGN **65227** message. The data block name matches the ECU reference. For example:

```
<ecu0>
          <timestamp>2012-12-09T05:15:32Z</timestamp>
          <value>00ff00000000ffff</value>
</ecu0>
```

JSON

The result, if any, is returned as an object with timestamp and value fields, with the value as a hexadecimal string representing the binary payload of the PGN **65227** message. The name of the object matches the ECU reference. For example:

```
{ “ecu0” : { “timestamp” :  “2012-12-09T05:15:32Z”,
             “value” : “00ff00000000ffff” } }
```

vehicle/ecus

The **vehicle/ecus** URI gets a list of detected vehicle ECUs.

URI path

```
vehicle/ecus
```

Supported request methods

GET

Query the system for the list of addressable ECUs in the vehicle. The data record returned is a list of URIs corresponding to the “children” of **vehicle/ecus** in the web services data tree.

Each ECU in the system is referenced by a J1939/CAN bus number, and the ECU address number. A standard ECU in vehicles, for instance, is at address **0**, which is defined as the primary engine. The reference designation for the engine on CAN bus **1** is **can1ecu0**. Both the CAN bus number and the ECU address number in the reference designator are expressed in decimal. ECU address numbers are generally sparse.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned within an **ecus** block. For example:

```
<ecus>
         <element>vehicle/ecus/can0ecu0</element>
         <element>vehicle/ecus/can0ecu2</element>
         <element>vehicle/ecus/can1ecu25</element>
</ecus>
```

JSON

The result URI strings are collected in an array assigned to an ecus field. For example:

```
{ “ecus” : [ “vehicle/ecus/can0ecu0”,
             “vehicle/ecus/can0ecu2”,
             “vehicle/ecus/can1ecu5” ] }
```

vehicle/ecus/*ecu_reference*

The **vehicle/ecus/*ecu_reference*** URI gets a list of descriptive items for an ECU.

URI path

```
vehicle/ecus/ecu_reference
```

Supported request methods

GET

Queries the system for the list of data elements describing a specific engine control unit (ECU) in the vehicle. The data record returned is a list of URIs corresponding to the "children" of **vehicle/ecus/ecu** reference in the web services data tree. ECU references include a J1939/CAN bus number and ECU address number as described in vehicle/ecus.

An example element describing an ECU is a serial number. The VIN is another element that should reside in at least one vehicle ECU.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within a block with a name corresponding to the named ECU. For example:

```
<can0ecu2>
          <element>vehicle/ecus/can0ecu2/make</element>
          <element>vehicle/ecus/can0ecu2/model</element>
          <element>vehicle/ecus/can0ecu2/serial_number</element>
          :
          <element>vehicle/ecus/can0ecu2/VIN</element>
</can0ecu2>
```

JSON

The result URI strings are collected in an array assigned to a field corresponding to the named ECU. For example:

```
{ "can0ecu2"  : [ "vehicle/ecus/can0ecu2/make",
                  "vehicle/ecus/can0ecu2/model",
                  "vehicle/ecus/can0ecu2/serial_number",  ...
                  "vehicle/ecus/can0ecu2/VIN"  ] }
```

vehicle/ecus/*ecu_reference*/*ecu_info_item*

The **vehicle/ecus/ecu_reference/ecu_info_item** URI gets a specific ECU information element.

URI path

```
vehicle/ecus/ecu_reference/ecu_info_item
```

Supported request methods

GET

Queries the system for a specific element describing a specific ECU. The set of valid ***ecu_info_item*** values is available in vehicle/ecus/ecu_reference.

Supported content types

XML

The result is returned in an element with a name corresponding with the item being requested. For example:

```
<VIN>1HGBH41JXMN109186</VIN>
```

JSON

The result is assigned to a field corresponding to the requested value. For example:

```
{ “VIN” : “1HGBH41JXMN109186” }
```

Asynchronous data delivery registration: subscriptions, alarms, and the Event Channel

Because each data element the web services can address has a unique URI, it is possible to query any required data on demand. However, since each query incurs an overhead cost, it can be a somewhat inefficient mechanism if data elements must be polled to detect changes. You can use data alarm and subscription features to detect these changes.

Subscriptions

A data subscription is a registration with the system, instructing that specified URIs should be delivered to a remote entity at specified intervals, asynchronously. Any URI except files can be registered in a subscription.

Alarms

Alarms allow you to inform the device that it should “watch” specified vehicle bus values and prepare to push asynchronous updates regarding those values if a configured test is true. For example, you can set an alarm to receive notification that a vehicle's engine speed has exceeded a particular RPM value.

Event Channel

All notifications of subscribed data or alarms are delivered via an Event Channel. The Event Channel is the mechanism for delivering asynchronous, out-of-band data, such as alarms and periodic updates from subscriptions.

The Configuration interfaces can be used to prepare the device to generate out-of-band data. Delivery of out-of-band data occurs on a dedicated TCP channel only, independent of the normal HTTP traffic. As a result, browsers without custom extensions are not really the intended targets of these out-of-band channels. See Event channel for more information.

URIs for managing subscriptions and alarms

The following URI paths manage subscriptions and alarms:

subscriptions

subscriptions/short_name

alarms

alarms/short_name

subscriptions

Data subscriptions are managed as a set of records, where each record describes a specific element to be delivered on a periodic basis. If desired, you can apply multiple subscriptions to a single URI, though this is of limited value. The subscriptions URI operates like a data store. Initially, the store is empty. The existence of a record in the data store corresponds with a subscription registered with the system. You can query, add, change, and remove records at will. The records have names that are arbitrary to the system, supplied via the request URI.

URI path

```
subscriptions
```

Supported request methods

GET

Queries the system for the list of subscription records. The record returned is a list of URIs corresponding to the “children” of **subscriptions** in the web services data tree.

Supported content types:

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within a **subscriptions** block. For example:

```
<subscriptions>
          <element>subscriptions/short_name_1</element>
          <element>subscriptions/short_name_2</element>
          :
          <element>subscriptions/short_name_N</element>
</subscriptions>
```

JSON

The result URI strings are collected in an array assigned to a **subscriptions** field. For example:

```
{ “subscriptions” : [  “subscriptions/short_name_1”,
                       “subscriptions/short_name_2”, ...
                       “subscriptions/short_name_N” ] }
```

subscriptions/*short_name*

The **subscriptions/*short_name*** URI manipulates a web service subscription record.

URI path

```
subscriptions/short_name
```

Subscription configuration records

Subscription configuration records include:

- **uri**: The URI of the element to be delivered.
- **buffer**: The buffering policy for the subscription, which can include:
  - **queue**: If the Event Channel is closed, updates are placed in the Event Channel delivery queue.
  - **discard**: Messages are discarded if the event channel is closed.
- **interval**: The number of seconds between updates. This must be an integer greater than zero.

Supported request methods

GET

Queries the system for the record associated with the specified short name, which is supplied by the user when the record is first stored in the device.

PUT

If the requested record does not yet exist, adds a new record with the requested URI to the system. If the requested record already exists in the system, the existing record is modified to match the supplied record data.

DELETE

Removes the specified record from the system.

Supported content types

XML

The **GET** and **PUT** forms use a subscription record. For example:

```
<subscription>
         <uri>vehicle/data/EngineSpeed</uri>
          <buffer>queue</buffer>
          <interval>10</interval>
</subscription>
```

JSON

The **GET** and **PUT** forms create an object assigned to a subscription field. In **PUT** requests, numeric fields need not be quoted, but **GET** responses feature all fields and values as quoted strings, regardless of type. For example:

```
{ “subscription” : { “uri” :  “vehicle/data/EngineSpeed”,
                     “buffer” :  “queue”, “interval” : “10” }}
```

alarms

The **alarms** URI gets a list of web service alarm records.

Data alarms are managed as a set of records, with each record describing a specific alarm condition. Multiple alarm conditions can be applied to a single URI. For any sample, all alarm conditions are tested and matching alarm conditions are pushed into the event channel, relative to buffering controls and the existence of an event channel.

Not all URIs support alarms. If a URI does not support alarms, an attempt to create an alarm results in an HTTP error.

Note In the initial WVA release, only vehicle data URIs support alarms. The interface is defined so that other URIs can be added to alarms without changing the registration methods.

The alarms URI operates like a data store. Initially, the store is empty. The existence of a record in the store corresponds with an alarm condition registered with the system. The external user can query, add, change, and remove records at will. The records have names that are arbitrary to the system, which you supply through the request URI.

URI path

```
alarms
```

Supported request methods

GET

Queries the system for the list of alarm records. The returned record is a list of URIs corresponding to the “children” of **alarms** in the web services data tree.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within an **alarms** block. For example:

```
<alarms>
        <element>alarms/short_name_1</element>
        <element>alarms/short_name_2</element>
        :
        <element>alarms/short_name_N</element>
</alarms>
```

JSON

The result URI strings are collected in an array assigned to an **alarms** field. For example:

```
{ “alarms” : [ “alarms/short_name_1”,
               “alarms/short_name_2”, ...
               “alarms/short_name_N” ] }
```

alarms/*short_name*

The **alarms/*short_name*** URI manipulates a web services alarm record.

URI path

```
alarms/short_name
```

Alarm configuration records

Alarm configuration records include:

- **uri**: The URI of the element to test for alarms

Note In the initial release, only vehicle data URIs support alarms. The interface is defined such that other URIs can be added to alarms without changing the registration methods.

- **type**: The type of alarm, which can include:
  - **above**: Triggered if a detected value exceeds the threshold.
  - **below**: Triggered if a detected value is below the threshold.
  - **change**: Triggered if the detected value is different from the most recent value recorded. The expected frequency of updates is limited by the internal implementation associated with the supported URIs. For example, the frequency of a vehicle data alarm is limited by the frequency with which the ECUs in the system push the requested items are pushed into the vehicle bus.
  - **delta**: Triggered if the newest detected value is different from the previous value reported in an event by more than the configured threshold. When the alarm is added or modified, the current value becomes the previous value. If no current value is available, the next change triggers the alarm, regardless of threshold.
- **buffer**: The buffering policy for the alarm, which can include:
  - **queue**: If the Event Channel is closed, an alarm indication is placed in the Event Channel delivery queue. See Event Channel buffering policy.
  - **discard**: If the event channel is closed, messages are discarded.
- **threshold**: For scalar elements, the threshold value to be tested based on the alarm type.
- **interval**: The minimum number of seconds that must pass before another indication of the same alarm should be delivered. This value is an integer and must be greater than or equal to **0**.

Supported request methods

GET

Queries the system for the record associated with the specified short name. The short name was supplied by the user when the record was first stored in the device.

PUT

If the requested record does not yet exist, a new record (with the requested URI) is added to the system. If the requested record already exists in the system, the existing record is modified to match the supplied record data.

DELETE

Removes the specified record from the system.

Supported content types

XML

The **GET** and **PUT** forms use an alarm record. For example:

```
<alarm>
         <uri>vehicle/data/EngineSpeed</uri>
         <type>above</type>
         <buffer>discard</buffer>
         <threshold>8000.0</threshold>
         <interval>60</interval>
</alarm>
```

JSON

The **GET** and **PUT** forms create an object assigned to an alarm field. In PUT requests, numeric fields need not be quoted, but **GET** responses will feature all fields and values as quoted strings, regardless of type. For example:

```
{ “alarm” : { “uri” :  “vehicle/data/EngineSpeed”,
              “type” : “above”,  “buffer” : “discard”,
              “threshold” : “8000.0”,  “interval” : “60” } }
```

Event channel

Listed below are details on the Event Channel functionality and considerations on using the Event Channel in application programming.

- Assumptions regarding the Event Channel
- Event Channel service configuration
- Event Channel buffering policy
- Event message records

Assumptions regarding the Event Channel

- The WVA device supports only one Event Channel at a time.
- The remote peer is responsible for creating the Event Channel, as only it knows when it is capable of handling event messages.
- The connection is made to a configurable service port on the WVA device.
- The Event Channel is used to deliver asynchronous, out-of-band data, such as alarms and periodic updates from subscriptions to vehicle data endpoints. Set the TCP port number to a value between 1025 and 65535 that is not already assigned to another service. The TCP port number is set in the Network Service Configuration page.
- You can add and remove alarm and subscription registrations freely. These registrations inform the system on which alarm and subscription messages should be generated, and when. The messages are placed in the Event Channel for consumption by a remote entity. The messages are not retrieved via web services.
- The Event Channel is assumed to be on a local, high bandwidth connection. Therefore, the size of the data expression is limited in its importance.
- The Event Channel is not compressed.
- The Event Channel is not encrypted.
- The Event Channel is write-only by the WVA device.
- If the remote peer does not consume the event channel notifications, messages can be dropped by the device when the TCP send path is full.
- The WVA device does not read from the TCP connection.
- All notifications pushed into the Event Channel use a common, prearranged MIME content type.

Event Channel service configuration

The Event Channel functionality runs in parallel to the bulk of web services. It must be configurable by the system as part of the standard system configuration mechanism. You can configure the following parameters affecting Event Channel operation:

- Enable or disable the Event Channel network service in the product.

  When enabled, the TCP port number defined for the Event Channel network service listens for connections. The Event Channel network service is enabled or disabled in the Network Service Configuration page.

- The Event Channel is used to deliver asynchronous, out-of-band data, such as alarms and periodic updates from subscriptions to vehicle data endpoints. Set the TCP port number to a value between 1025 and 65535 that is not already assigned to another service. The TCP port number is set in the Network Service Configuration page.
- The MIME content type used for any notifications is determined by the content type of the most recent registration. Since most reasonable uses will use one content type or other for all data manipulation, this establishes the Event Channel format without requiring extra configuration. See RESTful interface principles in the web services.

Event Channel buffering policy

As alarms, subscriptions, and other web services elements need to push data to a remote entity via the Event Channel, they are submitted for transmission to the Event Channel manager. Each submission includes the data to transmit and the buffering policy preferred by that specific message. This buffering policy is referenced by the channel manager, as it determines whether to immediately transmit a message or queue the message for transmission.

The buffering behavior for Event Channel messages when there is no Event Channel is configurable via web services on a per-event basis.

- If there is no Event Channel present, and the policy is to discard, the message is dropped.
- If there is no Event Channel present, and the policy is to queue, the message is added to the pending message queue (see note, below).
- If there is an Event Channel, and the queue is empty, an attempt to transmit is made. If the transmit fails (for instance, the connection has recently been closed), the buffering policy is consulted.
- If there is an Event Channel, but there are messages in the queue, the message is added to the queue.

Note In the initial implementation of web services, the queue only retains the 64 most recent submissions of any type.

Event message records

Event message records are written to the Event Channel as simple wrappers around the standard data delivery form for the item the event was configured for.

The wrapper includes:

- An indication of the type of record, such as an alarm or data from a subscription.
- The user-supplied short name associated with the event record.
- The URI associated with the event value, for convenience.
- A sequence number that increments with each event. This sequence number is unique to the alarm or subscription and may be used to detect dropped events.
- A timestamp indicating when the event was generated. This timestamp is in addition to the vehicle data timestamp, which indicates when the data was obtained.

The following XML and JSON code examples demonstrate how the records might be seen in the Event Channel.

Example event message record: periodic delivery of engine speed (RPM) as a subscription

XML

```
<data>
     <short_name>RPM_Every_Minute</short_name>
     <uri>vehicle/data/EngineSpeed</uri>
     <sequence>2</sequence>
     <timestamp>2012-12-07T20:17:32Z</timestamp>
     <EngineSpeed>
           <timestamp>2012-12-07T20:17:32Z</timestamp>
           <value>3600.0</value>
     </EngineSpeed>
</data>
```

JSON

```
{ data : { “short_name” : “RPM_Every_Minute”,
             “uri” :  “vehicle/data/EngineSpeed”,
             “sequence” : 2,
             “timestamp” : “2012-12-07T20:17:32Z”,

             “EngineSpeed” : { “timestamp”  : “2012-12-07T20:17:32Z”,
                               “value” :  “3600.0” } } }
```

Example event message record: delivery of vehicle speed (RPM) as an alarm

The data delivered with an alarm is the value read that triggered the alarm. It is the responsibility of the remote consumer to either “remember” what the alarm associated with this short name is for, or to read the alarm record to determine the alarm configuration.

XML

```
<alarm>
      <short_name>Speed_Limit</short_name>
      <uri>vehicle/data/VehicleSpeed</uri>
      <sequence>2</sequence>
      <timestamp>2012-12-07T20:17:32Z</timestamp>
      <VehicleSpeed>
           <timestamp>2012-12-07T20:18:33Z</timestamp>
           <value>93.5</value>
      </VehicleSpeed>>
</alarm>
```

JSON

```
{ “alarm” : { “short_name” : “Speed_Limit”,
                “uri” : “vehicle/data/VehicleSpeed”,
               "sequence" : 2,
               "timestamp" : "2012-12-07T20:17:32Z",
               “VehicleSpeed” : { “timestamp” : “2012-12-07T20:18:33Z”,
                                  “value” : “93.5” } } }
```

Hardware interfaces

You can directly interact with some of the WVA hardware interfaces. The hardware interfaces must be enumerated, and operations defined.

Hardware interface URIs

hw

hw/buttons

hw/buttons/button_name

hw/leds

hw/leds/led_name

hw/time

hw/buzzer

hw/fw_update

hw/reboot

hw

The **hw** URI gets a list of categories of manageable hardware interfaces.

URI path

```
hw
```

Supported request methods

GET

Queries the system for the list of categories of hardware interfaces manageable by web services. The data record returned is a list of URIs corresponding to the “children” of **hw** in the web services data tree.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within an **hw** block. For example:

```
<hw>
        <element>hw/buttons</element>
        <element>hw/leds</element>
        <element>hw/time</element>
</hw>
```

JSON

The result URI strings are collected in an array assigned to a hw field. For example:

```
{ “hw” : [ “hw/buttons”, “hw/leds”,  “hw/time” ] }
```

hw/buttons

The **hw/buttons** URI gets a list of manageable buttons.

URI path

```
hw/buttons
```

Supported request methods

GET

Query the system for the list of addressable buttons manageable by web services. The data record returned is a list of URIs corresponding to the “children” of **hw** in the web services data tree.

Supported content types:

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are wrapped in element tags, and returned in a **buttons** block. For example:

```
<buttons>
        <element>hw/buttons/button_1</element>
        <element>hw/buttons/button_2</element>
        :>
        <element>hw/buttons/button_N</element>
</buttons>
```

JSON

The result URI strings are collected in an array assigned to a buttons field. For example:

```
{ “buttons” : [ “hw/buttons/button_1”,
                “hw/buttons/button_2”, ...
                “hw/buttons/button_N” ] }
```

hw/buttons/*button_name*

The **hw/buttons/*button_name*** URI reads the state of the named button.

URI path

```
hw/buttons/button_name
```

Supported request methods

GET

Queries the state of the named button.

Supported content types

XML

Button records can have one of the following as contents:

```
<button>down</button>
<button>up</button>
```

JSON

Button records can have one of the following as contents:

```
{ “button” : “down” }
{ “button” : “up” }
```

hw/leds

The **hw/leds** URI gets a list of manageable LEDs.

URI path

```
hw/leds
```

Supported request methods

GET

Query the system for the list of addressable LEDs manageable by web services. The data record returned is a list of URIs corresponding to the “children” of **hw/leds** in the web services data tree.

On the WVA, there is a **dimmer** feature that is exposed as an LED for control purposes.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are wrapped in **element** tags, and returned within an **leds** block. For example:

```
<leds>
        <element>hw/leds/led_1</element>
        <element>hw/leds/led_2</element>
        :
        <element>hw/leds/led_N</element>
</leds>
```

JSON

The result URI strings are collected in an array assigned to a **buttons** field. For example:

```
{ “leds” : [ “hw/leds/led_1”, “hw/leds/led_2”,  ...
                “hw/leds/led_N” ] }
```

hw/leds/*led_name*

The **hw/leds/*led_name*** URI manipulates the named LED.

On the WVA, you can use the special LED name **dimmer** to manipulate the hardware LED dimmer (**off** for bright, **on** for dim).

URI path

```
hw/leds/led_name
```

Supported request methods

GET

Queries the state of the named LED.

PUT

Changes the state of the named LED.

Supported content types

XML

LED records may have one of the following as contents:

```
<led>on</led>
<led>off</led>
```

JSON

LED records can have one of the following as contents:

```
{ “led” : “on” }
{ “led” : “off” }
```

hw/time

The **hw/time** URI gets the system time for the WVA's real time clock in the WVA. Time values must follow standard XML date and time formatting rules. Times must be expressed in UTC (Coordinated Universal Time).

URI path

```
hw/time
```

Supported request methods

GET

Queries the device system time, in UTC.

PUT

Set the device system time, in UTC.

Supported content types:

XML

```
<time>2013-01-05T09:10:00Z</time>
```

JSON

```
{ “time” : “2013-01-05T09:10:00Z” }
```

hw/buzzer

The **hw/buzzer** URI manipulates the buzzer, also known as the audible alarm.

URI path

```
hw/buzzer
```

Supported request methods

GET

Queries the state of the buzzer.

PUT

Change the state of the buzzer.

Supported content types

XML

Buzzer records can have one of the following as contents:

```
<buzzer>on</buzzer>
<buzzer>off</buzzer>
```

JSON

Buzzer records may have one of the following as contents:

```
{ “buzzer” : “on” }
{ “buzzer” : “off” }
```

hw/fw_update

The **hw/fw_update** URI initiates or tests the status of a firmware update. Updating the firmware in the WVA is a three stage process:

1. The firmware update image must be transferred to the device. This transfer operation can be performed through Digi Remote Manager, the web user interface, or web services. See Filesystem interfaces for more information.
2. This web services URI can be used to initiate a firmware update using the uploaded file.
3. This web services URI can be used to determine the status of the firmware update.

URI path

```
hw/fw_update
```

Supported request methods

GET

Query the state of the firmware update process (if any). Possible firmware update states include:

- **running**: Normal running, no update in progress.
- **updating**: Firmware update is in progress.
- **updatefailed**: The most recent firmware update failed.
- **unknown**

PUT

Initiate a firmware update with a target file already in the filesystem.

Supported content types

GET records

XML

```
<fw_update><status>update_state</status></fw_update>
```

JSON

```
{ “fw_update”        : { “status” : “update_state” }        }
```

PUT records

XML

```
<fw_update><filename>file_path</filename></fw_update>
```

JSON

```
{“fw_update” : { “filename” : “file_path”        } }
```

hw/reboot

The **hw/reboot** URI reboots the WVA system.

URI path

```
hw/reboot
```

Supported request methods

PUT

Makes the WVA system perform an orderly reboot. The contents of the payload of the message are unimportant, and ignored.

State interfaces

The WVA state interfaces read information about the WVA's internal state. State information is read-only. A state is organized in groups of related items.

Some state groups contain multiple instances. For example, a product with multiple network interfaces has one unique instance for network settings per interface.

Some state groups contain nested subgroups, organized in a tree structure.

Each final state group contains a number of information elements. An element contains a single state information value. Each element is addressed individually with a unique URI.

The request URI for state interfaces contains a **state_path**. The path is a list of groups, subgroups, and instances separated by "/". Subgroups and instances are optional, or may be repeated.

The state interface URIs have this general format:

```
state/group/subgroup/instance/element
```

State interface URIs

state

state/state_path (containing subgroups or instances)

state/state_path (containing elements)

state/state_path/element_name

state

The **state** URI gets a list of state categories.

URI path

```
state
```

Supported request method

GET

Queries the system for a list of URIs corresponding with the state groups in the system.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned within a **state** block. For example:

<state>

```
<state>
          <element>state/device_info</element>
          <element>state/device_stats</element>
          <element>state/interface_info</element>
</state>
```

JSON

The result URI strings are collected in an array assigned to a state field. For example:

```
{ “state” : [ “state/device_info”, “state/device_stats”,
              “state/interface_info” ]  }
```

state/*state_path* (containing subgroups or instances)

The **state/*state_path*** URI gets list of subgroups or instances for the named state path. The ***state_path*** could incorporate indexes for state path elements as independent path elements following the indexed element name. For example, a query for **state/interface** returns a subtree including available interfaces. A query for **state/interface_info/wlan0** returns data for that specific interface only.

URI path

```
state/state_path
```

Supported request method

GET

Queries the system for a list of URIs corresponding with the subgroups or instances of the named state group in the system.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in element tags, and returned in a block corresponding to the name of the state group. For example:

```
<interface_info>
      <element>state/interface_info/eth0</element>
      <element>state/interface_info/wlan0</element>
</interface_info>
```

JSON

The result URI strings are collected in an array assigned to a field corresponding to the name of the state group. For example:

```
{ “interface_info” : [ “state/interface_info/eth0”,
                       “state/interface_info/wlan0” ] }
```

state/*state_path* (containing elements)

The **state/*state_path*** URI gets a list of elements for the named state path.

URI path

```
state/state_path
```

The ***state_path*** could incorporate indexes for state path elements as independent path elements following the indexed element name. For example, a query for **state/interface** returns a subtree including available interfaces.

Supported request method

GET

Queries the system for a list of URIs corresponding with the elements of the named state group in the system.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned in a block corresponding to the name of the state group. For example:

```
<interface_info  name="eth0">
       <element>state/interface_info/eth0/mac</element>
       <element>state/interface_info/eth0/ip</element>
</interface_info>
```

JSON

The result URI strings are collected in an array assigned to a field corresponding to the name of the state group. For example:

```
{ "interface_info" : [ "state/interface_info/eth0/mac",
                       "state/interface_info/eth0/ip" ] }
```

state/state_path/*element_name*

The **state/state_path/*element_name*** URI queries the system for a specific element of the named state group in the system.

URI path

```
state/state_path/element_name
```

Supported request method

GET

Queries the system for a specific element of the named state group in the system.

Supported content types

XML

The result is returned in an element with a name corresponding with the item being requested. For example:

```
<mac> 00:40:9d:68:77:aa</mac>
```

JSON

The result is assigned to a field corresponding to the requested value. For example:

```
{ “mac” : “00:40:9d:68:77:aa”  }
```

Filesystem interfaces

The WVA offers a mechanism to access files and directories for a variety of purposes, but primarily information storage. Occasional uses might include pushing a firmware update to the WVA, or examining the system event log.

The WVA has a volume, named **userfs** that you can use for custom file storage. Files in this volume are stored in a hierarchical directory structure. There are several system-specific directories and files in **userfs** that you cannot manipulate.

In the WVA filesystem, a **path** refers to a specification for a directory within the volume. Individual files have a **filename** in a particular "path."

Other volume names correspond to storage space on USB devices. An example USB volume name is **sda1**.

For consistency with a wide range of existing Digi products, the WVA has two directory paths pre-existing when the system is created: **/userfs/WEB** and **/userfs/WEB/python**. Both paths are writable, and you can create custom directory trees below these paths.

Note The WVA filesystem interface is used to manipulate specific files in the embedded system at relatively fixed paths. An external application that uses web services must know the paths to the files, and know which paths and names are directories versus files. The filesystem interface itself does not provide hints as to the types of file names found in the directory tree.

Filesystem interface URIs

files/volume/path

files/volume/path?type=dir

files/volume/path/filename

files/*volume*/*path*

The **files**/***volume***/***path*** gets a list of directory contents of the path, or deletes an empty directory.

URI path

```
files/volume/path
```

Where:

volume is either userfs or a USB flash drive name.

path refers to a directory in this system, and is constructed from a list of directory names separated by a **/**, indicating a specific location within the directory tree. Example directory paths include:

```
userfs/WEB
userfs/WEB/python
```

If the path is omitted, the root directory of the volume is manipulated.

Supported request methods

GET

Queries the system for the directory listing associated with the specified path.

DELETE

Removes the specified directory path if it is empty.

Supported content types

HTML

For a **GET**, the result URIs are turned into URLs relative to the device, and returned in an HTML list, allowing browsing of the directory structure.

XML

The **GET** form creates a directory listing record, with the top level node the same as the trailing directory name in the path. The full URI to each entry in the directory is included in the list. An example **GET** from **/userfs/WEB/python** might include:

```
<file_list>
         <file>files/userfs/WEB/python/name1</file>
         <file>files/userfs/WEB/python/name2</file>
</file_list>
```

JSON

The **GET** form returns a list of URIs corresponding to the contents of the requested directory. The list is assigned to a field in a new object corresponding to the trailing directory name in the path. An example **GET** from **/userfs/WEB/python** might include:

```
{ “file_list” : [ “files/userfs/WEB/python/name1”,
                  “files/userfs/WEB/python/name2”  ] }
```

files/*volume*/*path*?type=dir

The **files/*volume*/*path*?type=dir** URI creates a new directory. The named path must not yet exist. The parent directory of the named path must exist.

URI path

```
files/volume/path?type=dir
```

Supported request method

PUT

Create a directory with the specified path.

files/*volume*/*path*/*filename*

The **files/*volume*/*path*/*filename*** URI manipulates the named file in the named path.

URI path

```
files/volume/path/filename
```

Supported request methods

GET

Queries the system for the file at the specified path with the supplied filename. The file contents are returned as the payload of the message response.

PUT

If the named file does not yet exist, adds a new file (with the requested URI) to the system. If the requested file already exists in the system, changes the existing file. The payload of the PUT command become the contents of the file.

DELETE

Removes the specified file from the system.

Configuration interfaces

The WVA has an interface to the configuration settings for the device. In addition, WVA web services are available for device configuration. Configuration settings are organized in groups of related settings, which are intended to be manipulated together. Some settings groups have multiple instances. For example, a product with multiple network interfaces has one unique instance for network settings for each network interface.

Configuration interface URIs

config

config/settings_group

config/settings_group (with instances)

config/settings_group/instance_specifier

config/factory_default

config

The **config** URI gets a list of configuration setting groups.

URI path

```
config
```

Supported request methods

GET

Queries the system for a list of URIs corresponding with the settings groups in the system.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned in a **config** block. For example:

```
<config>
      <element>config/system</element>
      <element>config/button</element>
      <element>config/led_control</element>
</config>
```

JSON

The result URI strings are collected in an array assigned to a **config** field. For example:

```
{ “config” : [ “config/system”, “config/button”,
               “config/led_control” ] }
```

config/*settings_group*

The **config/*settings_group*** URI manipulates a single settings record, for settings without an instance specifier.

URI path

```
config/settings_group
```

Supported request methods

GET

Queries the settings, returning the named settings instance record.

PUT

Changes the settings instance to match the supplied record.

Supported content types

XML

Settings are returned inside tags matching the name of the settings group, with the appropriate instance specifier inserted if applicable. For example, here is a settings group without an instance specifier:

```
<http>
          <enable>on</enable>
          <port>80</port>
</http>
```

Here is an example of a settings group with an instance specifier:

```
<interface  name="wlan0">
          <static>off</static>
          <dhcp>on</dhcp>
          <ip>0.0.0.0</ip>
          <subnet>0.0.0.0</subnet>
          <gateway>0.0.0.0</gateway>
</interface>
```

JSON

Settings are returned as an object with a field whose name matches the settings group, and whose value is an object with fields matching the settings record fields. If there is an instance specifier associated with the record, it is not present in the JSON; it only is present in the request URI.

Here is an example of a settings group without an instance specifier:

```
{  “http” : { “enable” : “on”, “port” : 80 } }
```

Here is an example of a settings group with an instance specifier:

```
{  “interface” : { “static” : “off”, “dhcp” : “on”,
   “ip” : “0.0.0.0”, “subnet” : “0.0.0.0”,
   “gateway” : “0.0.0.0” } }
```

config/*settings_group* (with instances)

The **config/*settings_group***URI gets a list of known instance specifiers, for settings that have them. Each settings group has its own unique elements, and those settings records are not documented here. They closely match the format available from the device via Digi's RCI protocol. The method for pulling the current configuration and pushing configuration changes to individual settings records follows.

URI path

```
config/settings_group
```

Supported request method

GET

Queries the system for a list of URIs corresponding to the instances of the named settings group in the system.

Supported content types

HTML

The result URIs are turned into URLs relative to the device, and returned in an HTML list.

XML

The result URIs are each wrapped in **element** tags, and returned within a block corresponding to the name of the settings group. For example:

```
<interface>
        <element>config/interface/wlan0</element>
</interface>
```

JSON

The result URI strings are collected in an array assigned to a **config** field. For example:

```
{ “interface” : [ “config/interface/wlan0” ] }
```

config/*settings_group*/*instance_specifier*

The **config/*settings_group*/*instance_specifier*** URI manipulates a single settings record, for settings with an instance specifier.

URI path

```
config/settings_group/instance_specifier
```

Supported request methods

GET

Queries the settings, returning the named settings instance record.

PUT

Changes the settings instance to match the supplied record.

Supported content types

XML

Settings are returned inside tags matching the name of the settings group, with the appropriate instance specifier inserted if applicable. For example, here is a settings group without an instance specifier:

```
<http>
          <enable>on</enable>
          <port>80</port>
</http>
```

Here is an example of a settings group with an instance specifier:

```
<interface  name="wlan0">
          <static>off</static>
          <dhcp>on</dhcp>
          <ip>0.0.0.0</ip>
          <subnet>0.0.0.0</subnet>
          <gateway>0.0.0.0</gateway>
</interface>
```

JSON

Settings are returned as an object with a field whose name matches the settings group, and whose value is an object with fields matching the settings record fields. If there is an instance specifier associated with the record, it is not present in the JSON; it only is present in the request URI.

Here is an example of a settings group without an instance specifier:

```
{  “http” : { “enable” : “on”, “port” : 80 } }
```

Here is an example of a settings group with an instance specifier:

```
{  “interface” : { “static” : “off”, “dhcp” : “on”,
   “ip” : “0.0.0.0”, “subnet” : “0.0.0.0”,
   “gateway” : “0.0.0.0” } }
```

config/factory_default

The **config/factory_default** URI resets configurable settings to their factory defaults.

URI path

```
config/factory_default
```

Supported request method

PUT

Causes a reset of the configurable settings to their factory defaults. Any content-related headers (for example, **Accept** and **Content-Type**) are ignored. The payload of the message is unimportant, and is ignored.

Password interface

A password interface is available to manage the password for protected URIs.

password

The **password** URI sets the password for **admin** username.

URI path

```
password
```

Supported request methods

PUT

Sets the **admin** password required to modify protected items.

Supported content types

XML

```
<password>new_password</password>
```

JSON

```
{ “password” : “new_password” }
```

HTTP response codes

This topic provides the HTTP response codes generated by the web server and the web services code, along with the contexts in which the response codes may be returned. When using the RESTful interface, the RESTful web services API calls use HTTP response codes to indicate the status of any transaction.

200 (“OK”)

This code is the standard code for a successful response. The document payload in a **GET** request is as documented. A **PUT**, **POST**, or **DELETE** request has had its effect.

400 (“Bad Request”)

Returned under the following conditions:

- The URI cannot be parsed, particularly if the URI is too long for the internal buffering of the web services engine.
- The web services are unable to parse a document successfully based on the content type.
- Failure when attempting to set the time.
- Failure parsing the fields of a record in a **PUT** operation. For example, the URI specified for an alarm does not reference something that supports alarms.

401 ("Unauthorized")

The request requires authentication. The client should repeat the request with HTTP basic authentication using the admin login and password.

If the request already included authentication, the login and password are incorrect.

403 (“Forbidden”)

Returned when the specified URI in the request URL is not one that can be managed by the web services user due to permissions embedded within the system. This failure is most common when when you are manipulating files in the filesystem. See Filesystem interfaces.

404 (“Not Found”)

Returned under the following conditions:

- The specified URI in the request URL is not one recognized by the web services system.
- For alarms and subscriptions, a **GET** or **DELETE** of an as-yet unknown URI is performed. Note that a **PUT** can be made to a previously unknown URI to create an alarm or subscription without generating this error.

405 (“Method Not Allowed”)

Returned when the requested URI is recognized, but the HTTP method (for example, **GET** or **PUT**) in use is not supported in conjunction with that URI. Seee Index of web services resources to determine which methods are compatible with which URIs.

406 (“Not Acceptable”)

Returned when an HTTP method (for example, **GET**) requests a response, but the server cannot deliver a document with that content type for the specified URI. This failure could happen if a request arrived with an **Accept: text/html** header, and the URI was XML or JSON only.

414 (“Request-URI Too Long”)

Returned when the requested URI is longer than the web services system is able to parse.

415 (“Unsupported Media Type”)

Returned when an HTTP method (for example, **PUT**) supplies a document with a content type that is unexpected for the specified URI. This failure could happen if a document arrived with a **Content-type: text/html** header, and the URI was XML or JSON only.

500 (“Internal Server Error”)

Returned when there are unexpected errors unrelated to the requests or responses themselves. Possible conditions resulting in this code include:

- An unexpected failure registering an alarm.
- An unexpected failure querying the vehicle bus subsystem.
- An inability to read or write the internal resources associated with LEDs.

503 (“Service Unavailable”)

Returned when a data query could not be fully processed due to a transient state in the web server. Possible conditions resulting in this code include:

- A temporary lack of memory to allocate for parsing requests or generating responses can generate this response.
- A vehicle bus data request is made to a URI known to potentially be available on the bus, but the Digi device has not yet received the specified data.

Programming

These topics discuss aspects of developing application programs for the WVA.

WVA file system

The WVA has a Linux-based filesystem. This section gives an overview of the key directories of the filesystem and common operations performed on directories and files.

Important directories

The **/WEB/python/** directory is for user-specific files, such as custom applications. Subdirectories can be created in this area for the customer's applications. This area is read-write.

The **/WEB/logging** directory contains system log files, including eventlog.txt, python.log, and digi.log. These files are read-only. For more information on these files, see the description of the system log in the Troubleshooting section of the *Wireless Vehicle Bus Adapter Getting Started Guide*.

For more information on the filesystem, see Filesystem interfaces.

Access/browse the filesystem from device interfaces

There are several ways you can interact with the filesystem resident on the WVA:

- Through the web services filesystem interfaces. These resources are used to browse and create directories and manipulate files. See Filesystem interfaces.
- Through the web interface for several operations, including firmware updates and backup/restore operations. See Configure the WVA in the *Wireless Vehicle Bus Adapter Getting Started Guide*.
- Through the Digi Remote Manager File Management functions. See Managing Device Files in the *Digi Remote Manager User Guide*.

Demo application and resources for Android developers

The source code for a WVA demo application is available on GitHub, and includes a library and source code repository for details on requirements, setup, and building the application. The WVA Android library provides an API for accessing the web services and event channel of a WVA.

For information see:

- Wireless Vehicle Bus Adapter Android demo application
- WVA Android Library source code repository README on GitHub
- *Wireless Vehicle Bus Adapter Getting Started Guide*
- *WVA Android Library Tutorial*

Real time clock

The WVA has a real time clock. The time for this clock is set by the web services resource hw/time.

In the application code, note that the application must set the real-time clock before any firmware updates can be performed.

Security features in the WVA

There are several security features in the WVA.

- For Wi-Fi security, the WVA uses WPA2 with pre-shared key (PSK).
- For user authentication, the WVA uses HTTP authentication.

Security for the Wi-Fi communications channel

The basic channel for communication between the WVA and a connected device is a Wi-Fi communication channel.

The security in place over the Wi-Fi communications channel varies depending on the network connection type:

- **When the WVA is the access point**: The Wi-Fi channel is secured using WPA2 with a pre-shared key, also known as WPA-PSK. WPA2 provides encryption over the channel, while the pre-shared key provides authentication.
- **When the connected device is the access point**: The connected device selects which kind of security key is in place. Some devices, such some models of smart phones, do not have security.
- **When in Wi-Fi Direct mode**: The Wi-Fi channel is secured using WPA-PSK, with the key being selected dynamically per the Wi-Fi Direct specification.

Security for activities performed over the Wi-Fi communications channel

Several activities performed over the Wi-Fi communications channel implement security.

Activity	Protocol	Security in place
Web server, including web interface	HTTP/HTTPS	By default, the web interface uses HTTPS (encrypted), with HTTP basic authentication. The WVA devices creates a unique certificate for self-identification. Any display of warnings about certificates is handled through the web browser. In applications, code can handle the certificate management step.
Web services basic request/response	HTTP/HTTPS	Some web services are protected with HTTP basic authentication. See the Index of web services resources. The **Protected URI** column identifies protected resources.
Web Services Event Channel	TCP	The Event Channel is a read-only channel using a dedicated TCP port. The channel is unencrypted.

Activity	Protocol	Security in place
Digi Remote Manager	EDP over SSL over TCP	Digi always requires that the channel for Digi Remote Manager communications from a Digi device be secure and encrypted, and that the server be verified.

Modifying the security model

You can enable a different security model through the password web services resource. This resource sets the password for the **admin** username for the WVA.

CAUTION! Changing the **admin** password can make pairing of the WVA with a connecting device more difficult. A lost password could result in users not being able to access the WVA.

Power management

The WVA has a power management scheme built into its device configuration settings, on the **Configuration > Power Management** page in the web interface. The default for power management is that all of Power Management is disabled (off).

For configuration information, see Configure power management settings in the *Wireless Vehicle Bus Adapter Getting Started Guide*.

The basic power management settings allow for control over the following:

- **Sleep mode**: Determines whether the WVA goes into sleep mode when no engine activity (**RPM=0**) is detected. The default is disabled (**off**).
- **Wake on movement**: Determines whether the WVA wakes from sleep mode upon detection of vibration, typically a sharp vibration to the vehicle frame. The default is disabled (**off**).
- **Wake on alternator**: Determines whether the WVA wakes from sleep mode upon detection of the vehicle's alternator running on alternator power rather than the vehicle battery. The default is disabled (**off**).
- **Wake on Button press**: Controls whether the WVA wakes from sleep mode by pressing the button. The default is disabled (**off**).
- **Periodic Wake Settings**: Allows for a periodic or timed wake from sleep mode for the device. The default is that periodic wake is disabled (off). If enabled, the wake timer can range from **1** to **1440** minutes (**24** hours), and the default is **5** minutes.