Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Tech Support: support@parallax.com

Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Tech Support: (888) 997-8267

Parallax Wi-Fi Module Firmware Guide

(#32420D DIP version, and 32420S SIP version)

This documentation is written for Parallax Wi-Fi Module firmware v1.0 (2016-11-02 18:04:30). For details about the Parallax Wi-Fi Module features and hardware, see the Downloads section of the 32420S or 32402D product pages at www.parallax.com.

Table of Contents

Module Interfaces

The Parallax Wi-Fi Module provides a wireless communication link between a microcontroller and IP-networked devices. It has two wireless network interfaces, Station and AP (a.k.a. "Soft AP") for communicating with wireless devices, and one wired interface (called Serial) for communicating with a microcontroller.

Wireless Interfaces

One (or both) of the wireless interfaces (Station and/or AP) can be made active at any time. Typically, the AP (SoftAP) interface is used to configure its Station interface to connect to a shared Wi-Fi Access Point for long-term use. After configuration, the AP (SoftAP) interface should be turned off since it is "open" (insecure). However, the SoftAP interface can serve as the sole wireless interface for long-term use, if desired.

With the module powered up, its Associate LED (marked ASC or ASSOC) indicates which wireless interface is active and what kind of connectivity is available.

Associate (ASC) LED Display

	STA (no IP) *no connection*	**STA** (has IP) *connect via STA*	**AP** *connect via AP*	**STA+AP** (no STA IP) *connect via AP*	**STA+AP** (has STA IP) *connect via STA or AP*
LED	OFF	long OFF, blink ON	ON	long OFF, long ON	long OFF, blink ON, long ON

Any visible "ON" state of the Associate LED indicates there is *some kind* of wireless access possible, either via the STA (Station) or AP (Soft AP) interfaces. When the AP interface is active, it *always* has an IP address- wireless access to the module is always possible via the AP interface during any of the last three conditions shown.

Any of the above modes can be configured via the module's Serial interface or Wireless interface (using Serial or Network commands) or via the convenient Wi-Fi Networks web page, http://<*ip_address*>/wifi/wifi.html, as long as there's an available wireless connection. In a situation where there is no wireless connection (STA mode with no IP address) the module can be forced to turn on its AP (SoftAP) interface by toggling its RES pin low/high four times within two seconds. If the module is plugged into the WX socket of a Parallax development board, the board's Reset button can pressed/released at this rate to force AP mode, then a Wi-Fi device (laptop/tablet/smartphone) can connect to the module's SSID to get to it's Wi-Fi Networks web page.

Commands for use over the wireless interfaces are detailed in the Network Commands section.

Serial Interfaces

The Serial interface consists of direct-connect pins, Data Out (DO) and Data In (DI), that use an asynchronous serial protocol similar to RS-232. The DO and DI pins are the only connections used to establish a bi-directional wired communication interface with a microcontroller. Commands for use over the wired serial interface are detailed in the Serial Commands section.

Serial Interface Modes

There are two modes of communication over the Serial interface; Transparent and Command mode. Transparent mode lets data pass through without modification. Command mode provides run-time control to an attached microcontroller; allowing it to set or check module settings, establish connections, and process communications in special ways. By default, the module's Serial interface begins in Transparent mode.

Setting Command Mode

The serial interface can be set to Command mode in three ways:

1. Via its Settings web page
    a. Connect through its SoftAP interface:
        i. Configure a device (computer, smartphone, etc.) to join its access point
        ii. Navigate the device's browser to http://192.168.4.1/settings.html
    -- *OR* --
    b. Connect through its Station interface:
        i. Use a device (computer, smartphone, etc.) on the same shared access point
        ii. Navigate the device's browser to http://<module_IP_address>/settings.html
            1. Find the module IP address with the command-line Prop-Loader tool
                a. `$ proploader -W`
    -- *THEN* --
    c. Change the *Serial Commands* setting to *Enabled*
2. Via a network command
    a. **POST** /wx/setting?name=cmd-enable&value=1
3. Via a Break Condition on the Serial interface itself
    a. Set the DI pin low for $\geq \frac{30}{Communications\ Baud\ Rate}$seconds

Once in Command mode, communication from the microcontroller will be interpreted and processed as special commands, or passed through as-is, depending on the context and format of that communication.

Setting Transparent Mode

The serial interface can be set to Transparent mode in three ways:

1. via its Settings web page
    a. Follow steps 1a or 1b, above, then,
    b. Change the *Serial Commands* setting to *Disabled*
2. via a network command
    a. **POST** /wx/setting?name=cmd-enable&value=0
3. via a serial command
    a. **SET**:cmd-enable,0

In Transparent mode all communication from the microcontroller passes through to the network as-is. This mode uses the module as a wireless serial interface, transporting serial traffic using the RS-232-like asynchronous serial protocol over the DO and DI pins, and transporting IP traffic using a subset of the TELNET protocol over port 23.

Network Commands

The following are commands (also known as requests) for use over the module's wireless network interface. They can be issued via a web browser (using a special set of web pages written in HTML and Javascript, for example), or with developer tools or dedicated applications (that communicate using HTTP, WebSocket, or TCP protocols). *A handy developer tool is the Chrome browser application "Postman" found in the Chrome web store.*

IMPORTANT: Network commands and arguments are **case-sensitive**.

Network Command Syntax Legend:

BOLD	- command or setting name
Normal	- case-sensitive command argument; must be entered exactly as shown
Italicized	- item must be replaced by user input
[optional...]	- square brackets denote optional, possibly repeating items
<data>	- stream of data delivered in the body of the command

User File System

The user file system is a built-in, non-volatile storage area for hosting custom files. Users can store custom files (such as web pages and microcontroller application images) for client access at a later time. For example, after placing an html page called “test.html” into the user file system using a POST command, the file can be accessed by entering http://*<module_ip_address>*/files/test.html into their web browser.

POST /userfs/format

Reformats the user file system, removing all files from the logical **/files** path in the process.

POST /userfs/write?file=*name <file_data>*

Writes a file called *name* to the user file system containing *<file_data>* (from the body of the **POST** request). If a file already exists with that name, it is marked as deleted and the new file replaces it. However, the space occupied by the old copy is not recovered until reformatted.

GET /files/*name*

Retrieves the *name* file from the user file system.
For example, if a file named "myfile.html" exists in the file system, the command "**GET** http://*<module_ip_address>*/files/myfile.html" will retrieve it.

Propeller Loader

The Parallax Wi-Fi Module includes built-in Propeller application loader capability through the use of these POST commands.

POST /propeller/load?*argument* [&*argument...*] *<propeller_application>*

Loads a small (< 2 KByte) Propeller .binary file to Propeller RAM over the HTTP connection. This is commonly used to deliver a small second-stage loader application which then handles a full Propeller application delivery over TCP. The body of the **POST** contains the Propeller application to load. The body of the reply from the module contains the response sent by the Propeller, if any.

Arguments

baud-rate=*rate*

The baud *rate* used for loading the Propeller application image. If missing, the default is that of the module's "loader-baud-rate" setting.

final-baud-rate=*rate*

The baud *rate* that is switched to after the load completes. This determines the Serial interface speed for use during Propeller run time. If missing, the default is that of "baud-rate" (if provided) or "loader-baud-rate" (if "baud-rate" not provided).

reset-pin=*pin*

The module I/O *pin* used to reset the Propeller – i.e. it is connected to the Propeller's RESn pin. If missing, the default is that of the module's "reset-pin" setting.

response-size=*size*

The *size* of the response expected from the Propeller after the loaded Propeller application starts. This is typically used to alert the caller that the

second-stage loader is initialized and ready for further communication. If missing, or if its value is zero, no response is expected from the Propeller application.

response-timeout=*ms*

The number of milliseconds (*ms*) to wait for a response from the Propeller application. This is only used if response-size is set to a non-zero value.

<propeller_application>

The Propeller application image to load. This data should be provided in the body of the **POST** command and must be less than 2 Kb (kilobytes) in size.

POST /propeller/load-file?*argument* [&*argument...*]

Loads a Propeller application to RAM from the user file system. Application must be a .binary image file previously stored using "**POST** /userfs/write...".

Arguments

file=*name*

The *name* of the file to load. This must be a .binary image file previously stored in the user file system with "**POST** /userfs/write...".

baud-rate=*rate*

The baud *rate* used for loading the Propeller application image. If missing, the default is that of the module's "loader-baud-rate" setting.

final-baud-rate=*rate*

The baud *rate* that is switched to after the load completes. This determines the Serial interface speed for use during Propeller run time. If missing, the default is that of "baud-rate" (if provided) or "loader-baud-rate" (if "baud-rate" not provided).

reset-pin=*pin*

The module I/O *pin* used to reset the Propeller – i.e. it is connected to the Propeller's RESn pin. If missing, the default is that of the module's "reset-pin" setting.

POST /propeller/reset [?reset-pin=*pin*]

Resets the Propeller; similar to pressing and releasing the Reset button on the Propeller development board.

Pin

The module I/O *pin* used to reset the Propeller – it is connected to the Propeller's RESn pin. If missing, the default is that of the module's "reset-pin" setting.

WebSockets

GET /ws/*

Used to establish a websocket connection to the module. The header must contain the fields "Upgrade: websocket" and "Connection: Upgrade".

Settings

The Parallax Wi-Fi Module has many settings that determine default modes and behaviors. Using the *settings* commands below, they can be retrieved and adjusted in either a temporary or persistent fashion.

GET /wx/setting?name=*setting*

Retrieves the value of the named *setting*. Note: If *setting* is an I/O pin, such as **pin-gpio2**, the **GET** command will set the I/O pin to an input direction and read its state, leaving it set as an input. Use the **POST** command to set an I/O pin to an output.

Settings

version (read only)

View the firmware version string.

module-name

The module name appears as its SSID. This is limited to 32 characters.

wifi-mode

Module Wi-Fi mode can be "STA" (Station), "AP" (SoftAP), or both modes at once: "STA+AP" (Station+SoftAP).

wifi-ssid (read only)

View the currently connected access point SSID, if any.

station-ipaddr (read only)

View the current Station IP address – if connected in STA (Station) mode to an access point.

station-macaddr (read only)

View the Station MAC address – a fixed address uniquely identifying the STA (Station) wireless interface.

softap-ipaddr (read only)

View the current SoftAP IP address – if serving in AP (SoftAP) mode.

softap-macaddr (read only)

View the SoftAP MAC address – a fixed address uniquely identifying the AP (SoftAP) wireless interface.

cmd-start-char

The character used to denote the start of a Serial interface command or response. The default is 0xFE (254).

cmd-enable

Enable or disable Command mode on the Serial interface. When command mode is disabled, all data is passed through as-is (transparently) in both directions between the network and the attached microcontroller. Value can be 0 (disabled; the default) or 1 (enabled).

cmd-events

Enable or disable the delivery of events on the Serial interface. When events are disabled, notifications of connections, disconnections, and the arrival of data are reported through the **POLL** command. When events are enabled, notifications of connections, disconnections, and arrival of data are sent to the MCU immediately. The form of these notifications is identical to the responses to the **POLL** command except that the "=" is replaced by a "!". Value can be 0 (disabled; the default) or 1 (enabled).

loader-baud-rate

The baud rate of the Serial interface used for the Propeller Loader process. See Propeller Loader. Valid values are: 1200, 4800, 9600, 19200, 38400, 57600, 74880, 115200 (the default), 230400, 460800, and 921600. The proploader program, and the applications that use it (like SimpleIDE), always override this setting.

baud-rate

The baud rate of the Serial interface when not in the process of loading code. Valid values are: 1200, 4800, 9600, 19200, 38400, 57600, 74880, 115200 (the default), 230400, 460800, and 921600.

stop-bits

The number of stop bits per byte on the Serial interface when not in the process of loading code. Value can be 1 (the default), 1.5, or 2.

dbg-baud-rate

The baud rate of serial debug data on the module's DBG pin. Valid values are: 1200, 4800, 9600, 19200, 38400, 57600, 74880, 115200 (the default), 230400, 460800, and 921600.

dbg-stop-bits

The number of stop bits per byte of serial debug data on the DBG pin. Value can be 1 (default), 1.5, or 2.

reset-pin

The I/O pin used to reset the Propeller; default is 12. The proploader program, and applications that use it (like SimpleIDE), always override this setting.

connect-led-pin

The I/O pin used to indicate the module is associated with an access point (in Station mode); default is 5. This pin is connected to the Associate LED (ASC) on the module and on some development boards.

rx-pullup

Determines whether or not a pullup resistor is enabled on the DI pin. This is needed by some non-Parallax Wi-Fi ESP-based modules. Value 0 (default) = disabled; value 1 = enabled.

pin-gpio0

The logic level (0 or 1) of the PGM pin.

pin-gpio1

The logic level (0 or 1) of the RX pin.

pin-gpio2

The logic level (0 or 1) of the DBG pin.

pin-gpio3

The logic level (0 or 1) of the TX pin.

pin-gpio4

The logic level (0 or 1) of the SEL pin.

pin-gpio5

The logic level (0 or 1) of the ASC pin.

pin-gpio12

The logic level (0 or 1) of the DTR pin.

pin-gpio13

The logic level (0 or 1) of the CTS pin.

pin-gpio14

The logic level (0 or 1) of I/O pin 14.

pin-gpio15

The logic level (0 or 1) of the RTS pin.

POST /wx/setting?name=*setting*&value=*value*

Adjusts a module *setting* to *value*. Valid *settings* are those not marked as "(read only)" in the **GET** command descriptions, above. Note: If *setting* is an I/O pin, such as **pin-gpio2**, the **POST** command will set the I/O pin to an output direction during the process of setting its state. Use the **GET** command to set an I/O pin to an input.

Setting

The name of the desired *setting*; described in the **GET** command, above.

Value

The value to change the *setting* to. Each *setting* has a range of values described in the **GET** command, above.

POST /wx/save-settings

Makes the current module settings persistent– saved across module resets and power cycles. Without this command, all changes to settings will only last during the current module session.

POST /wx/restore-settings

Restores the module settings from persistent storage. Use this to undo temporary changes to settings.

POST /wx/restore-default-settings
Restores the factory default module settings. This has a temporary effect unless followed by a "**POST** /wx/save-settings" command.

Serial Commands

The commands in this section are for use over the wired serial interface (DI and DO pins). They can be issued via an attached microcontroller using the RS-232-like asynchronous serial protocol.

IMPORTANT: For any serial commands to be processed correctly, the module must first be placed in Command mode. See the Setting Command Mode section to do this.

Serial Command Format

Commands are expressed in either text form, like "**CLOSE**" (five characters), or in token form, like **232** (or hexadecimal E8; one byte which means CLOSE). The module's command interpreter automatically understands each form but always responds back in text form. Text form is easier to read and type on a terminal, with required colon ':' and comma ',' delimiters, and token form is convenient for a microcontroller to transmit (extra characters and delimiters are eliminated). See the Tokens section for a list of command and argument tokens.

Command transmissions to the Wi-Fi Module always start with A *begin* marker byte (B) and finish with an *end* marker byte (E). The byte values of the *begin* and *end* markers themselves may be configurable in future firmware, so all syntax examples show them as the symbols B and E. By default, the *begin* (B) marker is a byte value of 254 (hexadecimal FE) and the *end* (E) marker is a byte value of 13 (hexadecimal 0D; also known as a carriage return or as "\r" in some programming languages).

Serial Stream Example (22 bytes on DI pin)

-	p	a	s	s	t	h	r	u	-	B	C	O	M	M	A	N	D	E	.	.	.

In command mode, all data bytes on the serial interface's DI pin leading up to a *begin* (B) marker are passed through to the wireless interface. Data that appears in-between a *begin* (B) and *end* (E) marker is parsed by the Wi-Fi Module as a command. Data immediately after the *end* (E) marker is passed through to the wireless interface– except for after **SEND** and **REPLY** commands which specify that a certain number of bytes of binary payload follows. Binary payload is not translated; it can contain any data bytes, even those equivalent to the *begin* (B) and *end* (E) markers without problem.

Data received on the wireless interface is processed by the module and stored until the microcontroller requests it (with **POLL** or **RECV**, for example) at which point the Wi-Fi Module formats it as specified by the command and transmits it on the serial interface's DO pin.

Example Serial Command (DI pin)

Using these rules, the example syntax for the **CLOSE** command is:

B **CLOSE**:*handle_id* E

To transmit this command in text form (8+ bytes):

- send the *begin* byte value (254 by default)
- followed by the six characters "CLOSE:"
- then one or more characters representing the decimal value *handle_id*
- and finally the *end* byte value (13 by default)

To transmit this command in token form (5 bytes):

- send the *begin* byte value (254 by default)
- followed by the CLOSE command's token byte value 232
- then the byte indicator value 252 and the byte-sized value *handle_id*
- and finally the *end* byte value (13 by default)

Command responses from the Wi-Fi Module always start with two bytes, a *begin* (B) marker followed by an equal sign (=), and finish with an *end* (E) marker.

Example Serial Response (DO pin)

A successful **CHECK wifi-mode** command responds in the following form:

b = S,*mode* E

Assuming the current mode is STA+AP, the response will be (11 bytes):

- the *begin* byte value (254 by default)
- the nine characters "=S,STA+AP"
- and finally the *end* byte value (13 by default)

Microcontroller Networking

A microcontroller attached to the Wi-Fi Module's serial interface uses serial commands to set up Listeners and Connections to actively manage communication with the network. In this way, the Wi-Fi Module + microcontroller pair becomes a server and/or client. Listeners stay active until the microcontroller issues a CLOSE command or a Break Condition. Connections stay active until either the remote side closes the connection or the microcontroller issues a Break Condition.

While in Command mode, any Break Condition issued on the DI pin will deactivate (clear) all Listeners and Connections that were previously set up and will leave the serial interface in Command mode.

IMPORTANT: Serial commands and arguments are **case-sensitive**.

Serial Command Syntax Legend

BOLD	- command or setting name
Normal	- case-sensitive command argument; must be entered exactly as shown
Italicized	- item must be replaced by user input
[optional]	- square brackets denote optional items
<data>	- stream of 1 or more data bytes delivered immediately after the command

General

b (empty) E

A request that consists of a *begin* (B) marker followed immediately by an *end* (E) marker will do nothing but respond with b = S,0 E . This empty command provides a way for the microcontroller to check if the Wi-Fi Module is functioning.

Connections

b **LISTEN**:*protocol*,*path* E

Activate a listener process to monitor HTTP or WebSocket *protocol* activity on port 80 with a specified *path*. Remote clients that connect to, request action of, and disconnect from *path* are noted by the listener and cause it to alert the attached microcontroller via the **POLL** command.

Protocol

The Internet *protocol* to listen for. Value can be "HTTP" or "WS" (WebSocket), or corresponding token values, to establish a port 80 listener.

Path

The *path* part of the URL that the remote client uses to access this module and its resources. The *path* can end in an asterisk '*' to match anything that begins with *path*.

Returns b = S,id E on success, or b = E,code E on error; see Error Codes. Use the returned listener *id* with **PATH** or **CLOSE** commands.

There are a maximum of four listeners available. Issuing additional **LISTEN** commands when all four listeners are already established will result in a response of b = E,4 E (NO_FREE_LISTENER) until a **CLOSE** command is used to free a listener.

Using **LISTEN**, the microcontroller tells the module what incoming requests it is able to handle – specifically supporting a particular *protocol* on port 80 and

targeting one or more URL *paths*. The module stores these *paths* and *protocols* in available listeners. When an HTTP or WebSocket connection is attempted by a client, the module checks for any active listeners that match the *path* and *protocol*. If it finds a match, it assigns an unused connection and accepts the request, storing any information about the request in the connection. If there is no match, or all connections are currently used, it rejects the request.

The microcontroller should monitor for related events (such as described above) using the **POLL** command, and must respond accordingly for the request to be processed properly.

There is a one-to-many relationship between listeners and connections since clients can make multiple requests on the same *path* and *protocol* while the microcontroller is processing them. When the microcontroller calls **POLL**, the module looks through its active connections and, for those that were accepted by a listener, delivers a 'G' (GET), 'P' (POST), or 'W' (WebSocket) response plus the associated connection handle and listener id. The microcontroller can then use the connection handle to fetch things like arguments or body content, and in the case of HTTP, send a **REPLY**.

HTTP connections are automatically closed after a **REPLY** is sent, freeing that connection handle for future requests; however, WebSocket connections must be explicitly terminated by calling **CLOSE**. Note that an HTTP connection *can* also be closed manually with the **CLOSE** command if the microcontroller does not intend to service the request.

b **CONNECT**:*address*,*port* E

Attempt a TCP connection to *address* on *port*.

Address

The destination *address* of the target to connect to.

Port

The *port* number to attempt communication on.

Returns b=S,handle E on success, or b=E,code E on error; see Error Codes. Use the returned connection *handle* with **PATH**, **CLOSE**, **SEND**, or **RECV** commands.

There are a maximum of four connections available; shared between successful **CONNECT** requests and active listener connections. Issuing additional **CONNECT** commands when all four connections are already established will result in a response of b=E,5 E (NO_FREE_CONNECTION) until a **CLOSE** command is used to free a connection.

b **CLOSE**:*handle_id* E

Terminate an established connection or listener via its *handle* or *id* (respectively), freeing it to rejoin the available connection or listener pools.

Handle_id

The connection *handle* or listener *id* to terminate.

Returns b =S,0 E on success, or b =E,code E on error; see Error Codes.

b **POLL**[:*filtermask*] E

Check for activity like incoming HTTP **GET**/**POST** requests, HTTP/WebSocket/TCP connections/disconnections, and incoming WebSocket/TCP data.

Optionally, the event activity can be filtered with *filtermask*, a 32-bit binary value where each bit corresponds to a connection handle and limits event responses to just those connections. Example: b "**POLL**:"1<<5 E will cause the module to respond with an event only if one has occurred on connection handle 5.

Returns event notifications in the form of event_char:value1,value2 as shown below. Value1 and value2 have different meanings depending on event_char and the context of the response.

b =G:handle,id E

Received HTTP GET request that matched a listener *id's* path and is now assigned to the connection *handle* for processing.

Handle

The connection identifier to use for this request.

Id

The identifier of the listener that matched the request.

Use:

- **PATH**:*handle* to get the associated connection path
- **PATH**:*id* to get the associated listener path (which may end in '*')
- *Id* as a unique identifier of the associated listener; alternative of using **PATH**:*id* or **PATH**:*handle* and parsing the path
- **ARG**:*handle*,... to retrieve HTTP GET query arguments
- **REPLY**:*handle*,... [+ **SEND**:*handle*,...] to respond to GET request

The 'G' event connection *handle* is automatically closed after a **REPLY** or **REPLY+SEND** is used to respond to the request. Do not manually **CLOSE** a 'G' event connection *handle* except to reject the request.

b = P:handle,id E

Received HTTP POST request that matched a listener *id's* path and is now assigned to the connection *handle* for processing.

Handle

The connection identifier to use for this request.

Id

The identifier of the listener that matched the request.

Use:

- **PATH**:*handle* to get the associated connection path
- **PATH**:*id* to get the associated listener path (which may end in '*')
- *Id* as a unique identifier of the associated listener; alternative of using **PATH**:*id* or **PATH**:*handle* and parsing the path
- **ARG**:*handle,...* to retrieve HTTP POST query/body arguments
- **RECV**:*handle,...* to get the HTTP body
- **REPLY**:*handle,...* [+ **SEND**:*handle,...*] to respond to POST request

The 'P' event connection *handle* is automatically closed after a **REPLY** or **REPLY+SEND** is used to respond to the request. Do not manually **CLOSE** a 'P' event connection *handle* except to reject the request.

b = W:handle,id E

Received a WebSocket request that matched a listener *id's* path and is now assigned to the connection *handle* for processing.

Handle

The connection identifier to use for this request.

Id

The identifier of the listener that matched the request.

Use:

- **PATH**:*handle* to get the associated connection path
- **PATH**:*id* to get the associated listener path (which may end in '*')
- *Id* as a unique identifier of the associated listener; alternative of using **PATH**:*id* or **PATH**:*handle* and parsing the path
- **SEND**:*handle,...* to transmit data to the remote
- **POLL** to wait for 'D' events, then **RECV**:*handle,...* to receive data

The 'W' event connection *handle* is persistent; it remains open until it is manually **CLOSE**ed, or is terminated by the remote. The connection *handle* can not be reused for future events if it is not closed first; make sure to close it when it has completely served its purpose.

b = D:handle,size E

WebSocket or TCP data was received. The 'D' event means data of *size* arrived on an already-established WebSocket or TCP connection *handle*.

Handle

The connection identifier to use for this request.

Size

The number of bytes of data received.

Use:

- **PATH**:*handle* to get the associated connection path (if WebSocket)
- *Handle* as a unique identifier of the associated WebSocket or TCP connection; alternative of using **PATH**:*handle* and parsing the path
- **RECV**:*handle*,... to receive the incoming data
- **SEND**:*handle*,... to transmit data to the remote

The 'D' event connection *handle* is persistent; it represents an established WebSocket or TCP connection and remains open until it is manually **CLOSE**ed, or is terminated by the remote. The connection *handle* can not be reused for future events if it is not closed first; make sure to close it when it has completely served its purpose.

b = S:handle,0 E

A **REPLY** or **SEND** operation completed successfully.

Handle

The connection identifier for the **REPLY** or **SEND** that completed.

b = X:handle,code E

A connection *handle* was disconnected due to reason *code*.

Handle

The connection identifier associated with this event.

Code

The communication event reason; see Network Codes.

b = N:0,0 E

No connection activity has occurred since the last **POLL** command.

b = E:handle,code E

A communication error occurred.

Handle

The connection identifier associated with this error.

Code

The communication error code; see Error Codes.

b **RECV**:*handle,max_count* E

Retrieve incoming HTTP body or WebSocket/TCP data.

Handle

An active connection *handle*; returned by **CONNECT** or **POLL**.

Max_count

The maximum number of bytes to receive.

Returns b = S,count E (on success) followed by *count* bytes up to *max_count* of payload, or returns b = E,code E on error; see Error Codes.

b **SEND**:*handle,count* E *<data>*

Transmit WebSocket/TCP data, or extended HTTP body (after **REPLY** command).

Handle

An active connection *handle*; returned by **CONNECT** or **POLL**.

Count

The number of bytes of *<data>* in this transmission.

<data>

The *count*-bytes of payload to send to the remote.

Returns b = S,0 E on success, or b = E,code E on error; see Error Codes.

WEB/HTTP

b **PATH**:*handle_id* E

Retrieve the path associated with a connection *handle* or listener *id*.

Handle_id

An active connection *handle* or listener *id*; returned by **LISTEN** or **POLL**.

Returns b = S,path E on success, or b = E,code E on error; see Error Codes.

HTTP

b **ARG**:*handle,name* E

Retrieve HTTP GET/POST's *name* argument (in query or body) on connection *handle*.

Handle

An active connection *handle*; returned by **POLL**.

Name

The argument name to retrieve.

Returns b = S,value E on success, or b = E,code E on error; see Error Codes.

b **REPLY**:*handle*,*rcode*,[*total_count*[,*count*]] E [<*data*>]

Transmit HTTP <*data*> in response to a GET or POST request. **REPLY** can send up to 1,024 bytes– if more is required, **REPLY** can be followed by one or more **SEND** commands to transmit the entire <*data*> set.

Handle

An active connection *handle*; returned by **POLL**.

Rcode

The desired HTTP response code for the reply.

Total_count

The total size of the <*data*>. If <*data*> is empty (0 bytes in size), *total_count*, *count*, and <*data*> may be omitted. For other <*data*> sizes, see description of *count* and <*data*>.

Count

The number of bytes of <*data*> (up to 1,024) included with this **REPLY** command. *Count* defaults to *total_count* and may be omitted if <*data*> size ≤ 1024 bytes. For sizes > 1,024, see description of <*data*>.

<*data*>

The data for the HTTP reply must follow the **REPLY** command. If *total_count* = 0 (or is omitted), then *count* must also be 0 (or omitted) and <*data*> must also be omitted; otherwise, at least *total_count* and <*data*> must be included. If <*data*> is ≤ 1,024 bytes, *total_count* must be specified, *count* can be omitted, and the entire <*data*> must follow the command. If <*data*> is > 1,024 bytes, *total_count* must be specified, *count* must be up to 1,024, *count* bytes of <*data*> must follow the command, and the remainder of <*data*> should be sent via one or more **SEND** commands.

Returns b = S,0 E on success, or b = E,code E on error; see Error Codes.

Settings

The Parallax Wi-Fi Module's *settings* determine default modes and behaviors. Using the *settings* commands below, they can be retrieved and adjusted.

b **CHECK**:*setting* E

Retrieve the current value of a *setting*. Note: If *setting* is an I/O pin, such as **pin-gpio2**, the **CHECK** command will set the I/O pin to an input direction and read its state, leaving it set as an input. Use the **SET** command to set an I/O pin to an output.

Setting

The desired *setting* to read. *Setting* may be any of those described in the Network Command - Settings - GET command section. For example: **CHECK:module-name** retrieves the module's name.

Returns b = S,value E on success (where *value* is the current value of the *setting*), or returns b = E,code E on error; see Error Codes.

b **SET**:*setting*,*value* E

Change the *setting* to *value*. Note: If *setting* is an I/O pin, such as **pin-gpio2**, the **SET** command will set the I/O pin to an output direction during the process of setting its state. Use the **CHECK** command to set an I/O pin to an output.

Setting

The desired *setting* to change. *Setting* may be any of those described in the Network Command - Settings - GET command section. For example: **SET:wifi-mode**,STA changes the module's Wi-Fi interface to station mode.

Value

The value to change the *setting* to. Each *setting* has a range of values described in the Network Command - Settings - GET command section.

Returns b = S,0 E on success, or b = E,code E on error; see Error Codes.

b **JOIN**:*ssid*,*passphrase* E

Attempt to join a network via the *ssid* access point using *passphrase*.

Ssid

The desired access point's SSID name.

Passphrase

The desired access point's passphrase.

Returns b = S,0 E on success, or b = E,code E on error; see Error Codes.

Tokens

The following single byte tokens can be used in place of text command names and parameters to decrease serial transmission time and microcontroller application size.

Command Name Tokens

```
TKN_JOIN       = 0xEF
TKN_CHECK      = 0xEE
TKN_SET        = 0xED
TKN_POLL       = 0xEC
TKN_PATH       = 0xEB
TKN_SEND       = 0xEA
TKN_RECV       = 0xE9
TKN_CLOSE      = 0xE8
TKN_LISTEN     = 0xE7
TKN_ARG        = 0xE6
TKN_REPLY      = 0xE5
TKN_CONNECT    = 0xE4
```

Each of the above tokens can be used in place of the corresponding command name and the following colon. For example: TKN_JOIN will replace "**JOIN:**". Note that you must not follow one of the tokens with a colon.

Parameter Tokens

```
TKN_INT8       = 0xFD
TKN_UINT8      = 0xFC
TKN_INT16      = 0xFB
TKN_UINT16     = 0xFA
TKN_INT32      = 0xF9
TKN_UINT32     = 0xF8
```

The above tokens precede a binary value. The INT8 and UINT8 tokens are followed by a single byte, the INT16 and UINT16 tokens are followed by two bytes, low byte first. The INT32 and UINT32 tokens are followed by four bytes, low byte first. None of these tokens should be followed by a comma to separate it from any following parameter.

```
TKN_HTTP       = 0xF7
TKN_WS         = 0xF6
TKN_TCP        = 0xF5
TKN_STA        = 0xF4
TKN_AP         = 0xF3
TKN_STA_AP     = 0xF2  // inserts "STA+AP"
```

The above tokens insert strings with the same name. Note that because of symbol name restrictions the string "STA+AP" must be inserted using the SSCP_TKN_STA_AP token.

Error Codes

```
1          Invalid request
```

```
2           Invalid argument
3           Wrong argument count
4           No free listener
5           No free connection
6           Lookup failed
7           Connect failed
8           Send failed
9           Invalid state
10          Invalid size
11          Disconnected
12          Unimplemented
13          Busy
14          Internal error
```

Network Codes

```
0           No error
-1          Out of memory error
-2          <undefined>
-3          Timeout
-4          Routing problem
-5          Operation in progress
-6          <undefined>
-7          Total number exceeds the set maximum
-8          Connection aborted
-9          Connection reset
-10         Connection closed
-11         Not connected
-12         Illegal argument
-13         <undefined>
-14         UDP send error
-15         Already connected
-16..-27    <undefined>
-28         ssl handshake failed
-29..-60    <undefined>
-61         ssl application invalid
```

References

Source code is in the Parallax GitHub account: https://github.com/parallaxinc/Parallax-ESP

Based on ESP-HTTPD by Jeroen Domburg and ESP-LINK by Thorsten von Eicken.