EZ-Serial WICED BLE Firmware Platform User Guide

Doc. No. 002-21887 Rev. **

Cypress Semiconductor

198 Champion Court

San Jose, CA 95134-1709

www.cypress.com

Contents

1. Introduction

This document provides a complete guide to the EZ-Serial platform for the EZ-BLE WICED module, CYBLE-013025-00. The guide covers the following:

- Cypress Serial Port Profile (CYSPP) UART-to-BLE bridge functionality
- GPIO status and control connections
- GAP Peripheral operation
- GATT Server data transfers
- Customizable GATT structures
- Security features such as encryption, pairing, and bonding
- API protocol allowing full control over all of these behaviors from an external host

1.1 How to Use This Guide

The high-level concepts covered in this document are organized into the following categories:

- System description and functional overview (Chapter 1, Introduction and Chapter 2, Getting Started)
- Firmware configuration examples (Chapter 3, Operational Examples)
- Complete design examples (Chapter 4, Application Design Examples)
- API protocol implementation examples for external MCU (Chapter 5, Host API Library)
- Troubleshooting guides (Chapter 6, Troubleshooting)
- Reference material (Chapter 7, API Protocol Reference through 10, Configuration Example Reference)

The following approach provides a good way to gain familiarity with EZ-Serial quickly:

Read through Chapter 1 (Introduction) and Chapter 2 (Getting Started) for a functional overview.

Find at least one example from Chapter 3 (Operational Examples) that is interesting or relevant to your intended design. Follow along with the described configuration on a development kit for a true hands-on experience. These examples provide excellent out-of-the-box feature demonstration:

- How to Get Started in CYSPP Mode with Zero Custom Configuration
- How to Define Custom Local GATT Services and Characteristics
- How to Detect and Process Written Data from a Remote Client
- How to Bond With or Without MITM Protection

Find at least one design example from Chapter 4 (Application Design Examples) that is similar to the type of system you intend to use an EZ-Serial-based EZ-BLE WICED module with, especially noting the functional capabilities provided by the configuration and GPIO connections.

If you are combining EZ-Serial with an external host microcontroller, read through Chapter 5 (Host API Library) to understand how the external MCU will need to communicate with the module.

Spend a few minutes reading through the guides in Chapter 6 (Troubleshooting) to avoid unnecessary frustration later on in the event that something does not behave in the way you expect.

Note the reference material available in this document to allow fast access to additional information and resources available from Cypress. When in doubt, always consult the API reference for helpful information and related content concerning any API command, response, or event.

EZ-BLE WICED modules support only a binary API protocol. However, text API protocol format is used to describe the application in this document for ease of reading. Refer to Section 2.4.2, Using the API Protocol in Binary Mode for the actual binary APIs that should be used during development. Along with this user guide, a Python script is provided to translate the text commands from or to binary ones, which can be downloaded from the Cypress website.

Throughout the guide, you will find API methods referenced in the following format:

- gap_set_adv_parameters (SAP, ID=4/23)

These links contain three important parts:

- Proper descriptive name (e.g., "**gap_set_parameters**"), unique among all other methods.
- Group/method ID values (e.g., "4/23"), present in the 4-byte header when using the binary API protocol (see Section 2.4.2, Using the API Protocol in Binary Mode).

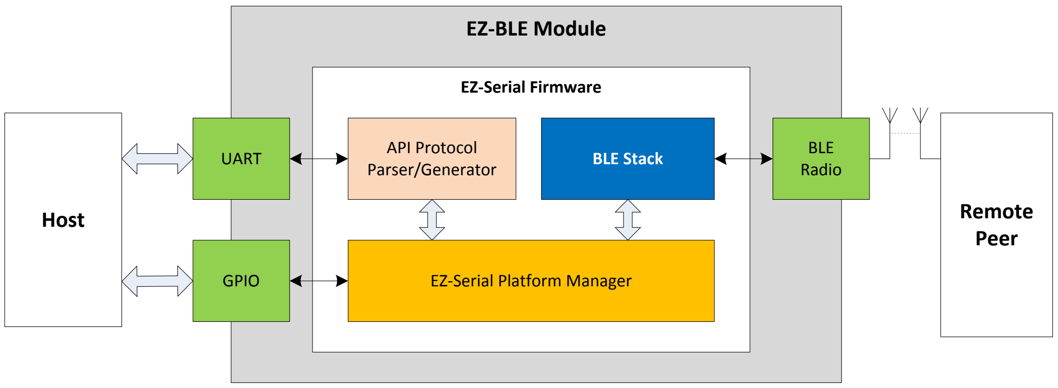
Click on any linked API method for detailed reference material in Chapter 7 (API Protocol Reference).

1.2 Block Diagram

The WICED EZ-Serial platform is built on top of EZ-BLE WICED modules from Cypress. Depending on the specific application, this platform may utilize an external host device, such as a microcontroller (MCU), connected to the module via UART, GPIO pins, or both. EZ-BLE WICED modules communicate with a remote device using the Bluetooth Low Energy (BLE) protocol.

Note that all GPIO pins on CYBLE-013025-00 are pre-defined with EZ-Serial. See Section 8.1 (GPIO Pin Map for Supported Modules) for details.

Figure 1-1. EZ-Serial System Block Diagram



1.3 Functional Overview

EZ-Serial provides an easy way to access the most commonly needed hardware and communication features in BLE-based applications. To accomplish this, the firmware implements a binary API protocol over the UART interface and exposes a number of status and control signals through the module’s GPIO pins.

1.3.1 BLE Communication Features

The EZ-Serial platform for WICED BLE modules has the following BLE-related features:

- Bluetooth 4.1 support on compatible modules
- Slave connection role (no master role support)
- Peripheral and Broadcaster GAP roles (no Central or Observer role support)
- Server GATT role (no Client role support)
- Customizable GATT database definition
- Encryption, bonding, and protection from man-in-the-middle (MITM) threats
- CYSPP (Cypress Serial Port Profile) mode for bidirectional serial data transmission
- Efficient low-power operation

1.3.2 Hardware and Communication Features

The EZ-Serial platform also implements a number of features that rely on internal CYW20737 chipset features and local interfaces:

- Binary API protocol
- On-demand ADC conversion
- Configurable PWM output

1.3.3 Firmware Overwrite

EZ-Serial for EZ-BLE WICED modules is a ready-to-use platform intended to satisfy a wide variety of application design requirements with minimal effort. If you have use cases that cannot be handled easily with the EZ-Serial platform, use the WICED Smart SDK to build your application firmware image. You can flash a custom firmware image onto any module via the HCI UART interface and completely replace the existing EZ-Serial image at any time. To return to EZ-Serial later, simply download the latest image from the Cypress website and flash it using the same mechanism.

For details on where to find these images, see Section 2.6.1 (Latest EZ-Serial Firmware Image).

1.4 Cypress EZ-BLE WICED Module Support

EZ-Serial WICED BLE firmware images exist only for the CYBLE-013025-00 module, based on the CYW20737 chip.

Table 1-1. Supported Devices

Devices
CYBLE-013025-00

For details on which pins support which functions, see Section 8.1 (GPIO Pin Map for Supported Modules) for pin definition on the CYBLE-013025-00 module.

2. Getting Started

2.1 Prerequisites

For a streamlined experience, we recommend that you have the following parts available:

- CYBLE-013025-EVAL EZ-BLE™ Module Arduino Evaluation Board
- Computer with serial terminal software such as Tera Term, Realterm, or PuTTY
- *Optional:* CY5677 CySmart Bluetooth® Low Energy (BLE) 4.2 USB Dongle
- *Optional:* BLE-capable mobile device such as an iPad, iPhone, or Android phone/tablet

The CYBLE-013025-EVAL board has a USB-to-UART bridge built in. The optional CySmart BLE dongle used with the matching CySmart software supports various Client-side functions such as connection establishment and GATT exploration without a BLE-capable smartphone or tablet..

You can control EZ-Serial over a UART interface without additional GPIOs; refer to Chapter 4 (Application Design Examples) for detail. However, we recommend using the CYBLE-013025-EVAL board for the best experience learning and prototyping due to its more comprehensive design and peripheral support.

2.2 Factory Default Behavior

The default configuration of EZ-Serial firmware is shown below:

- UART interface configured for 115200 baud, 8 data bits, no parity, 1 stop bit
- UART flow control disabled (signals from the module are not generated, signals from the host are ignored)
- CYSPP serial data transfer profile **enabled in auto-start mode**

When the module is powered on or reset, it will generate the system_boot (BOOT, ID=2/1) API event. This is only one example of one API method used by the platform; refer to Chapter 7 (API Protocol Reference) for details on the structure and behavior of the API protocol.

Since the EZ-Serial platform for EZ-BLE WICED modules only supports binary mode, the boot event will appear as shown below. The EZ-Serial firmware version shown below is 1.1.2 build 31. This information will change in the event a new firmware version is released.

```
80 12 02 01 1F 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20 7D
```

In text mode (not supported on this module), the same boot event would look like this:

```
@E,003B,BOOT,E=0101021F,S=02020355,P=0103,H=B1,C=00,A=20737A1A21D3
```

This text-mode string of data indicates:

- `@E` – an event has occurred
- `003B` – there are 59 bytes (0x3B) of content to follow
- `BOOT` – the event which occurred is the **BOOT** event
- `E=0101021F` – the EZ-Serial application version is 1.1.2 build 31 (0x1F)
- `S=2020355` – the BLE stack component version is 2.2.3 build 85 (WICED Smart SDK)
- `P=0103` – the protocol version is 1.3
- `H=B1` – the hardware platform is CYBLE-013025-00
- `C=00` – cause (this is always zero as WICED modules do not support this feature)

- A=20737A1A21D3 – the public Bluetooth MAC address of this module is 20:73:7A:1A:21:D3

NOTE: The version data and MAC address shown here are examples only. Actual values may differ.

Once the system boots, EZ-Serial will automatically start the CYSPP connection process by advertising as peripheral device. When this occurs, the gap_adv_state_changed (ASC, ID=4/2) API event will follow the boot event:

```
80 02 04 02 01 03 25
```

In text mode (not supported on this module), the same advertisement state change event would look like this :

```
@E,000E,ASC,S=01,R=03
```

NOTE: EZ-Serial for EZ-BLE WICED modules only supports binary mode. A simple python based script is provided along with this User Guide to translate a text format command/event from/to binary format. Please refer to Section 12. Binary/Text Conversion Python Script for additional details. For readability, the command/response/events in this guide use text mode format along with binary format.

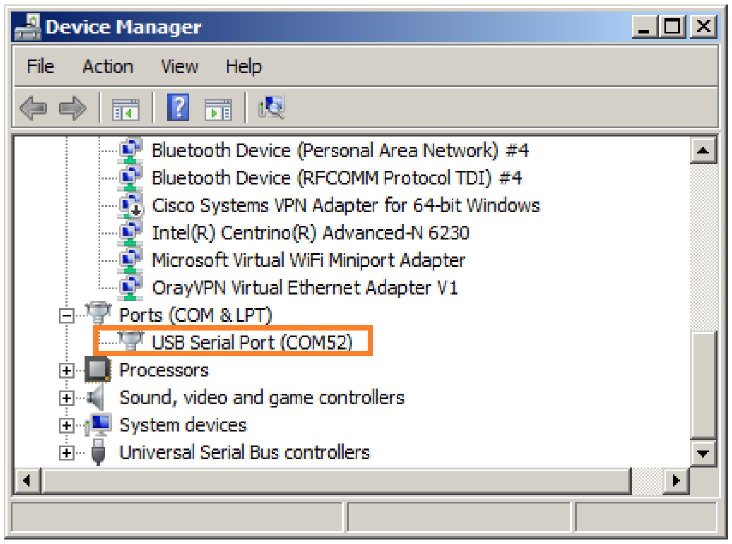
2.3 Connecting a Host Device

EZ-Serial communicates with an external host device, such as a microcontroller, using serial data (UART) and simple GPIO signals for status and control. Depending on your application, you may need to use one, both, or neither of these in your final design. Chapter 4 (Application Design Examples) describes each of these use cases.

2.3.1 Connecting the CYBLE-013025-EVAL Kit

When using the recommended evaluation kit for prototyping, simply connect the micro-USB cable between your PC and the evaluation board. This provides power to the module and a communication interface (UART) via the onboard USB-to-UART bridge. Once you have connected the cable and allowed any necessary drivers to install, a new virtual COM port will become available, as shown in Figure 2-1.

Figure 2-1. Virtual Serial Port from Evaluation Board



Bluetooth Device (Personal Area Network) #4
Bluetooth Device (RFCOMM Protocol TDI) #4
Cisco Systems VPN Adapter for 64-bit Windows
Intel(R) Centrino(R) Advanced-N 6230
Microsoft Virtual WiFi Miniport Adapter
OrayVPN Virtual Ethernet Adapter V1
Ports (COM & LPT)
USB Serial Port (COM52)
Processors
Sound, video and game controllers
System devices
Universal Serial Bus controllers

Note: COM11 is shown here, but your port number may be different.

You can then use this serial port with any compatible serial terminal software on your PC such as Tera Term, Realterm, or PuTTY.

2.3.2 Connecting the Serial Interface

You can also connect your own host or USB adapter for UART communication. The module's UART interface uses standard true-type logic (TTL) signals, with logic LOW at the GND (0 V) level and logic HIGH at the VDD level (typically 3.3 V).

WARNING: Do not connect the module directly to RS-232 signals which have VDD level range between ±3 ~ ±15. To prevent damage to the device, you must add voltage convertors before connecting to RS-232 signals.

EZ-Serial's UART interface has two required signals for data and two optional signals for flow control, if enabled:

- Required: **RXD** – Receive data (input), connect to host TXD (output)
- Required: **TXD** – Transmit data (output), connect to host RXD (input)
- Optional: **RTS** – Module-side flow control (output), connect to host CTS (input)
- Optional: **CTS** – Host-side flow control (input), connect to host RTS (output)

Refer to Section 8.1 (GPIO Pin Map for Supported Modules) for pin-to-function correlations.

NOTE: If you connect an external UART device or adapter to the CYBLE-013025-EVAL J2 header, ensure that you disconnect the onboard USB-to-UART bridge device by setting positions 1-4 of SW1 to the OFF position. Otherwise, the built-in USB-to-UART bridge interface will compete with the external interface as both devices attempt to drive the module's **P2 UART_RX** pin.

The default port settings are 115200 baud, 8 data bits, no parity, and one stop bit. Flow control is supported, but must be specifically enabled if desired.

You can change these settings using the system_set_uart_parameters (STU, ID=2/25) API command. UART transport settings are **protected**, which means they cannot be written to flash until they have first been applied to RAM. This prevents unintentional communication lockouts. Refer to Section 2.5.3 (Protected Configuration Settings) for details concerning protected settings.

If you experience any problems communicating over the serial interface, refer to Chapter 6 (Troubleshooting) for solutions to common issues.

2.3.3 Connecting GPIO Pins

EZ-Serial also supports GPIO connections for status signals (output) and control signals (input). These allow more flexible hardware design choices and more efficient operation than what the serial interface alone provides.

The firmware provides five single-function pins for status and control, aside from the two (or four if Flow Control is used) pins used for UART communication, one ADC pin, and one PWM output pin.

Table 2-1 summarizes the functions provided by these pins. For additional information, including module-specific pin assignments, operational side effects, and default logic states, refer to Chapter 8 (GPIO Reference). Note that some pins are active-HIGH, while some are active-LOW.

Table 2-1. GPIO Function Summary

Pin name	Direction	Functional Description
LP_MODE	Input	Low-power mode control. Assert (HIGH) to prevent sleep; de-assert (LOW) to allow sleep.
CYSPP	Input	CYSPP mode control. Assert (LOW) for CYSPP data mode; de-assert (HIGH) for command mode. **NOTE:** Asserting this pin will begin the CYSPP operation in the configured role even if the CYSPP profile is disabled in the platform configuration. See Section 2.4.3 (Using CYSPP Mode).
DATA_READY	Output	Data ready indicator. Asserted (LOW) when serial data is ready to be sent to the host; de-asserted (HIGH) after all data has been fully transmitted.

Pin name	Direction	Functional Description
CONNECTION	Output	Connection indicator. Asserted (LOW) when a BLE connection is established; de-asserted (HIGH) upon disconnection. **NOTE:** When CYSPP data mode is active using the CYSPP pin in the asserted (LOW) state, the **CONNECTION** pin is asserted only when a remote device has connected *and* completed the CYSPP GATT data characteristic subscription, indicating that the bidirectional data pipe is ready. It is de-asserted when data can no longer flow, either due to disconnection or because the data characteristic subscription is ended.
LP_STATUS	Output	Low-power state indicator. Asserted (HIGH) if the CPU is awake; de-asserted (LOW) if asleep.

For more details on GPIO functionality, see Chapter 8 (GPIO Reference).

2.4 Communicating with a Host Device

Once you have connected a host to the module via the serial interface, you can send and receive data. EZ-Serial supports two different modes of communication: **command mode** (API protocol communication and control) and **CYSPP mode** (transparent wireless cable replacement to remote device). The following sections describe these modes.

The active communication mode depends on the state of the **CYSPP** pin, which can be one of the following three options:

- **CYSPP** pin externally de-asserted (HIGH): **command mode**
- **CYSPP** pin externally asserted (LOW): **CYSPP mode**
- **CYSPP** pin left floating: **command mode** until activating CYSPP data pipe, then **CYSPP mode**

Ensure that the CYSPP pin is in the intended state at boot time to achieve the desired behavior. If you assert this pin, the API parser and generator become inactive, because all serial data is piped through the BLE connection (once established). You will experience what appears to be a lack of communication if you attempt to send API commands to the module while in CYSPP mode.

2.4.1 Using the API Protocol in Text Mode

The EZ-Serial platform for EZ-BLE WICED modules does not support text mode protocol operation. This feature is available only on EZ-Serial for Creator-based BLE modules.

2.4.2 Using the API Protocol in Binary Mode

EZ-Serial implements a binary-format API protocol that allows complete control of the platform using compact binary commands, responses, and events.

The binary protocol uses a fixed packet structure for every transaction in either direction. This fixed structure comprises a 4-byte header followed by an optional payload, terminating with a checksum byte. The payload carries information related to the command, response, or event. If present, this payload always comes immediately after the header and before the checksum byte.

Table 2-2. Binary Packet Structure

Header				Payload (optional)	Checksum
`[0]` Type	`[1]` Length	`[2]` Group	`[3]` ID	`[4...`*`N`*`-1]` Parameter(s)	`[N]` Summation

The checksum byte is calculated by starting from `0x99` and adding the value of each header and payload byte, rolling over back to 0 (instead of 256) to stay within the 8-bit boundary. The checksum byte itself is not included in the summation process. For the example 4-byte binary packet for the system_ping (/PING, ID=2/1) API command:

```
C0 00 02 01
```

Calculate the checksum as follows:

```
0x99 + 0xC0 + 0x00 + 0x02 + 0x01 = 0x15C
```

Retain only the final lower 8 bits (0x5C) for the 1-byte checksum value. The final 5-byte packet (including checksum) is:

```
C0 00 02 01 5C
```

The structure above allows a packet parser implementation to know exactly how much data to expect in advance any time a new packet begins to arrive, and to calculate the checksum as new bytes arrive.

The “Type” byte in the header contains information not only about the packet type (highest two bits), but also the memory scope (where applicable), and the highest three bits of the 11-bit “Length” value. For details on the binary packet format and flow, see the API structural definition in Section 7.1 (Protocol Structure and Communication Flow).

2.4.2.1 Binary Mode Protocol Characteristics

The **binary mode** protocol has the following general behavior:

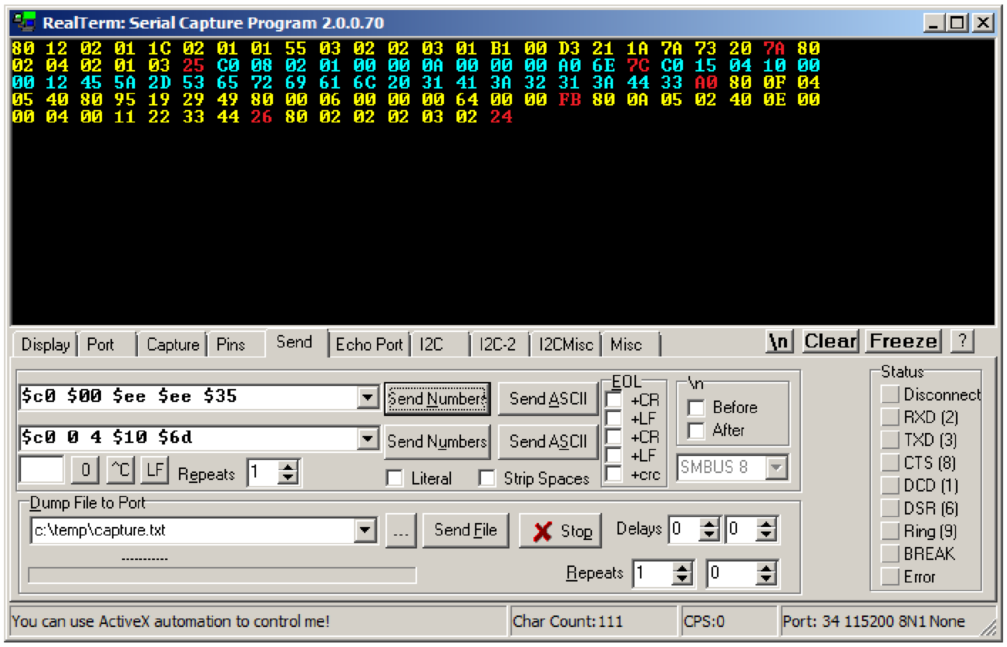
- Commands sent from the host must begin with a properly formatted 4-byte header.
- Commands must contain the number of payload bytes specified in the **Length** field from the header.
- Commands must end with a valid checksum byte, but no additional termination such as NULL or carriage return.
- Commands are always *immediately* followed by a response, if they are parsed correctly.
- Commands require all arguments to be supplied in the binary payload according to the protocol structural definition, in the right order (no arguments are optional).
- Commands with syntax errors are followed by a system_error (ERR, ID=2/2) API event with an error code indicating the nature of the problem, rather than a response packet.
- Commands must be fully transmitted within one second of the first byte, or the parser will time out and return to an idle state after triggering the system_error (ERR, ID=2/2) API event with a timeout error code.
- All multi-byte integer data is entered and expressed in little-endian byte order (e.g., 0x12345678 is **[78 56 34 12]**). Note that this applies only to API method arguments and parameters with a fixed width – 1, 2, or 4 – byte integers, and 6-byte MAC addresses.
- All multi-byte data passed inside a variable-length byte array (`uint8a` or `longuint8a`) remains in the original order provided by the source. This includes UUID data found during GATT discovery. If unsure, consult the API reference manual to verify the argument data type.
- Response payloads always begin with a 16-bit “result” value as the first parameter, indicating success or failure of the command triggering the response.
- The binary command header includes a single bit in the first byte, which performs the same duty as the ‘$’ character in text mode, to cause changed settings to be written to flash immediately instead of just RAM.

2.4.2.2 Binary Mode API Example

The easiest way to use binary command mode is with a host MCU or other application that has a complete parser and generator implementation available, such as the host API library example provided by Cypress and discussed in Chapter 5 (Host API Library).

However, it is also possible to test individual commands manually with a serial terminal application capable of entering and displaying binary data. Figure 2-2 shows an example of this type of test using Realterm, including hexadecimal representation of data. There is no local echo when binary mode is used, so the screenshot does not show the command packets sent to the module. To assist in identifying the packet types and boundaries, responses are colored **cyan**, events are **yellow**, and the final checksum byte of each packet is **red**.

Figure 2-2. Binary Command Mode Session with Realterm



```
80 12 02 01 1C 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20 7A 80
02 04 02 01 03 25 C0 08 02 01 00 00 0A 00 00 00 A0 6E 7C C0 15 04 10 00
00 12 45 5A 2D 53 65 72 69 61 6C 20 31 41 3A 32 31 3A 44 33 A0 80 0F 04
05 40 80 95 19 29 49 80 00 06 00 00 00 64 00 00 FB 80 0A 05 02 40 0E 00
00 04 00 11 22 33 44 26 80 02 02 02 03 02 24
```

$c0 $00 $ee $ee $35

$c0 0 4 $10 $6d

Repeats 1

c:\temp\capture.txt

Delays 0 0

Repeats 1 0

You can use ActiveX automation to control me!

Char Count: 111

CPS:0

Port: 34 115200 8N1 None

NOTE: This is helpful for testing, but not the most efficient way to communicate in binary mode.

Each binary packet (including the checksum byte) is described in Table 2-3. For better comparison between text mode and binary mode, the API transactions demonstrated here are the same as those used in the text mode example. Note that multi-byte integer data such as the 6-byte MAC address and the 16-bit advertisement interval are transmitted in little-endian byte order.

Table 2-3. Binary Mode Communication Example

Direction	Content	Detail
←RX	`80 12 02 01 1F 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20 7A`	system_boot (BOOT, ID=2/1) API event received: **app** = 1.1.2 build 31 **stack** = 2.2.3 build 85 **protocol** = 1.3 **hardware** = CYBLE-013025-00 module **boot cause =** N/A **MAC address =** 20:73:7A:1A:21:D3
←RX	`80 02 04 02 01 03 25`	gap_adv_state_changed (ASC, ID=4/2) API event received: **state** = 1 (active) **reason** = 3 (CYSPP operation)
TX→	`C0 00 02 01 5C` *(not visible)*	system_ping (/PING, ID=2/1) API command sent to ping the local module to verify proper communication
←RX	`C0 08 02 01 00 00 0A 00 00 00 A0 6E 7C`	system_ping (/PING, ID=2/1) API response received: **result** = 0 (success) **runtime** = 10 seconds **fraction** = 28320/32768
TX→	`C0 00 04 10 6D` *(not visible)*	gap_get_device_name (GDN, ID=4/16) API command sent to get the configured Device Name
←RX	`C0 15 04 10 00 00 12 45 5A 2D 53 65 72 69 61 6C 20 31 41 3A 32 31 3A 44 33 A0`	gap_get_device_name (GDN, ID=4/16) API response received: **result** = 0 (success) **name** = "EZ-Serial 1A:21:D3"

Direction	Content	Detail
←RX	`80 0F 04 05 40 80 95 19 29 49 80 00 06 00 00 00 64 00 00 FB`	gap_connected (C, ID=4/5) API event received: **handle** = 40 **peer** = 80:49:29:19:95:80 **addr_type** = 0 (public) **interval** = 6 (7.5 ms) **slave_latency** = 0 **supervision_timeout** = 0x64 (100 = 1 second) **bond** = 0 (not bonded)
←RX	`80 0A 05 02 40 0E 00 00 04 00 11 22 33 44 26`	gatts_data_written (W, ID=5/2) API event received: **conn_handle** = 4 **attr_handle** = 0x1F (31) **type** = 0 (simple write) **data** = 4 bytes `[11 22 33 44]`
TX→	`C0 00 EE EE 35` *(not visible)*	Invalid API command (group and ID bytes set to **0xEE**) sent to demonstrate binary mode error event
←RX	`80 02 02 02 03 02 24`	system_error (ERR, ID=2/2) API event received: **reason** = 0x0203 (Unrecognized Command)

Refer to the reference material in Chapter 7 (API Protocol Reference) for details concerning each of these API methods and the binary packet format, including information on all header fields and supported data types.

2.4.3 Using CYSPP Mode

EZ-Serial implements a special CYSPP profile that provides a simple method to send and receive serial data over a BLE connection. This operational mode is separate from the normal command mode where the API protocol may be used. When CYSPP data mode is active, any data received from an external host will be transmitted to the remote peer, and any data received from the remote peer will be sent out through the hardware serial interface to the external host.

2.4.3.1 Starting CYSPP Operation

You can start CYSPP mode using any of these three methods:

1. Assert (LOW) the **CYSPP** pin externally. You may connect this pin to ground in hardware designs that require CYSPP operation only and never need API communication. You can also use this pin to enter CYSPP mode even if the CYSPP profile is disabled in the platform configuration.
2. Use the p_cyspp_start (.CYSPPSTART, ID=10/2) API command. You can use this command to enter CYSPP mode even if the CYSPP profile is disabled in the platform configuration.
3. Have a remote GATT Client connect and subscribe to the CYSPP acknowledged data characteristic (enabling indications) or unacknowledged data characteristic (enabling notifications). This method will enter CYSPP mode only if the CYSPP profile is enabled in the platform configuration.

When starting CYSPP mode locally using either the **CYSPP** pin or the p_cyspp_start (.CYSPPSTART, ID=10/2) API command, the data pipe will not be immediately available because the remote device must still connect and set up proper GATT data subscriptions. If 100% data delivery is required in this context, the Host should monitor the **CONNECTION** pin to determine when it is safe to begin sending data from the Host for BLE transmission. Once the **CONNECTION** pin is asserted while the **CYSPP** pin is also asserted, the Host may send and receive data over CYSPP.

NOTE: Externally asserting (LOW) the **CYSPP** pin will always begin CYSPP operation, even if the profile has been disabled in the platform configuration via the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command. If you do not require CYSPP operation, you should ensure that this pin remains electrically floating or externally de-asserted (HIGH).

2.4.3.2 Sending and Receiving Data in CYSPP Data Mode

Once you have started CYSPP mode, the EZ-Serial platform will take care of the rest of the connection process and data pipe construction on the module side. If you are using modules running EZ-Serial firmware on both ends of the connection, then simply start CYSPP mode with complementary roles (Peripheral on one end, Central on the other), and the modules will automatically connect and prepare the data pipe using the processes described below. Keep in mind that CYSPP operation supports only the peripheral role on EZ-BLE WICED modules. To achieve Central-role CYSPP operation, you will need to create your own program using WICED SMART or use an EZ-BLE module in PSoC Creator with pre-programmed EZ-Serial firmware.

A non-Cypress device such as a BLE-enabled smartphone will frequently be used for one end of the connection; you must configure it to follow the same procedure.

For configuration examples in each mode, see Section 3.2 (Cable Replacement Examples with CYSPP).

1. EZ-Serial begins advertising with configured advertisement settings.
2. Upon connection, a remote peer must subscribe to one of the two “Data” characteristics:
   a. Acknowledged Data, enable indications (guaranteed reliability)
   b. Unacknowledged Data, enable notifications (faster potential throughput)
3. Remote peer may optionally subscribe to the “RX Flow Control” characteristic to allow the Server communicate whether it is safe to write new data.
4. EZ-Serial will assert the **CONNECTION** pin, indicating that CYSPP is ready to send and receive data.
5. The data pipe remains open until the central device disconnects or unsubscribes from the data characteristic, or the CYSPP pin is de-asserted locally.

2.4.3.3 Exiting CYSPP Mode

Once in CYSPP mode, the API parser is logically disconnected from incoming serial data, so you will not be able to send any commands to the module. However, you can still exit CYSPP mode in two ways:

1. De-assert (HIGH) the **CYSPP** pin externally.
2. Have the remote GATT Client unsubscribe from the relevant CYSPP data characteristic (applies only when the CYSPP pin is not externally asserted).

When the CYSPP operation ends, EZ-Serial returns to command mode.

WARNING: It is not possible to use an API command to exit the CYSPP data mode, because the API parser is not available while in this mode. If your design needs to switch between modes on demand, include external access to the **CYSPP** pin so you can control the operational mode.

2.4.3.4 Customizing CYSPP Behavior for Specific Needs

While the default behavior is suitable in many cases, there are configuration settings that allow a great deal of control over this behavior. The following list describes which options can be changed, and how to do so:

- CYSPP mode uses the system’s configured UART host transport settings for sending and receiving serial data. To change these settings, use the system_set_uart_parameters (STU, ID=2/25) API command.
- CYSPP mode uses the system’s configured radio transmit power setting for all BLE communication. To change this setting, use the system_set_tx_power (STXP, ID=2/21) API command.
- CYSPP mode supports special incoming data packetization modes. This helps make radio transmissions and data delivery more efficient in a variety of use cases. To change these settings, use the p_cyspp_set_packetization (.CYSPPSK, ID=10/7) API command.
- When operating in Peripheral mode, CYSPP uses the system’s configured advertisement parameters, including the advertisement and scan response packet content (which may be based on the device name) and the system’s whitelist. To change these settings, use one or more of the following API commands:
  - gap_set_adv_parameters (SAP, ID=4/23)
  - gap_set_adv_data (SAD, ID=4/19)
  - gap_set_sr_data (SSRD, ID=4/21)
  - gap_set_device_name (SDN, ID=4/15)

2.4.3.5 Understanding CYSPP Connection Keys

EZ-Serial also supports CYSPP connection keys, which improve usability in environments where multiple CYSPP-capable devices are operating in an automated configuration. This feature allows an advertising peripheral device to broadcast an arbitrary 4-byte value that a scanning device can filter against, searching either for a masked range of devices or a single specific device.

CYSPP connection keys are not set in the factory default configuration; CYSPP Peripheral advertisements contain a “0” key. To change this, use the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command, and specifically the “**local_key**” argument of this command as described in the following section. Note that the “**remote_key**”, and “**remote_mask**” arguments apply only to CYSPP Central operation, which is not supported on this platform. They are left in to maintain protocol consistency across multiple platforms.

2.4.3.6 Using the CYSPP Peripheral Connection Key

The CYSPP Peripheral connection key affects only the content of the advertisement packet while the module is in an advertising state. The CYSPP Peripheral role does not include any filtering behavior; filtering is left to the scanning device that is operating in the CYSPP Central role.

When the CYSPP profile is enabled, the platform-managed advertising packet contains a special Manufacturer Data field to hold the local connection key value. It is not stored elsewhere, such as in a GATT characteristic. This advertisement packet field has the following structure:

Table 2-4. CYSPP Peripheral Connection Key Manufacturer Data Field Structure

Length	Type	Company ID	Connection Key
`07`	`FF`	*`b0 b1`*	*`b0 b1 b2 b3`*

The Company ID value is a 16-bit value that the Bluetooth SIG assigns to member companies that have requested them (see resources on www.bluetooth.com). The factory default value is the Cypress company identifier, 0x0131, but you can change this with the same command used to change other CYSPP parameters. Note that both the Company ID and the Connection Key values are broadcast in little-endian byte order.

Use the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command and enter the desired 32-bit value for the “**local_key**” argument to apply a new Peripheral connection key. Changes take effect immediately, even if the module is already advertising in the CYSPP peripheral role.

WARNING: EZ-Serial incorporates only the CYSPP Peripheral connection key into the advertising packet if you have not enabled user-defined advertisement content. If you have configured user-defined advertisement content instead as described in Section 3.3.3 (How to Customize Advertisement and Scan Response Data), then changing this value will have no effect. You must ensure that your user-defined advertisement packet contains an equivalent field in order to allow scanning devices to filter properly.

Example 1: Update CYSPP peripheral key to 0x11223344

Direction	Text Content	Binary Content	Effect
TX→	`.CYSPPSP,E=2,G=0,C=0131,L=11223344,R=0,M=0,P=1,S=0,F=2`	C0 13 0A 03 02 00 31 01 44 33 22 11 00 00 00 00 00 00 00 00 01 00 02 5A	Apply new CYSPP configuration
←RX	`@R,000E,.CYSPPSP,0000`	C0 02 0A 03 00 00 68	Response indicates success

2.4.3.7 CYSPP Configuration and Pin States

Table 2-5 describes the relationship between the state of the CYSPP pin and the CYSPP firmware configuration managed with the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command. Note the following two key behaviors concerning hardware control versus software control:

- Asserting the **CYSPP** pin externally always triggers automatic CYSPP. This occurs even if you have disabled the profile in software.
- CYSPP data mode (where the API is suppressed and all serial data is channeled to the remote peer) ultimately depends on the state of the **CYSPP** pin. EZ-Serial pulls this pin to the appropriate logic level based on internal

CYSPP state changes when CYSPP is enabled, but you can override the pulled state with an external host or hardware design feature.

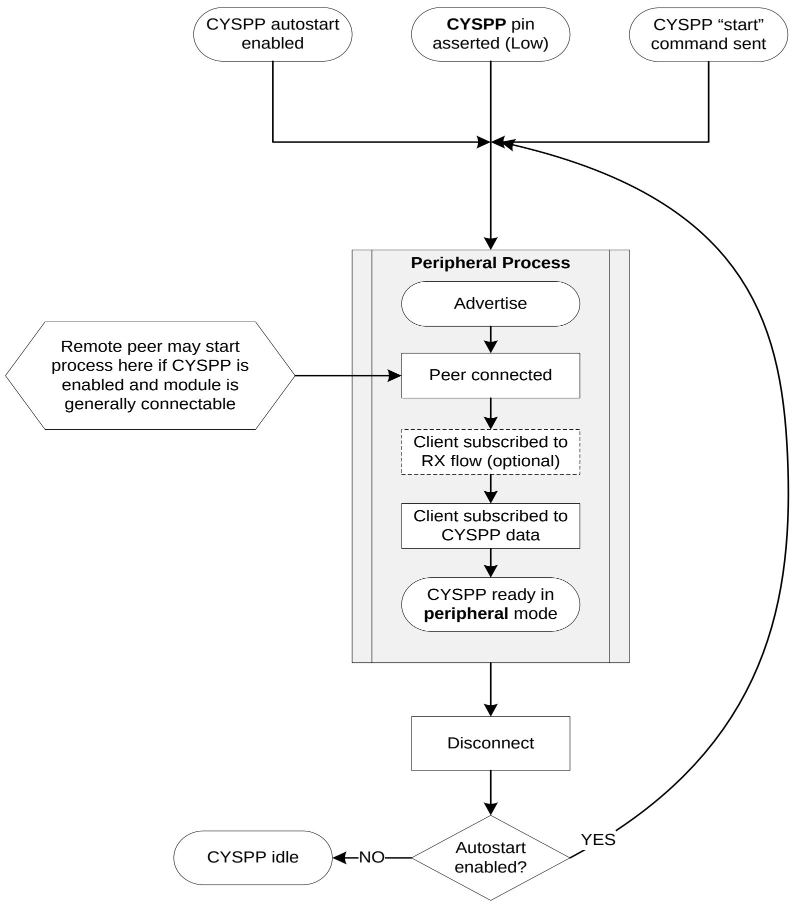
Table 2-5. CYSPP Configuration and Pin Relationship

CYSPP pin state	CYSPP “enable” value in configuration	CYSPP Operation
Floating **(assumed default)**	Disabled	**Inactive.** All advertising, scanning, connections, GATT subscriptions, GATT transfers, etc., occur via API commands and events. CYSPP GATT structure is not visible to a remote Client.
	Enabled	**Idle until start.** When started via the p_cyspp_start (.CYSPPSTART, ID=10/2) API command, the module begins advertising. API events (boot, stage changes, connections, etc.) are **visible** over UART until the CYSPP data connection is opened between the local device and remote peer. The **CYSPP** pin is pulled LOW when this occurs, at which point the API is suppressed and the serial interface may be used only for CYSPP data pipe. This mode continues until the remote host disconnects or unsubscribes.
	Autostart **(factory default)**	**Automatic.** Same behavior as in the “Enabled” case, except that CYSPP operation begins automatically at boot time and restarts upon disconnection.
Externally driven HIGH (de-asserted)	Disabled	**Inactive.** All advertising, connections, GATT subscriptions, GATT transfers, etc., occur via API commands and events. CYSPP GATT structure is not visible to a remote Client.
	Enabled	**Idle until start, command mode retained.** When started via the p_cyspp_start (.CYSPPSTART, ID=10/2) API command, module begins advertising. API events (BOOT, stage changes, connections, etc.) are **visible** over UART. API communication continues throughout the process; CYSPP data from the remote host is never raw/transparent unless the host asserts the **CYSPP** pin.
	Autostart	**Automatic.** The same behavior as in the “Enabled” case, except that CYSPP operation begins automatically at boot time and restarts upon disconnection. API events continues to be visible while **CYSPP** pin is de-asserted (HIGH).
Externally driven LOW (asserted)	Doesn’t matter	**Active regardless of firmware configuration.** Automatic advertising begins at boot time. API events (boot, state changes, connections, etc.) are **not be visible** over UART, because API communication is always suppressed when **CYSPP** pin is asserted.

2.4.3.8 *CYSPP State Machine*

Figure 2-3 shows the way EZ-Serial manages CYSPP operation, depending on firmware configuration and the logic states of the **CYSPP** pins.

Figure 2-3. CYSPP State Machine


CYSPP autostart enabled
CYSPP pin asserted (Low)
CYSPP "start" command sent
Peripheral Process
Advertise
Remote peer may start process here if CYSPP is enabled and module is generally connectable
Peer connected
Client subscribed to RX flow (optional)
Client subscribed to CYSPP data
CYSPP ready in peripheral mode
Disconnect
Autostart enabled?
YES
NO
CYSPP idle

2.5 Configuration Settings, Storage, and Protection

The EZ-Serial platform provides methods to customize its many built-in functions. It's important to understand how these settings are stored and changed in different contexts to avoid unexpected behavior.

2.5.1 Factory, Boot, and Runtime Settings

EZ-Serial implements three different "layers" of configuration data, each of which serves a unique purpose. Table 2-6 below describes each type of configuration storage in detail.

Table 2-6. Configuration Setting Storage Layers

Layer	Details
Factory (FLASH)	**Description:** Factory-level settings are hard-coded into the firmware image and stored in flash, and cannot be changed independently by the user. They are used for runtime-level settings until/unless customized boot-level values exist. Using the system_factory_reset (/RFAC, ID=2/5) API command reverts to these values. **Content:** These values contain only platform configuration settings, but no custom GATT structure definitions or value data. **Data retention during chipset reset: YES** These values are retained upon power cycles and chipset reset conditions.
Boot (FLASH)	**Description:** Boot-level settings are set by the user and stored in flash, and applied to the runtime-level area for active use when the module boots. (If no customized boot-level settings have been set by the user, the factory-level settings are applied instead upon first boot.) These values can be modified using API commands, and they are erased when performing a factory reset. **Content:** These values contain both platform configuration settings and any custom GATT structure definitions. Actual GATT characteristic values such as those written by a remote Client are not included in this data. **Data retention during chipset reset: YES** These values are retained during power cycles and chipset reset conditions.
Runtime (RAM)	**Description:** Runtime-level settings are used as the active configuration set that controls EZ-Serial's behavior at all times, with a few exceptions as noted in the "Automatic" section below. API commands that set or get configuration values access this layer of configuration data unless explicitly noted otherwise. **Content:** These values contain platform configuration settings, custom GATT structure definitions, and GATT characteristic values written from a remote Client. **Data retention during chipset reset: NO** These values are not retained during power cycles and chipset reset conditions. Any runtime settings or GATT database structure definitions should be written to flash with the relevant API command(s) before performing a reset.

2.5.2 Saving Runtime Settings in Flash

Storing settings in flash memory is critical to allow predictable, long-term customized behavior without needing to reconfigure each time. EZ-Serial provides two ways to accomplish this:

1. Use the system_store_config (/SCFG, ID=2/4) API command to write all current runtime-level settings to the boot-level configuration. This applies a snapshot of the current configuration to flash in one step. This method should be used if you are unsure which settings have changed between boot-level and runtime-level values, or if you want to test out a new set of options before making them permanent.
2. Set the "flash" memory scope bit in the binary command packet header when writing new configuration values with relevant commands. This method should be used if you know exactly which settings need to be changed, because it does not require the final use of the system_store_config (/SCFG, ID=2/4) API command afterward.

Note that while the flash memory scope bit may be used with any command; doing so is relevant only for commands that either read or write configuration values directly. For other commands, these flags will be silently ignored. See the API reference material in Chapter 7 (API Protocol Reference) for details.

To ensure the longest flash memory life, writes to flash should be as infrequent as possible in production-ready designs. Settings that must be changed frequently should be modified in RAM and only written to flash if required. Note that the internal chipsets used in the EZ-BLE modules that run EZ-Serial have a minimum flash endurance rating of 100,000 cycles.

2.5.3 Protected Configuration Settings

A small number of configuration values have the potential to put the module into a state where it is no longer possible to communicate over the serial interface as intended. To help avoid this potential problem, a few settings are classified as **protected**. This means that they must be changed at the runtime level only (RAM) before they may be applied to the boot-level (flash) area. Currently, only one command affects protected settings:

- system_set_uart_parameters (STU, ID=2/25)

The changes that are most likely to cause an unintended communication lockout are serial transport reconfigurations, such as selecting a baud rate that is not supported by the host. To store new values in flash for protected configuration settings, you must either send the same command twice with the flash memory scope bit/character used only the second time, or else use the system_store_config (/SCFG, ID=2/4) API command to write all runtime-level settings to the boot level after first setting the new value in RAM only. This forces the flash write to occur using the new configuration, which can only occur if communication is still possible.

2.6 Where to Find Related Material

This guide refers to firmware images and example source code files that must be accessed separately from this document.

2.6.1 Latest EZ-Serial Firmware Image

You can find the latest available EZ-Serial firmware image files on Cypress’s website:

http://www.cypress.com/ez-serial

These images are suitable for both HCI UART-based re-flashing through WICED Smart SDK chip loading tools. See Section 3.7 (Device Firmware Update Examples) for details about how to flash these firmware images onto target modules.

2.6.2 Latest Host API Protocol Library

You can find the latest host API protocol library source code examples on Cypress website:

http://www.cypress.com/ez-serial

2.6.3 Comprehensive API Reference

While this guide contains many specific functional examples, these are not intended to provide a full reference to all possible functionality provided by the API. See Chapter 7 (API Protocol Reference) of this document for detailed material concerning the API structure and protocol.

3. Operational Examples

EZ-Serial provides a great platform on which to build a wide variety of BLE applications. The sections below describe many common operations that you can experiment with or combine together to create the behavior needed for your application.

3.1 System Setup Examples

These examples demonstrate basic platform behavior and configuration of the system.

NOTE: The first example shown below provides low-level detail and explanation of some API protocol formatting features, while all other examples assume a basic understanding of the mechanics of the protocol and will only show example snippets in text format. For details on the API methods used in each case and the binary equivalents of each command, response, and event, see Chapter 7 (API Protocol Reference).

3.1.1 How to Identify the Running Firmware and BLE Stack Version

The EZ-Serial firmware, BLE stack, and protocol version details can be obtained from the API event generated at boot time, or on demand using an API command.

3.1.1.1 Getting Version Details from Boot Event

Capture and process the system_boot (BOOT, ID=2/1) API event that occurs when the module is powered on or reset. This event includes the application version, stack version, protocol version, boot cause, and unique Bluetooth MAC address.

This event will be similar to that shown below, expressed in hexadecimal notation:

Header	Payload	Checksum
`80 12 02 01`	`1F 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20`	`7D`

To simplify manual interpretation in this guide, individual parameters within the payload are separately underlined.

The payload data in the event text/binary examples shown above is described in Table 3-1.

Table 3-1. Payload Detail for Boot Event

Binary Data	Details	Interpretation
`1F 02 01 01`	EZ-Serial application version	Version 1.1.2 build 31 (0x1F)
`55 03 02 02`	WICED Smart SDK	2.2.3 build 85 (0x55)
`03 01`	API protocol version	Version 1.3
`B1`	Hardware ID	CYBLE-013025-00 module
`00`	Cause for boot event	Unused/unavailable
`D3 21 1A 7A 73 20`	MAC address	20:73:7A:1A:21:D3

3.1.1.2 Getting Version Details On Demand

Use the system_query_firmware_version (/QFV, ID=2/6) API command to request version details at any time. The response to this command contains the same initial information in the system_boot (BOOT, ID=2/1) API event, but it does not include the boot cause or the module's Bluetooth MAC address.

The text-mode response to this API command is as shown below:

```
@R,002C,/QFV,0000,E=0101021C,S=02020355,P=0103,H=B1
```

The binary-mode response packet is as shown below:

Header	Payload	Checksum
C0 0D 02 06	06 00 00 1C 02 01 01 55 03 02 02 03 01 B1	9F

To simplify manual interpretation in this guide, individual parameters within the payload are separately underlined.

3.1.2 How to Change the Serial Communication Parameters

Use the system_set_uart_parameters (STU, ID=2/25) API command to reconfigure the serial interface used for host communication. This command affects **protected** settings, and therefore it must be applied in RAM first before it can be written to flash.

All data entered via text mode must be expressed in **hexadecimal** notation. Table 3-2 lists common baud rates and their hexadecimal equivalents:

Table 3-2. Common UART Baud Rates and Hex Equivalents

Baud Rate	Hex Equivalent
1,200	4B0
2,400	960
4,800	12C0
9,600	2580
14,400	3840
19,200	4B00
28,800	7080
38,400	9600
57,600	E100
115,200 (default)	1C200
230,400	38400
460,800	70800
921,600	E1000

NOTE: EZ-Serial supports non-standard baud rates not listed in the table above, and should remain below 3 percent clock error due to the use of an internal fractional clock divider. While this is within the tolerance level required by many UART interfaces, you should measure the actual bit timing with an oscilloscope or logic analyzer to verify that the baud rate is operating within required tolerance for your host device.

Example 1. Set UART to 38400 baud, even parity, flow control enabled, and store in flash

Direction	Text Content	Binary Content	Effect
TX→	`STU,B=9600,A=0,C=0,F=1,D=8,P=0,S=1`	C0 0A 02 19 00 96 00 00 00 00 01 08 00 01 1E	Set new UART parameters (RAM only) – “38400” decimal is “9600” hex
←RX	`@R,0009,STU,0000`	C0 02 02 19 00 00 76	Response indicates success
Change host UART parameters to match new settings here before sending additional data			
TX→	`STU$,B=9600,A=0,C=0,F=1,D=8,P=0,S=1`	D0 0A 02 19 00 96 00 00 00 00 01 08 00 01 2E	Write UART settings to flash
←RX	`@R,000A,STU$,0000`	D0 02 02 19 00 00 86	Response indicates success

Example 2. Set UART to 115200 baud, no parity, flow control disabled, and store in RAM only

Direction	Text Content	Binary Content	Effect
TX→	`"STU,B=1C200,A=0,C=0,F=0,D=8,P=0,S=1`	C0 0A 02 19 00 C2 01 00 00 00 00 08 00 01 4A	Apply new UART parameters
←RX	`@R,0009,STU,0000`	[C0 02 02 19 00 00 76	Response indicates success

3.1.3 How to Change the Device Name and Appearance

Use the gap_set_device_name (SDN, ID=4/15) API command to set a new friendly device name at any time, and the gap_set_device_appearance (SDA, ID=4/17) API command to set a new appearance value.

EZ-Serial uses the device name and appearance to populate the GAP service's name and appearance characteristic values in the GATT database. If EZ-Serial is allowed to automatically manage the advertisement and scan response data content (default behavior), then it also includes up to 29 bytes of the device name in the scan response packet. (The limit of 29 bytes is due to a BLE specification limit on the maximum scan response payload, which is 31 bytes; the other two bytes are needed for the field length and field type values that are part of the device name field.)

NOTE: EZ-Serial limits the device name length to **64 bytes** to minimize internal SRAM requirements.

Using EZ-Serial's special **macro codes**, described in Section 7.5 (Macro Definitions), you can enter a single text string which is expanded internally to include module-specific values—in this case, the Bluetooth MAC address. This is shown in the first example below.

The device appearance value is a 16-bit field made up of a 10-bit and 6-bit subfield. Allowed values are defined by the Bluetooth SIG and can be found at developer.bluetooth.org.

Changes made to the device name and appearance values take effect immediately. They are written to the local GATT characteristics for these two values (always present), and the device name is updated in the scan response packet if user-defined advertisement content has not been enabled with the gap_set_adv_parameters (SAP, ID=4/23) API command.

Example 3. Set device name with partial MAC address incorporation

Direction	Text Content	Binary Content	Effect
TX→	`SDN$,N=EZ-Serial %M4:%M5:%M6`	D0 16 04 0F 15 45 5A 2D 53 65 72 69 61 6C 20 25 4D 34 3A 25 4D 35 3A 25 4D 36 5C	Set new device name in flash using 4th, 5th, and 6th MAC bytes (module-specific)
←RX	`@R,000A,SDN$,0000`	D0 02 04 0F 00 00 7E	Response indicates success

This configured name results in an actual name of "EZ-Serial 1A:21:D3" assuming that the module in use has a MAC address of `20:73:7A:1A:21:D3` (as is used in other examples throughout this document).

Example 4. Set device appearance to "Generic Computer" (0x0080)

Direction	Text Content	Binary Content	Effect
TX→	`SDA$,A=0080`	D0 02 04 11 80 00 00	Set new appearance value in flash
←RX	`@R,000A,SDA$,0000`	D0 02 04 11 00 00 80	Response indicates success

3.1.4 How to Change the Output Power

Use the system_set_tx_power (STXP, ID=2/21) API command to set a new radio transmit power level. The argument to this command is not the dBm value directly, but rather a set of predefined values representing a fixed range from **-18 dBm** to **+4 dBm**. Table 3-3 lists each allowed value. Argument values of 3 and 4 provide the same output power; similarly, argument values of 5, 6, and 7 provide the same output power levels.

Table 3-3. Supported TX Power Output Options

Argument Value	Power Level	Comments
1	-18 dBm	
2	-12 dBm	
3, 4	-4 dBm	
5, 6, 7	+0 dBm	Default Value
8	+4 dBm	

Changes to the configured output power will take effect immediately.

Example 5. Set output power to -4 dBm

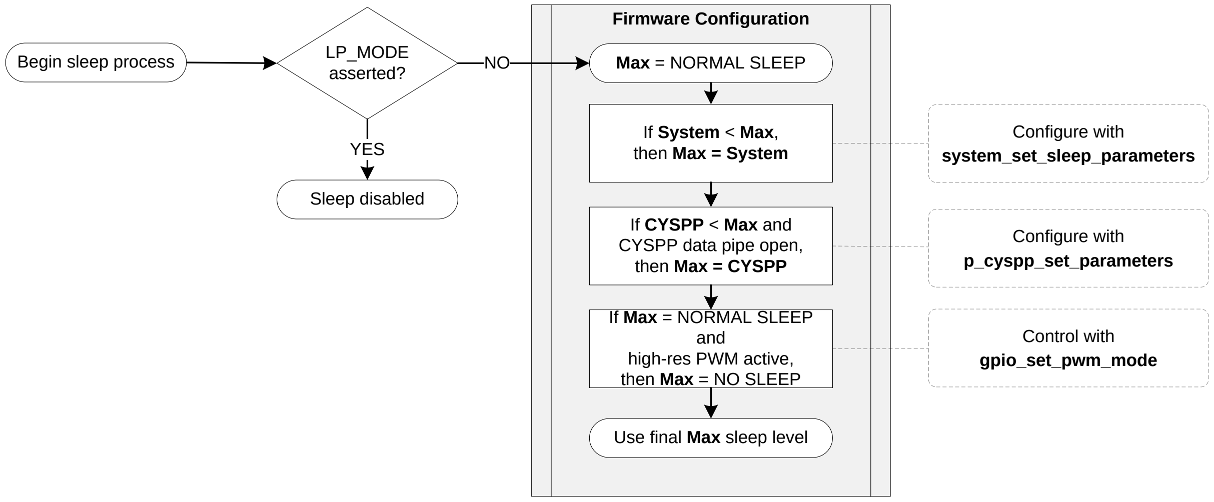
Direction	Text Content	Binary Content	Effect
TX→	`STXP,P=3`	C0 01 02 15 03 74	Set new TX power (RAM only)
←RX	`@R,000A,STXP,0000`	C0 02 02 15 00 00 72	Response indicates success

3.1.5 How to Manage Sleep States

EZ-Serial manages transitions between CPU active and sleep states automatically. It chooses the mode requiring the lowest safe power consumption according to the current operational state and configuration, including transitioning into normal sleep mode between BLE radio events (advertising, scanning, or while connected).

EZ-Serial uses the maximum allowed sleep level based on combined data from the system-wide sleep setting, CYSPP data mode sleep setting (if CYSPP data mode is active), PWM output state, and **LP_MODE** pin state. Figure 3-1 below describes the sleep level determination logic.

Figure 3-1. EZ-Serial Sleep State Behavior


Begin sleep process
LP_MODE asserted?
NO
YES
Sleep disabled
Firmware Configuration
Max = NORMAL SLEEP
If System < Max, then Max = System
If CYSPP < Max and CYSPP data pipe open, then Max = CYSPP
If Max = NORMAL SLEEP and high-res PWM active, then Max = NO SLEEP
Use final Max sleep level
Configure with system_set_sleep_parameters
Configure with p_cyspp_set_parameters
Control with gpio_set_pwm_mode

In outline form, the sleep state logic follows this process:

1. If the **LP_MODE** pin is asserted, remain in **Active** mode. Otherwise:
2. Select the *lowest* value (**0** = no sleep, **1** = normal sleep) among the following:
   a. The system sleep level configured with system_set_sleep_parameters (SSLP, ID=2/19) API command.
   b. The CYSPP-specific sleep level configured with the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command, if the CYSPP data pipe is open (connected and in CYSPP data mode).
   c. **No** sleep if high-resolution PWM output is enabled with the gpio_set_pwm_mode (SPWM, ID=9/11) API command.

NOTE: EZ-Serial does not allow changes to the sleep level calculation hierarchy order. For example, if CYSPP sleep level is "1" (sleep) but system-wide sleep is level "0" (no sleep), then the system-wide setting will override the CYSPP setting because it is a lower value. EZ-Serial will always select the lowest applicable value for the current operational state.

3.1.5.1 Configuring the System-Wide Sleep Level

Configure the system-wide sleep level using the system_set_sleep_parameters (SSLP, ID=2/19) API command. When sleep is not prevented by asserting the **LP_MODE** pin, this value is the first "default" sleep level limit applied when calculating which sleep mode to use.

Active PWM output limits the effective sleep level in any state to **no sleep** (value = 0). If the CYSPP data pipe is open (connected and in CYSPP data mode), then the CYSPP-specific sleep level may further limit the effective maximum sleep level. Figure 3-1 shows how EZ-Serial determines the sleep level to use.

EZ-Serial platform for WICED BLE modules allows **no sleep** (value = 0) as the factory default system-wide sleep level, for a simpler out-of-the-box experience concerning UART communication. However, you can change this to allow **normal sleep** to improve average current consumption.

Example 6. Change system-wide sleep level to normal sleep

Direction	Text Content	Binary Content	Effect
TX→	`SSLP,L=1`	C0 01 02 13 01 70	Set new system sleep level to "normal sleep"
←RX	`@R,000A,SSLP,0000`	C0 02 02 13 00 00 70	Response indicates success
*When the module is in normal sleep mode, it cannot receive commands; the host needs proper use of the **LP_MODE** pin as described in Section 3.1.5.3 (Preventing Sleep with the LP_MODE Pin) before transmitting the command.*			

3.1.5.2 Configuring the CYSPP Data Mode Sleep Level

Configure the CYSPP data mode sleep level using the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command. When sleep is not disabled using the **LP_MODE** pin, this value is the second limit applied when determining the sleep mode to use. The system-wide sleep level takes precedence over the CYSPP sleep level. Further, PWM output limits the effective maximum sleep level in any state to **no sleep** (value = 0), regardless of other settings. Figure 3-1 shows how EZ-Serial determines the sleep level to use.

Setting the CYSPP data mode sleep level to **normal sleep** (value = 1) or **no sleep** (value = 0) ensures that EZ-Serial does not use a sleep level beyond that setting whenever a CYSPP data pipe is open (connected and in CYSPP data mode). The factory default setting for this option is to allow **normal sleep** (value = 1), but keep in mind that factory default values also set the system-wide sleep level limit to **no sleep** (value = 0), which prevent normal sleep at all times unless you reconfigure it.

Example 7. Limit CYSPP-specific sleep level to normal sleep

Direction	Text Content	Binary Content	Effect
TX→	`.CYSPPSP,E=2,G=0,C=0131,L=0,R=0,M=0,P=1,S=1,F=2`	C0 13 0A 03 02 00 31 01 00 00 00 00 00 00 00 00 00 00 00 00 01 01 02 B1	Set new CYSPP sleep level to "normal sleep"
←RX	`@R,000E,.CYSPPSP,0000`	C0 02 0A 03 00 00 68	Response indicates success

3.1.5.3 *Preventing Sleep with the LP_MODE Pin*

Assert (LOW) the **LP_MODE** control pin to prevent the module from sleeping under any circumstances. Properly asserting and de-asserting this pin surrounding host-to-module UART transmissions provides a more efficient power consumption while still allowing normal sleep at all other times.

NOTE: The **LP_STATUS** output pin provides an externally accessible signal to determine whether the CPU is currently awake (HIGH) or asleep (LOW).

3.1.5.4 *Managing Host and Module Sleep Simultaneously*

In applications that include both an external host MCU and a BLE module, usually both components need to sleep in order to save as much power as possible. The DATA_READY pin is asserted (LOW) whenever there is UART data in the output buffer and not yet fully clocked out of the module. Using this pin as the wakeup signal for the MCU is the recommended way to allow the module to alert the host whenever some interaction needs to occur.

Sometimes, the external MCU takes long enough to wake up that it loses the first few bits or bytes of the incoming UART data from the module. If the host needs extra time to wake up and RTS/CTS flow control is unavailable, then you should enable UART flow control in EZ-Serial with the system_set_uart_parameters (STU, ID=2/25) API command and then control the module's CTS pin from a host GPIO. When CTS is held in the deasserted (HIGH) state, the module waits to send any outgoing UART data. The host can complete its wakeup process and then assert (LOW) the module's CTS pin to allow serial data transmission when ready.

Real flow control support on the host MCU is not necessary in this case, and you can leave the module's RTS pin disconnected. However, you still need to enable flow control within EZ-Serial. Flow control is not enabled by default.

To summarize the complete cycle:

1. Host sets the module CTS pin HIGH to prevent UART transmission.
2. Host enables the DATA_READY pin falling-edge interrupt.
3. Host puts the CPU to sleep.
4. Module asserts (LOW) its DATA_READY pin when relevant activity occurs.
5. Host CPU wakes up.
6. Host sets the module CTS pin LOW to allow UART transmission.
7. Module transmits data to the host for processing.

3.1.6 How to Perform a Factory Reset

You can perform a factory reset using system_factory_reset (/RFAC, ID=2/5) API command.

EZ-Serial generates the

system_factory_reset_complete (RFAC, ID=2/3) API event immediately after erasing all settings, and before performing the final module reset to boot to the factory default state. The platform generates this event using the previously configured parser and transport mode. While this event is typically not processed by an external host during a hardware-triggered factory reset, it helps to verify the intended flow when controlling the module via software. After the reset completes, the system_boot (BOOT, ID=2/1) API event occurs.

To trigger a factory reset over the serial interface, use the system_factory_reset (/RFAC, ID=2/5) API command.

Example 8. Perform a factory reset

Direction	Text Content	Binary Content	Effect
TX→	`/RFAC`	C0 00 02 05 60	Trigger factory reset
←RX	`@R,000B,/RFAC,0000`	C0 02 02 05 00 00 62	Response indicates success
←RX	`@E,0005,RFAC`	80 00 02 03 1E	Event indicates factory reset completed
Short delay while chipset reset and boot process occurs, ~150 ms			
←RX	@E,003B,BOOT,E=0101021C,S=02020355,P=0103,H=B1,C=00,A=20737A1A21D3	80 12 02 01 1C 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20 7A	

3.2 Cable Replacement Examples with CYSPP

EZ-Serial's CYSPP implementation provides a simple way to use a BLE connection to manage a bidirectional stream of serial data. Both ends of the connection must support CYSPP, including the ability to either provide or make use of the CYSPP GATT structure for data flow. The EZ-Serial firmware can only operate as a GAP Peripheral and CYSPP Server device (typical when communicating with a smartphone)).

NOTE: EZ-Serial platform for WICED BLE modules do not support GAP Central behavior, For GAP Central and CYSPP Client device behavior, customers can use EZ-Serial with BLE modules on PSoC Creator or use WICED SMART to develop a program.

See Section 2.4.3 (Using CYSPP Mode) for a description of how CYSPP mode behaves generally and how it affects API communication.

3.2.1 How to Get Started in CYSPP Mode with Zero Custom Configuration

The factory default configuration enables the CYSPP profile in "auto-start" mode. With this configuration, the module begins advertising as soon as it has power.

3.2.1.1 How to Start CYSPP Out of the Box in Peripheral Mode

EZ-Serial's factory default configuration automatically starts CYSPP operation in the Peripheral role after booting. To establish a CYSPP data pipe, simply scan and connect from a remote device, then subscribe to RX flow control (optional) and the desired acknowledged or unacknowledged data characteristic as described in Section 2.4.3.2 (Sending and Receiving Data in CYSPP Data Mode).

A second EZ-Serial module running in CYSPP Central/Client mode (e.g., a BLE module based on PSoC Creator) will perform all required Client-side steps automatically. EZ-Serial shows all GATT events relating to CYSPP setup until the CYSPP data pipe is fully opened.

Example 9. Complete boot and CYSPP connection process in peripheral mode

Direction	Text Content	Binary Content	Effect
←RX	`@E,003B,BOOT,E=0101021C,S=02020355,P=0103,H=B1,C=00,A=20737A1A21D3`	80 12 02 01 1C 02 01 01 55 03 02 02 03 01 B1 00 D3 21 1A 7A 73 20 7A	Boot event
←RX	`@E,000E,ASC,S=01,R=03`	80 02 04 02 01 03 25	CYSPP-triggered advertisement started
←RX	`@E,0035,C,C=40,A=00A050422A0F,T=00,I=0006,L=0000,O=0064,B=00`	80 0F 04 05 40 0F 2A 42 50 A0 00 00 06 00 00 00 64 00 00 46	Connection established with remote device
←RX	`@E,001A,W,C=40,H=0015,T=00,D=0200`	80 08 05 02 40 15 00 00 02 00 02 00 81	Remote Client writes [02 00] to Client Characteristic Configuration Descriptor for RX flow control to enable indications from that characteristic.
←RX	`@E,000C,.CYSPP,S=04`	80 01 0A 01 04 29	CYSPP status update (0x04): • 0x04: Subscribed to RX flow control
←RX	`@E,001A,W,C=40,H=0012,T=00,D=0100`	80 08 05 02 40 12 00 00 02 00 01 00 7D	Remote Client writes [01 00] to Client Characteristic Configuration Descriptor for unacknowledged data to enable notifications from that characteristic.
←RX	`@E,000C,.CYSPP,S=05`	80 01 0A 01 05 2A	CYSPP status update (0x05): • 0x04: Subscribed to RX flow control • 0x01: Subscribed to unacknowledged data
Host may now send data to the module for delivery to the remote peer, received data comes from peer			

3.3 GAP Peripheral Examples

GAP Peripheral operation is one of the most common use cases for BLE designs because it is usually the simplest way to communicate with a smartphone operating as a Central device.

The Bluetooth specification defines different types of roles for the devices on each end of a BLE link:

- **Link layer**
  - Master – device which initiates a connection (always GAP Central)
  - Slave – device which accepts a connection (always GAP Peripheral)
- **GAP layer**
  - Central – device which initiated a connection (always LL master)
  - Peripheral – device which accepted a connection (always LL slave)
  - Broadcaster – device which is advertising in a non-connectable state
  - Observer – device which is scanning without initiating a connection
- **GATT layer**
  - Client – device which accesses data from a remote GATT Server
  - Server – device which provides Attribute data to be accessed remotely

Link layer roles are defined at the moment a connection is initiated based on which side initiates the connection.

The GAP layer provides four different roles, two of which involve connections (Central and Peripheral) and two of which are connectionless (Broadcaster and Observer). The link layer and GAP layer roles are closely related, particularly when a connection is involved.

The GATT layer role is independent of other behavior. A single device may even perform GATT duties in both the Client and Server roles. A common example of this is an iOS device providing the Apple Notification Center Service as a GATT Server, even though it is connected to a Peripheral device and acting as a GATT Client to that device.

EZ-Serial for the EZ-BLE WICED modules only supports **slave** link layer role, **Peripheral** or **Broadcaster** GAP roles, and **GATT Server** functionality. It does not support master, Central/Observer, or Client behavior due to resource limitations on the module.

3.3.1 How to Advertise as Peripheral Device

Advertising is the BLE activity which allows scanning devices to observe and connect to Peripherals. It is required in order for a connection to be initiated, but it may also be done in a non-connectable way (called “broadcasting”). EZ-Serial supports non-connectable broadcasting even while connected.

EZ-Serial gives you full control over when and how to advertise by using the gap_start_adv (/A, ID=4/8) API command and the gap_set_adv_parameters (SAP, ID=4/23) API command.

When the advertising state changes, the gap_adv_state_changed (ASC, ID=4/2) API event occurs. This event includes the new state as well as a code showing the reason why the state changed.

NOTE: If you do not have any automatic advertisement timeout set, then advertisements continue until you explicitly stop them or a remote device initiates a connection.

Example 10. Start advertising with custom parameters

Direction	Text Content	Binary Content	Effect
TX→	`/A,M=2,T=0,I=A0,C=6,F=0,O=1E`	C0 08 04 08 02 00 A0 00 06 00 1E 00 33	Begin advertising with custom arguments
←RX	`@R,0008,/A,0000`	C0 02 04 08 00 00 67	Response indicates success
←RX	`@E,000E,ASC,S=01,R=00`	[80 02 04 02 01 00 22	Event indicates advertising state changed to “active”

3.3.2 How to Stop Advertising as Peripheral Device

To explicitly stop advertising, use the gap_stop_adv (/AX, ID=4/9) API command, or open a connection to the module from a remote BLE Central device.

Example 11. Stop advertising

Direction	Text Content	Binary Content	Effect
TX→	`/AX`	C0 00 04 09 66	Stop advertising
←RX	`@R,0009,/AX,0000`	C0 02 04 09 00 00 6	Response indicates success
←RX	`@E,000E,ASC,S=00,R=00`	80 02 04 02 00 00 21	Event indicates advertising state changed to “inactive” due to user request

3.3.3 How to Customize Advertisement and Scan Response Data

You can customize the content of the main advertisement payload and scan response payload with the gap_set_adv_data (SAD, ID=4/19) and gap_set_sr_data (SSRD, ID=4/21) API commands, respectively.

NOTE: If you intend to use user-defined advertisement content, you must explicitly enable this in the advertisement parameters. Normally, the EZ-Serial platform manages the content in the advertisement and scan response packets automatically based on the platform configuration, including the device name and which profiles are enabled. If you set custom content but do not configure EZ-Serial to use that content, advertisement and scan response payloads remain automatically managed.

Key features and requirements for customizing data:

- Each of the advertisement and scan response packet payloads may have a maximum of 31 bytes. This is a BLE specification limit.
- Advertisement data in both packets should follow the correct **[Length, Type, Value...]** format required by the Bluetooth specification. Malformed data within advertisements can prevent proper scanning by remote devices. The **Length** value does not include itself, but does include the **Type** byte and all bytes in the remaining **Value** data.
- Each packet may contain as many fields as will fit in 31 bytes. Place multiple fields one right after the other with no special separator. Since each field begins with a “length” value, a scanning device is always able to properly identify the end of each field.
- Advertisement packets include the Bluetooth connection address (public or random) outside of the payload data. This does not count towards the 31-byte limit.
- The main advertisement packet is always transmitted while advertising. It typically includes things like connectable flags, important supported service UUIDs, and a custom manufacturer data field. Place any data that is critical for the remote device to see inside the main advertisement packet.
- The scan response packet is transmitted only when a remote device is performing an **active** scan. During an active scan, the scanning device send a **scan request** to any discovered advertising device immediately after receiving the main advertisement packet. The scan response packet typically includes the friendly name of the advertising device, and occasionally also includes transmit power, more manufacturer data, or other useful but less critical data that a remote scanning device may not need to see.

Detailed information on approved field types and their intended contents can be found the Bluetooth specification. Table 3-4 lists fields that are most commonly used:

Table 3-4. Common Advertisement Field Types

Type	Description	Value
0x01	Flags field – 1 byte of data	1 byte (bitfield)
0x02	Partial list of 16-bit UUIDs for supported GATT services	2*N bytes (UUIDs)
0x03	Complete list of 16-bit UUIDs for supported GATT services	2*N bytes (UUIDs)
0x04	Partial list of 32-bit UUIDs for supported GATT services	4*N bytes (UUIDs)
0x05	Complete list of 32-bit UUIDs for supported GATT services	4*N bytes (UUIDs)
0x06	Partial list of 128-bit UUIDs for supported GATT services	16*N bytes (UUIDs)

Type	Description	Value
0x07	Complete list of 128-bit UUIDs for supported GATT services	16*N bytes (UUIDs)
0x08	Shortened local name	0-29 bytes (Text string)
0x09	Complete local name	0-29 bytes (Text string)
0x0A	TX power level	1 byte (dBm as signed integer)
0xFF	Manufacturer data	3-29 bytes (company ID + data)

EZ-Serial does not validate advertisement or scan response payload content, and the underlying BLE stack has only limited validation on the Flags field. You must ensure that any customized data within either of these packets is correctly formatted. While the module will transmit whatever payload data is configured, scanning devices may not correctly identify your device if the data is malformed or missing (especially the Flags field).

The stack requires that the Flags field, if present, must have the final two bits set so that they match the Discovery Mode setting used when starting advertisements. For BLE-only devices which do not support "classic" BR/EDR Bluetooth behavior, this means that the flags byte will almost always be one of these three values:

- 0x04: Non-discoverable/broadcast-only (common for beacon-only devices)
- 0x05: Limited discoverable
- 0x06: General discoverable (most common for connectable devices)

See gap_start_adv (/A, ID=4/8) API command for additional reference on discoverable modes.

Table 3-5 provides examples for reference:

Table 3-5. Examples of Well-Formed Advertisement Fields

Byte content	Field Description
`02 01 06`	**Length:** 2 bytes **Type:** Flags (0x01) **Value:** LE General Discoverable Mode, BR/EDR Not Supported
`05 02 09 18 0D 18`	**Length:** 5 bytes **Type:** Complete list of 16-bit UUIDs for supported GATT services (0x02) **Value:** 0x1809 (Health Thermometer), 0x180D (Heart Rate)
`07 08 57 69 64 67 65 74`	**Length:** 7 bytes **Type:** Shortened local name (0x08) **Value:** "Widget"
`09 FF 31 01 AA BB CC DD EE FF`	**Length**: 9 bytes **Type:** Manufacturer data (0xFF) **Value:** Company ID = 0x0131 (Cypress Semiconductor) Data = `[AA BB CC DD EE FF]`

These four example fields require 25 bytes when combined, including each of the four **Length** values. They can be placed in a single advertisement packet if desired:

```
02 01 06 05 02 09 18 0D 18 07 08 57 69 64 67 65 74 09 FF 31 01 AA BB CC DD EE FF
```

Here, the shortened name is included in the same packet as the more critical information. This is uncommon, but not prohibited. The name typically goes in the scan response packet because there it cannot fit into the advertisement packet, but any field may be in any location as long as the scanning device knows what to expect.

Example 12. Set custom advertisement and scan response data

Direction	Text Content	Binary Content	Effect
TX→	`SAP,M=2,T=0,I=30,C=7,L=0,O=0,F=2`	C0 09 04 17 02 00 30 00 07 00 00 00 02 B8	Enable user-defined advertisement and scan response content
←RX	`@R,0009,SAP,0000`	C0 02 04 17 00 00 7	Response indicates success
TX→	`SAD,D=020106060209180D18`	C0 0A 04 13 09 02 01 06 06 02 09 18 0D 18 DA C0 02 04 13 00 00 72	Set new advertisement content (RAM only), Flags and 16-bit UUID fields

Direction	Text Content	Binary Content	Effect
←RX	`@R,0009,SAD,0000`	C0 02 04 13 00 00 72	Response indicates success
TX→	`SSRD,D=0708576964676574`	C0 09 04 15 08 07 08 57 69 64 67 65 74 F6	Set new scan response content (RAM only), Complete local name field
←RX	`@R,000A,SSRD,0000`	C0 02 04 15 00 00 74	Response indicates success

Example 13. Set advertisement and scan response data to value similar to factory defaults

Direction	Text Content	Binary Content	Effect
TX→	`SAP,M=2,T=0,I=30,C=7,L=0,O=0,F=1`	C0 09 04 17 02 00 30 00 07 00 00 00 01 B7	Enable user-defined advertisement and scan response content
←RX	`@R,0009,SAP,0000`	C0 02 04 17 00 00 76	Response indicates success
TX→	`SAD,D=020106110700a10c2000089a9ee21115a133333365`	C0 16 04 13 15 02 01 06 11 07 00 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 70	Set new advertisement content (RAM only)
←RX	`@R,0009,SAD,0000`	C0 02 04 13 00 00 72	Response indicates success
TX→	`SSRD,D=1309455a2d53657269616c2045333a38333a3546`	C0 15 04 15 14 13 09 45 5A 2D 53 65 72 69 61 6C 20 45 33 3A 38 33 3A 35 46 D5	Set new scan response content (RAM only)
←RX	`@R,000A,SSRD,0000`	C0 02 04 15 00 00 74	Response indicates success

3.4 GATT Server Examples

BLE data transfer operations between two connected devices most often occur through the GATT layer, with a Server on one side and a Client on the other side. The GATT Server makes use of a pre-defined attribute structure, which the Client may remotely discover and use as needed. The GATT Server defines what data is available and how it may be accessed, and has limited ability to push data to the Client if the Client has subscribed to receive these types of updates.

3.4.1 How to Define Custom Local GATT Services and Characteristics

EZ-Serial implements a dynamic GATT structure that can be modified at runtime and stored in flash. Note that the structure itself and values stored within data characteristics (other than default values defined when creating new entries) are stored in RAM only, and is stored to flash until explicitly calling command gatts_store_db (/SGDB, ID=5/4) or system_store_config (/SCFG, ID=2/4).

The EZ-Serial platform contains a few pre-defined GATT elements in the factory default configuration. EZ-Serial requires these for correct operation, and they cannot be removed or modified. However, additional structural elements are entirely customizable.

A GATT structure is fundamentally made up of individual attributes, each of which has a unique numeric handle, a UUID that is 16 bits, 32 bits, or 128 bits wide, and a value container. Attribute handles start at 1 and may go up to 0xFFFF (65535). No two attributes may have the same handle. WICED BLE EZ-Serial firmware internally use two structures to store individual attribute:

`gatts_db[]`, an array of GATT entry structures containing the fixed-length portion of each entry (type, permissions, length, and the 16-bit length prefix value from the data array)

`gatts_db_const_data[]`, an array of UINT8 bytes containing the variable-length portion of each entry (the payload from the data array)

EZ-Serial provides the gatts_create_attr (/CAC, ID=5/1) API command to create a new custom attribute, which in the WICED Smart EZ-Serial firmware takes the following arguments:

uint8 type

uint8 permissions

uint16 length

longuint8a data

The first 6 bytes of this packed structure (through the 16-bit length prefix on data) is a match for the GATT entry structure. Any payload data in the data structure goes in the constant data pool instead.

In order to use this correctly, you must have some prior knowledge of correct GATT structures, especially in the case of a characteristic declaration which includes additional metadata beyond just the value attribute's UUID. The following pseudocode demonstrates how you would use this command to add the Generic Access service with a writeable Device Name value (though in EZ-Serial you should not do this since it is part of the default structure and cannot be removed):

```
// syntax: gatts_create_attr(type, permissions, length, data[])
gatts_create_attr(0, 0x02, 4,  [0x00, 0x28, 0x00, 0x18])
gatts_create_attr(0, 0x02, 7,  [0x03, 0x28, 0x02, 0x03, 0x00, 0x00, 0x2A])
gatts_create_attr(1, 0x0B, 64, [])
gatts_create_attr(0, 0x02, 7,  [0x03, 0x28, 0x02, 0x05, 0x00, 0x01, 0x2A])
gatts_create_attr(1, 0x02, 2,  [])
gatts_create_attr(0, 0x02, 7,  [0x03, 0x28, 0x02, 0x07, 0x00, 0x04, 0x2A])
gatts_create_attr(1, 0x02, 8,  [])
```

Each characteristic declaration (2nd, 4th, and 6th entries) contain the 0x2803 structural UUID, 0x02 properties byte, then the 16-bit attribute handle (0x0003, 0x0005, and 0x0007 respectively) corresponding to the value attribute. EZ-Serial requires the value attribute to come immediately after the declaration.

WARNING: Modifications to the custom GATT structure require flash write operations, which can potentially disrupt BLE connectivity. Therefore, you should only make changes to the GATT database while there is no active BLE connection to avoid the possibility of a connection loss.

3.4.1.1 Understanding Custom GATT Limitations

The dynamic GATT implementation in EZ-Serial contains some built-in entries to provide required EZ-Serial functionality, leaving the remaining space available for custom entries. Each entry is assembled by two structures:

GATT attribute entry: containing the fixed-length portion of each entry (type, permissions, length, and the 16-bit length prefix value from the data array)

GATT data array: containing the variable-length portion of each entry.

The table below lists each relevant value on both platforms:

Table 3-6. Dynamic GATT Structural Limitations

Category	**Built-in**	**CYBLE-013025-00**	
		Total	**Avail.**
SRAM reserved for GATT attribute entries	21*6 = 126 bytes	128*6=768 bytes	107*6=642 bytes
SRAM reserved for GATT data arrays	38+87 = 125 bytes	768 bytes	643 bytes
Flash memory room reserved for storing GATT data base	251 bytes	1536 bytes	1285 bytes

Attempting to create a new custom attribute which exceeds any of these bounds will generate an error result indicating the nature of the limitation. See Section 7.4 , Error Codes for details.

3.4.1.2 Building Custom Services and Characteristics

The GATT database is made up of one or more primary services. Each primary service has a service declaration (UUID 0x2800) and includes one or more characteristics. Each characteristic has a characteristic declaration (UUID 0x2803) and a value attribute (any UUID not in the above list), and often has additional characteristic-related descriptors in the 0x2900 range.

UUIDs indicate the purpose of each attribute, but may be (and often are) repeated through the complete database. For example, a database containing three services will contain three separate attributes which all have the UUID 0x2800, which is the official "Primary Service Declaration" UUID defined by the Bluetooth SIG. Table 3-7 lists notable pre-defined structural definition UUIDs from the Bluetooth SIG.

Table 3-7. Bluetooth SIG Structural UUIDs

UUID	Description
0x2800	Primary Service Declaration
0x2801	Secondary Service Declaration
0x2802	Include Declaration
0x2803	Characteristic Declaration
0x2900	Characteristic Extended Properties
0x2901	Characteristic User Description
0x2902	Client Characteristic Configuration
0x2903	Server Characteristic Configuration
0x2904	Characteristic Format
0x2905	Characteristic Aggregate Format

Further detail on these and other official identifiers can be found on the Bluetooth SIG website.

When defining GATT elements at runtime, you must enter each attribute in the correct order based on the desired structure. Any entries that do not conform to the correct order requirement will be rejected with a validation error. The only case where a validation *warning* is allowed is when you define a new service or characteristic declaration and have not yet entered the subsequent attributes which must follow. You can use the gatts_validate_db (/VGDB, ID=5/3) API command at any time to perform an integrity check on the current GATT structure to see whether additional attributes are expected.

The required order for each complete characteristic definition (declaration, value, and optional descriptors) is dictated by the internal BLE stack as follows:

Table 3-8. Required Characteristic Attribute Order

Order	UUID	Description	Required
#1	0x2803	Characteristic Declaration	**Yes**
#2	<custom>	Characteristic Value	**Yes**
#3	0x2900	Characteristic Extended Properties	No
#4	0x2901	Characteristic User Description	No
#5	0x2902	Client Characteristic Configuration	No
#6	0x2903	Server Characteristic Configuration	No
#7	0x2904	Characteristic Format	No
#8	0x2905	Characteristic Aggregate Format	No

Any optional attributes may be omitted as long as all provided attributes are supplied in the above order.

For details on how to use custom GATT creation API commands to add support for Bluetooth SIG official services such as Device Information, Health Thermometer, and others, see Section 10.2 (Adopted Bluetooth SIG GATT Profile Structure Snippets) and the API reference material for gatts_create_attr (/CAC, ID=5/1).

3.4.1.3 *Choosing Correct GATT Permissions*

It is critical to use correct permissions when defining any custom GATT structural elements. See Section 10.2 (Adopted Bluetooth SIG GATT Profile Structure Snippets) for example definitions, and you may notice certain patterns. Here are the recommended guidelines for the most common entries:

- **Service declarations (UUID = 0x2800)**

  PERM =0x02

  - PERM_READABLE

  Characteristic properties are not needed because they do not apply.

- **Characteristic declarations (UUID = 0x2803)**

  PERM =0x02
  - PERM_READABLE

  Characteristic properties = <actual properties>
- **Characteristic value attributes (type = 0x0000)**

  PERM =0x89
  - PERM_VARIABLE_LENGTH
  - PERM_WRITE_REQ
  - PERM_SERVICE_UUID_128 (if this service has a 128-bit UUID)

  Characteristic properties value is not required because it has been defined in previous characteristic declarations.
- **Characteristic user description attributes (UUID = 0x2901)**

  PERM =0x02
  - PERM_READABLE

  Characteristic properties = 0x02 (read)
- **Client characteristic configuration attributes (UUID = 0x2902)**

  PERM =0x0A
  - PERM_READABLE
  - PERM_WRITE_REQ

  Characteristic properties = 0x0A (read + write)

In general, structural elements such as service and characteristic declarations should be read-only, but should have no particular security restrictions on them. This ensures that a connected Client is able to discover the database structure correctly, even if additional security is required to execute read and/or write operations on the characteristic value attributes. Some Android devices are known to have problems during discovery if the declaration descriptors themselves have extra security requirements.

3.4.2 How to List Local GATT Services, Characteristics, and Descriptors

Listing the local GATT structure can be helpful in certain cases, even though it is typically the remote GATT structure that requires discovery. This is especially true because you can dynamically change the local GATT structure at runtime. EZ-Serial provides three commands for local discovery.

3.4.2.1 Discovering Local GATT Services

Use the gatts_discover_services (/DLS, ID=5/6) API command to obtain a list of services in the local GATT database.

Example 14. Local GATT service discovery with factory default structure (no custom attributes)

Direction	Text Content	Binary Content	Effect
TX→	`/DLS,B=0,E=0`	C0 04 05 06 00 00 00 00 68	Request to discover all local services
←RX	`@R,0011,/DLS,0000,C=0003`	C0 04 05 06 00 00 03 00 6B	Response indicates success, 3 records to follow
←RX	`@E,0024,DL,H=0001,R=0007,T=2800,P=00,U=0018`	80 0A 05 01 01 00 07 00 00 28 00 02 00 18 73	Service 0x1800, start=1, end=7
←RX	`@E,0024,DL,H=0008,R=000B,T=2800,P=00,U=0118`	80 0A 05 01 01 00 07 00 00 28 00 02 00 18 73	Service 0x1801, start=8, end=11 (0x0B)
←RX	`@E,0040,DL,H=000C,R=0015,T=2800,P=00, U=00A10C2000089A9EE21115A133333365`	80 18 05 01 0C 00 15 00 00 28 00 10 00 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 44	Service 0x6533…A100, start=12 (0x0C), end=21 (0x15)

3.4.2.2 Discovering Local GATT Characteristics

Use the gatts_discover_characteristics (/DLC, ID=5/7) API command to obtain a list of characteristics in the local GATT database.

Example 15. Local GATT characteristic discovery with factory default structure (no custom attributes)

Direction	Text Content	Binary Content	Effect
TX→	`/DLC,B=0,E=0,S=0`	C0 06 05 07 00 00 00 00 00 00 6B	Request to discover all local characteristics
←RX	`@R,0011,/DLC,0000,C=0007`	C0 04 05 07 00 00 07 00 70	Response indicates success, 7 records to follow
←RX	`@E,0024,DL,H=0002,R=0003,T=2803,P=02,U=002A`	80 0A 05 01 02 00 03 00 03 28 02 02 00 2A 87	Char 0x2A00, decl handle=2, value handle=3, perm=0x02
←RX	`@E,0024,DL,H=0004,R=0005,T=2803,P=02,U=012A`	80 0A 05 01 02 00 03 00 03 28 02 02 00 2A 87	Char 0x2A01, decl handle=4, value handle=5, perm=0x02
←RX	`@E,0024,DL,H=0006,R=0007,T=2803,P=02,U=042A`	80 0A 05 01 04 00 05 00 03 28 02 02 01 2A 8C	Char 0x2A04, decl handle=6, value handle=7, perm=0x02
←RX	`@E,0024,DL,H=0009,R=000A,T=2803,P=22,U=052A`	80 0A 05 01 09 00 0A 00 03 28 22 02 05 2A BA	Char 0x2A05, decl handle=9, value handle=10, perm=0x22
←RX	`@E,0040,DL,H=000D,R=000E,T=2803,P=28, U=01A10C2000089A9EE21115A133333365`	80 18 05 01 0D 00 0E 00 03 28 28 10 01 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 6A	Char 0x6533…A101, decl handle=13, value handle=14, perm=0x28
←RX	`@E,0040,DL,H=0010,R=0011,T=2803,P=14, U=02A10C2000089A9EE21115A133333365`	80 18 05 01 10 00 11 00 03 28 14 10 02 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 5D	Char 0x6533…A102, decl handle=16, value handle=17, perm=0x14
←RX	`@E,0040,DL,H=0013,R=0014,T=2803,P=20, U=03A10C2000089A9EE21115A133333365`	80 18 05 01 13 00 14 00 03 28 20 10 03 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 70	Char 0x6533…A103, decl handle=19, value handle=20, perm=0x20

3.4.2.3 Discovering Local GATT Descriptors

Use the gatts_discover_descriptors (/DLD, ID=5/8) API command to obtain a list of descriptors in the local GATT database.

Example 16. Local GATT descriptor discovery with factory default structure (no custom attributes)

Direction	Text Content	Binary Content	Effect
TX→	`/DLD,B=0,E=0,S=0,C=0`	C0 08 05 08 00 00 00 00 00 00 00 00 6E	Request to discover all local descriptors
←RX	`@R,0011,/DLD,0000,C=0015`	C0 04 05 08 00 00 15 00 7F	Response indicates success, 21 records to follow
←RX	`@E,0024,DL,H=0001,R=0007,T=2800,P=00,U=0028`	80 0A 05 01 01 00 07 00 00 28 00 02 00 28 83	UUID 0x2800 (Primary Service), start=1, end=7
←RX	`@E,0024,DL,H=0002,R=0003,T=2803,P=02,U=0328`	80 0A 05 01 02 00 03 00 03 28 02 02 03 28 88	UUID 0x2803 (Characteristic), decl=2, value handle=3
←RX	`@E,0024,DL,H=0003,R=0000,T=0000,P=02,U=002A`	80 0A 05 01 03 00 00 00 00 00 02 02 00 2A 5A	UUID 0x2A00 (Device Name), handle=3, perm=0x02
Additional records omitted for brevity			

Direction	Text Content	Binary Content	Effect
←RX	`@E,0024,DL,H=000C,R=0015,T=2800,P=00,U=0028`	80 0A 05 01 0C 00 15 00 00 28 00 02 00 28 9C	UUID 0x2800 (Primary Service), start=12, end=21
←RX	`@E,0024,DL,H=000D,R=000E,T=2803,P=28,U=0328`	80 0A 05 01 0D 00 0E 00 03 28 28 02 03 28 C4	UUID 0x2803 (Characteristic), decl=13, value handle=14, perm=0x28
←RX	`@E,0040,DL,H=000E,R=0000,T=0000,P=28, U=01A10C2000089A9EE21115A133333365`	80 18 05 01 0E 00 00 00 00 00 28 10 01 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 32	UUID 0x6533…A101 (Acknowledged Data Characteristic:), handle=14, perm=0x28
←RX	`@E,0024,DL,H=000F,R=0000,T=2902,P=0A,U=0229`	80 0A 05 01 0F 00 00 00 02 29 0A 02 02 29 9A	UUID 0x2902 (CCCD), handle=15, perm=0x0A
←RX	`@E,0024,DL,H=0010,R=0011,T=2803,P=14,U=0328`	80 0A 05 01 10 00 11 00 03 28 14 02 03 28 B6	UUID 0x2803 (Characteristic), decl=16, value handle=17, perm=0x28
←RX	`@E,0040,DL,H=0011,R=0000,T=0000,P=14, U=02A10C2000089A9EE21115A133333365`	80 18 05 01 11 00 00 00 00 00 14 10 02 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 22	UUID 0x6533…A102 (Unacknowledged Data Characteristic), handle=17, perm=0x28
←RX	`@E,0024,DL,H=0012,R=0000,T=2902,P=0A,U=0229`	80 0A 05 01 12 00 00 00 02 29 0A 02 02 29 9D	UUID 0x2902 (CCCD), handle=18, perm=0x0A
←RX	`@E,0024,DL,H=0013,R=0014,T=2803,P=20,U=0328`	80 0A 05 01 13 00 14 00 03 28 20 02 03 28 C8	UUID 0x2803 (Characteristic), decl=19, value handle=20, perm=0x28
←RX	`@E,0040,DL,H=0014,R=0000,T=0000,P=20, U=03A10C2000089A9EE21115A133333365`	80 18 05 01 14 00 00 00 00 00 20 10 03 A1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 32	UUID 0x6533…A103 (RX Flow Characteristic), handle=20, perm=0x20
	`@E,0024,DL,H=0015,R=0000,T=2902,P=0A,U=0229`	80 0A 05 01 15 00 00 00 02 29 0A 02 02 29 A0	UUID 0x2902 (CCCD), handle=21, perm=0x0A

3.4.3 How to Read and Write Local GATT Attribute Values

Read and write local GATT values using the gatts_read_handle (/RLH, ID=5/9) and gatts_write_handle (/WLH, ID=5/10) API commands, respectively.

It is always possible to locally read any attribute, and locally write any attribute that supports the write operation. Some attributes, such as service and characteristic declarations, contain only constant data (stored in flash) that is not meant to be modified with a typical GATT write command. If you intend to change the structure of the GATT database itself, use the gatts_create_attr (/CAC, ID=5/1) and gatts_delete_attr (/CAD, ID=5/2) API commands.

3.4.3.1 Reading Local GATT Data

You can read the value of a local attribute using the gatts_read_handle (/RLH, ID=5/9) API command. EZ-Serial will return the current value in the response.

Example 17. Read local Device Name characteristic

Direction	Text Content	Binary Content	Effect
TX→	`/RLH,H=3`	C0 02 05 09 03 00 6C	Read attribute with handle = 3
←RX	`@R,0031,/RLH,0000, D=455A2D53657269616C2031 413A32313A4433`	[C0 16 05 09 00 00 12 00 45 5A 2D 53 65 72 69 61 6C 20 31 41 3A 32 31 3A 44 33 9B	Response indicates success, hex data is “EZ-Serial 1A:21:D3”

3.4.3.2 Writing Local GATT Data

You can write the value of a local attribute using the gatts_write_handle (/WLH, ID=5/10) API command. This command replaces any existing data in the attribute and is limited by the maximum length of the attribute in the GATT structure.

Writing data does not automatically push a notification or indication packet to a remote Client, even if the Client has subscribed to either of these types of pushed updates. See Section 3.4.4 (How to Notify and Indicate Data to a Remote Client) for details on how to push data.

Example 18. Write "ABCD" at beginning of local Device Name characteristic

Direction	Text Content	Binary Content	Effect
TX→	`/WLH,H=3,D=41424344`	C0 08 05 0A 03 00 04 00 41 42 43 44 81	Write "ABCD" (hex) into attribute with handle = 3
←RX	`@R,000A,/WLH,0000`	[C0 02 05 0A 00 00 6A	Response indicates success
TX→	`/RLH,H=3`	C0 02 05 09 03 00 6C	Read attribute with handle = 3 to verify
←RX	`@R,0031,/RLH,0000,D=41424344`	C0 08 05 09 00 00 04 00 41 42 43 44 7D	Response indicates success, data shows expected value

3.4.4 How to Notify and Indicate Data to a Remote Client

Notifying and indicating both allow a Server to push updates to a Client without the Client specifically requesting the latest values. These transfer mechanisms provide an efficient way to send real-time updates without constant polling from the Client side, saving power for use cases such as remote sensors or any interrupt-driven activities.

Notifications and indications both transmit data from the Server to the Client, but notifications are **unacknowledged**, while indications are **acknowledged**. You can transmit multiple notifications during a single connection interval, but you can only transmit one indication every two connection intervals (one interval for the transmission and one for the acknowledgement).

Although the Server decides when to push data to the Client using these methods, the Client retains ultimate control over whether the Server may transmit at all, via the use of "subscription" bits for each type of transfer. All GATT characteristics which support either the "notify" or "indicate" operation must have a "Client Characteristic Configuration Descriptor" (CCCD) within the set of attributes making up the complete characteristic structure. For example, the "Service Changed" characteristic (UUID 0x2A05) within the "Generic Attribute" service (UUID 0x1801) is made up of three separate attributes:

Table 3-9. Service Changed GATT Characteristic Structure

Handle	UUID	Description
0x0009	0x2803	Characteristic declaration
0x000A	0x2A05	Service change value attribute
0x000B	0x2902	Client Characteristic Configuration Descriptor (CCCD)

This characteristic supports the "indicate" operation. In order for a Client to subscribe to indications, it must set Bit 1 (0x02) of the value in the CCCD. This descriptor holds a 16-bit value, so the correct operation on the Client side is to write `[ 02 00 ]` to the 0x000B handle.

For characteristics that support the "notify" operation, the correct subscription flag is Bit 0 (0x01).

Notification and indication subscriptions do not persist across multiple connections.

3.4.4.1 Notifying Data to a Remote Client

Use the gatts_notify_handle (/NH, ID=5/11) API command to notify data to a remote Client. You must use a handle corresponding to a value attribute for a characteristic for which the remote Client has already subscribed to notifications by writing 0x0001 to the relevant CCCD. First, you need create a CCCD value as follows:.

NOTE: Notifying data to a Client requires an active connection.

Example 19. Notify a four-byte value to a Client manually using the customized characteristic with CCCD

Direction	Text Content	Binary Content	Effect
TX→	/CAC,T=00,P=2,L=0012,D=002800B10C2000089A9EE21115A133333365	C0 18 05 01 00 02 12 00 12 00 00 28 00 B1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 89	Create a new CCCD value as follows:. First, create new service, UUID=.
←RX	@R,0018,/CAC,0000,H=0016,V=0001	C0 06 05 01 00 00 16 00 01 00 7C	Response indicates success.
TX→	/CAC,T=00,P=2,L=0015,D=032828180001B10C2000089A9EE21115A133333365	[C0 1B 05 01 00 02 15 00 15 00 03 28 28 18 00 01 B1 0C 20 00 08 9A 9E E2 11 15 A1 33 33 33 65 D6	Then, create a characteristic.
←RX	@R,0018,/CAC,0000,H=0017,V=0001	C0 06 05 01 00 00 17 00 01 00 7D	Response indicates success.
TX→	/CAC,T=01,P=B9,L=0014,D=	C0 06 05 01 01 89 14 00 00 00 03	Create a value for the above characteristic.
←RX	@R,0018,/CAC,0000,H=0018,V=0000	C0 06 05 01 00 00 18 00 00 00 7D	Response indicates success.
TX→	/CAC,T=00,P=0A,L=04,D=0229	C0 08 05 01 00 0A 04 00 02 00 02 29 A2	Create CCCD.
←RX	@R,0018,/CAC,0000,H=0019,V=0000	C0 06 05 01 00 00 19 00 00 00 7E	
←RX	@E,0035,C=40,A=00A05012C722,T=00,I=0007,L=0000,O=000A,B=0	80 0F 04 05 40 22 C7 12 50 A0 00 00 07 00 00 00 0A 00 00 6D	Connected from peer device
←RX	@E,001A,W,C=40,H=0019,T=00,D=0100	80 08 05 02 40 19 00 00 02 00 01 00 84	Subscribe service by peer device.
TX→	`/NH,C=40,H=18,D=41424344`	C0 08 05 0B 40 18 00 04 41 42 43 44 D7	Notify “ABCD” (hex) via attribute with handle = 17 (0x11).
←RX	`@R,0009,/NH,0000`	C0 02 05 0B 00 00 6B	Response indicates success.

3.4.4.2 *Indicating Data to a Remote Client*

Use the gatts_indicate_handle (/IH, ID=5/12) API command to indicate data to a remote Client. You must use a handle corresponding to a value attribute for a characteristic for which the remote Client has already subscribed to indications by writing 0x0002 to the relevant CCCD.

NOTE: Indicating data to a Client requires an active connection.

Example 20. Indicate a start/end handle range to a Client through the Service Changed Characteristic

Direction	Text Content	Binary Content	Effect
←RX	`@E,001A,W,C=40,H=000B,T=00,D=0200`	80 08 05 02 40 0B 00 00 02 00 02 00 77	Remote Client writes 0x002 to handle 0x0B to subscribe the Service Changed Characteristic.
TX→	`/IH,C=40,H=A,D=1D002500`	C0 08 05 0C 40 0A 00 04 1D 00 25 00 02	Write 1D002500 via attribute with handle = 10 (0x0A)
←RX	`@R,0009,/IH,0000`	C0 02 05 0C 00 00 6C	Response indicates success.
←RX	`@E,000F,IC,C=40,H=000A`	80 03 05 03 40 0A 00 6E	Event indicates Client has confirmed receipt of data.

3.4.5 How to Detect and Process Written Data from a Remote Client

Write operations from a remote GATT Client generates the gatts_data_written (W, ID=5/2) API event, containing the handle and value data as well as the remote connection handle from the device that initiated the request. This event occurs only if the write succeeds and was not blocked due to incorrect permissions, insufficient encryption or authentication levels, or invalid length or offset.

NOTE: EZ-Serial does not currently implement an API event for read requests.

3.5 Security and Encryption Examples

EZ-Serial supports built-in Bluetooth security technologies for safeguarding sensitive data transmitted wirelessly, including privacy and encryption.

3.5.1 How to Bond With or Without MITM Protection

Bonding between two devices requires first generating and exchanging encryption keys and then permanently storing encryption data along with the information required to identify the bonded device and reuse the same keys again in the future. The mechanism of pairing depends on which side (master or slave) initiates the pairing request, and the I/O capabilities of each side.

NOTE: While the Bluetooth specification allows pairing (generation and exchange of encryption keys) without bonding (permanent storage of encryption data), most common smartphones, tablets, and computer operating systems require performing both at the same time if you need encryption. The encryption-only arrangement (no bonding) is supported only between modules that support pairing without bonding.

EZ-Serial supports pairing with or without MITM protection enabled. The factory default settings apply the so-called “just works” method, with no passkey entry and no MITM protection. EZ-Serial also supports the configuration of a fixed passkey to be used during the pairing process instead of no passkey.

3.5.1.1 Controlling Automatic Pairing Request Acceptance

EZ-Serial’s default behavior is to accept all compatible pairing requests that come in from other devices. However, your application may benefit from having more control over the pairing process. To change this, clear Bit 0 (0x01) of the **`flags`** value in the smp_set_security_parameters (SSBP, ID=7/11) API command. Subsequent pairing requests generates the smp_pairing_requested (P, ID=7/2) API event; you must respond with the smp_send_pairreq_response (/PR, ID=7/5) API command to accept or reject the request.

The example below assumes that you have already connected to a remote peer device. An active connection is required for any type of pairing operation to succeed. However, configuration of security settings may be done either before or after connecting.

Example 21. Disable automatic acceptance of incoming pairing requests, store in flash, then pair from remote peer

Direction	Text Content	Binary Content	Effect
TX→	`SSBP$,M=11,B=1,K=10,P=0,I=3,F=0`	D0 06 07 0B 11 01 10 00 03 00 A6	Clear Bit 0 (auto-accept)
←RX	`@R,000B,SSPB$,0000`	D0 02 07 0B 00 00 7D	Response indicates success; stored in flash.
←RX	`@E,001B,P,C=40,M=10,B=01,K=10,P=00`	80 05 07 02 40 10 01 10 00 88	Event indicates incoming pairing request.
TX→	`/PR,C=40,R=0`	C0 03 07 05 40 00 00 A8	Send pairing request response with “0” result (accept).
←RX	`@R,0009,/PR,0000`	C0 02 07 05 00 00 67	Response indicates success.
←RX	`@E,000E,ENC,C=40,S=01`	80 02 07 04 40 01 67	Event indicates encryption status changed.
←RX	`@E,001B,B,B=40,A=00A05008B3A8,T=00`	80 08 07 01 40 A8 B3 08 50 A0 00 00 BC	Event indicates new bond entry created.
←RX	`@E,000F,PR,C=40,R=0000`	80 03 07 03 40 00 00 66	Event indicates pairing process completed successfully.

3.5.1.2 *Pairing and Bonding in “Just Works” Mode Without MITM Protection*

The simplest way to bond requires no special passkey entry or display. If your device has no input or output capabilities, you must use this mode for pairing since MITM protection requires numeric display or entry (or both) to function correctly.

The example below assumes that you have already connected to a remote peer device. An active connection is required for any type of pairing operation to succeed. However, configuration of security settings may be done either before or after connecting.

Example 22. Configure simple pairing without MITM protection, then initiate pairing

Direction	Text Content	Binary Content	Effect
TX→	`SSBP,M=11,B=1,K=10,P=0,I=3,F=1`	C0 06 07 0B 11 01 10 00 03 01 97	Set “No Input / No Output” I/O (Factory default).
←RX	`@R,000A,SSPB,0000`	C0 02 07 0B 00 00 6D	Response indicates success.
TX→	`/P,C=40,B=0,K=10,M=11,P=0`	C0 05 07 03 40 11 00 10 00 C9	Initiate pairing request to remote peer.
←RX	`@R,0008,/P,0000`	C0 05 07 03 40 11 00 10 00 C9	Response indicates success.
←RX	`@E,000E,ENC,C=40,S=01`	80 02 07 04 40 01 67	Event indicates encryption status changed (peer accepted).
←RX	`@E,001B,B,B=40,A=00A05008B3A8,T=00`	80 08 07 01 40 A8 B3 08 50 A0 00 00 B	Event indicates new bond entry created.
←RX	`@E,000F,PR,C=40,R=0000`	80 03 07 03 40 00 00 66	Event indicates pairing process completed successfully.

3.5.1.3 *Pairing and Bonding with a Fixed Passkey*

EZ-Serial supports the configuration of a fixed passkey to be used during the pairing process instead of either no passkey or a random one. You can choose a fixed 6-digit value between 000000 and 999999 by using the smp_set_fixed_passkey (SFPK, ID=7/13) API command and configuring the local I/O capabilities to the “Display Only” value with the smp_set_security_parameters (SSBP, ID=7/11) API command.

NOTE: The fixed passkey takes effect only if you enable fixed passkey use by setting Bit 1 (0x02) of the security flags parameter and set the “Display Only” I/O capabilities value (0x00) using the smp_set_security_parameters (SSBP, ID=7/11) API command. If both of these conditions are not met, the stack reverts to the default behavior of using a random passkey.

The example below assumes that you have already connected to a remote peer device. An active connection is required for any type of pairing operation to succeed. However, configuration of security settings may be done either before or after connecting.

Example 23. Configure “123456” fixed passkey value and required I/O capabilities, then pair from remote peer

Direction	Text Content	Binary Content	Effect
TX→	`SSBP,M=11,B=1,K=10,P=0,I=0,F=3`	C0 06 07 0B 11 01 10 00 00 03 96	Set “Display Only” I/O, enable fixed passkey use flag bit (0x02).
←RX	`@R,000A,SSPB,0000`	C0 02 07 0B 00 00 6D	Response indicates success.
TX→	`SFPK,P=1E240`	C0 04 07 0D 40 E2 01 00 94	Set fixed passkey value (1E240 hex = **123456** dec).
←RX	`@R,000A,SFPK,0000`	C0 02 07 0D 00 00 6F	Response indicates success.
←RX	`@E,000E,ENC,C=40,S=01`	80 02 07 04 40 01 67	Event indicates encryption status changed (peer entered key).
←RX	`@E,001B,B,B=40,A=00A05012C722,T=0`	80 08 07 01 40 22 C7 12 50 A0 00 00 54	Event indicates new bond entry created.
←RX	`@E,000F,PR,C=40,R=0000`	80 03 07 03 40 00 00 66	Event indicates pairing process completed successfully.

3.6 Performance Testing Examples

This section covers techniques to achieve optimal performance in specific contexts.

3.6.1 How to Maximize Throughput to a Remote Peer

Throughput concerns how much data you can move across a link within a specific period of time, usually expressed in bytes per second or bits per second (8 bits per byte). In the case of BLE, the following guidelines help improve the average throughput:

- **Minimize the connection interval.** The BLE specification allows 7.5 ms minimum connection interval. Data transfers are specifically timed during BLE connections, and more frequent transfers mean higher potential throughput.
  - When operating in the **GAP Peripheral** role, the remote Central determines the initial interval, and you must request an update with the gap_update_conn_parameters (/UCP, ID=4/3) API command after connecting. The remote peer (master/central device) may either accept or reject this request. Note that if the remote peer rejects the request, it does not notify the requesting device; the only evidence of the rejection is the lack of a subsequent gap_connection_updated (CU, ID=4/8) API event.
- **Maximize the payload size for GATT transfers.** It takes much longer to send 20 one-byte packets than one 20-byte packet, due to the low transmission duty cycle required by the BLE protocol. If your application has five 16-bit sensor measurement values that are used to the remote peer on the same interval, use a single characteristic to send all 10 bytes at once rather than using five separate characteristics.
- **Use unacknowledged transfers.** You can push more unacknowledged data through in a single connection interval than you can with acknowledged transfers. A typical acknowledged data transfer requires two full connection intervals to complete (one for the transfer and one for the acknowledgement), but multiple unacknowledged transfers can be used in sequence within the same interval—up to one packet every 1.25 ms, if supported by the remote Client. Typically, standalone full-stack modules cannot buffer and process data quite this fast, but it is often possible to achieve something near this level of throughput. Note that making this change may require additional application logic to provide a packet delivery/retry request mechanism.
  - For **Server-to-Client** transfers, use the "notify" operation instead of "indicate."

These actions help increase the observed throughput, but simultaneously increase power consumption. Keep this trade-off in mind to choose the right balance between power consumption and throughput.

Example 24. Request a connection parameter update to 7.5-ms interval, no latency, 1-second timeout

Direction	Text Content	Binary Content	Effect
TX→	`/UCP,C=40,I=6,L=0,O=64`	C0 07 04 03 40 06 00 00 00 64 00 11	Request connection update to 7.5 ms (6 * 1.25 ms), no slave latency, 1-second supervision timeout.
←RX	`@R,000A,/UCP,0000`	C0 02 04 03 00 00 62	Response indicates success; request sent to remote peer.
←RX	`@E,001D,CU,H=40,I=0006,L=0000,O=0064`	80 07 04 08 40 06 00 00 00 64 00 D6	Event indicates new connection parameters accepted.

3.6.1.1 How to Maximize Throughput to an iOS Device

Apple devices began supporting BLE technology with the iPhone 4S and iOS 5. iOS devices have additional limitations on top of those mandated in the Bluetooth specification.

The following additional guidelines apply for maximizing iOS throughput:

- When operating in the GAP Central role, the latest iOS devices limit the minimum connection interval of 30 ms (or 11.25 ms when connecting to HID devices). If the peripheral requests a shorter connection interval than this, the iOS device rejects the request.
- iOS devices limit unacknowledged GATT data transfers (write-no-response or notify) to a maximum of four per connection interval, according to widespread observations.
- iOS 5 added support for GAP Peripheral role operation, which includes support for 7.5-ms intervals as required by the Bluetooth specification. However, switching GAP roles may not be suitable depending on other application requirements, and requires a notably different mobile app development approach with its own side effects. In

addition, EZ-Serial for EZ-BLE WICED modules requires Peripheral-mode operation on the module, so the remote Client must use the Central role.

Refer to the Core Bluetooth Programming Guide on the Apple Developer website for official guidelines.

Example 25. Request a connection parameter update to 30-ms interval, no latency, 1-second timeout

Direction	Text Content	Binary Content	Effect
TX→	`/UCP,C=40,I=18,L=0,O=64`	[C0 07 04 03 40 18 00 00 00 64 00 23	Request connection update to 30 ms (24 * 1.25 ms), no slave latency, 1-second supervision timeout.
←RX	`@R,000A,/UCP,0000`	C0 02 04 03 00 00 62	Response indicates success; request sent to remote peer.
←RX	`@E,001D,CU,H=04,I=0010,L=0000,O=0064`	80 07 04 08 40 18 00 00 00 64 00 E8	Event indicates new connection parameters accepted.

3.6.1.2 How to Maximize Throughput to an Android Device

Android devices officially began supporting BLE technology with the Android 4.3 release, though Android 4.4 and onward greatly improved stability and supported functionality.

The following additional guidelines apply for maximizing Android throughput:

- Android 4.4.2 and earlier releases only support a single connection interval of 48.75 ms.
- Android 4.4.3 and later releases support intervals down to 7.5 ms when requested by the remote device, even though the default interval is still 48.75 ms when first establishing the connection.
- Newer Android handsets allow up to six unacknowledged GATT transfers in a single connection interval.

3.6.2 How to Minimize Power Consumption

You can reduce power consumption by making the BLE radio active as infrequently as your application allows. Specific actions described in this section help decrease average consumption, but also decreases the potential throughput. Keep this trade-off in mind to choose the right balance between power consumption and throughput.

If you have not already done so, ensure that the best possible CPU sleep mode for your application is configured as described in Section 3.1.5 (How to Manage Sleep States). This will ensure that the CPU is not taking more power than necessary. If the CPU is fully or partially awake more often than necessary, relative improvements possible using the methods described below may not make a notable difference.

3.6.2.1 How to Minimize Power Consumption While Broadcasting

To reduce power consumption in an advertising state:

- **Maximize the advertisement interval while broadcasting.** The BLE specification allows advertising at any interval between 20 ms and 10240 ms. Increasing the interval means fewer transmissions within a given time period. For example, a device advertising at 500 ms will use roughly 20% of the power required by that same device advertising at 100 ms. Use the gap_set_adv_parameters (SAP, ID=4/23) API command to change the default advertisement interval, or the gap_start_adv (/A, ID=4/8) API command to use a non-default interval at the moment you enter an advertising state.

  Side effects:

  - Scanning devices are less likely to detect each advertisement packet, due to the reduced probability of the scanning device actively receiving on the same channel at the same time as the advertisement transmission occurs.
  - Connections may take longer to establish, because this process begins with the same scanning process and requires detection of a connectable advertisement packet from the target device.
- **Don't use all three advertisement channels.** The BLE spectrum dedicates three channels to advertisement packets, spread across the 2.4-GHz Bluetooth RF spectrum to help ensure reception in busy RF environments. Most BLE devices advertise on all three channels, but you can selectively advertise on only one or two of these channels using the gap_set_adv_parameters (SAP, ID=4/23) or gap_start_adv (/A, ID=4/8) API commands. Advertising on only one channel requires roughly 33% of the power needed when using all three.

Side effects:

- o Scanning devices are less likely to detect advertisement packets for the same reason as above—there are fewer advertisement packets being transmitted, which reduces the probability of actively receiving on the correct channel at the correct time.
- o The advertising device cannot combat RF interference as effectively. If you enable only one advertisement channel, but that portion of the RF spectrum is extremely congested, then a scanning device may not be able to detect advertisement packets at all even if the timing lines up correctly.

- **If connections are not required, use a non-connectable/non-scannable mode.** When a Peripheral device is connectable (accepting new connections) or scannable (accepting scan request packets while advertising), the BLE radio switches to a receiving state for approximately 150 µs after every advertisement packet to listen for a connection request or scan request packet. When using all three advertising channels, this means three complete TX-RX cycles occur repeatedly at the configured advertisement interval. If a Peripheral device needs to broadcast only, you can configure a broadcast-only advertising mode with the gap_set_adv_parameters (SAP, ID=4/23) or gap_start_adv (/A, ID=4/8) API commands. This prevents the radio from switching into a receiving state after each transmission, saving both time and power.

  Side effects:

  - o Any data configured in the scan response packet payload is never transmitted. Most often, this is the friendly device name.

- **Minimize the advertisement and/or scan response data payload length**. Regardless of the configured advertisement interval, the advertisement payload also has a significant effect on the amount of time spent on transmissions. The advertisement payload may be between 0 and 31 bytes, and the BLE RF protocol uses a symbol rate of 1 Mbit/sec, which translates to 8 µs per byte. The fixed encapsulation and overhead data in every advertisement or scan response packet takes roughly 140 µs to transmit, but the payload can add up to 248 µs to this duration. In other words, a 31-byte payload (~390 µsec) requires twice as much transmission time as a 7-byte payload (~195 µs).

  In most cases, the application design requires very specific content in the advertisement payload. However, you should optimize this as much as possible if low power consumption is critical for the application. You can configure custom advertisement data content with the gap_set_adv_data (SAD, ID=4/19) and gap_set_adv_parameters (SAP, ID=4/23) API commands, as described in Section 3.3.3 (How to Customize Advertisement and Scan Response Data).

3.6.2.2 How to Minimize Power Consumption While Connected

To reduce power consumption in a connected state:

- **Maximize the connection interval.** The BLE specification allows a connection interval from 7.5 ms to 4000 ms.
  - o When operating in the **GAP Peripheral** role, the remote Central determines the initial interval; you must request an update after connecting if you need to change it. The remote peer may either accept or reject this request.

- **Use non-zero slave latency.** While this affects only power consumption on the slave/peripheral device during a connection, the slave latency setting can drastically improve power efficiency in many applications. This setting controls how many connection intervals the slave may skip if it has no data to send to the connected master device. Once the allowed number of intervals have occurred, the slave must respond regardless of whether it has any new data to send. The slave *may* respond at any interval.

  With the default “0” slave latency setting, the slave must acknowledge the master’s connection maintenance packets at every interval. In applications requiring infrequent data transfers, this wastes a great deal of power. Increasing the slave latency value to “3” allows the slave to respond every four intervals instead of every interval, for an average power reduction of 75% while connected. Applications such as environmental sensors and human input devices can benefit greatly from non-zero slave latency.

  The slave latency value may not be higher than the maximum number that allows the calculated value for **`[conn_interval * slave_latency]`** to remain below the **`supervision_timeout`** value, because otherwise the connection would time out regularly.

  Side effects:

  - o If the slave has no data to send, the master must wait until the slave latency period passes before it can send or request data to or from the slave. The slave will not be aware of any requests from the master until it enables its radio again. This can result in noticeable delays especially when using long connection intervals. For example, a 500-ms connection interval and slave latency setting of “3” could create a master-to-slave response delay of up to two full seconds. To mitigate this, select a balanced combination of connection interval and slave latency values that provides acceptable master-side delay and slave-side power consumption.

  - Non-zero slave latency interval increases the possibility of a connection timeout in non-optimal RF environments. The master triggers a supervision timeout condition if it does not receive an acknowledgement from the slave before the timeout period elapses. The master resends any connection maintenance packet that is not acknowledged, but if the slave has already switched back to a low-power state between required response intervals, the master's attempted retries may be ignored for too long. To mitigate this, select a longer supervision timeout, shorter connection interval, and/or lower slave latency value to achieve required connection stability in the target environment.

- **Use unacknowledged transfers.** Acknowledged transfers involve more data sent over the air to handle the acknowledgement. This results in higher average consumption. If you do not need application-level data transfer confirmations, use unacknowledged methods instead.
  - For **Server-to-Client** transfers, use the "notify" operation instead of "indicate."

3.7 Device Firmware Update Examples

See Section 2.6.1 (Latest EZ-Serial Firmware Image) for information on where to find the latest EZ-Serial firmware images.

3.7.1 How to Update Firmware Locally Using UART

If you have access to the HCI UART interface, you can use standard WICED Smart SDK software and a UART interface to flash a new firmware image onto the module. Details about how to do this are available on the Cypress website.

Updating firmware via this method always returns to factory default settings and removes any bonding data and custom GATT structure.

4. Application Design Examples

Examples in this section describe the hardware design and platform configuration necessary for some common types of applications. You can use any of these exactly as described for your design, or modify as needed.

4.1 Smart MCU Host with 4-Wire UART and Full GPIO Connections

This design takes allows maximum functionality with an external host microcontroller, including efficient sleep state control and optional CYSPP communication.

4.1.1 Hardware Design

Include the following design elements in your hardware:

1. Module **UART_TX** pin to host UART RX pin
2. Module **UART_RX** pin to host UART TX pin
3. Module **UART_CTS** pin to host UART RTS pin
4. Module **UART_RTS** pin to host UART CTS pin
5. Module **CYSPP**, and **LP_MODE** pins to digital output host GPIOs
6. Module **LP_STATUS**, **DATA_READY**, and **CONNECTION** pins to high-impedance digital input host GPIOs

4.1.2 Module Configuration

Most configuration settings will depend on your communication requirements. However, you may wish to make one or more of the following changes:

- Change Device Name with gap_set_device_name (SDN, ID=4/15)
- Change CYSPP connection key and/or security requirements with p_cyspp_set_parameters (.CYSPPSP, ID=10/3)
- Enable system-wide Deep Sleep with system_set_sleep_parameters (SSLP, ID=2/19)
- Enable flow control and optionally change UART parameters with system_set_uart_parameters (STU, ID=2/25)

4.1.3 Host Configuration

The external host must match EZ-Serial's configured UART communication. The factory default settings are 115200,8/N/1 with no flow control. However, you should enable and use flow control if the host supports it.

Use the host API library examples described in Chapter 5 (Host API Library) to facilitate easy API communication between the host and the module, making sure to properly assert and deassert the module's **LP_MODE** pin as described in Section 2.3.3 (Connecting GPIO Pins).

Enable a falling-edge interrupt on the **DATA_READY** signal to allow the host to know when it needs to parse incoming serial API or CYSPP data. This pin remains asserted (LOW) until no more data exists in the module's serial transmit buffer.

Monitor the **CONNECTION** signal for a simple indicator of BLE connectivity without needing to parse all possible API events from the module. This can be especially helpful when using CYSPP mode.

4.2 Dumb Terminal Host with CYSPP and Simple GPIO State Indication

This design takes advantage of the factory-default EZ-Serial configuration and support for automatic CYSPP connectivity. It is best suited for applications where the external host cannot or does not need to impose any control over the EZ-Serial platform via API commands or events.

4.2.1 Hardware Design

Include the following design elements in your hardware:

1. Module **CYSPP** pin to GND (force CYSPP data mode at all times, no API communication)
2. Module **UART_TX** pin to host UART RX pin
3. Module **UART_RX** pin to host UART TX pin
4. Optional for flow control:
   a. Module **UART_CTS** pin to host UART RTS pin
   b. Module **UART_RTS** pin to host UART CTS pin
5. Optional for connectivity status:
   a. Module **CONNECTION** pin to LED (active LOW)

4.2.2 Module Configuration

The factory default configuration provides most of the behavior required. However, you may wish to make one or more of the following changes:

- Change device name with gap_set_device_name (SDN, ID=4/15).
- Change CYSPP connection key and/or security requirements with p_cyspp_set_parameters (.CYSPPSP, ID=10/3).
- Change system sleep settings with system_set_sleep_parameters (SSLP, ID=2/19).
- Change UART baud or other parameters with system_set_uart_parameters (STU, ID=2/25).

4.2.3 Host Configuration

The external host must match EZ-Serial's configured UART communication. The factory-default settings are 115200,8/N/1 with no flow control. However, you should enable and use flow control if the host supports it.

If the host supports a simple "enable" control line for whether it is safe to send data, use the module's **CONNECTION** pin. This signal is asserted (LOW) only when the CYSPP data pipe is fully established.

4.3 Module-Only Application with Beacon Functionality

This design requires no special external hardware and only minimal initial configuration to define the type of beaconing desired.

4.3.1 Hardware Design

For correct operation, the module only requires power to the supply pins. You may also wish to include test pad or header access to the UART interface and status pins such as **LP_STATUS** or **CONNECTION** during prototyping, because this can greatly simplify debugging if necessary.

4.3.2 Module Configuration

Make the following changes from the factory default configuration:

- Disable CYSPP mode with p_cyspp_set_parameters (.CYSPPSP, ID=10/3).
- Enable system-wide sleep mode with system_set_sleep_parameters (SSLP, ID=2/19).
- Configure non-connectable (broadcast-only) with gap_set_adv_parameters (SAP, ID=4/23).
- Configure custom advertisement data with the appropriate beacon content using gap_set_adv_data (SAD, ID=4/19).

4.3.3 Host Configuration

The simple automatic beacon design does not require any host hardware, and therefore needs no host configuration.

5. Host API Library

The host library implements a protocol parser/generator that communicates with the EZ-Serial firmware using the API protocol. The library is usually written in standard C and wraps all API methods into easy-to-use command functions or response/event callbacks. However, such a host API library is not provided with this EZ-Serial WICED BLE firmware platform. If it is required for system integration, you must create your own host API library based on the online host API library provided for EZ-Serial on BLE modules based on PSoC Creator. Attention should be paid because this EZ-Serial WICED BLE firmware platform has different features set from EZ-Serial on modules based on PSoC Creator. See previous chapters for details.

This section uses the online host API library for EZ-Serial on BLE modules based on PSoC Creator as examples to describe how to use the library as designed, how to port it to other platforms, or how to create your own library if the provided code is not suited for direct use or porting for any reason.

5.1 Host API Library Overview

5.1.1 High Level Architecture

The host library communicates with the EZ-Serial firmware platform, providing the host side of the command/response/event communication mechanism that the module implements. The host must perform the following over the UART interface:

- Read and parse incoming data (may be either response or event packets).
- Validate packets using checksum.
- Trigger application-defined callbacks when incoming packets arrive.
- Generate and send outgoing data (command packets).

The protocol parser and generator on the module side strictly follow these rules:

- Events may be generated by the module at any time.
- Every command received from the host immediately generates a response.
- An event generated (e.g., by a GPIO interrupt) while a command is being processed does not interrupt the command-response packet flow, but is sent out after the response packet is sent.

The parser and generator on the host side must operate under these assumptions.

5.1.2 Host Library Design

Host communication with an EZ-Serial-based module requires only that the incoming module-to-host byte stream is processed correctly, and that the outgoing host-to-module byte stream is properly formatted. To simplify this and provide a convenient layer of abstraction, the host API library provides a simple "parse" function for incoming bytes, and "wrapper" command functions that convert named parameter lists into binary packets ready for transmission.

Other than expecting standard C compiler functionality and little-endian byte order, the library is intentionally platform-agnostic. The source of incoming data does not matter; the internal methods process the data only after it arrives. The destination of outgoing data also does not matter; the internal methods perform only packetization and buffering of data so that it is ready to transmit. This improves portability because UART peripherals are accessed differently on different platforms, and a single library cannot provide support across all (or even very many) platforms if the UART peripheral implementation is built into the library itself.

5.2 Implementing a Project Using the Host API Library

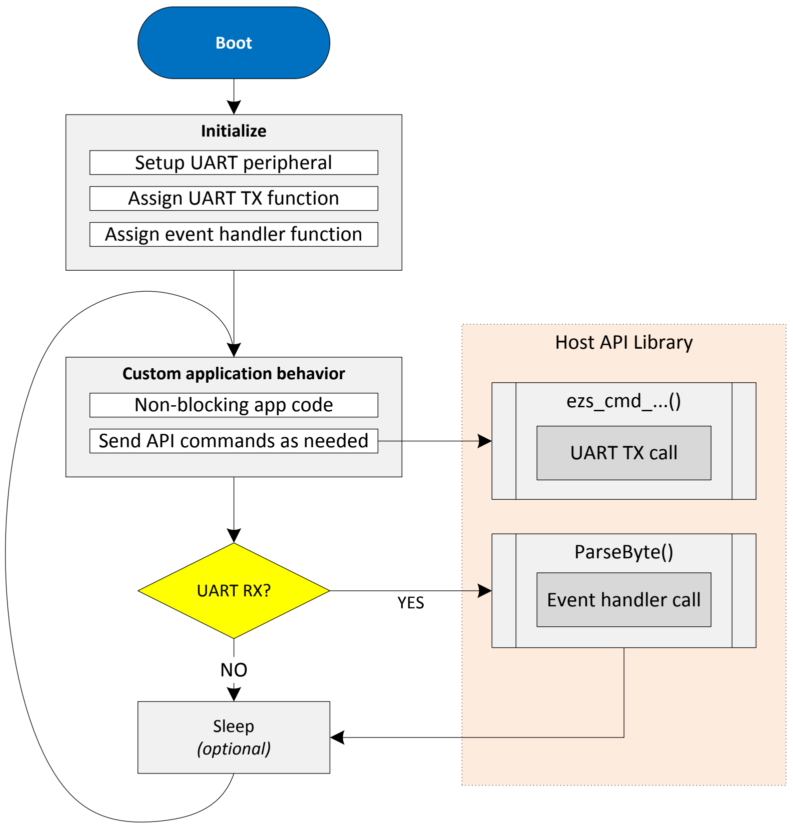
5.2.1 Basic Application Architecture

Any host application that uses the EZ-Serial API library must follow the same basic behavior:

1. Set up UART peripheral for incoming and outgoing data.
2. Assign hardware-specific input/output callback methods.
3. Monitor UART for incoming data, and send to parser.
4. Handle event/response packets sent to callback handler.
5. Call command wrapper functions as needed for application.

This process is shown in the following flowchart:

Figure 5-1. EZ-Serial Host API Library Application Flow



The host API library contains the core parsing and generating functions necessary to translate incoming data into callbacks and command function calls into binary packets.

5.2.2 Exposed API Functions

The generic host API implementation written in C provides the following methods:

Function	Description
`EZSerial_Init`	Initializes parser and callback functions used for event handling, serial output, and serial input
`EZSerial_Parse`	Processes incoming bytes and triggers the event callback function when response or event packet is successfully processed

Function	Description
`EZSerial_FillPacketMetaFromBinary`	Fills binary packet metadata in **ezs_packet_t** structure based on the 4-byte binary packet header content (used internally within **EZSerial_Parse**)
`EZSerial_SendPacket`	Sends binary packet and checksum byte using the host-specific output callback function
`EZSerial_WaitForPacket`	Reads the data using the host-specific input callback function in a blocking or non-blocking way depending on the timeout argument (calls **EZSerial_Parse** as part of its functionality)

The application is responsible for providing implementation functions for three methods, assigned to the function pointers below:

Function	Description
`EZSerial_AppHandler`	Called whenever a valid incoming packet is observed. This is strictly required in all cases. It is a core element of abstracting incoming packets into callback functions.
`EZSerial_HardwareOutput`	Called whenever the API generator needs to send data to the module over UART. This is required if you intend to use the **EZSerial_SendPacket** method, or the **ezs_cmd_...** macros which also use that method. If you are manually sending well-formed binary command packet data directly from your own application, this may be assigned as NULL.
`EZSerial_HardwareInput`	Called whenever the API parser needs to read data from the module over UART. This is required if you intend to use the **EZSerial_WaitForPacket** method, or the **EZS_WAIT_...** or **EZS_CHECK_...** macros which also use that method. If you are manually calling the **EZSerial_Parse** method after reading bytes in over UART, this may be assigned as NULL.

5.2.3 Command Macros

To simplify binary packet creation, the library implements packet builder macros that match the protocol definitions for each command method. For example:

- `ezs_cmd_system_ping()`
- `ezs_cmd_system_reboot()`
- `ezs_cmd_gap_start_adv(mode, type, interval, channels, filter, timeout)`

Commands which fall into the SET/GET categories and may access flash memory for retrieving or storing setting data have two separate command functions for each:

- RAM: `ezs_cmd_gatts_set_parameters(flags)`
- Flash: `ezs_fcmd_gatts_set_parameters(flags)`

To substantially reduce flash usage, these are defined as macros that make use of a single function that accepts variable arguments:

- `ezs_output_result_t ezs_cmd_va(uint16 index, uint8 memory, ...)`

This single method uses the supplied command table index (defined in the library header file as an enumerated list) and the packed binary protocol structure definition to determine the number of arguments needed for any given command and their data types.

This macro-based approach means that it is not possible for to perform type checking at compile time, but it also means that the entire command generator implementation uses a tiny quantity of flash memory (well under one KB as measured on one 8-bit MCU).

5.2.4 Convenience Macros

If the hardware-specific input and output functions are correctly defined, the library also provides macros to further abstract common behavior into simpler code.

Function	Description
`EZS_SEND_AND_WAIT(CMD, TIMEOUT)`	Sends a command and then calls **EZS_WAIT_FOR_RESPONSE**
`EZS_WAIT_FOR_PACKET(TIMEOUT)`	Calls **EZSerial_WaitForPacket** with type set to **any**
`EZS_WAIT_FOR_RESPONSE(TIMEOUT)`	Calls **EZSerial_WaitForPacket** with type set to **response**
`EZS_WAIT_FOR_EVENT(TIMEOUT)`	Calls **EZSerial_WaitForPacket** with type set to **event**
`EZS_CHECK_FOR_PACKET()`	Wrapper for **EZS_WAIT_FOR_PACKET(0)**, a non-blocking attempt to read data

The assignable "return value" (evaluated expression result) for all of these macros is a pointer to an `ezs_packet_t` object. If the process fails at any point for any reason—timeout, command transmission failure, incoming packet in progress, and so on—then the pointer value will be 0 (NULL).

5.3 Porting the Host API Library to Different Platforms

Because the API protocol uses a packet byte stream, the API host library expects matching byte ordering and packet structure mapping in order to avoid any extra processing overhead. The module (and low-level Bluetooth spec) uses little-endian byte ordering, so the host must as well for all multi-byte integer data.

The example application code provided with the library to demonstrate EZ-Serial API usage includes a block of code that can verify proper support and configuration of byte ordering and structure packing. While it is not possible to provide a single, comprehensive cross-platform implementation of a structure packing macro due to variations between compilers, it is possible to definitively test whether the existing code will work properly. This can quickly identify and avoid potential problems that are otherwise very difficult to troubleshoot.

No special C extensions are used; tested compilers are GCC or GCC-compliant and follow the default C89 ruleset because no additional extensions are enabled.

5.4 Using the API Definition JSON File to Create a Custom Library

The JSON schema used for the API definition has the following structure:

- `info` (single dictionary)
  - `date` – Definition revision date
  - `version` – API protocol definition version
- `groups` (list of dictionaries) […
  - `id` – Numeric ID assigned to group
  - `name` – Alpha name assigned to group (e.g., “gap”)
  - `commands` (list of dictionaries) […
    - `id` – Numeric ID assigned to command
    - `name` – Alpha name assigned to command (e.g., “start_adv”)
    - `flashopt` – Boolean flag indicating flash storage for settings
    - `parameters` (list of dictionaries) […
      - `type` – Data type (e.g., “uint16”)
      - `name` – Alpha name assigned to parameter (e.g., “mode”)
      - `textname` – text-mode equivalent (e.g., “M”)
      - `required` – Boolean flag indicating optional or required parameter
      - `format` – Intended data presentation format (e.g., “string” or “hex”)
      - `default` – Fixed default value if optional parameter
    - `returns` (list of dictionaries) […see `parameters`…]
    - `references` (single dictionary)
      - `commands` (dictionary)
      - `events` (dictionary)
  - `events` (list of dictionaries) […see `commands`…]

6. Troubleshooting

EZ-Serial is designed to be as robust and intuitive as possible, but it is always possible for something to go wrong. The instructions below can help narrow down the cause of failure in identify solutions in some cases.

6.1 UART Communication Issues

If you are unable to send or receive data as expected over the UART interface, perform the following steps:

1. Ensure that **VDD** and **GND** pins are properly connected (**VDDR** also requires power).
2. Ensure that **VDD** has a stable supply within the supported range (typically 3.3 V).
3. Ensure that UART data pins are properly connected:
    a. Module **UART_RX** to host TX
    b. Module **UART_TX** to host RX
4. If flow control is enabled or expected, ensure that the UART flow control pins are properly connected:
    a. Module **UART_RTS** to host CTS
    b. Module **UART_CTS** to host RTS
5. Ensure that the **CYSPP** pin is floating or HIGH to avoid entry into CYSPP mode. When CYSPP is active, API communication is disabled, and this can appear as a non-communicative state until a connection is established.
6. Drive or strongly pull the **LP_MODE** pin HIGH to disable normal sleep mode. This is not necessary in most cases, but it can help eliminate potential uncertainty during testing.
7. Reset the module and monitor the **UART_TX** pin during the boot process. If the module boots normally (**CYSPP** pin de-asserted), the system_boot (BOOT, ID=2/1) API event should occur at the configured baud rate. With factory default settings, these values are 115200 baud and text mode. If possible, verify activity using an oscilloscope or a logic analyzer.
8. If attempting to communicate using the API protocol, ensure that your command packet structures are correct per the definitions in Section 7.1 (Protocol Structure and Communication Flow).
9. If you are sending commands in binary mode and the commands in use have any variable-length arguments (data type of **uint8a** or **longuint8a**), ensure that the argument has the correct `<length> [data`$_0$`, data`$_1$`, ..., data`$_N$`]` format. Omitting the length byte will cause the API parser to interpret the packet incorrectly.

6.2 BLE Connection Issues

If you are unable to connect from a remote device, perform the following steps:

1. Ensure that the module is advertising in a connectable state. Start advertising specifically in the “connectable, undirected” mode using the gap_start_adv (/A, ID=4/8) API command, and watch for the expected gap_adv_state_changed (ASC, ID=4/2) API event indicating that the state actually changed to “active.”
2. Ensure you have set properly formed custom advertising data with gap_set_adv_data (SAD, ID=4/19) if you have disabled automatic advertising packet management with gap_set_adv_parameters (SAP, ID=4/23). Advertisement packets without a standard “Flags” field (usually `[ 02 01 06 ]`) do not appear in a generic scan. See Section 3.3.3 (How to Customize Advertisement and Scan Response Data).

6.3 GPIO Signal Issues

If you do not observe the expected behavior for GPIO input and/or output signals, perform the following steps:

1. Ensure that the pins that you have connected are correct based on your chosen module. See Section 8.1 (GPIO Pin Map for Supported Modules) for per-device pin map details.
2. If a special-function output pin is not sufficiently driving a connected external device's input logic, ensure that the external device is not also attempting to drive or strongly pull the pin in the opposite direction at the same time.

7. API Protocol Reference

This section describes the API protocol that EZ-Serial uses. This protocol allows an external host to control the module, in addition to any GPIO signals involved in the design. The protocol follows a strict set of rules to make deterministic host-side behavior possible.

The material in this revision of the User Guide describes version 1.3 of the API protocol.

7.1 Protocol Structure and Communication Flow

7.1.1 API Protocol Formats

EZ-Serial implements a unified set of functionality that can be accessed using binary API communication. Cypress text-based protocol APIs are also provided for ease of reading, as well as to generate binary API commands via the provided Python script.

7.1.1.1 Text Format Overview

Cypress text-based format is provided for ease of reading throughout this document. When communicating with an EZ-Serial WICED module, the binary format must be used. To convert text-based commands to binary format, a Python script has been provided.

7.1.1.2 Binary Format Overview

The binary protocol uses a fixed packet structure for every transaction in either direction. This fixed structure comprises a 4-byte header, followed by an optional payload of up to 2047 bytes (length specifier field is 11 bits wide).

No currently defined binary packet contains more than 520 payload bytes at this time, and very few contain more than 48. The API reference material below lists every fixed or minimum/maximum length value for all commands, responses, and events within the protocol.

The payload carries information related to the command, response, or event. If present, this payload always comes immediately after the header. All data in the payload is contained within one or more of the datatypes specified in Section 7.1.2 (API Protocol Data Types).

To simplify the implementation of parsers and generators both inside the firmware and on external host microcontrollers, any packet may have a maximum of one variable-length data member (byte array or string), and if present, it must be the last element in the payload.

7.1.2 API Protocol Data Types

The data types implemented for individual parameters/arguments in the API protocol are described below, including representative text and binary examples.

In both text and binary modes, all negative numbers are represented in two’s complement form. In this form, the most significant bit is the sign bit, which indicates a negative number if set. The remaining bits count upward from the bottom of the selected (positive or negative) range. For example, the value 0x80 is the bottom of the “int8” range, -128.

Table 7-1. API Protocol Data Types

Type	Bytes	Description	Example
`uint8`	1	Unsigned 8-bit integer. Range is 0 to 255.	Text Mode: - “10” = 0x10, decimal 16 - “9A” = 0x9A, decimal 154 Binary Mode: - [10] = 0x10, decimal 16 - [9A] = 0x9A, decimal 154
`int8`	1	Signed 8-bit integer. Range is -128 to 127.	Text Mode: - “10” = 0x10, decimal 16 - “9A” = 0x9A, decimal -102 Binary Mode: - [10] = 0x10, decimal 16 - [9A] = 0x9A, decimal -102
`uint16`	2	Unsigned 16-bit integer. Range is 0 to 65,535.	Text Mode: - “1234” = 0x1234, decimal 4,660 - “9ABC” = 0x9ABC, decimal 39,612 Binary Mode: *(little-endian)* - [34 12] = 0x1234, decimal 4,660 - [BC 9A] = 0x9ABC, decimal 39,612
`int16`	2	Signed 16-bit integer. Range is -32,768 to 32,767.	Text Mode: - “1234” = 0x1234, decimal 4,660 - “9ABC” = 0x9ABC, decimal -25,924 Binary Mode: *(little-endian)* - [34 12] = 0x10, decimal 4,660 - [BC 9A] = 0x9ABC, decimal -25,924
`uint32`	4	Unsigned 32-bit integer. Range is 0 to 4,294,967,295.	Text Mode: - “12345678” = 0x12345678 decimal 305,419,896 - “9ABCDEF0” = 0x9ABCDEF0, decimal 2,596,069,104 Binary Mode: *(little-endian)* - [78 56 34 12] = 0x12345678 decimal 305,419,896 - [F0 DE BC 9A] = 0x9ABCDEF0 decimal 2,596,069,104
`int32`	4	Signed 32-bit integer. Range is -2,147,438,648 to 2,147,483,647.	Text Mode: - “12345678” = 0x12345678 decimal 305,419,896 - “9ABCDEF0” = 0x9ABCDEF0, decimal -1,698,898,192 Binary Mode: *(little-endian)* - [78 56 34 12] = 0x12345678 decimal 305,419,896 - [F0 DE BC 9A] = 0x9ABCDEF0 decimal -1,698,898,192
`macaddr`	6	48-bit MAC address.	Text Mode: - “112233AABBCC” = 11:22:33:AA:BB:CC Binary Mode: *(little-endian)* - [CC BB AA 33 22 11] = 11:22:33:AA:BB:CC
`uint8a`	1+	Array of `uint8` bytes, with prefixed one-byte length value. Supported length is 0-255 bytes.	Text Mode: *(length omitted, detected automatically)* - “41424344” = Length 4, Data [41 42 43 44] - “1122334455” = Length 5, Data [11 22 33 44 55] Binary Mode: - [04 41 42 43 44] = Ln. 4, [41 42 43 44] - [05 11 22 33 44 55] = Ln. 5, [11 22 33 44 55]

Type	Bytes	Description	Example
`longuint8a`	2+	Array of `uint8` bytes, with prefixed two-byte length value. Supported length is 0-65535 bytes.	Text Mode: *(length omitted, detected automatically)* - "41424344" = Length 4, Data [41 42 43 44] - "1122334455" = Length 5, Data [11 22 33 44 55] Binary Mode: - [04 00 41 42 43 44] = Length 4, Data [41 42 43 44] - [05 00 11 22 33 44 55] = Length 5, Data [11 22 33 44 55] Note the 16-bit **length** prefix in binary mode is transmitted in little-endian byte order, so the value 0x0005 is sent as [05 00].
`string`	1+	String of `uint8` bytes, with prefixed one-byte length value. Length is 0-255 bytes.	These two datatypes are represented in binary exactly the same way as uint8a and longuint8a data, but in text mode they are entered and displayed exactly as-is, with the assumption that they contain printable ASCII characters. An example of a string value entered and displayed in this way is the Device Name value.
`longstring`	2+	String of `uint8` bytes, with prefixed two-byte length value. Length is 0-65535 bytes.	

7.1.3 Binary Format Details

7.1.3.1 Byte Ordering and Structure Packing

The protocol implements a collection of common data types representing signed and unsigned integers, arrays of binary bytes, arrays of printable characters, and certain technology-specific data (6-byte MAC address).

In text mode, all data except **string/longstring** values are represented as ASCII hexadecimal characters, without a leading "`0x`" or other prefix. For example, the decimal value 154 is shown or entered as "`9A`". Leading zeros may be omitted. Also, in text mode, all multi-byte integer and MAC address data shall be entered in big-endian byte order. For example, the value 0x1234 is entered or displayed as "`1234`". The MAC address 11:22:33:AA:BB:CC is entered or displayed as "`112233AABBCC`".

In binary mode, all multi-byte integers and MAC address data must be transmitted serially in little-endian byte order. For example, the value 0x1234 is two bytes transmitted as `[ 34 12 ]`, and the MAC address 11:22:33:AA:BB:CC is six bytes transmitted as `[ CC BB AA 33 22 11 ]`.

The Bluetooth Low Energy specification mandates little-endian byte order internally, so data from the stack is naturally presented to the application layer in this byte order. Further, many common embedded processors use little-endian data storage, including the Arm® Cortex®-M0 in Cypress EZ-BLE modules. As a result, host MCU firmware can read in a serial byte stream into a contiguous SRAM buffer, and define a structure like the following:

```
typedef struct {
    uint16 app;
    uint32 stack;
    uint16 protocol;
    uint8 hardware;
    uint8 cause;
    macaddr address;
} ezs_evt_system_boot_t;
```

The host MCU application can directly map this structure onto the packet buffer in memory with no additional byte-swap operations. Accessing any one of the structure members gives correct access to the data in the packet. This arrangement allows for minimal flash usage and CPU execution time.

7.1.3.2 *Binary Packet Header*

The binary packet 4-byte header structure is described in the table below:

Table 7-2. Binary Packet Header Structure

Byte	Field(s)	Description
0	[7:6] - Type [5:4] - Memory [2:0] - Length MSB	**Type:** The “Type” field is a 2-bit value (MSB aligned) indicating whether the packet is a command, response, or event. Options are as follows: - 00: RESERVED - 01: RESERVED - 10: Event (module-to-host) - 11: Response (module-to-host), and Command (host-to-module) Protocol methods follow this convention when the “Type” value is aligned properly: - Commands sent to the module begin with 0xC0 - Responses sent to the host begin with 0xC0 - Events sent to the host begin with 0x80 **Memory:** The “Memory” field is a 2-bit value (MSB aligned) indicating whether a command sent accesses the runtime value stored in RAM, or the boot value stored in flash. This field is ignored for commands which do not read or write configuration data stored in either flash or RAM. Options are as follows: - 00: Runtime (RAM) - 01: Boot (Flash) - 10: RESERVED - 11: RESERVED The values stored in RAM and flash may be the same, if the user has not modified the runtime value separately from the boot value since the last power-on or reset. **Length MSB:** The length MSB field contains the upper three bits of the payload length value (11 bits total). See below for length detail. The “Type”, “Memory”, and “Length MSB” bitfields are positioned within Byte 0 as follows: `0b` **`TTMM`** `0`**`LLL`** The remaining bit in the middle is currently reserved and should always be set to zero.
1	Length LSB	This value indicates the number of bytes in the payload. It may be 0 to indicate no payload, or any value up to the 11-bit maximum of 2047 (combining the LSB and MSB fields together). Typically, packets fit easily within a 64-byte buffer. However, a few packets such as local GATT reads and writes may potentially be much longer than this. Protocol methods which may require or generate atypically long packets are documented specifically.
2	Group ID	All protocol methods are organized into logically separate groups, such as GAP, GATT Server, L2CAP, CYSPP, etc. This byte represents the group ID, between 0 and 255. A single group ID applies to all commands, responses, and events within that group.
3	Method ID	Within each group and packet type, every protocol method has a unique ID between 0 and 255. Command/response pairs always have matching IDs. Command/response pairs and events are separate collections and may have overlapping method IDs, each in a set starting from 0.

7.2 API Commands and Responses

All commands and responses implemented in the API protocol are described in detail below. API events are documented separately in Section 7.3). A master list of all possible error codes resulting from commands can be found in Section 7.4 (Error Codes).

Important things to note about the reference material in the following sections:

- The 16-bit "**result**" code is common to every response, and always occupies the same position in the packet (immediately after the binary header or text name). For simplicity, this "**result**" field is omitted from each list of response parameters in the tables below.
- The "Text" column in each "Command Arguments" table contains the text code for each argument. Required arguments have a red asterisk (*) next to their text codes. Optional arguments in text mode will not have a red asterisk.
- All command arguments are **required** in binary mode, due to the fact that binary parsing depends on predictable argument position and byte width for proper data identification and unpacking.
- The "Command-Specific Result Codes" list appearing for some commands do not include some errors that may result from command entry or protocol format mistakes. These common errors include:

```
o 0x0203 – EZS_ERR_PROTOCOL_UNRECOGNIZED_COMMAND
o 0x0206 – EZS_ERR_PROTOCOL_SYNTAX_ERROR
o 0x0207 – EZS_ERR_PROTOCOL_COMMAND_TIMEOUT
o 0x0209 – EZS_ERR_PROTOCOL_INVALID_CHECKSUM
o 0x020A – EZS_ERR_PROTOCOL_INVALID_COMMAND_LENGTH
o 0x020B – EZS_ERR_PROTOCOL_INVALID_PARAMETER_COUNT
o 0x020C – EZS_ERR_PROTOCOL_INVALID_PARAMETER_VALUE
o 0x020D – EZS_ERR_PROTOCOL_MISSING_REQUIRED_ARGUMENT
o 0x020E – EZS_ERR_PROTOCOL_INVALID_HEXADECIMAL_DATA
o 0x020F – EZS_ERR_PROTOCOL_INVALID_ESCAPE_SEQUENCE
o 0x0210 – EZS_ERR_PROTOCOL_INVALID_MACRO_SEQUENCE
```

Refer to Section 7.4 (Error Codes) for details on these and other error codes.

Commands and responses are broken down into the following groups:

- System Group (ID=2)
- GAP Group (ID=4)
- GATT Server Group (ID=5)
- SMP Group (ID=7)
- GPIO Group (ID=9)
- CYSPP Group (ID=10)

7.2.1 System Group (ID=2)

System methods relate to the core device and describe functionality such as boot status, setting or obtaining device address info, and resetting to an initial state.

Commands within this group are listed below:

- system_ping (/PING, ID=2/1)
- system_reboot (/RBT, ID=2/2)
- system_dump (/DUMP, ID=2/3)
- system_store_config (/SCFG, ID=2/4)
- system_factory_reset (/RFAC, ID=2/5)
- system_query_firmware_version (/QFV, ID=2/6)
- system_query_random_number (/QRND, ID=2/8)
- system_set_bluetooth_address (SBA, ID=2/13)
- system_get_bluetooth_address (GBA, ID=2/14)
- system_get_sleep_parameters (GSLP, ID=2/20)

- system_set_tx_power (STXP, ID=2/21)
- system_get_tx_power (GTXP, ID=2/22)
- system_set_transport (ST, ID=2/23)
- system_get_transport (GT, ID=2/24)
- system_set_uart_parameters (STU, ID=2/25)
- system_get_uart_parameters (GTU, ID=2/26)

Events within this group are documented in Section 7.3.1, System Group (ID=2).

7.2.1.1 system_ping (/PING, ID=2/1)

Test API communication.

Pinging the module verifies that the host and the module can communicate properly in API mode. The module should immediately generate a well-formed response to this command if communication is working correctly. Host-side initialization routines often begin with this step.

Runtime values returned in the response to this command are calculated based on the built-in 32768-Hz watch clock oscillator (WCO) that is used to manage low-power operation of the BLE stack. No external hardware is required for this functionality.

Note: Pinging the module does not serve any purpose other than to verify proper communication, or to obtain runtime since reset. You do not need to ping at regular intervals to keep a connection alive or prevent the module from entering low-power states. The platform automatically maintains BLE connections unless commanded otherwise. See Section 3.1.5 (How to Manage Sleep States) for details of sleep behavior.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`01`	None.
RSP	`C0`	`08`	`02`	`01`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/PING`	`0x000B`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint32`	`runtime`	`R`	Number of seconds since boot
`uint16`	`fraction`	`F`	Fraction of a second (units are 1/32768)

7.2.1.2 system_reboot (/RBT, ID=2/2)

Reboot module.

A module reboot takes effect immediately. Any configuration settings not stored in flash revert to their boot-level values, and any active connections are terminated without clean closure (remote peer will detect a supervision timeout). See Section 2.5.2 (Saving Runtime Settings in Flash) for details about how to store settings in flash to make them persist across reboots and power cycles.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`02`	None.
RSP	`C0`	`02`	`02`	`02`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/RBT`	`0x000A`	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Commands:

- system_store_config (/SCFG, ID=2/4) – Use to store all configuration items in flash before rebooting, if desired

Related Events:

- system_boot (BOOT, ID=2/1) – Occurs once the reboot process completes

7.2.1.3 system_dump (/DUMP, ID=2/3)

Dump current device configuration or state information.

Performing a system dump generates a sequence of system_dump_blob (DBLOB, ID=2/5) API events, each containing up to 16 bytes, until all data transmission is complete. You can provide this information for troubleshooting if requested by Cypress support staff.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`02`	`03`	None.
RSP	`C0`	`04`	`02`	`03`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/DUMP`	`0x0012`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`type`	`T`	Type of information to dump: • 0 = Runtime configuration data (default) • 1 = Boot-level configuration data • 2 = Factory-level configuration data • 3 = System state data

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`length`	`L`	Number of bytes to be dumped: • Configuration data is 674 bytes (0x02A2) • State data is 1,955 bytes (0x07A3)

Related Commands:

- system_store_config (/SCFG, ID=2/4)

Related Events:

- system_dump_blob (DBLOB, ID=2/5)

7.2.1.4 system_store_config (/SCFG, ID=2/4)

Store all configuration settings into flash.

This command applies all runtime settings into the boot-level configuration area stored in non-volatile flash. See Section 2.5 (Configuration Settings, Storage, and Protection) for details about different configuration areas.

> **WARNING:** This command briefly halts CPU execution, and may cause connectivity loss for any open connections if this occurs during a precise moment when low-level BLE interrupts require processing. If possible, use this command only while not connected to avoid this potential issue.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`04`	None.
RSP	`C0`	`02`	`02`	`04`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/SCFG`	`0x000B`	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Commands:

- system_factory_reset (/RFAC, ID=2/5)

7.2.1.5 system_factory_reset (/RFAC, ID=2/5)

Reset all settings to factory defaults and reboot.

This command reverts all configuration settings back to the values stored in the factory default area. After applying these default values, the system reboots immediately.

> **WARNING:** If you have configured custom serial communication settings using the system_set_transport (ST, ID=2/23) API command, using this command will undo these changes and may prevent a working communication until you reconfigure your host device to the factory default transport settings. See Section 2.2 (Factory Default Behavior) for details about these settings.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`05`	None.
RSP	`C0`	`02`	`02`	`05`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/RFAC`	`0x000B`	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Events:

- system_factory_reset_complete (RFAC, ID=2/3) – Occurs after the settings are reset
- system_boot (BOOT, ID=2/1) – Occurs after the system reboots

7.2.1.6 *system_query_firmware_version (/QFV, ID=2/6)*

Query EZ-Serial firmware version info.

This command provides the same version details that the system_boot (BOOT, ID=2/1) event contains.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`06`	None.
RSP	`C0`	`0D`	`02`	`06`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/QFV`	`0x002C`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint32`	`app`	`E`	Application version number (e.g.: 0x0101021F = 1.1.2 build 31)
`uint32`	`stack`	`S`	BLE stack version number (e.g.: 0x02020355 = 2.2.3 build 85)
`uint16`	`protocol`	`P`	API protocol version number (e.g.: 0x0103 = 1.3)
`uint8`	`hardware`	`H`	Hardware identifier: • 0x01 = CYBLE-01201X-X0 • 0x02 = CYBLE-014008-00 • 0x03 = CYBLE-022001-00 • 0x04 = CYBLE-2X20XX-X1 • 0x05 = CYBLE-2120XX-X0 • 0x06 = CYBLE-212020-01 • 0x07 = CYBLE-214009-00 • 0x08 = CYBLE-214015-01 • 0x09 = CYBLE-222005-00 • 0x0A = CYBLE-222014-01 • 0x0B = CYBLE-224110-00 • 0x0C = CYBLE-224116-01 • 0xB1 = CYBLE-013025-00

Related Events:

- system_boot (BOOT, ID=2/1)

7.2.1.7 *system_query_random_number (/QRND, ID=2/8)*

Query random number generator for 8-byte pseudo-random sequence.

This command provides simple access to the random number generator in the EZ-BLE module's chipset. The query always provides exactly eight bytes of random data.

> **NOTE:** This pseudo-random generation mechanism is FIPS PUB 140-2 compliant.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`08`	None.
RSP	`C0`	`0B`	`02`	`08`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/QRND`	`0x001E`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8a`	`data`	`D`	Random 8-byte sequence (1 length byte equal to 0x08, followed by 8 data bytes) **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

7.2.1.8 system_set_bluetooth_address (SBA, ID=2/13)

Configure a new public Bluetooth address.

This address is visible to remote scanning or connected devices, as long as the module is not operating with privacy enabled. EZ-Serial uses a fixed public address by default, which is generated dynamically based on unique properties of the chipset inside each module (including wafer/die data). Normally, you do not need to change the Bluetooth address using this command.

NOTE: When privacy is enabled, remote peer devices see a random address instead of the fixed address. Central or Peripheral privacy is not the same as encryption. See related commands and example usage for detail.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`06`	`02`	`0D`	None.
RSP	`C0`	`02`	`02`	`0D`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SBA`	`0x0009`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`Macaddr`	`address`	`A`	New public Bluetooth address. Set all six 0x00 bytes to revert to factory-provided address.

Response Parameters:

None.

Related Commands:

- system_get_bluetooth_address (GBA, ID=2/14)
- smp_set_privacy_mode (SPRV, ID=7/9)

7.2.1.9 *system_get_bluetooth_address (GBA, ID=2/14)*

Obtain the current public Bluetooth address.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	02	0E	None.
RSP	C0	08	02	0E	None.

Text Info:

Text Name	Response Length	Category	Notes
GBA	0x0018	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
macaddr	address	A	Current public Bluetooth address

Related Commands:

- system_set_bluetooth_address (SBA, ID=2/13)
- smp_set_privacy_mode (SPRV, ID=7/9)

7.2.1.10 *system_set_sleep_parameters (SSLP, ID=2/19)*

Configure new system-wide sleep settings.

In order to maintain the required activity (including BLE communication, PWM output, and UART output), EZ-Serial will not automatically enter Deep Sleep mode even if it is configured as normal sleep mode.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	01	02	13	None.
RSP	C0	02	02	13	None.

Text Info:

Text Name	Response Length	Category	Notes
SSLP	0x000A	SET	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	level	L	New maximum system-wide sleep level: • 0 = Sleep disabled (factory default) • 1 = Normal sleep when possible

Response Parameters:

None.

Related Commands:

- system_get_sleep_parameters (GSLP, ID=2/20)
- gpio_set_pwm_mode (SPWM, ID=9/11) – Configure PWM output
- p_cyspp_set_parameters (.CYSPPSP, ID=10/3) – Configure new CYSPP parameters, including CYSPP data mode sleep level

Example Usage:

- Section 3.1.5.1 (Configuring the System-Wide Sleep Level)

7.2.1.11 system_get_sleep_parameters (GSLP, ID=2/20)

Obtain the current system-wide sleep settings.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`14`	None.
RSP	`C0`	`03`	`02`	`14`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GSLP`	`0x000F`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`level`	`L`	Current maximum system-wide sleep level: • 0 = Sleep disabled (factory default) • 1 = Normal sleep when possible

Related Commands:

- system_set_sleep_parameters (SSLP, ID=2/19)

7.2.1.12 system_set_tx_power (STXP, ID=2/21)

Configure new transmit power for all outgoing radio communications.

This power setting affects all transmissions, including advertising, scan requests and connection requests, and all packets sent during an active connection. Changes take effect as soon as the next transmitted packet begins.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`02`	`15`	None.
RSP	`C0`	`02`	`02`	`15`	None.

Text Info:

Text Name	Response Length	Category	Notes
`STXP`	`0x000A`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`power`	`P`	Available power value can be set, the value must be in the range of 1 and 8. The default set value is 7. See 0 for details on the TX output power map.

Response Parameters:

None.

Related Commands:

- system_get_tx_power (GTXP, ID=2/22)

7.2.1.13 *system_get_tx_power (GTXP, ID=2/22)*

Obtain current transmit power for all outgoing radio communications.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	02	16	None.
RSP	C0	03	02	16	None.

Text Info:

Text Name	Response Length	Category	Notes
GTXP	0x000F	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
uint8	power	P	Current active power level value, it should be in the range of 1 and 8. See 0 for details on the TX output power map.

Related Commands:

- system_get_tx_power (GTXP, ID=2/22)

7.2.1.14 *system_set_transport (ST, ID=2/23)*

Configure new host communication interface.

This command configures the interface used for wired external host communication. If a change is successful, EZ-Serial will send the response packet in the *original* configuration, and then switch to the new transport interface.

NOTE: The current EZ-Serial release supports only the UART transport interface. No other options are available.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	01	02	17	None.
RSP	C0	02	02	17	None.

Text Info:

Text Name	Response Length	Category	Notes
ST	0x0008	SET	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	interface	I	New host transport interface: • 1 = UART (factory default)

Response Parameters:

None.

Related Commands:

- system_get_transport (GT, ID=2/24)
- system_set_uart_parameters (STU, ID=2/25)

7.2.1.15 system_get_transport (GT, ID=2/24)

Obtain the current host transport setting.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	02	18	None.
RSP	C0	03	02	18	None.

Text Info:

Text Name	Response Length	Category	Notes
GT	0x000D	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
uint8	interface	I	Current host transport interface: • 1 = UART (factory default)

Related Commands:

- system_set_transport (ST, ID=2/23)
- system_get_uart_parameters (GTU, ID=2/26)

7.2.1.16 system_set_uart_parameters (STU, ID=2/25)

Configure new UART settings for host communication.

This command configures the UART peripheral behavior used for wired external host communication when the host transport interface is set to “UART” with the system_set_transport (ST, ID=2/23) API command. If a change is successful, EZ-Serial will send the response packet using the *original* configuration, and then apply the new UART settings.

> **NOTE:** This command affects **protected settings**, which means you cannot immediately apply changes to flash. In order to store new settings in non-volatile memory, you must send the command once without the flash storage bit/flag, and then re-send the same command again with the flash storage bit/flag set. This prevents accidental permanent communication lockout resulting from flash-stored settings that the connected host cannot use. For detail, refer to Section 2.5.3 (Protected Configuration Settings).

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	0A	02	19	None.
RSP	C0	02	02	19	None.

Text Info:

Text Name	Response Length	Category	Notes
STU	0x0009	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint32`	`baud`	`B`	UART baud rate: • Minimum = 300 baud (0x12C) • Factory default = 115,200 baud (0x1C200) • Maximum = 2,000,000 baud (0x1E8480)
`uint8`	`autobaud`	`A`	Auto-detect UART baud rate at boot: • 0 = Disabled (factory default, must always be disabled in current version)
`uint8`	`autocorrect`	`C`	Auto-correct UART clock to compensate for wide temperature variation: • 0 = Disabled (factory default, must always be disabled in current version)
`uint8`	`flow`	`F`	UART RTS/CTS flow control: • 0 = Disabled (factory default) • 1 = Enabled
`uint8`	`databits`	`D`	UART data bits: • 8 = 8 data bits (factory default)
`uint8`	`parity`	`P`	UART parity: • 0 = Disabled (factory default) • 1 = Odd parity • 2 = Even parity
`uint8`	`stopbits`	`S`	UART stop bits: • 1 = 1 stop bit (factory default) • 2 = 2 stop bits •

Response Parameters:

None.

Related Commands:

- system_set_transport (ST, ID=2/23)
- system_get_uart_parameters (GTU, ID=2/26)

Example Usage:

- Section 3.1.2 (How to Change the Serial Communication Parameters)

7.2.1.17 system_get_uart_parameters (GTU, ID=2/26)

Obtain the current UART settings for host communication.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`02`	`1A`	None.
RSP	`C0`	`0C`	`02`	`1A`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GTU`	`0x0032`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint32`	`baud`	`B`	UART baud rate: • Minimum = 300 baud (0x12C) • Factory default = 115,200 baud (0x1C200) • Maximum = 2,000,000 baud (0x1E8480)
`uint8`	`autobaud`	`A`	Auto-detect UART baud rate at boot: • 0 = Disabled (factory default, **must always be disabled in current version**)
`uint8`	`autocorrect`	`C`	Auto-correct UART clock to compensate for wide temperature variation: • 0 = Disabled (factory default, **must always be disabled in current version**)
`uint8`	`flow`	`F`	UART RTS/CTS flow control: • 0 = Disabled (factory default) • 1 = Enabled
`uint8`	`databits`	`D`	UART data bits: • 8 = 8 data bits (factory default)
`uint8`	`parity`	`P`	UART parity: • 0 = Disabled (factory default) • 1 = Odd parity • 2 = Even parity
`uint8`	`stopbits`	`S`	UART stop bits: • 1 = 1 stop bit (factory default) • 2 = 2 stop bits

Related Commands:

- system_get_transport (GT, ID=2/24)
- system_set_uart_parameters (STU, ID=2/25)

7.2.2 GAP Group (ID=4)

GAP methods relate to the Generic Access Protocol layer of the Bluetooth stack, which includes management of scanning and advertising, connection establishment, and connection maintenance.

Commands within the GAP group are listed below:

- gap_update_conn_parameters (/UCP, ID=4/3)
- gap_disconnect (/DIS, ID=4/5)
- gap_add_whitelist_entry (/WLA, ID=4/6)
- gap_delete_whitelist_entry (/WLD, ID=4/7)
- gap_start_adv (/A, ID=4/8)
- gap_stop_adv (/AX, ID=4/9)
- gap_query_peer_address (/QPA, ID=4/12)
- gap_query_rssi (/QSS, ID=4/13)
- gap_query_whitelist (/QWL, ID=4/14)
- gap_set_device_name (SDN, ID=4/15)
- gap_get_device_name (GDN, ID=4/16)
- gap_set_device_appearance (SDA, ID=4/17)
- gap_get_device_appearance (GDA, ID=4/18)
- gap_set_adv_data (SAD, ID=4/19)
- gap_get_adv_data (GAD, ID=4/20)
- gap_set_sr_data (SSRD, ID=4/21)
- gap_get_sr_data (GSRD, ID=4/22)
- gap_set_adv_parameters (SAP, ID=4/23)

- gap_get_adv_parameters (GAP, ID=4/24)

Events within this group are documented in Section 7.3.2, GAP Group (ID=4).

7.2.2.1 gap_update_conn_parameters (/UCP, ID=4/3)

Request a connection parameter update for an active connection.

Use this command to change the connection interval, slave latency, and supervision timeout for an active connection. If the parameter update is successful, EZ-Serial will generate the gap_connection_updated (CU, ID=4/8) API event after applying new parameters. This will only occur if one or more of the parameters changes from its previous value.

The behavior following this command depends on the link-layer role (master or slave) of the device which initiated the request. The master device has final authority over connection parameters. The EZ-BLE WICED version of EZ-Serial supports operation only in the slave role.

If used while in the **slave** role (connection from peer initiated remotely):

- New connection parameters must be confirmed by the master
- Local device will generate the gap_connection_updated (CU, ID=4/8) event if master accepts parameters

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`07`	`04`	`03`	None.
RSP	`C0`	`02`	`04`	`03`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/UCP`	`0x000A`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Handle of connection to update **(Ignored in current release due to internal BLE stack functionality, set to 0)**
`uint16`	`interval`	`I*`	Connection interval
`uint16`	`slave_latency`	`L*`	Slave latency
`uint16`	`supervision_timeout`	`O*`	Supervision timeout

Response Parameters:

None.

Related Commands:

None.

Related Events:

- gap_connection_updated (CU, ID=4/8)

7.2.2.2 gap_disconnect (/DIS, ID=4/5)

Close an open connection to a remote device.

Use this command to cleanly close an established connection with a remote peer device. The connection must first have been fully opened, indicated by the gap_connected (C, ID=4/5) API event.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`04`	`05`	None.
RSP	`C0`	`02`	`04`	`05`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/DIS`	`0x000A`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Handle of connection to disconnect **(Ignored in current release due to internal BLE stack functionality, set to 0)**

Response Parameters:

None.

Related Commands:

None.

Related Events:

- gap_disconnected (DIS, ID=4/6)

7.2.2.3 gap_add_whitelist_entry (/WLA, ID=4/6)

Add a new Bluetooth address to the whitelist.

The whitelist is an optional filter for determining which remote peers are allowed to connect, or which the local module may try to connect to. When whitelist filtering is active, any devices which are not on the whitelist are not be allowed to connect with the module. You can control whitelist filter usage during advertising, scanning, or outgoing connect attempts.

> **NOTE:** You can only use this command while disconnected. Changes to the whitelist are not allowed during a connection.

Each whitelist entry is made up of two parts: the peer's Bluetooth address, and the type of address (public or private). You must specify the correct address type for each peer based on the type of address it is using. This information is available in scan results and connection details.

> **NOTE:** The BLE stack in EZ-Serial automatically mirrors the bonded device list into the whitelist. This behavior accommodates the most common use case for the whitelist, and you may not need any manual additions or removals from the whitelist.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`07`	`04`	`06`	None.
RSP	`C0`	`03`	`04`	`06`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/WLA`	`0x000F`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`macaddr`	`address`	`A`*	Bluetooth address
`uint8`	`type`	`T`	Address type: • 0 = Public (default) • 1 = Random/private

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`count`	`C`	Updated whitelist entry count

Command-Specific Result Codes:

None.

Related Commands:

- gap_delete_whitelist_entry (/WLD, ID=4/7)
- gap_query_peer_address (/QPA, ID=4/12)
- gap_set_adv_parameters (SAP, ID=4/23) – Configure whitelist filter for advertising

Related Events:

None.

7.2.2.4 gap_delete_whitelist_entry (/WLD, ID=4/7)

Remove a Bluetooth address from the whitelist.

Use this command to remove a specific device from the whitelist if it is already present. For details on whitelist behavior, refer to documentation for the gap_add_whitelist_entry (/WLA, ID=4/6) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`07`	`04`	`07`	None.
RSP	`C0`	`03`	`04`	`07`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/WLD`	`0x000F`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`macaddr`	`address`	`A`	Bluetooth address
`uint8`	`type`	`T`	Address type: • 0 = Public (default) • 1 = Random/private

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`count`	`C`	Updated whitelist entry count

Related Commands:

- gap_add_whitelist_entry (/WLA, ID=4/6)

7.2.2.5 gap_start_adv (/A, ID=4/8)

Start advertising.

This command begins advertising using the specified parameters, or using the pre-configured default advertising parameters if in text mode and some arguments are omitted. EZ-Serial must not already be advertising in order for this command to succeed. However, it is possible to advertise and scan simultaneously.

EZ-Serial will generate the gap_adv_state_changed (ASC, ID=4/2) API event when the advertising state changes.

NOTE: You can start advertising while connected only if you specify "0" (broadcast-only) for the **`mode`** argument. The BLE stack does not support being connected and connectable at the same time.

NOTE: When using the "scannable, undirected" type or "non-connectable, undirected" setting for the **`type`** argument, the advertisement interval must be **100 ms** (0xA0) or greater, per the Bluetooth specification. Shorter intervals than this will result in an error response.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`08`	`04`	`08`	None.
RSP	`C0`	`02`	`04`	`08`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/A`	`0x0008`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Discovery mode: • 0 = Non-discoverable/broadcast-only • 1 = Limited discovery • 2 = General discovery
`uint8`	`type`	`T`	Advertisement type: • 0 = Connectable, undirected • 1 = Connectable, directed • 2 = Scannable, undirected • 3 = Non-connectable, undirected
`uint16`	`interval`	`I`	Advertisement interval (625 µs units): • Minimum = 0x0020 (32 * 0.625 ms = **20 ms**) • Maximum = 0x4000 (16384 * 0.625 ms = **10.24 seconds**)
`uint8`	`channels`	`C`	Advertisement channel selection bitmask (at least one bit must be set): • Bit 0 (0x1) = Channel 37 • Bit 1 (0x2) = Channel 38 • Bit 2 (0x4) = Channel 39
`uint8`	`filter`	`F`	Advertisement filter policy: • 0 = Scan request and connect request from any • 1 = Scan request whitelist-only, connect request from any • 2 = Scan request from any, connect request whitelist-only • 3 = Scan request and connect request whitelist-only
`uint16`	`timeout`	`O`	Advertisement timeout (seconds): • 0 to disable

Response Parameters:

None.

Related Commands:

- gap_stop_adv (/AX, ID=4/9)
- gap_set_adv_data (SAD, ID=4/19)
- gap_set_sr_data (SSRD, ID=4/21)
- gap_set_adv_parameters (SAP, ID=4/23)

Related Events:

- gap_adv_state_changed (ASC, ID=4/2)

Example Usage:

- Section 3.3.1 (How to Advertise as Peripheral Device)

7.2.2.6 gap_stop_adv (/AX, ID=4/9)

Stop advertising.

This command immediately stops advertising if it is currently active. Note that advertising may have started as a result of the gap_start_adv (/A, ID=4/8) API command, or due to specific configuration settings (GAP parameters, CYSPP profile) that automatically begin advertising.

EZ-Serial will generate the gap_adv_state_changed (ASC, ID=4/2) API event when the advertising state changes.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	04	09	None.
RSP	C0	02	04	09	None.

Text Info:

Text Name	Response Length	Category	Notes
/AX	0x0009	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Commands:

- gap_start_adv (/A, ID=4/8)

Related Events:

- gap_adv_state_changed (ASC, ID=4/2)

7.2.2.7 gap_query_peer_address (/QPA, ID=4/12)

Query remote peer Bluetooth address.

This command returns the Bluetooth address of the currently connected remote peer device. An active connection is required in order to use this command successfully.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	01	04	0C	None.
RSP	C0	09	04	0C	None.

Text Info:

Text Name	Response Length	Notes
/QPA	0x001E	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	conn_handle	C	Handle of connection for which to query remote peer address **(Ignored in current release due to internal BLE stack functionality, set to 0)**

Response Parameters:

Data Type	Name	Text	Description
`macaddr`	`address`	`A`	Peer Bluetooth address
`uint8`	`address_type`	`T`	Address type

Related Commands:

- gap_query_rssi (/QSS, ID=4/13)

7.2.2.8 gap_query_rssi (/QSS, ID=4/13)

This command returns the remote signal strength indication (RSSI) value detected in the packet received most recently from the currently connected remote peer device. An active connection is required in order to use this command successfully.

> **NOTE:** RSSI values in real-world environments often fall in the -50 dBm to -70 dBm range. An RSSI value at this level does not necessarily indicate a poor connection.

The RSSI value returned in the response is expressed as a signed 8-bit integer. In text mode, it will appear in two's complement form. Positive numbers in this form fall in the range [0, 127] and are as they appear. Negative numbers fall in the range [128, 255] and should have 256 subtracted from them to obtain the real value.

Examples:

- 0x03 = **+3 dBm**
- 0xFF = **-1 dBm** (0xFF = 255 - 256 = -1)
- 0xF0 = **-16 dBm** (0xF0 = 240 - 256 = -16)
- 0xC5 = **-59 dBm** (0xC5 = 197 - 256 = -59)

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`04`	`0D`	None.
RSP	`C0`	`03`	`04`	`0D`	None.

Text Info:

Text Name	Response Length	Notes
`/QSS`	`0x000F`	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Handle of connection for which to query signal strength **(Ignored in current release due to internal BLE stack functionality, set to 0)**

Response Parameters:

Data Type	Name	Text	Description
`int8`	`rssi`	`R`	RSSI value in dBm (between -85 and +5), or 0 if used while not connected

Related Commands:

- gap_query_peer_address (/QPA, ID=4/12)

7.2.2.9 gap_query_whitelist (/QWL, ID=4/14)

Request a list of whitelisted devices.

This command provides access to the current whitelist. The response from this command includes the number of devices on the whitelist, and the response is followed by that many gap_whitelist_entry (WL, ID=4/1) API events which provide details for each entry.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`0E`	None.
RSP	`C0`	`03`	`04`	`0E`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/QWL`	`0x000F`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`count`	`C`	Whitelist entry count

Related Commands:

- gap_add_whitelist_entry (/WLA, ID=4/6)
- gap_delete_whitelist_entry (/WLD, ID=4/7)

Related Events:

- gap_whitelist_entry (WL, ID=4/1)

7.2.2.10 gap_set_device_name (SDN, ID=4/15)

Configure a new device name.

This is typically a UTF-8 string value that is stored in the Device Name characteristic (UUID 0x2A00) in the local GATT structure. This characteristic is part of the GAP service (UUID 0x1800). The GAP service is mandatory for all Bluetooth Smart devices, and the Device Name characteristic is a mandatory part of the GAP service.

Using this command affects the value in the local GATT Server Device Name characteristic, and the local name field in the automatically managed scan response packed used for advertising.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01-41`	`04`	`0F`	Variable-length command payload, minimum of 1 (0x01), maximum of 65 (0x41)
RSP	`C0`	`02`	`04`	`0F`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SDN`	`0x0009`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`string`	`name`	`N`	New device name (0-64 bytes, raw ASCII data when in text mode)

Response Parameters:

None.

Related Commands:

- gap_get_device_name (GDN, ID=4/16)

Example Usage:

- Section 3.1.3 (How to Change the Device Name and Appearance)

7.2.2.11 gap_get_device_name (GDN, ID=4/16)

Obtain the current device name.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`10`	None.
RSP	`C0`	`03-43`	`04`	`10`	Variable-length response payload, minimum of 3 (0x03), maximum of 67 (0x43)

Text Info:

Text Name	Response Length	Category	Notes
`GDN`	`0x000C-0x004C`	GET	Variable-length response payload, minimum of 12 (0x0C), maximum of 76 (0x4C)

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`string`	`name`	`N`	Current device name (0-64 bytes, raw ASCII data when in text mode)

Related Commands:

- gap_set_device_name (SDN, ID=4/15)

7.2.2.12 gap_set_device_appearance (SDA, ID=4/17)

Configure a new device name.

Define the device appearance value. This is a 16-bit value which is stored in the Appearance characteristic (UUID 0x2A01) in the local GATT structure. This characteristic is part of the GAP service (UUID 0x1800). The GAP service is mandatory for every Bluetooth Smart device, and the Appearance characteristic is a mandatory part of the GAP service.

Using this command affects the value in the local GATT Server Device Appearance characteristic.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`02`	`04`	`11`	None.
RSP	`C0`	`02`	`04`	`11`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SDA`	`0x0009`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint16`	`appearance`	`A`	New device appearance value (factory default is 0x0000)

Response Parameters:

None.

Related Commands:

- gap_get_device_appearance (GDA, ID=4/18)

7.2.2.13 gap_get_device_appearance (GDA, ID=4/18)

Obtain the current device appearance value.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`12`	None.
RSP	`C0`	`04`	`04`	`12`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GDA`	`0x0010`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`appearance`	`A`	Current device appearance value

Related Commands:

- gap_set_device_appearance (SDA, ID=4/17)

7.2.2.14 gap_set_adv_data (SAD, ID=4/19)

Configure new custom advertisement packet data.

Define a new byte sequence for the primary advertisement packet data payload. This content is visible to all scanning devices performing a passive or active scan when the EZ-BLE module is in an advertising state.

> **NOTE:** EZ-Serial automatically manages advertisement content unless you enable the use of user-defined data with the gap_set_adv_parameters (SAP, ID=4/23) API command. If you only set custom data but do not enable user-defined content, the data here remains unused.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01-20`	`04`	`13`	Variable-length command payload, minimum of 1 (0x01), maximum of 32 (0x20)
RSP	`C0`	`02`	`04`	`13`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SAD`	`0x0009`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8a`	`data`	`D`	New advertisement payload data (0-31 bytes) **NOTE:** `uint8a` data type requires one prefixed “length” byte before binary parameter payload

Response Parameters:

None.

Related Commands:

- gap_start_adv (/A, ID=4/8)
- gap_get_adv_data (GAD, ID=4/20)
- gap_set_sr_data (SSRD, ID=4/21)
- gap_set_adv_parameters (SAP, ID=4/23)

Example Usage:

- Section 3.3.3 (How to Customize Advertisement and Scan Response Data)

7.2.2.15 gap_get_adv_data (GAD, ID=4/20)

Obtain the current custom advertisement packet data.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`14`	None.
RSP	`C0`	`03-22`	`04`	`14`	Variable-length response payload, minimum of 3 (0x03), maximum of 34 (0x22)

Text Info:

Text Name	Response Length	Category	Notes
`GAD`	`0x000D-0x004B`	GET	Variable-length response payload, minimum of 13 (0x0D), maximum of 75 (0x4B)

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8a`	`data`	`D`	Current advertisement payload data (0-31 bytes) **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Related Commands:

- gap_set_adv_data (SAD, ID=4/19)

7.2.2.16 gap_set_sr_data (SSRD, ID=4/21)

Configure new custom scan response packet payload.

This command defines a new byte sequence for the scan response packet. This content is visible to all scanning devices performing an active scan when the EZ-BLE module is in a scannable advertising state.

> **NOTE:** EZ-Serial automatically manages scan response content unless you enable the use of user-defined data with the gap_set_adv_parameters (SAP, ID=4/23) API command. If you only set custom data but do not enable user-defined content, the data here will remain unused.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01-20`	`04`	`15`	Variable-length command payload, minimum of 1 (0x01), maximum of 32 (0x20)
RSP	`C0`	`02`	`04`	`15`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SSRD`	`0x000A`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8a`	`data`	`D`	New scan response payload data (0-31 bytes) **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Response Parameters:

None.

Related Commands:

- gap_start_adv (/A, ID=4/8)
- gap_set_adv_data (SAD, ID=4/19)
- gap_get_sr_data (GSRD, ID=4/22)
- gap_set_adv_parameters (SAP, ID=4/23)

Example Usage:

- Section 3.3.3 (How to Customize Advertisement and Scan Response Data)

7.2.2.17 gap_get_sr_data (GSRD, ID=4/22)

Obtain the current custom scan response packet data.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`16`	None.
RSP	`C0`	`03-22`	`04`	`16`	Variable-length response payload, minimum of 3 (0x03), maximum of 34 (0x22)

Text Info:

Text Name	Response Length	Category	Notes
`GSRD`	`0x000D-0x004B`	GET	Variable-length response payload, minimum of 13 (0xD), maximum of 75 (0x4B)

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8a`	`data`	`D`	Current scan response payload data (0-31 bytes) **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Related Commands:

- gap_set_sr_data (SSRD, ID=4/21)

7.2.2.18 gap_set_adv_parameters (SAP, ID=4/23)

Configure new default advertisement parameters.

These parameters are used when sending the gap_start_adv (/A, ID=4/8) API command in text mode without specifying non-default arguments.

> **NOTE:** Setting Bit 0 (0x01) of the **flags** value using this command enables automatic advertisement on boot, as described. However, advertisements may automatically start even if this bit is cleared if the **enable** setting of CYSPP is set to the "enable + autostart" setting. Factory default settings include this value for the CYSPP feature.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	09	04	17	None.
RSP	C0	02	04	17	None.

Text Info:

Text Name	Response Length	Category	Notes
SAP	0x0009	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Discovery mode: • 0 = Non-discoverable/broadcast-only • 1 = Limited discovery • 2 = General discovery (factory default)
`uint8`	`type`	`T`	Advertisement type: • 0 = Connectable, undirected (factory default) • 1 = Connectable, directed • 2 = Scannable, undirected • 3 = Non-connectable, undirected
`uint16`	`interval`	`I`	Advertisement interval (625 µs units): • Minimum = 0x0020 (32 * 0.625 ms = **20 ms**) • Maximum = 0x4000 (16384 * 0.625 ms = **10.24 seconds**) • Factory default = 0x0030 (48 * 0.625 ms = **30 ms**)
`uint8`	`channels`	`C`	Advertisement channel selection bitmask: • Bit 0 (0x1) = Channel 37 • Bit 1 (0x2) = Channel 38 • Bit 2 (0x4) = Channel 39 • **NOTE:** At least one bit must be set, factory default is all **0x07** (all bits set)
`uint8`	`filter`	`L`	Advertisement filter policy: • 0 = Scan request and connect request from any (factory default) • 1 = Scan request whitelist-only, connect request from any • 2 = Scan request from any, connect request whitelist-only • 3 = Scan request and connect request whitelist-only
`uint16`	`timeout`	`O`	Advertisement timeout (seconds): • 0 to disable (factory default)
`uint8`	`flags`	`F`	Advertisement behavior flags bitmask: • Bit 0 (0x1) = Enable automatic advertising mode upon boot/disconnection • Bit 1 (0x2) = Use custom advertisement and scan response data • **NOTE:** Factory default = 0x00 (no bits set)

Response Parameters:

None.

Related Commands:

- gap_start_adv (/A, ID=4/8)
- gap_get_adv_parameters (GAP, ID=4/24)

7.2.2.19 *gap_get_adv_parameters (GAP, ID=4/24)*

Obtain the current advertisement parameters.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`04`	`18`	None.
RSP	`C0`	`0B`	`04`	`18`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GAP`	`0x0030`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Discovery mode: • 0 = Non-discoverable/broadcast-only • 1 = Limited discovery • 2 = General discovery (factory default)
`uint8`	`type`	`T`	Advertisement type: • 0 = Connectable, undirected (factory default) • 1 = Connectable, directed • 2 = Scannable, undirected • 3 = Non-connectable, undirected
`uint16`	`interval`	`I`	Advertisement interval (625 µs units): • Minimum = 0x0020 (32 * 0.625 ms = **20 ms**) • Maximum = 0x4000 (16384 * 0.625 ms = **10.24 seconds**) • Factory default = 0x0030 (48 * 0.625 ms = **30 ms**)
`uint8`	`channels`	`C`	Advertisement channel selection bitmask: • Bit 0 (0x1) = Channel 37 • Bit 1 (0x2) = Channel 38 • Bit 2 (0x4) = Channel 39 • **NOTE:** At least one bit must be set, factory default is all **0x07** (all bits set)
`uint8`	`filter`	`L`	Advertisement filter policy: • 0 = Scan request and connect request from any (factory default) • 1 = Scan request whitelist-only, connect request from any • 2 = Scan request from any, connect request whitelist-only • 3 = Scan request and connect request whitelist-only
`uint16`	`timeout`	`O`	Advertisement timeout (seconds): • 0 to disable (factory default)
`uint8`	`flags`	`F`	Advertisement behavior flags bitmask: • Bit 0 (0x1) = Enable automatic advertising mode upon boot/disconnection • Bit 1 (0x2) = Use custom advertisement and scan response data • **NOTE:** Factory default = 0x00 (no bits set)

Related Commands:

- gap_set_adv_parameters (SAP, ID=4/23)

7.2.3 GATT Server Group (ID=5)

GATT Server methods relate to the Server role of the Generic Attribute Protocol layer of the Bluetooth stack. These methods are used for working with the local GATT structure.

Commands within this group are listed below:

- gatts_create_attr (/CAC, ID=5/1)
- gatts_delete_attr (/CAD, ID=5/2)
- gatts_validate_db (/VGDB, ID=5/3)
- gatts_store_db (/SGDB, ID=5/4)
- gatts_dump_db (/DGDB, ID=5/5)
- gatts_discover_services (/DLS, ID=5/6)
- gatts_discover_characteristics (/DLC, ID=5/7)
- gatts_discover_descriptors (/DLD, ID=5/8)
- gatts_read_handle (/RLH, ID=5/9)
- gatts_write_handle (/WLH, ID=5/10)
- gatts_notify_handle (/NH, ID=5/11)
- gatts_indicate_handle (/IH, ID=5/12)
- gatts_set_parameters (SGSP, ID=5/14)
- gatts_get_parameters (GGSP, ID=5/15)

Events within this group are documented in Section 7.3.3, GATT Server Group (ID=5).

7.2.3.1 gatts_create_attr (/CAC, ID=5/1)

Add a new custom attribute to the local GATT structure.

The new attribute is given the next available handle. All handles are assigned sequentially. Attributes must be added in order, and are always appended to the next available position in the GATT structure.

New attributes must be entered such that the database always has a valid structure, other than possibly being incomplete while adding other required attributes. EZ-Serial rejects new attribute creation attempts that would result in an invalid structure and provide a validity report code from the list in Section 7.4.2 (EZ-Serial GATT Database Validation Error Codes).

See Section 3.4.1 (How to Define Custom Local GATT Services and Characteristics) and Section 10.2 (Adopted Bluetooth SIG GATT Profile Structure Snippets) for detailed instructions and example usage, including important guidelines for permission settings.

NOTE: Always configure structural declarations (types 0x2800 and 0x2803) to have unrestricted read permissions (0x01) and no write permissions (0x00) to ensure that Clients can properly discover the basic GATT database structure. Special security requirements should only be applied to characteristic value attributes or, in limited cases, related configuration descriptors.

Use the gatts_dump_db (/DGDB, ID=5/5) API command to list the current local GATT database entries in a format similar to what this command requires.

NOTE: EZ-Serial includes a fixed set of attributes as part of the core functionality, which cannot be deleted or modified. These attributes occupy the handle range from 1 (0x0001) to 21 (0x0015). Therefore, the first custom attribute created in a factory default state receives the handle value 22 (0x0016).

NOTE: Additions to and removals from the GATT structure are always stored in flash. As long as the “result” value in the response indicates success, the change are effective immediately and persist through power cycles and resets. The internal CPU is occupied for approximately 15 ms during each flash write operation; during this time, no other activity is processed (UART or BLE communication). Any UART data sent during this brief window is lost. Therefore, you should modify the GATT structure only while disconnected, and you should allow a gap of at least 20 ms between the end of one API command and the beginning of a new one. If you have enabled hardware flow control using the system_set_uart_parameters (STU, ID=2/25) API command, EZ-Serial blocks incoming data flow during flash writes to prevent serial data corruption or loss.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`09`	`05`	`01`	Variable-length command payload, value specified is minimum
RSP	`C0`	`06`	`05`	`01`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/CAC`	`0x0018`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`type`	`T`*	type: • 0 = structure • 1 = characteristic value Structural entries require constant data containing the definition Structural entries optionally allow additional RAM data beyond the constant length for descriptor value information, such a two-byte CCCD values . Characteristic value entries do not require any constant data, but may have it if a default boot-time value is desired
`uint8`	`perm`	`P`*	Permission bits: • Bit 0 (0x01) = Variable length • Bit 1 (0x02) = Readable • Bit 2 (0x04) = Write command (unacknowledged) • Bit 3 (0x08) = Write request (acknowledged) • Bit 4 (0x10) = Authenticated readable • Bit 5 (0x20) = Reliable write (includes prepared write) • Bit 6 (0x40) = Authenticated writeable
`uint16`	`length`	`L`*	Indicates how many bytes of RAM are allocated for the definition (structure) or content (characteristic value)
`longuint8a`	`data`	`D`*	Data (UUID or default attribute value where applicable) Data may include UUID, characteristic properties byte and/or value. **NOTE:** `longuint8a` data type requires two prefixed "length" bytes before binary parameter payload. Characteristic properties: • Bit 0 (0x01) = Broadcast • Bit 1 (0x02) = Read • Bit 2 (0x04) = Write without response • Bit 3 (0x08) = Write • Bit 4 (0x10) = Notify • Bit 5 (0x20) = Indicate • Bit 6 (0x40) = Signed write • Bit 7 (0x80) = Extended properties (requires 0x2900) Characteristic declaration store the UUID of the Characteristic value attribute. So its 'D' will be: 0x2803 (UUID)+ Characteristic properties(1 byte)+ handle of value attribute(2 byte) + UUID of value attribute.

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`handle`	`H`	New attribute handle (0x0001-0xFFFF)
`uint16`	`valid`	`V`	GATT database validity status

Related Commands:

- gatts_delete_attr (/CAD, ID=5/2)
- gatts_validate_db (/VGDB, ID=5/3)
- gatts_dump_db (/DGDB, ID=5/5)

Related Events:

- gatts_db_entry_blob (DGATT, ID=5/4)

Example Usage:

- Section 3.4.1 (How to Define Custom Local GATT Services and Characteristics)
- Section 10.2 (Adopted Bluetooth SIG GATT Profile Structure Snippets)

7.2.3.2 gatts_delete_attr (/CAD, ID=5/2)

Remove one or more attributes from the GATT structure.

If you use this command without a handle in text mode or you supply handle value 0 in either text or binary mode, the highest attribute number (most recently added) is removed. If you supply a non-zero handle, the attribute with that handle **and all higher handles** are removed.

After removing an attribute with this command, the local GATT database may no longer be strictly valid. See Section 7.4.2 (EZ-Serial GATT Database Validation Error Codes) for possible validity states. Use the gatts_dump_db (/DGDB, ID=5/5) API command to list the current local GATT database entries.

> **NOTE:** EZ-Serial includes a fixed set of attributes as part of the core functionality, which cannot be deleted or modified. These attributes occupy the handle range from 1 (0x0001) to 28 (0x001C). Therefore, you cannot delete any attribute with a handle value less than 29 (0x001D).

> **NOTE:** Additions to and removals from the GATT structure are always stored in flash. As long as the "result" value in the response indicates success, the change is effective immediately and persists through power cycles and resets. The internal CPU is occupied for approximately 15 ms during each flash write operation; during this time, no other activity is processed (UART or BLE communication). Any UART data sent during this brief window is lost. Therefore, you should modify the GATT structure only while disconnected, and you should allow a gap of at least 20 ms between the end of one API command and the beginning of a new one. If you have enabled hardware flow control using the system_set_uart_parameters (STU, ID=2/25) API command, EZ-Serial blocks incoming data flow during flash writes to prevent serial data corruption or loss.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`02`	`05`	`02`	None.
RSP	`C0`	`08`	`05`	`02`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/CAD`	`0x001F`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint16`	`handle`	`H`	Attribute handle to remove (includes all higher attributes)

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`count`	`C`	Number of attributes deleted from GATT structure

Data Type	Name	Text	Description
`uint16`	`next_handle`	`H`	Next available attribute handle after removal
`uint16`	`valid`	`V`	GATT database validity status

Related Commands:

- gatts_create_attr (/CAC, ID=5/1)
- gatts_validate_db (/VGDB, ID=5/3)
- gatts_dump_db (/DGDB, ID=5/5)

7.2.3.3 gatts_validate_db (/VGDB, ID=5/3)

Check to ensure that the custom GATT structure has no malformed or missing elements.

Use this command to check for errors in the custom GATT structure configured in EZ-Serial. The dynamic GATT implementation automatically tests for validity issues when making changes to the structure with the gatts_create_attr (/CAC, ID=5/1) and gatts_delete_attr (/CAD, ID=5/2) API commands, but this command provides the same test result upon request without making or attempting any modifications. See Section 7.4.2 (EZ-Serial GATT Database Validation Error Codes) for possible validity states.

EZ-Serial allows only one non-valid state, indicated by GATTS_DB_VALID_WARNING_NOT_ENOUGH_ATTRIBUTES code (0x0001). This non-valid state is unavoidable during custom attribute creation, because attributes must be added one at a time, and every new service or characteristic requires multiple attributes. All other non-valid states prevent the addition of a custom attribute in the first place. Therefore, running this command should result only in a valid state (0x0000) or the warning state noted here (0x0001).

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`05`	`03`	None.
RSP	`C0`	`04`	`05`	`03`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/VGDB`	`0x0012`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`valid`	`V`	GATT database validity status

Related Commands:

- gatts_create_attr (/CAC, ID=5/1)
- gatts_delete_attr (/CAD, ID=5/2)
- gatts_dump_db (/DGDB, ID=5/5)

7.2.3.4 gatts_store_db (/SGDB, ID=5/4)

Store the current custom GATT structure in flash.

> **NOTE: This command has been deprecated and has no effect when used.** As of the latest firmware build, GATT database changes are always written instantly to flash when using either gatts_create_attr (/CAC, ID=5/1) or gatts_delete_attr (/CAD, ID=5/2).

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	05	04	None.
RSP	C0	02	05	04	None.

Text Info:

Text Name	Response Length	Category	Notes
/SGDB	0x000B	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Commands:

- gatts_create_attr (/CAC, ID=5/1)
- gatts_delete_attr (/CAD, ID=5/2)
- gatts_validate_db (/VGDB, ID=5/3)
- gatts_dump_db (/DGDB, ID=5/5)

7.2.3.5 gatts_dump_db (/DGDB, ID=5/5)

List current local GATT database attributes.

This command produces a series of gatts_db_entry_blob (DGATT, ID=5/4) API events, one for each attribute in the current local GATT database. The output is similar to that of the gatts_discover_descriptors (/DLD, ID=5/8) API command, but in a format that more closely matches the input parameters of the gatts_create_attr (/CAC, ID=5/1) API command.

You can choose to dump only those attributes in the user-definable range (0x001D and above), or include fixed attributes as well (0x0001 and above) for complete reference.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	01	05	05	None.
RSP	C0	04	05	05	None.

Text Info:

Text Name	Response Length	Notes
/DGDB	0x0012	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	include_fixed	F	Include fixed attributes: • 0 = Start from handle 0x001D, do not include fixed attributes (default) • 1 = Start from handle 0x0001, include fixed attributes

Response Parameters:

Data Type	Name	Text	Description
uint16	count	C	Number of entries to be returned

Related Commands:

- gatts_create_attr (/CAC, ID=5/1)
- gatts_delete_attr (/CAD, ID=5/2)
- gatts_validate_db (/VGDB, ID=5/3)

- gatts_discover_descriptors (/DLD, ID=5/8)

Related Events:

- gatts_db_entry_blob (DGATT, ID=5/4)

7.2.3.6 gatts_discover_services (/DLS, ID=5/6)

Request a list of all services in the local GATT structure.

This allows convenient discovery of services within the local GATT database. This command does not require an active connection because it concerns only local resources. Normally, you should not need to use this command except during development because the application should already know all relevant details about its own local GATT structure. To find all services in the local database, use “0” for both arguments, or explicitly set 0x0001 and 0xFFFF for the beginning and end handles.

The gatts_discover_result (DL, ID=5/1) API events resulting from this command will be generated when any local GATT services discovered.

For local GATT database information that more closely matches the input format required for the gatts_create_attr (/CAC, ID=5/1) API command, use the gatts_dump_db (/DGDB, ID=5/5) API command instead.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`04`	`05`	`06`	None.
RSP	`C0`	`04`	`05`	`06`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/DLS`	`0x0011`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint16`	`begin`	`B`	Handle to begin searching
`uint16`	`end`	`E`	Handle to end searching (inclusive)

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`count`	`C`	Number of entries to be returned

Related Commands:

- gatts_dump_db (/DGDB, ID=5/5)
- gatts_discover_characteristics (/DLC, ID=5/7)
- gatts_discover_descriptors (/DLD, ID=5/8)

Related Events:

- gatts_discover_result (DL, ID=5/1)

Example Usage:

Note: Any attribute that requires authentication (bonding) must also require encryption. If you enable the authentication bit, ensure that you also enable the encryption bit, or the command will be rejected with an error result.

- How to List Local GATT Services, Characteristics, and Descriptors

7.2.3.7 gatts_discover_characteristics (/DLC, ID=5/7)

Request a list of all characteristics in the local GATT structure.

This allows convenient discovery of characteristics within the local GATT database. This command does not require an active connection because it concerns only local resources. Normally, you should not need to use this command except during development because the application should already know all relevant details about its own local GATT structure. To find all characteristics in the local database, use “0” for both arguments, or explicitly set 0x0001 and 0xFFFF for the beginning and end handles.

The gatts_discover_result (DL, ID=5/1) API events resulting from this command will be generated when any local GATT characteristics discovered.

For local GATT database information that more closely matches the input format required for the gatts_create_attr (/CAC, ID=5/1) API command, use the gatts_dump_db (/DGDB, ID=5/5) API command instead.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`06`	`05`	`07`	None.
RSP	`C0`	`04`	`05`	`07`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/DLC`	`0x0011`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint16`	`begin`	`B`	Handle to begin searching
`uint16`	`end`	`E`	Handle to end searching (inclusive)
`uint16`	`service`	`S`	Service UUID filter (0 for all) – **Currently not implemented in firmware, set to 0**

Response Parameters:

Data Type	Name	Text	Description
`uint16`	`count`	`C`	Number of entries to be returned

Related Commands:

- gatts_dump_db (/DGDB, ID=5/5)
- gatts_discover_services (/DLS, ID=5/6)
- gatts_discover_descriptors (/DLD, ID=5/8)

Related Events:

- gatts_discover_result (DL, ID=5/1)

Example Usage:

- Section 3.4.2 (How to List Local GATT Services, Characteristics, and Descriptors)

7.2.3.8 gatts_discover_descriptors (/DLD, ID=5/8)

Request a list of all descriptors in the local GATT structure.

This allows convenient discovery of descriptors within the local GATT database. This command does not require an active connection because it concerns only local resources. Normally, you should not need to use this command except during development because the application should already know all relevant details about its own local GATT structure. To find all descriptors in the local database, use “0” for both arguments, or explicitly set 0x0001 and 0xFFFF for the beginning and end handles, respectively.

The gatts_discover_result (DL, ID=5/1) API events resulting from this command wil be generated when any local GATT descriptors discovered.

For local GATT database information that more closely matches the input format required for the gatts_create_attr (/CAC, ID=5/1) API command, use the gatts_dump_db (/DGDB, ID=5/5) API command instead.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	08	05	08	None.
RSP	C0	04	05	08	None.

Text Info:

Text Name	Response Length	Category	Notes
/DLD	0x0011	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
uint16	begin	B	Handle to begin searching
uint16	end	E	Handle to end searching (inclusive)
uint16	service	S	Service UUID filter (0 for all) **(Ignored in current release, set to 0)**
uint16	characteristic	C	Characteristic UUID filter (0 for all) **(Ignored in current release, set to 0)**

Response Parameters:

Data Type	Name	Text	Description
uint16	count	C	Number of entries to be returned

Related Commands:

- gatts_dump_db (/DGDB, ID=5/5)
- gatts_discover_services (/DLS, ID=5/6)
- gatts_discover_characteristics (/DLC, ID=5/7)

Related Events:

- gatts_discover_result (DL, ID=5/1)

Example Usage:

- Section 3.4.2 (How to List Local GATT Services, Characteristics, and Descriptors)

7.2.3.9 gatts_read_handle (/RLH, ID=5/9)

Read the value of an attribute in the local GATT Server.

This command does not require an active connection because it concerns only local resources.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	02	05	09	None.
RSP	C0	04+	05	09	Variable-length response payload, value specified is minimum.

Text Info:

Text Name	Response Length	Category	Notes
/RLH	0x000D+	ACTION	Variable-length response payload, value specified is minimum.

Command Arguments:

Data Type	Name	Text	Description

Data Type	Name	Text	Description
`uint16`	`attr_handle`	`H*`	Handle of attribute to read value from

Response Parameters:

Data Type	Name	Text	Description
`longuint8a`	`data`	`D`	Data read from attribute **NOTE:** `longuint8a` data type requires two prefixed "length" bytes before binary parameter payload

Related Commands:

- gatts_write_handle (/WLH, ID=5/10)

7.2.3.10 gatts_write_handle (/WLH, ID=5/10)

Write a new value to an attribute in the local GATT Server.

This command does not require an active connection because it concerns only local resources.

> **NOTE:** Writing data to a local characteristic value attribute does not automatically trigger a notification or indication of that data to a connected Client, even if the Client has subscribed to notifications or indications for the characteristic. This command affects only the value stored locally in RAM if the Client performs a GATT read operation later. To push data to a Client that subscribed to notifications or indications, use the gatts_notify_handle (/NH, ID=5/11) or gatts_indicate_handle (/IH, ID=5/12) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`04`	`05`	`0A`	Variable-length command payload, value specified is minimum.
RSP	`C0`	`02`	`05`	`0A`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/WLH`	`0x000A`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint16`	`attr_handle`	`H*`	Handle of attribute to write new value to
`longuint8a`	`data`	`D*`	New data to write to attribute **NOTE:** `longuint8a` data type requires two prefixed "length" bytes before binary parameter payload

Response Parameters:

None.

Related Commands:

- gatts_read_handle (/RLH, ID=5/9)
- gatts_notify_handle (/NH, ID=5/11)
- gatts_indicate_handle (/IH, ID=5/12)

7.2.3.11 gatts_notify_handle (/NH, ID=5/11)

Notify a new attribute value to a remote GATT Client.

> **NOTE:** This command does not change any locally stored values for the notified attribute. To modify the data stored locally in RAM for the attribute in question, use the gatts_write_handle (/WLH, ID=5/10) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	06	05	0B	Variable-length command payload, value specified is minimum.
RSP	C0	02	05	0B	None.

Text Info:

Text Name	Response Length	Category	Notes
/NH	0x0009	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	conn_handle	C	Connection handle to use for notification **(Ignored in current release due to internal BLE stack functionality, set to 0)**
uint16	attr_handle	H*	Handle of attribute to notify
uint8a	data	D*	Data to push to remote Client via notification **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Response Parameters:

None.

Related Commands:

- gatts_write_handle (/WLH, ID=5/10)
- gatts_indicate_handle (/IH, ID=5/12)

7.2.3.12 gatts_indicate_handle (/IH, ID=5/12)

Indicate a new attribute value to a remote GATT Client.

If successful, pushing an indicated value to a remote Client results in the gatts_indication_confirmed (IC, ID=5/3) API event occurring after the Client acknowledges the transfer.

Because this method requires Client acknowledgement, you cannot attempt another GATT operation until this confirmation event arrives. A single acknowledged transfer requires two connection intervals: one for the actual data transfer, and one for the acknowledgement. Using this type of transfer has effects on potential throughput; see Section 3.6.1 (How to Maximize Throughput to a Remote Peer) for details on alternative design choices.

> **NOTE:** This command does not change any locally stored values for the indicated attribute. To modify the data stored locally in RAM for the attribute in question, use the gatts_write_handle (/WLH, ID=5/10) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	06	05	0C	Variable-length command payload, value specified is minimum.
RSP	C0	02	05	0C	None.

Text Info:

Text Name	Response Length	Category	Notes
/IH	0x0009	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	conn_handle	C	Connection handle to use for indication

Data Type	Name	Text	Description
			(Ignored in current release due to internal BLE stack functionality, set to 0)
`uint16`	`attr_handle`	`H`*	Handle of attribute to indicate
`uint8a`	`data`	`D`*	Data to indicate **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Response Parameters:

None.

Related Commands:

- gatts_read_handle (/RLH, ID=5/9)
- gatts_write_handle (/WLH, ID=5/10)
- gatts_notify_handle (/NH, ID=5/11)

Related Events:

- gatts_indication_confirmed (IC, ID=5/3) - Occurs on the Server after the remote Client confirms receipt of indicated data

7.2.3.13 gatts_set_parameters (SGSP, ID=5/14)

Configure new GATT Server parameters.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`05`	`0E`	None.
RSP	`C0`	`02`	`05`	`0E`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SGSP`	`0x000A`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`flags`	`F`	GATT Server behavior flags bitmask: • Bit 0 (0x01) = Enable automatic response to acknowledged writes • **NOTE:** Factory default is 0x01 (all bits set)

Response Parameters:

None.

Related Commands:

- gatts_get_parameters (GGSP, ID=5/15)

7.2.3.14 gatts_get_parameters (GGSP, ID=5/15)

Obtain current GATT Server parameters.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`05`	`0F`	None.
RSP	`C0`	`03`	`05`	`0F`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GGSP`	`0x000F`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`flags`	`F`	GATT Server behavior flags bitmask: • Bit 0 (0x01) = Enable automatic response to acknowledged writes • **NOTE:** Factory default is 0x01 (all bits set)

Related Commands:

- gatts_set_parameters (SGSP, ID=5/14)

7.2.4 SMP Group (ID=7)

SMP methods relate to the Security Manager Protocol layer of the Bluetooth stack. These methods are used for working with privacy, encryption, pairing, and bonding between two devices.

Commands within this group are listed below:

- smp_query_bonds (/QB, ID=7/1)
- smp_delete_bond (/BD, ID=7/2)
- smp_pair (/P, ID=7/3)
- smp_send_pairreq_response (/PR, ID=7/5)
- smp_set_privacy_mode (SPRV, ID=7/9)
- smp_get_privacy_mode (GPRV, ID=7/10)
- smp_set_security_parameters (SSBP, ID=7/11)
- smp_get_security_parameters (GSBP, ID=7/12)
- smp_set_fixed_passkey (SFPK, ID=7/13)
- smp_get_fixed_passkey (GFPK, ID=7/14)

Events within this group are documented in Section 7.3.4, SMP Group (ID=7).

7.2.4.1 smp_query_bonds (/QB, ID=7/1)

Request a list of bonded devices.

This command accesses the current bonded device list. Bonded devices are those which have previously paired (exchanged encryption data) and bonded (stored the exchanged encryption data).

The response from this command includes the number of bonded devices, and the response are followed by that many smp_bond_entry (B, ID=7/1) API events that provide details for each device.

NOTE: EZ-Serial currently supports a maximum of four bonded devices at the same time. To bond with additional devices after all four bond slots are full, you must delete one of the existing bonds with the smp_delete_bond (/BD, ID=7/2) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`07`	`01`	None.
RSP	`C0`	`03`	`07`	`01`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/QB`	`0x000E`	ACTION	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`count`	`C`	Bond entry count

Related Commands:

- smp_pair (/P, ID=7/3) – Creates a new bond entry if pairing process succeeds with bonding enabled

Related Events:

- smp_bond_entry (B, ID=7/1) – Occurs once for each bonded device after requesting bond list

7.2.4.2 smp_delete_bond (/BD, ID=7/2)

Remove a bonded device.

This command removes the stored encryption key data for a device that has previously paired (exchanged encryption data) and bonded (stored the exchanged encryption data).

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`07`	`07`	`02`	None.
RSP	`C0`	`03`	`07`	`02`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/BD`	`0x000E`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`Macaddr`	`address`	`A`*	Bluetooth address
`uint8`	`type`	`T`	Address type: • 0 = Public (default) • 1 = Random/private

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`count`	`C`	Updated bond entry count

Related Commands:

- smp_query_bonds (/QB, ID=7/1)
- smp_pair (/P, ID=7/3) – Creates a new bond entry if pairing process succeeds with bonding enabled

7.2.4.3 smp_pair (/P, ID=7/3)

Initiate pairing process with a connected device.

> **NOTE:** EZ-Serial currently supports a maximum of four bonded devices at the same time. To bond with additional devices after all four bond slots are full, you must delete one of the existing bonds with the smp_delete_bond (/BD, ID=7/2) API command.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`05`	`07`	`03`	None.
RSP	`C0`	`02`	`07`	`03`	None.

Text Info:

Text Name	Response Length	Category	Notes
`/P`	`0x0008`	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Connection handle to use for pairing **(Ignored in current release due to internal BLE stack functionality, set to 0)**
`uint8`	`mode`	`M`	Security level setting reported to peer: always 0
`uint8`	`bonding`	`B`	Bond during pairing process: • 0 = Do not bond (exchange keys and encrypt only) • 1 = Bond (permanently store exchanged encryption data)
`uint8`	`keysize`	`K`	Encryption key size (7-16), value ignored if pairing initiated by slave device • **NOTE:** Factory default is 16 bytes (0x10)
`uint8`	`pairprop`	`P`	Pairing properties: always 0

Response Parameters:

None.

Related Commands:

- smp_send_pairreq_response (/PR, ID=7/5) – Use when remote device initiates pairing and auto-accept flag bit is not disabled
- smp_set_security_parameters (SSBP, ID=7/11) – Use to configure default security settings

Related Events:

- smp_pairing_requested (P, ID=7/2) – Occurs when remote device initiates pairing
- smp_pairing_result (PR, ID=7/3) – Occurs when pairing process completes (success or failure)
- smp_encryption_status (ENC, ID=7/4) – Occurs when encryption status changes during a pairing process

7.2.4.4 smp_send_pairreq_response (/PR, ID=7/5)

Send a response to a pairing request from a remote device.

EZ-Serial automatically accepts pairing requests unless **Bit 0** of the security behavior flags is cleared using the **`flags`** field in the smp_set_security_parameters (SSBP, ID=7/11) API command. If the auto-accept feature is disabled, use this command to manually accept or deny a remotely initiated pairing process.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`03`	`07`	`05`	None.

	Type	Length	Group	ID	Notes
RSP	C0	02	07	05	None.

Text Info:

Text Name	Response Length	Category	Notes
/PR	0x0009	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	conn_handle	C	Connection handle to use for sending response **(Ignored in current release due to internal BLE stack functionality, set to 0)**
uint16	response	R*	Response (0 = accept, non-zero = reject)

Response Parameters:

None.

Related Commands:

- smp_pair (/P, ID=7/3) – Used to initiate pairing

Related Events:

- smp_pairing_requested (P, ID=7/2) – Occurs when a remote device requests pairing
- smp_pairing_result (PR, ID=7/3) – Occurs after a pairing process completes (successfully or otherwise)

7.2.4.5 smp_set_privacy_mode (SPRV, ID=7/9)

Configure new privacy settings.

Use this command to enable or disable Peripheral or Central privacy. Enabling privacy in each mode causes the Bluetooth connection address used in related states to be random (private) instead of fixed (public). This can make passive profiling by a remote observer more difficult.

Peripheral privacy affects the Bluetooth connection address broadcast during advertisements, which the remote Central device may log or use for a scan request or connection request. Central privacy affects the Bluetooth connection address used for scan requests or connection requests when scanning for or communicating with a remote device.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	03	07	09	None.
RSP	C0	02	07	09	None.

Text Info:

Text Name	Response Length	Category	Notes
SPRV	0x000A	SET	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	mode	M	Privacy mode bitmask: • Bit 0 (0x01) = Enable peripheral privacy • **NOTE:** Factory default is 0x01 (peripheral privacy enabled)
uint16	interval	I	Randomization interval (seconds)

Response Parameters:

None.

Related Commands:

- smp_get_privacy_mode (GPRV, ID=7/10)

7.2.4.6 smp_get_privacy_mode (GPRV, ID=7/10)

Obtain current privacy settings.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`07`	`0A`	None.
RSP	`C0`	`05`	`07`	`0A`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GPRV`	`0x0016`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Privacy mode bitmask: • Bit 0 (0x01) = Enable peripheral privacy • **NOTE:** Factory default is 0x00 (no bits set)
`uint16`	`interval`	`I`	Randomization interval (seconds)

Related Commands:

- smp_set_privacy_mode (SPRV, ID=7/9)

7.2.4.7 smp_set_security_parameters (SSBP, ID=7/11)

Configure new security and bonding parameters.

These parameters are used when the smp_pair (/P, ID=7/3) API command is used without specifying non-default arguments. These values are reported to the remote device as part of the pairing process and affect the type of key generation and exchange that takes place during pairing and bonding.

> **NOTE:** Changing the I/O capabilities affects the command/event flow necessary to complete a pairing and bonding process. See the related commands and events for details concerning each one's use. Also, MITM protection requires I/O capabilities other than "No Input + No Output" in order to function correctly.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`06`	`07`	`0B`	None.
RSP	`C0`	`02`	`07`	`0B`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SSBP`	`0x000A`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Security level setting reported to peer: Don't care and always 11

Data Type	Name	Text	Description
`uint8`	`bonding`	`B`	Bond during pairing process: • 0 = Do not bond (exchange keys and encrypt only) • 1 = Bond (permanently store exchanged encryption data)
`uint8`	`keysize`	`K`	Encryption key size (7-16), value ignored if pairing initiated by slave device • **NOTE:** Factory default is 16 bytes (0x10)
`uint8`	`pairprop`	`P`	Pairing properties: Don't care and always 0
`uint8`	`io`	`I`	I/O capabilities: • 0 = Display Only – ability to convey a 6-digit number to user • 1 = Display + Yes/No – display and the ability to have user indicate "yes" or "no" • 2 = Keyboard Only – ability for the user to enter '0' through '9' and "yes" or "no" • 3 = No Input + No Output – no ability to display or input anything (factory default) • 4 = Keyboard + Display – ability to provide full numeric input and display
`uint8`	`flags`	`F`	Security behavior flags bitmask: • Bit 0 (0x01) = Enable auto-accept for incoming pairing requests • Bit 1 (0x02) = Enable use of fixed passkey during pairing • **NOTE:** Factory default is 0x01

Response Parameters:

None.

Related Commands:

- smp_pair (/P, ID=7/3)
- smp_send_pairreq_response (/PR, ID=7/5)
- smp_get_security_parameters (GSBP, ID=7/12)
- smp_set_fixed_passkey (SFPK, ID=7/13)

Related Events:

- smp_pairing_requested (P, ID=7/2)
- smp_pairing_result (PR, ID=7/3)
- smp_encryption_status (ENC, ID=7/4)

7.2.4.8 smp_get_security_parameters (GSBP, ID=7/12)

Obtain current security and bonding parameters.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`07`	`0C`	None.
RSP	`C0`	`08`	`07`	`0C`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GSBP`	`0x0028`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`Mode`	`M`	Security level setting reported to peer: Don't care and always 11
`uint8`	`bonding`	`B`	Bond during pairing process: • 0 = Do not bond (exchange keys and encrypt only) • 1 = Bond (permanently store exchanged encryption data)

Data Type	Name	Text	Description
`uint8`	`keysize`	`K`	Encryption key size (7-16), value ignored if pairing initiated by slave device • **NOTE:** Factory default is 16 bytes (0x10)
`uint8`	`pairprop`	`P`	• Pairing properties: Don't care and always 0
`uint8`	`Io`	`I`	I/O capabilities: • 0 = Display Only – ability to convey a 6-digit number to user • 1 = Display + Yes/No – display and the ability to have user indicate “yes” or “no” • 2 = Keyboard Only – ability for the user to enter ‘0’ through ‘9’ and “yes” or “no” • 3 = No Input + No Output – no ability to display or input anything (factory default) • 4 = Keyboard + Display – ability to provide full numeric input and display
`uint8`	`Flags`	`F`	Security behavior flags bitmask: • Bit 0 (0x01) = Enable auto-accept for incoming pairing requests • Bit 1 (0x02) = Enable use of fixed passkey during pairing • **NOTE:** Factory default is 0x01

Related Commands:

- smp_set_security_parameters (SSBP, ID=7/11)

7.2.4.9 smp_set_fixed_passkey (SFPK, ID=7/13)

Configure new fixed passkey value.

While the Bluetooth specification describes that the passkey should be randomized during pairing, you can configure a fixed (non-random) 6-digit passkey between 000000 and 999999 using this command and configuring the local I/O capabilities to the “Display Only” value.

> **NOTE:** The fixed passkey defined here takes effect only if you enable fixed passkey use by setting Bit 1 (0x02) of the security flags parameter and set the “Display Only” I/O capabilities value (0x00) using the smp_set_security_parameters (SSBP, ID=7/11) API command. If both of these conditions are not met, then the stack will revert to the default behavior of using a random passkey.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`04`	`07`	`0D`	None.
RSP	`C0`	`02`	`07`	`0D`	None.

Text Info:

Text Name	Response Length	Category	Notes
`SFPK`	`0x000A`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint32`	`passkey`	`P`	Fixed passkey value • Minimum = 0 (‘000000’ decimal entry during pairing) • Maximum = 0xF423F (‘999999’ decimal entry during pairing) • **NOTE:** Factory default is 0

Response Parameters:

None.

Related Commands:

- smp_pair (/P, ID=7/3)
- smp_send_pairreq_response (/PR, ID=7/5)
- smp_get_fixed_passkey (GFPK, ID=7/14)
- smp_set_security_parameters (SSBP, ID=7/11)

Related Events:

- smp_pairing_requested (P, ID=7/2)
- smp_pairing_result (PR, ID=7/3)
- smp_encryption_status (ENC, ID=7/4)

Example Usage:

- Section 3.5.1.3 (Pairing and Bonding with a Fixed Passkey)

7.2.4.10 smp_get_fixed_passkey (GFPK, ID=7/14)

Obtain current fixed passkey value.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`07`	`0E`	None.
RSP	`C0`	`08`	`07`	`0E`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GFPK`	`0x0015`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint32`	`passkey`	`P`	Fixed passkey value • Minimum = 0 ('000000' decimal entry during pairing) • Maximum = 0xF423F ('999999' decimal entry during pairing) • **NOTE:** Factory default is 0

Related Commands:

- smp_set_fixed_passkey (SFPK, ID=7/13)

7.2.5 GPIO Group (ID=9)

GPIO methods relate to the physical pins on the module.

Commands within this group are listed below:

- gpio_query_adc (/QADC, ID=9/2)

Events within this group are documented in Section 7.3.5 GPIO Group (ID=9).

7.2.5.1 gpio_query_adc (/QADC, ID=9/2)

Read the immediate analog voltage level on the selected channel.

EZ-Serial provides a single dedicated ADC input pin (**ADC0**) for reading analog voltages. The ADC supports an input voltage range of **0 V** minimum to VDD (usually **3.3V)** maximum. Use this command to perform a single ADC conversion. Once the conversion completes, the module transmits the result back in response parameters.

See Section 8.1 (GPIO Pin Map for Supported Modules) for a pin map table showing ADC pin assignment.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	01	09	02	None.
RSP	C0	02	09	02	None.

Text Info:

Text Name	Response Length	Category	Notes
/QADC	0x000B	ACTION	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	channel	N*	ADC channel (0 only)
uint8	reference	R	Voltage reference for conversion **(Ignored in current release, set to 0 and VDD will be used)**

Response Parameters:

Data Type	Name	Text	Description
uint16	value	A	Raw ADC conversion value, 0 – 2047 (0x0 – 0x7FF)
uint32	uvolts	U	Scaled ADC result in microvolts, 0 – VDD (0x0 – 0x325AA0 if VDD is 3.3V)

7.2.5.2 gpio_set_pwm_mode (SPWM, ID=9/11)

Configure new PWM output behavior for selected channel.

EZ-Serial provides a dedicated PWM output pin (**PWM0**). Enabling PWM on the channel means you cannot use that pin for other generic I/O. To return a PWM channel pin to standard functionality, use the gpio_set_pwm_mode (SPWM, ID=9/11) API command to disable PWM output on that pin. See Section 8.1 (GPIO Pin Map for Supported Modules) for a pin map table showing pin availability and default assignment.

NOTE: Enabling PWM output automatically prevents the CPU from entering normal sleep under any circumstances. This happens because the high-frequency clock required to generate the PWM signal cannot operate while the CPU is in sleep. To allow normal sleep mode again, you must disable all PWM output. Refer to Section 3.1.5 (How to Manage Sleep States) for further detail.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	08	09	0B	None.
RSP	C0	02	09	0B	None.

Text Info:

Text Name	Response Length	Category	Notes
SPWM	0x000A	SET	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	channel	N*	Channel number (0 only)
uint8	enable	E	Enable PWM output (0 to disable, 1 to enable)
uint8	divider	D	Clock divider value (24 MHz input): • Minimum = 0 (factory default) • Maximum = 255 • **NOTE:** Divider denominator is `divider+1`, so “`0`” is “divide by 1”

Data Type	Name	Text	Description
`uint8`	`prescaler`	`S`	PWM prescaler value: • 0 = 1x (no prescaling) • 1 = 2x • 2 = 4x • 3 = 8x • 4 = 16x • 5 = 32x • 6 = 64x • 7 = 128x • **NOTE:** Factory default is 0 (1x, no prescaling)
`uint16`	`period`	`P`	Period (0-1023)
`uint16`	`compare`	`C`	Compare (0-1023, must not be greater than `period`)

Response Parameters:

None.

Related Commands:

- gpio_get_pwm_mode (GPWM, ID=9/12)

7.2.5.3 gpio_get_pwm_mode (GPWM, ID=9/12)

Obtain current PWM output behavior for selected channel.

See Section 8.1 (GPIO Pin Map for Supported Modules) for a pin map table showing pin availability and default assignment.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`09`	`0C`	None.
RSP	`C0`	`09`	`09`	`0C`	None.

Text Info:

Text Name	Response Length	Category	Notes
`GPWM`	`0x0027`	GET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`channel`	`N`*	Channel number (0 only)

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`enable`	`E`	Enable PWM output (0 to disable, 1 to enable)
`uint8`	`divider`	`D`	Clock divider value (24 MHz input): • Minimum = 0 (factory default) • Maximum = 255 • **NOTE:** Divider denominator is `divider+1`, so "`0`" is "divide by 1"

Data Type	Name	Text	Description
`uint8`	`prescaler`	`S`	PWM prescaler value: • 0 = 1x (no prescaling) • 1 = 2x • 2 = 4x • 3 = 8x • 4 = 16x • 5 = 32x • 6 = 64x • 7 = 128x • **NOTE:** Factory default is 0 (1x, no prescaling)
`uint16`	`period`	`P`	Period (0-1023)
`uint16`	`compare`	`C`	Compare (0-1023, must not be greater than `period`)

Related Commands:

- gpio_set_pwm_mode (SPWM, ID=9/11)

7.2.6 CYSPP Group (ID=10)

CYSPP methods relate to the Cypress Serial Port Profile.

Commands within this group are listed below:

- p_cyspp_start (.CYSPPSTART, ID=10/2)
- p_cyspp_set_parameters (.CYSPPSP, ID=10/3)
- p_cyspp_get_parameters (.CYSPPGP, ID=10/4)

Events within this group are documented in Section 7.3.6 CYSPP Group (ID=10).

You can find further details and examples concerning CYSPP operation here:

- Section 2.4.3 (Using CYSPP Mode)
- Section 3.1.5.2 (Configuring the CYSPP Data Mode Sleep Level)
- Section 3.2 (How to Perform a Factory Reset)

7.2.6.1 p_cyspp_start (.CYSPPSTART, ID=10/2)

Activate CYSPP operation.

Use this command to start CYSPP via the API protocol, rather than asserting the **CYSPP** pin or configuring automatic start with the p_cyspp_set_parameters (.CYSPPSP, ID=10/3) API command.

Refer to Section 2.4.3.8 (CYSPP State Machine) for details about how CYSPP moves between different operational states.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`00`	`0A`	`02`	None.
RSP	`C0`	`02`	`0A`	`02`	None.

Text Info:

Text Name	Response Length	Category	Notes
`.CYSPPSTART`	`0x0011`	ACTION	None.

Command Arguments:

None.

Response Parameters:

None.

Related Commands:

- p_cyspp_set_parameters (.CYSPPSP, ID=10/3)

Related Events:

- p_cyspp_status (.CYSPP, ID=10/1)

7.2.6.2 p_cyspp_set_parameters (.CYSPPSP, ID=10/3)

Configure new CYSPP behavior settings.

Use this command to control how CYSPP behaves. You can find example usage and practical explanations of how these settings affect behavior in Section 2.4.3 (Using CYSPP Mode) and Section 3.2 (Cable Replacement Examples with CYSPP).

NOTE: Disabling CYSPP with this API method causes EZ-Serial to hide the relevant GATT database attributes from Client discovery. All other visible attributes remain the same and keep their original handles, but those inside the CYSPP attribute range are hidden and are unusable by connected Clients. This remains in effect until you enable the profile again or assert the **CYSPP** pin.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`13`	`0A`	`03`	None.
RSP	`C0`	`02`	`0A`	`03`	None.

Text Info:

Text Name	Response Length	Category	Notes
`.CYSPPSP`	`0x000E`	SET	None.

Command Arguments:

Data Type	Name	Text	Description
`uint8`	`enable`	`E`	Enable CYSPP profile: • 0 = Disable • 1 = Enable • 2 = Enable + auto-start (factory default)
`uint8`	`role`	`G`	GAP role to use: • 0 = Peripheral/Server (factory default) •
`uint16`	`company`	`C`	Company ID value for automatic advertisement payload Manufacturer Data: • **NOTE:** Factory default is 0x0131 (Cypress Semiconductor)
`uint32`	`local_key`	`L`	Local connection key to present while advertising (peripheral role)
`uint32`	`remote_key`	`R`	Remote connection key to search for while scanning (central role) – not applicable on EZ-BLE WICED platform
`uint32`	`remote_mask`	`M`	Bitmask for bits in remote key which must match for a central-role connection – not applicable on EZ-BLE WICED platform
`uint8`	`sleep_level`	`P`	Maximum sleep level while connected with open CYSPP data pipe: • 0 = Sleep disabled • 1 = Sleep when possible (factory default) • **NOTE:** System-wide sleep overrides this if it is set to a lower level
`uint8`	`server_security`	`S`	CYSPP Server security requirement to allow writing CYSPP data from a Client: • 0 = No security required • 1 = Encryption required • 2 = Bonding required • 3 = Encryption and bonding required

Data Type	Name	Text	Description
`uint8`	`client_flags`	`F`	Client GATT usage flags while operating CYSPP in the central role • Bit 0 (0x01) = Use acknowledged data transfers • Bit 1 (0x02) = Enable CYSPP RX flow control **NOTE:** Factory default is 0x02 (RX flow only)

Response Parameters:

None.

Related Commands:

- p_cyspp_start (.CYSPPSTART, ID=10/2)
- p_cyspp_get_parameters (.CYSPPGP, ID=10/4)

Related Events:

- gap_adv_state_changed (ASC, ID=4/2) – May occur if CYSPP is set to start automatically in peripheral role
- p_cyspp_status (.CYSPP, ID=10/1)

Example Usage:

- Section 2.4.3 (Using CYSPP Mode)
- Section 3.1.5.2 (Configuring the CYSPP Data Mode Sleep Level)
- Section 3.2 (Cable Replacement Examples with CYSPP)

7.2.6.3 p_cyspp_get_parameters (.CYSPPGP, ID=10/4)

Obtain current CYSPP behavior settings.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	`C0`	`01`	`0A`	`04`	None.
RSP	`C0`	`15`	`0A`	`04`	None.

Text Info:

Text Name	Response Length	Category	Notes
`.CYSPPGP`	`0x004F`	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`enable`	`E`	Enable CYSPP profile: • 0 = Disable • 1 = Enable • 2 = Enable + auto-start (factory default)
`uint8`	`role`	`G`	GAP role to use: • 0 = Peripheral/Server (factory default) • 1 = Central/Client
`uint16`	`company`	`C`	Company ID value for automatic advertisement packet payload Manufacturer Data: • **NOTE:** Factory default is 0x0131 (Cypress Semiconductor)
`uint32`	`local_key`	`L`	Local connection key to present while advertising (peripheral role)
`uint32`	`remote_key`	`R`	Remote connection key to search for while scanning (central role)
`uint32`	`remote_mask`	`M`	Bitmask for bits in remote key which must match for a central-role connection
`uint8`	`sleep_level`	`P`	Maximum sleep level while connected with open CYSPP data pipe: • 0 = Sleep disabled

Data Type	Name	Text	Description
			• 1 = Normal sleep when possible • **NOTE:** System-wide sleep overrides this if it is set to a lower level
`uint8`	`server_security`	`S`	CYSPP Server security requirement for writing CYSPP data from a Client: • 0 = No security required • 1 = Encryption required • 2 = Bonding required • 3 = Encryption and bonding required
`uint8`	`client_flags`	`F`	Client GATT usage flags while operating CYSPP in the Central role • Bit 0 (0x01) = Use acknowledged data transfers • Bit 1 (0x02) = Enable CYSPP RX flow control **NOTE:** Factory default is 0x02 (RX flow only)

Related Commands:

- p_cyspp_set_parameters (.CYSPPSP, ID=10/3)

7.2.6.4 p_cyspp_set_packetization (.CYSPPSK, ID=10/7)

Control how incoming serial data from an external host is packetized for CYSPP transmission.

Use this command to control whether or how incoming serial data is assembled into specific packets for transmission to the remote peer over a CYSPP connection. Packetization does not affect the content or ordering of serial data in any way, but only affects certain buffering and transmission timing.

NOTE: CYSPP packetization does not affect any *outgoing* UART serial data (module-to-host), nor does it affect incoming serial data while in command mode (i.e., the CYSPP data pipe is not open). It impacts *only* the incoming serial data while CYSPP data mode is active.

At 115200 baud, a single byte takes about 80 microseconds to transfer. EZ-Serial checks for new bytes at least every 20 microseconds and processes whatever is available. Because of this, a continuous serial byte stream from an external host may be delivered to a remote CYSPP peer with multiple GATT transfers even if all of the data could fit in a single packet (e.g., two bytes sent as two single-byte transfers). Although the data is always delivered completely and in the correct order, this results in potentially unnecessary complexity on the receiving end, which must buffer and combine incoming data if it does not handle it as a continuous data stream.

To address this behavior, EZ-Serial provides this API command to control incoming data packetization. There are five different modes:

- **Mode 0: Immediate**

  This mode reads and transmits data as quickly as possible, always sending as much data as is available as soon as the BLE stack allows a new transmission**.** In this mode, the first byte or two bytes of a new transmission are usually sent in a single packet even if more data is arriving at the same time.

  The *[wait]* and *[length]* settings are irrelevant in this mode.

- **Mode 1: Anticipate (factory default with 5 ms wait and 20 byte length)**

  This mode waits up to *[wait]* milliseconds in anticipation for at least *[length]* bytes to arrive from the external host. If the target byte count is reached before the wait time expires, all available bytes are transmitted immediately. If the configured wait time expires before reaching the target byte count, all available bytes are transmitted at that time. Anticipate mode is suitable for most general operations and does not negatively impact the throughput if the incoming serial data arrives fast enough to keep the UART receive buffer full.

  The *[wait]* setting must be between 1 and 255. The *[length]* setting must be between 1 and 128, which is the internal UART RX software buffer size.

- **Mode 2: Fixed**

  This mode waits indefinitely until at least *[length]* bytes have been read, then transmits exactly that many bytes. Fixed mode is best used in cases where the host sends chunks of data which are always of the same size. Setting a *[length]* value that is larger than the GATT MTU payload size results in multiple transmissions once all data has been buffered. For example, a fixed packet length of 32 bytes with the default GATT MTU size of 23 bytes (usable payload size of 20 bytes) results in one 20-byte packet followed by one 12-byte packet. The MTU depends on the value negotiated by the Client after connection.

The *[length]* setting must be between 1 and 128, which is the internal UART RX software buffer size. The *[wait]* setting is irrelevant in this mode.

- **Mode 3: Variable**

  This mode requires an additional *length* value from the host before each packet to indicate how many bytes to expect. EZ-Serial consumes this byte (it is *not* transmitted to the remote peer), and then waits until exactly that many bytes to have been read before transmitting them. Variable mode is suitable for applications that require packets of differing lengths and which can accommodate an extra transmitted byte from the host indicating each packet's length.

  For example, the host can send [***04*** 61 62 63 64] to transmit the 4-byte ASCII string "abcd" to the remote peer in a single packet. Or, the host can send [***05*** 61 62 63 64 65 ***03*** 66 67 68] to transmit "abcdefgh" in two packets ("abcde" followed by "def").

  The prefixed packet length byte must not be greater than 128. Values greater than this will be capped at 128. The *[wait]* and *[length]* settings are irrelevant in this mode.

- **Mode 4: End-of-packet**

  This mode buffers the data until the configured end-of-packet byte is encountered in the data stream, or until either the MTU payload size or UART RX buffer has filled. End-of-packet (EOP) mode allows variable-length packets without knowing in advance how long the packet will be.

  The EOP byte defaults to 0x0D (the carriage return byte, often expressed as '`\r`' in code). However, you can change it to any value between 0x00 and 0xFF. When the EOP byte occurs in the data stream, all buffered data up to that point **including the EOP byte itself** will be transmitted to the remote side.

  In this mode, EZ-Serial will also transmit buffered data under two other conditions:

  1. If the GATT MTU payload size is less than the UART RX buffer size (128 bytes) and enough data is buffered to fill a single GATT packet, one packet's worth of data is transmitted. The default GATT MTU is 23 bytes with a usable payload size of 20 bytes.
  2. If the GATT MTU payload size is greater than the UART RX buffer size (128 bytes) and the RX buffer is full, 128 bytes of data are transmitted. This can only occur in cases where the connected Client has negotiated a GATT MTU greater than 131 bytes (actual transmit payload is MTU - 3 bytes).

For the "Anticipate" mode (1), you must consider the UART baud rate when choosing the *[wait]* and *[length]* values. A 5-ms wait time is suitable for a 20-byte target length at 115200 baud, but this is not enough time to read in 20 bytes at 9600 baud (for example). If you change the baud rate, be sure to choose a *[wait]* value that allows the target packet length to be filled under normal operating conditions. Table 7-3 below contains "safe" wait values for 20-byte packets at common baud rates for reference.

Table 7-3. Common UART Timing for 20-Byte Packets

Baud Rate	Single Bit Duration	20 Bytes at 8/N/1 (200 Bits)	Safe Wait Value Example
9600	104 us	~21 ms	32 ms (0x20)
38400	26.1 us	~5.2 ms	10 ms (0x0A)
57600	17.4 us	~3.5 ms	5 ms (0x05)
115200	8.68 us	~1.7 ms	5 ms (0x05)
230400	4.34 us	868 us	2 ms (0x02)
460800	2.17 us	434 us	1 ms (0x01)
921600	1.09 us	217 us	1 ms (0x01)

The single-bit duration for any baud rate can be calculated in microseconds using this equation:

Bit time = 1,000,000 us / *[baud]*

Standard UART settings of 8 data bits, no parity, and 1 stop bit yield a total of 10 bits per byte. For a 20-byte packet, this requires allowance for 200 bits.

NOTE: If the packet length used in Anticipate, Fixed, Variable, or End-of-Packet modes exceeds the GATT MTU usable payload size (20 bytes on many platforms), the packets are broken apart to fit within this lower-level constraint. For example, using Fixed mode with *[length]* set to 32 bytes results in two transmitted packets each time the target length is reached: first a 20-byte packet and then a 12-byte packet.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	04	0A	07	None.
RSP	C0	02	0A	07	None.

Text Info:

Text Name	Response Length	Category	Notes
.CYSPPSK	0x000E	SET	None.

Command Arguments:

Data Type	Name	Text	Description
uint8	mode	M	Packetization mode: • 0 = Immediate: transmit incoming data as soon as possible • 1 = Anticipate: wait a short time to attempt a minimum buffer threshold • 2 = Fixed: buffer and send packets of exactly one size • 3 = Variable: specify the size of every packet with a prefixed length byte • 4 = End-of-packet: transmit data when specific byte occurs in stream • **NOTE:** Factory default is 1 (Anticipate)
uint8	wait	W	Anticipation delay (milliseconds), used only in “Anticipate” mode: • Minimum = 0x01 (1 millisecond) • Maximum = 0x80 (128 bytes) • **NOTE:** Factory default is 0x5 (5 milliseconds)
uint8	length	L	Fixed/anticipated packet length (bytes), used only in “Anticipate” or “Fixed” mode: • Minimum = 0x01 (1 byte) • Maximum = 0x80 (128 bytes) • **NOTE:** Factory default is 0x14 (20 bytes, standard GATT MTU)
uint8	eop	E	End-of-packet byte: • **NOTE:** Factory default is 0x0D (‘\r’ carriage return)

Response Parameters:

None.

Related Commands:

- p_cyspp_get_packetization (.CYSPPGK, ID=10/8)

7.2.6.5 p_cyspp_get_packetization (.CYSPPGK, ID=10/8)

Obtain current CYSPP packetization settings.

Binary Header:

	Type	Length	Group	ID	Notes
CMD	C0	00	0A	08	None.
RSP	C0	05	0A	08	None.

Text Info:

Text Name	Response Length	Category	Notes
.CYSPPGK	0x001D	GET	None.

Command Arguments:

None.

Response Parameters:

Data Type	Name	Text	Description
`uint8`	`mode`	`M`	Packetization mode: • 0 = Immediate: transmit incoming data as soon as possible • 1 = Anticipate: wait a short time to attempt a minimum buffer threshold • 2 = Fixed: buffer and send packets of exactly one size • 3 = Variable: specify the size of every packet with a prefixed length byte • 4 = End-of-packet: transmit data when specific byte occurs in stream • **NOTE:** Factory default is 1 (Anticipate)
`uint8`	`wait`	`W`	Anticipation delay (milliseconds), used only in "Anticipate" mode: • Minimum = 0x01 (1 millisecond) • Maximum = 0x80 (128 bytes) • **NOTE:** Factory default is 0x5 (5 milliseconds)
`uint8`	`length`	`L`	Fixed/anticipated packet length (bytes), used only in "Anticipate" and "Fixed" modes: • Minimum = 0x01 (1 byte) • Maximum = 0x80 (128 bytes) • **NOTE:** Factory default is 0x14 (20 bytes, standard GATT MTU)
`uint8`	`eop`	`E`	End-of-packet byte: • **NOTE:** Factory default is 0x0D ('\r' carriage return)

Related Commands:

- p_cyspp_set_packetization (.CYSPPSK, ID=10/7)

7.3 API Events

All events implemented in the API protocol are described in detail below. API commands and responses are documented separately in Section 7.2 (API Commands and Responses).

A master list of all possible error codes appearing in certain events can be found in Section 7.4 (Error Codes).

Commands and responses are broken down into the following groups:

- System Group (ID=2)
- GAP Group (ID=4)
- GATT Server Group (ID=5)
- SMP Group (ID=7)
- GPIO Group (ID=9)
- CYSPP Group (ID=10)

7.3.1 System Group (ID=2)

System methods relate to the core device, describing things like boot, device address info, and resetting to an initial state.

Events within this group are listed below:

- system_boot (BOOT, ID=2/1)
- system_error (ERR, ID=2/2)
-
- system_factory_reset_complete (RFAC, ID=2/3)
- system_dump_blob (DBLOB, ID=2/5)

Commands within this group are documented in Section 7.2.1, System Group (ID=2).

7.3.1.1 system_boot (BOOT, ID=2/1)

EZ-Serial module has booted and is ready to process commands.

Binary Header:

Type	Length	Group	ID	Notes
80	12	02	01	None.

Text Info:

Text Name	Event Length	Notes
BOOT	0x003B	None.

Event Parameters:

Data Type	Name	Text	Description
uint32	app	E	Application version number
uint32	stack	S	BLE stack version number
uint16	protocol	P	API protocol version number
uint8	hardware	H	Hardware identifier: • 0x01 = CYBLE-01201X-X0 • 0x02 = CYBLE-014008-00 • 0x03 = CYBLE-022001-00 • 0x04 = CYBLE-2X20XX-X1 • 0x05 = CYBLE-2120XX-X0 • 0x06 = CYBLE-212020-01 • 0x07 = CYBLE-214009-00 • 0x08 = CYBLE-214015-01 • 0x09 = CYBLE-222005-00 • 0x0A = CYBLE-222014-01 • 0x0B = CYBLE-224110-00 • 0x0C = CYBLE-224116-01 • 0xB1 = CYBLE-013025-00
uint8	cause	C	Cause of boot event: always 0
macaddr	address	A	Public Bluetooth address

Related Commands:

- system_reboot (/RBT, ID=2/2)
- system_factory_reset (/RFAC, ID=2/5)

7.3.1.2 system_error (ERR, ID=2/2)

System error has occurred.

This may be triggered by a malformed command, an operation that failed or could start due to an invalid operational state, or a low-level hardware failure. See Section 7.4 (Error Codes) for a list of all possible errors.

Binary Header:

Type	Length	Group	ID	Notes
80	02	02	02	None.

Text Info:

Text Name	Event Length	Notes
ERR	0x000B	None.

Event Parameters:

Data Type	Name	Text	Description
`uint16`	`error`	`E`	Error code describing what went wrong

7.3.1.3 *system_factory_reset_complete (RFAC, ID=2/3)*

Factory reset complete.

This event will occur after sending the system_factory_reset (/RFAC, ID=2/5) API command, or asserting (LOW) the **FACTORY_TR** and **CYSPP** pins at boot time. EZ-Serial transmits this event using the originally configured host interface settings (if different from the default). After generating this event, the module reboots immediately and the default settings take effect.

> **NOTE:** If you triggered a factory reset using the GPIO method at boot time, the final reboot back into an operational state occurs only after you deassert one or both of the pins. This safeguard prevents an endless loop of factory resets if both pins remain asserted.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`00`	`02`	`03`	None.

Text Info:

Text Name	Event Length	Notes
`RFAC`	`0x0005`	None.

Event Parameters:

None.

Related Commands:

- system_factory_reset (/RFAC, ID=2/5)

7.3.1.4 *system_dump_blob (DBLOB, ID=2/5)*

Single data blob of requested configuration type or system state.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`04-14`	`02`	`05`	Variable-length event payload, minimum of 4 (0x04), maximum of 20 (0x14).

Text Info:

Text Name	Event Length	Notes
`DBLOB`	`0x0015-0x0035`	Variable-length event payload, minimum of 21 (0x15), maximum of 53 (0x35)

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`type`	`T`	Type of information being dumped: • 0 = Runtime configuration data • 1 = Boot-level configuration data • 2 = Factory-level configuration data • 3 = System state data
`uint16`	`offset`	`O`	Blob start offset
`uint8a`	`data`	`D`	Dumped blob of data **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload

Related Commands:

- system_dump (/DUMP, ID=2/3)

7.3.2 GAP Group (ID=4)

GAP methods relate to the Generic Access Protocol layer of the Bluetooth stack, which includes management of scanning, advertising, connection establishment, and connection maintenance.

Events within this group are listed below:

- gap_whitelist_entry (WL, ID=4/1)
- gap_adv_state_changed (ASC, ID=4/2)
- gap_connected (C, ID=4/5)
- gap_disconnected (DIS, ID=4/6)7.3.2.5

Commands within this group are documented in Section 7.3.2, GAP Group (ID=4).

7.3.2.1 gap_whitelist_entry (WL, ID=4/1)

Details about a single entry in the whitelist table.

Binary Header:

Type	Length	Group	ID	Notes
80	07	04	01	None.

Text Info:

Text Name	Event Length	Notes
WL	0x0017	None.

Event Parameters:

Data Type	Name	Text	Description
macaddr	address	A	Bluetooth address
uint8	type	T	Address type: • 0 = Public • 1 = Random/private

Related Commands:

- gap_add_whitelist_entry (/WLA, ID=4/6)
- gap_query_whitelist (/QWL, ID=4/14)

7.3.2.2 gap_adv_state_changed (ASC, ID=4/2)

Indicates that the module has started or stopped advertising, due to a scheduled timeout, automated process, or intentional action.

Binary Header:

Type	Length	Group	ID	Notes
80	02	04	02	None.

Text Info:

Text Name	Event Length	Notes
ASC	0x000E	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`state`	`S`	Advertising state: • 0 = Stopped • 1 = Active
`uint8`	`reason`	`R`	Reason for state change: • 0 = User command • 1 = GAP automatic advertisement enabled • 2 = Configured timeout expired • 3 = CYSPP operation state change • 6 = Disconnection

Related Commands:

- gap_start_adv (/A, ID=4/8)
- gap_stop_adv (/AX, ID=4/9)
- gap_set_adv_parameters (SAP, ID=4/23)
- p_cyspp_start (.CYSPPSTART, ID=10/2)
- p_cyspp_set_parameters (.CYSPPSP, ID=10/3)

7.3.2.3 gap_connected (C, ID=4/5)

Connection established with a remote device.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`0F`	`04`	`05`	None.

Text Info:

Text Name	Event Length	Notes
`C`	`0x0035`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Connection handle
`macaddr`	`address`	`A`	Bluetooth address
`uint8`	`type`	`T`	Address type: • 0 = Public • 1 = Random/private
`uint16`	`interval`	`I`	Connection interval
`uint16`	`slave_latency`	`L`	Slave latency
`uint16`	`supervision_timeout`	`O`	Supervision timeout
`uint8`	`bond`	`B`	Bond entry (0 for no bond)

Related Commands:

- gap_update_conn_parameters (/UCP, ID=4/3)
- gap_disconnect (/DIS, ID=4/5)

Related Events:

- gap_disconnected (DIS, ID=4/6)

7.3.2.4 gap_disconnected (DIS, ID=4/6)

Connection to a remote device has closed.

For a list of possible disconnection reasons, see the 0x900 range of codes in Section 7.4.1 (EZ-Serial System Error Codes). These are the most common reasons:

- 0x0908 – Page timeout (unexpected loss of connectivity, no response within supervision timeout)
- 0x0913 – Remote user terminated connection (cleanly closed from remote side)
- 0x0916 – Connection terminated by local host (cleanly closed from local side)
- 0x093E – Connection failed to be established (connection initiated locally, but peer did not respond to request)

Binary Header:

Type	Length	Group	ID	Notes
`80`	`03`	`04`	`06`	None.

Text Info:

Text Name	Event Length	Notes
`DIS`	`0x0010`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Connection handle
`uint16`	`reason`	`R`	Reason for disconnection

Related Commands:

- gap_disconnect (/DIS, ID=4/5)

7.3.2.5 gap_connection_updated (CU, ID=4/8)

Active connection has negotiated and applied new parameters.

This event occurs on the slave side after a master requests new parameters or accepts the new parameters requested by the slave. It also occurs on the master side after a slave requests new parameters and the master accepts the request.

NOTE: A connection update request sent from a slave but rejected does not result in any events indicating the rejection. The slave must assume the original parameters are in effect until after it receives this API event.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`07`	`04`	`08`	None.

Text Info:

Text Name	Event Length	Notes
`CU`	`0x001D`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Connection handle
`uint16`	`interval`	`I`	Connection interval
`uint16`	`slave_latency`	`L`	Slave latency
`uint16`	`supervision_timeout`	`O`	Supervision timeout

Related Commands:

- gap_update_conn_parameters (/UCP, ID=4/3)

7.3.3 GATT Server Group (ID=5)

GATT Server methods relate to the Server role of the Generic Attribute Protocol layer of the Bluetooth stack. These methods are used for working with the local GATT structure.

Events within this group are listed below:

- gatts_discover_result (DL, ID=5/1)
- gatts_data_written (W, ID=5/2)
- gatts_indication_confirmed (IC, ID=5/3)
- gatts_db_entry_blob (DGATT, ID=5/4)

Commands within this group are documented in Section 7.2.3

7.3.3.1 gatts_discover_result (DL, ID=5/1)

Details about a single entry in the local GATT database.

This event occurs while discovering local services, characteristics, or descriptors.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`08+`	`05`	`01`	Variable-length event payload, value specified is minimum.

Text Info:

Text Name	Event Length	Notes
`DL`	`0x0020+`	Variable-length event payload, value specified is minimum.

Event Parameters:

Data Type	Name	Text	Description
`uint16`	`attr_handle`	`H`	Attribute handle
`uint16`	`attr_handle_rel`	`R`	Related attribute handle: • If discovering services, the **end handle** for the service group • If discovering characteristics, the **value handle** that holds the application data • If discovering descriptors, always 0 (not applicable)
`uint16`	`type`	`T`	Attribute type: • 0x2800 = Primary Service Declaration • 0x2801 = Secondary Service Declaration • 0x2802 = Include Declaration • 0x2803 = Characteristic Declaration • 0x2900 = Characteristic Extended Properties Descriptor • 0x2901 = Characteristic User Description Descriptor • 0x2902 = Client Characteristic Configuration Descriptor • 0x2903 = Server Characteristic Configuration Descriptor • 0x2904 = Characteristic Format Descriptor • 0x2905 = Characteristic Aggregate Format Descriptor • 0x0000 = Characteristic value attribute or user-defined structure (see UUID)

Data Type	Name	Text	Description
`uint8`	`properties`	`P`	Characteristic properties bitmask, only non-zero during **characteristic** discovery: • Bit 0 (0x01) = Broadcast • Bit 1 (0x02) = Read • Bit 2 (0x04) = Write without response • Bit 3 (0x08) = Write • Bit 4 (0x10) = Notify • Bit 5 (0x20) = Indicate • Bit 6 (0x40) = Signed write • Bit 7 (0x80) = Extended properties (will have 0x2900 descriptor)
`uint8a`	`uuid`	`U`	UUID **NOTE:** `uint8a` data type requires one prefixed "length" byte before binary parameter payload.

Related Commands:

- gatts_discover_services (/DLS, ID=5/6)
- gatts_discover_characteristics (/DLC, ID=5/7)
- gatts_discover_descriptors (/DLD, ID=5/8)

7.3.3.2 gatts_data_written (W, ID=5/2)

Remote GATT Client has written data to a local attribute.

A connected remote Client can write data to a local attribute using either acknowledged unacknowledged write operations Acknowledged writes require two full connection intervals to complete: one for the data transfer from Client to Server, and one for the acknowledgement back from Server to Client. Unacknowledged writes may occur multiple times within the same connection interval, and therefore provide greater throughput potential.

EZ-Serial automatically responds to acknowledged writes except in two cases:

- You have disabled automatic responses using the gatts_set_parameters (SGSP, ID=5/14) API command.
- The attribute written to has the "User data management" bit set in its properties value, set during creation with the gatts_create_attr (/CAC, ID=5/1) API command.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`06`	`05`	`02`	Variable-length event payload, value specified is minimum.

Text Info:

Text Name	Event Length	Notes
`W`	`0x0016+`	Variable-length event payload, value specified is minimum.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Handle of connection from which write came
`uint16`	`attr_handle`	`H`	Attribute handle
`uint8`	`type`	`T`	Write type: • 0x00 = Simple write – acknowledged • 0x01 = Write without response – unacknowledged • 0x80 = Simple write requiring manual response via API command
`longuint8a`	`data`	`D`	Written data **NOTE:** `longuint8a` data type requires two prefixed "length" bytes before binary parameter payload.

Related Commands:

None.

7.3.3.3 *gatts_indication_confirmed (IC, ID=5/3)*

Remote GATT Client has confirmed receipt of indicated data.

This event occurs after a Client receives and confirms data pushed using the gatts_indicate_handle (/IH, ID=5/12) API command.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`03`	`05`	`03`	None.

Text Info:

Text Name	Event Length	Notes
`IC`	`0x000F`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Handle of connection from which confirmation came
`uint16`	`attr_handle`	`H`	Attribute handle use for indication

Related Commands:

- gatts_indicate_handle (/IH, ID=5/12)

Related Events:

None.

7.3.3.4 *gatts_db_entry_blob (DGATT, ID=5/4)*

Single entry from the GATT structure definition.

This event presents local dynamic GATT attribute definition in a format which simplifies reentry using the gatts_create_attr (/CAC, ID=5/1) API command. For details about the data provided in this event, see Section 3.4.1 (How to Define Custom Local GATT Services and Characteristics).

NOTE: This event includes the attribute handle and the absolute group end value, neither of which is part of the data entered when creating a new custom attribute. Be sure to remove the handle and absolute group end if you are directly copying the content from these output lines into new commands manually.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`10-20`	`05`	`04`	Variable-length event payload, minimum of 16 (0x10), maximum of 32 (0x20)

Text Info:

Text Name	Event Length	Notes
`DGATT`	`0x0037-0x0057`	Variable-length event payload, minimum of 55 (0x37), maximum of 87 (0x57)

Event Parameters:

Data Type	Name	Text	Description
`uint16`	`handle`	`H`	Attribute handle (0x0001 – 0xFFFF)
`uint16`	`type`	`T`*	• 0 = structure • 1 = characteristic value

Data Type	Name	Text	Description
`uint8`	`perm`	`R`*	Permission bits: • Bit 0 (0x01) = Variable length • Bit 1 (0x02) = Readable • Bit 2 (0x04) = Write command (unacknowledged) • Bit 3 (0x08) = Write request (acknowledged) • Bit 4 (0x10) = Perm auth readable • Bit 5 (0x20) = Reliable write (includes prepared write) • Bit 6 (0x40) = Authenticated writable • Bit 7 (0x80) = UUID is 128 bits
`uint16`	`length`	`L`	Indicates how many bytes of RAM are allocated for the definition (structure) or content (characteristic value)
`longuint8a`	`data`	`U`	Data (UUID or default attribute value where applicable) **NOTE:** `longuint8a` data type requires two prefixed “length” bytes before binary parameter payload.

Related Commands:

- gatts_dump_db (/DGDB, ID=5/5)

7.3.4 SMP Group (ID=7)

SMP methods relate to the Security Manager Protocol layer of the Bluetooth stack. These methods are used for working with encryption, pairing, and bonding between two peers.

Events within this group are listed below:

- smp_bond_entry (B, ID=7/1)
- smp_pairing_requested (P, ID=7/2)
- smp_pairing_result (PR, ID=7/3)
- smp_encryption_status (ENC, ID=7/4)

Commands within this group are documented in Section 7.2.4 SMP Group (ID=7).

7.3.4.1 smp_bond_entry (B, ID=7/1)

Details about a single entry in the bonding table.

This event occurs once after a new bond is created as a result of the pairing process, or multiple times (based on bond list count) after requesting the bond list with the smp_query_bonds (/QB, ID=7/1) API command.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`07`	`07`	`01`	None.

Text Info:

Text Name	Event Length	Notes
`B`	`0x001B`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`handle`	`B`	Bonded device handle (1-4)
`macaddr`	`address`	`A`	Bluetooth address
`uint8`	`type`	`T`	Address type: • 0 = Public • 1 = Random/private

Related Commands:

- smp_query_bonds (/QB, ID=7/1)
- smp_pair (/P, ID=7/3)

7.3.4.2 smp_pairing_requested (P, ID=7/2)

Remote device has requested pairing.

When this event occurs, you must use the smp_send_pairreq_response (/PR, ID=7/5) API command to continue the process, unless the **auto-accept** bit is set in the **`flags`** setting of the smp_set_security_parameters (SSBP, ID=7/11) API command.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`05`	`07`	`02`	None.

Text Info:

Text Name	Event Length	Notes
`P`	`0x0016`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`conn_handle`	`C`	Connection handle
`uint8`	`mode`	`M`	Security level setting reported to peer: • 0x10 = Mode 1, Level 1 – No security • 0x11 = Mode 1, Level 2 – Unauthenticated pairing with encryption (no MITM) • 0x12 = Mode 1, Level 3 – Authenticated pairing with encryption (with MITM) • 0x21 = Mode 2, Level 2 – Unauthenticated pairing with data signing (no MITM) • 0x22 = Mode 2, Level 3 – Authenticated pairing with data signing (with MITM)
`uint8`	`bonding`	`B`	Bond during pairing process: • 0 = Do not bond (exchange keys and encrypt only) • 1 = Bond (permanently store exchanged encryption data)
`uint8`	`keysize`	`K`	Encryption key size (7-16), value ignored if pairing initiated by slave device
`uint8`	`pairprop`	`P`	Pairing properties: • Bit 0 (0x01): MITM enabled for Secure Connections (SC)

Related Commands:

- smp_pair (/P, ID=7/3)
- smp_send_pairreq_response (/PR, ID=7/5)
- smp_set_security_parameters (SSBP, ID=7/11)

Related Events:

- smp_pairing_result (PR, ID=7/3)

7.3.4.3 smp_pairing_result (PR, ID=7/3)

Pairing process has ended.

This event indicates that the pairing process is finished, successfully or otherwise. If the **`result`** parameter is 0, then pairing has completed successfully, and the smp_bond_entry (B, ID=7/1) API event follows if bonding is enabled. Any non-zero **`result`** value indicates failure.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`03`	`07`	`03`	None.

Text Info:

Text Name	Event Length	Notes
PR	0x000C	None.

Event Parameters:

Data Type	Name	Text	Description
uint8	conn_handle	C	Connection handle
uint16	result	R	Result

Related Commands:

- smp_pair (/P, ID=7/3)

Related Events:

- smp_encryption_status (ENC, ID=7/4)
- smp_bond_entry (B, ID=7/1)

7.3.4.4 smp_encryption_status (ENC, ID=7/4)

Encryption status has changed.

This event confirms that a link has transitioned between plaintext and encrypted status during the pairing process.

Binary Header:

Type	Length	Group	ID	Notes
80	02	07	04	None.

Text Info:

Text Name	Event Length	Notes
ENC	0x000E	None.

Event Parameters:

Data Type	Name	Text	Description
uint8	conn_handle	C	Connection handle
uint8	status	S	Encryption status: • 0 = Not encrypted • 1 = Encrypted

Related Commands:

- smp_pair (/P, ID=7/3)

Related Events:

- smp_pairing_result (PR, ID=7/3)

7.3.5 GPIO Group (ID=9)

GPIO methods relate to the physical pins on the module.

There are no GPIO-specific events supported in EZ-Serial for EZ-BLE WICED modules. Commands within this group are documented in Section 7.2.5, GPIO Group (ID=9).

7.3.6 CYSPP Group (ID=10)

CYSPP methods relate to the Cypress Serial Port Profile.

Events within this group are listed below:

- p_cyspp_status (.CYSPP, ID=10/1)

Commands within this group are documented in Section 7.2.6, CYSPP Group (ID=10).

7.3.6.1 p_cyspp_status (.CYSPP, ID=10/1)

CYSPP operational status has changed.

NOTE: If this event occurs within EZ-Serial and data mode is active (either Bit 0 or Bit 1 set and the CYSPP GPIO pin is not externally deasserted), the wired serial interface is logically disconnected from the API protocol parser and routed to CYSPP data pipe instead. For this reason, this event is never transmitted out the serial interface with Bit 5 set (0x20), because outgoing API events are suppressed while operating in CYSPP data mode.

Binary Header:

Type	Length	Group	ID	Notes
`80`	`01`	`0A`	`01`	None.

Text Info:

Text Name	Event Length	Notes
`.CYSPP`	`0x000C`	None.

Event Parameters:

Data Type	Name	Text	Description
`uint8`	`status`	`S`	CYSPP status bitmask: • Bit 0 (0x01) = Unacknowledged data subscribed • Bit 1 (0x02) = Acknowledged data subscribed • Bit 2 (0x04) = RX flow subscribed • Bit 3 (0x08) = RX flow blocked by remote Server • Bit 4 (0x10) = CYSPP peer support verified • Bit 5 (0x20) = Data mode active *(used internally)*

Related Commands:

- p_cyspp_start (.CYSPPSTART, ID=10/2)
- p_cyspp_set_parameters (.CYSPPSP, ID=10/3)

Example Usage:

Section 3.2 (Cable Replacement Examples with CYSPP)

7.4 Error Codes

7.4.1 EZ-Serial System Error Codes

The complete list of all result/error codes generated by EZ-Serial is contained in the table below. See the command and event reference in Section 7.2 (API Commands and Responses) and Section 7.3 (API Events) for specific details about each result within the context of the responses and events where they are triggered.

Table 7-4. EZ-Serial System Error Codes

Code (Hex)	Name	Description
0000	**EZS_ERR_SUCCESS**	**Operation successful, no error**
0100	**EZS_ERR_CORE**	**Core system error category**
0101	EZS_ERR_CORE_NULL_POINTER	Null pointer encountered *(internal error)*
0102	EZS_ERR_CORE_MALLOC_FAILED	Memory allocation failed *(internal error)*
0103	EZS_ERR_CORE_BUFFER_OVERFLOW	Buffer overflow *(internal error)*
0104	EZS_ERR_CORE_FEATURE_NOT_IMPLEMENTED	Unsupported feature *(internal error)*
0105	EZS_ERR_CORE_TASK_SCHEDULE_OVERFLOW	Task scheduling attempted but schedule is full
0106	EZS_ERR_CORE_TASK_QUEUE_OVERFLOW	Task queue attempted but queue is full
0107	EZS_ERR_CORE_INVALID_STATE	Invalid state for requested operation
0108	EZS_ERR_CORE_OPERATION_NOT_PERMITTED	Operation not permitted
0109	EZS_ERR_CORE_INSUFFICIENT_RESOURCES	Insufficient resources for requested action
010A	EZS_ERR_CORE_FLASH_WRITE_NOT_PERMITTED	Unable to perform flash write at this time
010B	EZS_ERR_CORE_FLASH_WRITE_FAILED	Flash write operation failed during write
010C	EZS_ERR_CORE_HARDWARE_FAILURE	Internal chipset hardware failure
010D	EZS_ERR_CORE_BLE_INITIALIZATION_FAILED	Could not initialize BLE stack
010E	EZS_ERR_CORE_REPEATED_ATTEMPTS	Repeated attempts to initialize BLE stack
010F	EZS_ERR_CORE_TX_POWER_READ	Could not read radio TX power
0110	EZS_ERR_CORE_DB_VERIFICATION_FAILED	Verification prevented custom attribute addition
0200	**EZS_ERR_PROTOCOL**	**Protocol error category**
0201	EZS_ERR_PROTOCOL_UNRECOGNIZED_PACKET_TYPE	Unsupported packet type for text parsing *(internal error)*
0202	EZS_ERR_PROTOCOL_UNRECOGNIZED_ARGUMENT_TYPE	Unsupported argument type for text parsing *(internal error)*
0203	EZS_ERR_PROTOCOL_UNRECOGNIZED_COMMAND	Command group/method not valid or unrecognized
0204	EZS_ERR_PROTOCOL_UNRECOGNIZED_RESPONSE	Response group/method invalid or unrecognized *(internal error)*
0205	EZS_ERR_PROTOCOL_UNRECOGNIZED_EVENT	Event group/method invalid or unrecognized *(internal error)*
0206	EZS_ERR_PROTOCOL_SYNTAX_ERROR	Syntax error while parsing text command
0207	EZS_ERR_PROTOCOL_COMMAND_TIMEOUT	Binary command packet transmission not completed in required time
0208	EZS_ERR_PROTOCOL_RESPONSE_PENDING	Command already sent but response still pending
0209	EZS_ERR_PROTOCOL_INVALID_CHECKSUM	Binary command packet has invalid checksum
020A	EZS_ERR_PROTOCOL_INVALID_COMMAND_LENGTH	Command length is greater than maximum
020B	EZS_ERR_PROTOCOL_INVALID_PARAMETER_COUNT	Incorrect number of parameters provided
020C	EZS_ERR_PROTOCOL_INVALID_PARAMETER_VALUE	Command parameter outside of acceptable range
020D	EZS_ERR_PROTOCOL_MISSING_REQUIRED_ARGUMENT	Text-mode command missing required arguments
020E	EZS_ERR_PROTOCOL_INVALID_HEXADECIMAL_DATA	Invalid hexadecimal data provided (not 0-9, A-F)
020F	EZS_ERR_PROTOCOL_INVALID_ESCAPE_SEQUENCE	Invalid escape sequence
0210	EZS_ERR_PROTOCOL_INVALID_MACRO_SEQUENCE	Invalid macro sequence
0211	EZS_ERR_PROTOCOL_FLASH_SETTINGS_PROTECTED	Attempted direct flash write of protected setting
0300	**EZS_ERR_GPIO**	**GPIO error category**
0301	EZS_ERR_GPIO_PORT_NOT_SUPPORTED	Selected port in GPIO command not supported

Code (Hex)	Name	Description
0400	**EZS_ERR_LL**	**Link layer error category**
0401	EZS_ERR_LL_CONTROLLER_BUSY	Link layer controller busy
0402	EZS_ERR_LL_NO_DEVICE_ENTITY	Device entity not available
0403	EZS_ERR_LL_NOT_IN_BOND_LIST	Device not found in bond list
0404	EZS_ERR_LL_DEVICE_ALREADY_EXISTS	Device already exists
0500	**EZS_ERR_GAP**	**GAP error category**
0501	EZS_ERR_GAP_INVALID_CONNECTION_HANDLE	Invalid connection handle specified
0502	EZS_ERR_GAP_CONNECTION_REQUIRED	Connection required, but none is available
0503	EZS_ERR_GAP_ROLE	Incorrect GAP role for this operation
0504	EZS_ERR_GAP_ADV_QUEUE_OVERFLOW	Advertisement queue attempted but queue is full
0600	**EZS_ERR_GATT**	**GATT error category**
0601	EZS_ERR_GATT_INVALID_ATTRIBUTE_HANDLE	Invalid attribute handle for GATT operation
0602	EZS_ERR_GATT_READ_NOT_PERMITTED	Read not permitted on this attribute
0603	EZS_ERR_GATT_WRITE_NOT_PERMITTED	Write not permitted on this attribute
0604	EZS_ERR_GATT_INVALID_PDU	Invalid PDU for requested operation
0605	EZS_ERR_GATT_INSUFFICIENT_AUTHENTICATION	Insufficient authentication for requested operation
0606	EZS_ERR_GATT_REQUEST_NOT_SUPPORTED	Request not supported
0607	EZS_ERR_GATT_INVALID_OFFSET	Invalid offset specified for requested operation
0608	EZS_ERR_GATT_INSUFFICIENT_AUTHORIZATION	Insufficient authorization for requested operation
0609	EZS_ERR_GATT_PREPARE_WRITE_QUEUE_FULL	Prepare write queue full, cannot prepare new write
060A	EZS_ERR_GATT_ATTRIBUTE_NOT_FOUND	Attribute not found in database
060B	EZS_ERR_GATT_ATTRIBUTE_NOT_LONG	Attribute not long when long operation requested
060C	EZS_ERR_GATT_INSUFFICIENT_ENC_KEY_SIZE	Insufficient encryption key size
060D	EZS_ERR_GATT_INVALID_ATTRIBUTE_LENGTH	Invalid attribute length
060E	EZS_ERR_GATT_UNLIKELY_ERROR	Unlikely error occurred, unknown cause
060F	EZS_ERR_GATT_INSUFFICIENT_ENCRYPTION	Insufficient encryption for requested operation
0610	EZS_ERR_GATT_UNSUPPORTED_GROUP_TYPE	Unsupported group type specified in Read By Group Type operation
0611	EZS_ERR_GATT_INSUFFICIENT_RESOURCES	Insufficient resources to perform operation
0680	EZS_ERR_GATT_CLIENT_NOT_SUBSCRIBED	Client has not subscribed to updates on characteristic (local error code when sending notifications or indications)
0700	**EZS_ERR_L2CAP**	**L2CAP error category**
0701	EZS_ERR_L2CAP_NOT_IN_BOND_LIST	Device not found in bond list
0702	EZS_ERR_L2CAP_PSM_WRONG_ENCODING	Wrong L2CAP PSM encoding
0703	EZS_ERR_L2CAP_PSM_ALREADY_REGISTERED	L2CAP PSM already registered
0704	EZS_ERR_L2CAP_PSM_NOT_REGISTERED	L2CAP PSM not registered
0705	EZS_ERR_L2CAP_CONNECTION_ENTITY_NOT_FOUND	L2CAP connection entity not found
0706	EZS_ERR_L2CAP_CHANNEL_NOT_FOUND	L2CAP channel not found
0707	EZS_ERR_L2CAP_PSM_NOT_IN_RANGE	L2CAP PSM is not in range
0800	**EZS_ERR_SMP**	**SMP error category**
0801	EZS_ERR_SMP_OOB_NOT_AVAILABLE	Out-of-band pairing data not available
0802	EZS_ERR_SMP_SECURITY_OPERATION_FAILED	Security operation failed
0803	EZS_ERR_SMP_MIC_AUTH_FAILED	Message integrity check authentication failed
0900	**EZS_ERR_SPEC**	**Bluetooth Core Specification error category**
0901	EZS_ERR_SPEC_UNKNOWN_HCI_COMMAND	Unknown HCI command
0902	EZS_ERR_SPEC_UNKNOWN_CONNECTION_IDENTIFIER	Unknown connection identifier
0903	EZS_ERR_SPEC_HARDWARE_FAILURE	Hardware failure
0904	EZS_ERR_SPEC_PAGE_TIMEOUT	Page timeout
0905	EZS_ERR_SPEC_AUTHENTICATION_FAILURE	**Authentication Failure**

Code (Hex)	Name	Description
0906	EZS_ERR_SPEC_PIN_OR_KEY_MISSING	**PIN or Key Missing**
0907	EZS_ERR_SPEC_MEMORY_CAPACITY_EXCEEDED	Memory capacity exceeded
0908	EZS_ERR_SPEC_CONNECTION_TIMEOUT	**Connection Timeout**
0909	EZS_ERR_SPEC_CONNECTION_LIMIT_EXCEEDED	Connection limit exceeded
090A	EZS_ERR_SPEC_SYNCHRONOUS_CONN_LIMIT _DEVICE_EXCEEDED	Synchronous connection limit to a device exceeded
090B	EZS_ERR_SPEC_ACL_CONNECTION_ALREADY_EXISTS	ACL connection already exists
090C	EZS_ERR_SPEC_COMMAND_DISALLOWED	Command disallowed
090D	EZS_ERR_SPEC_CONNECTION_REJECTED _LIMITED_RESOURCES	Connection rejected due to limited resources
090E	EZS_ERR_SPEC_CONNECTION_REJECTED _SECURITY_REASONS	Connection rejected due to security reasons
090F	EZS_ERR_SPEC_CONNECTION_REJECTED _UNACCEPTABLE_BDADDR	Connection rejected due to unacceptable BD_ADDR
0910	EZS_ERR_SPEC_CONNECTION_ACCEPT _TIMEOUT_EXCEEDED	Connection Accept Timeout exceeded
0911	EZS_ERR_SPEC_UNSUPPORTED_FEATURE _OR_PARAMETER_VALUE	Unsupported feature or parameter value
0912	EZS_ERR_SPEC_INVALID_HCI_COMMAND_PARAMETERS	Invalid HCI command parameters
0913	EZS_ERR_SPEC_REMOTE_USER_TERMINATED _CONNECTION	**Remote User Terminated Connection**
0914	EZS_ERR_SPEC_REMOTE_DEVICE_TERMINATED _LOW_RESOURCES	Remote device terminated connection due to low resources
0915	EZS_ERR_SPEC_REMOTE_DEVICE_TERMINATED _POWER_OFF	Remote device terminated connection due to power off
0916	EZS_ERR_SPEC_CONNECTION_TERMINATED _BY_LOCAL_HOST	**Connection Terminated by Local Host**
0917	EZS_ERR_SPEC_REPEATED_ATTEMPTS	Repeated attempts
0918	EZS_ERR_SPEC_PAIRING_NOT_ALLOWED	**Pairing Not Allowed**
0919	EZS_ERR_SPEC_UNKNOWN_LMP_PDU	Unknown LMP PDU
091A	EZS_ERR_SPEC_UNSUPPORTED_REMOTE _LMP_FEATURE	Unsupported remote feature / unsupported LMP feature
091B	EZS_ERR_SPEC_SCO_OFFSET_REJECTED	SCO offset rejected
091C	EZS_ERR_SPEC_SCO_INTERVAL_REJECTED	SCO interval rejected
091D	EZS_ERR_SPEC_SCO_AIR_MODE_REJECTED	SCO air mode rejected
091E	EZS_ERR_SPEC_INVALID_LMP_LL_PARAMETERS	Invalid LMP parameters / invalid LL parameters
091F	EZS_ERR_SPEC_UNSPECIFIED_ERROR	Unspecified error
0920	EZS_ERR_SPEC_UNSUPPORTED_LMP_LL _PARAMTER_VALUE	Unsupported LMP parameter value / Unsupported LL parameter value
0921	EZS_ERR_SPEC_ROLE_CHANGE_NOT_ALLOWED	Role change not allowed
0922	EZS_ERR_SPEC_LMP_LL_RESPONSE_TIMEOUT	LMP Response Timeout / LL Response Timeout
0923	EZS_ERR_SPEC_LMP_ERROR_TRANSACTION_COLLISION	LMP error transaction collision
0924	EZS_ERR_SPEC_LMP_PDU_NOT_ALLOWED	LMP PDU not allowed
0925	EZS_ERR_SPEC_ENCRYPTION_MODE_NOT_ACCEPTABLE	Encryption mode not acceptable
0926	EZS_ERR_SPEC_LINK_KEY_CANNOT_BE_CHANGED	Link key cannot be changed
0927	EZS_ERR_SPEC_REQUESTED_QOS_NOT_SUPPORTED	Requested QoS not supported
0928	EZS_ERR_SPEC_INSTANT_PASSED	Instant passed
0929	EZS_ERR_SPEC_PAIRING_WITH_UNIT_KEY _NOT_SUPPORTED	Pairing with unit key not supported
092A	EZS_ERR_SPEC_DIFFERENT_TRANSACTION_COLLISION	Different transaction collision
092B	/* 0x2B reserved */	Reserved
092C	EZS_ERR_SPEC_QOS_UNACCEPTABLE_PARAMETER = 0x092C	QoS unacceptable parameter

Code (Hex)	Name	Description
092D	EZS_ERR_SPEC_QOS_REJECTED	QoS rejected
092E	EZS_ERR_SPEC_CHANNEL_CLASSIFICATION _NOT_SUPPORTED	Channel classification not supported
092F	EZS_ERR_SPEC_INSUFFICIENT_SECURITY	Insufficient security
0930	EZS_ERR_SPEC_PARAMETER_OUT_OF _MANDATORY_RANGE	Parameter out of mandatory range
0931	/* 0x31 reserved */	Reserved
0932	EZS_ERR_SPEC_ROLE_SWITCH_PENDING = 0x0932	Role switch pending
0933	/* 0x33 reserved */	Reserved
0934	EZS_ERR_SPEC_RESERVED_SLOT_VIOLATION = 0x0934	Reserved slot violation
0935	EZS_ERR_SPEC_ROLE_SWITCH_FAILED	Role switch failed
0936	EZS_ERR_SPEC_EXTENDED_INQUIRY_RSP_TOO_LARGE	Extended inquiry response too large
0937	EZS_ERR_SPEC_SSP_NOT_SUPPORTED_BY_HOST	Secure simple pairing not supported by host
0938	EZS_ERR_SPEC_HOST_BUSY_PAIRING	Host busy - pairing
0939	EZS_ERR_SPEC_CONNECTION_REJECTED _NO_SUITABLE_CHANNEL	Connection rejected due to no suitable channel found
093A	EZS_ERR_SPEC_CONTROLLER_BUSY	Controller busy
093B	EZS_ERR_SPEC_UNACCEPTABLE _CONNECTION_PARAMETERS	Unacceptable connection parameters
093C	EZS_ERR_SPEC_DIRECTED_ADVERTISING_TIMEOUT	Directed advertising timeout
093D	EZS_ERR_SPEC_CONNECTION_TERMINATED _MIC_FAILURE	Connection terminated due to MIC failure
093E	EZS_ERR_SPEC_CONNECTION_FAILED _TO_BE_ESTABLISHED	**Connection Failed to be Established**
093F	EZS_ERR_SPEC_MAC_CONNECTION_FAILED	MAC connection failed
0940	EZS_ERR_SPEC_COARSE_CLOCK_ADJ_REJECTED	Coarse clock adjustment rejected but will try to adjust using clock dragging
EEEE	**EZS_ERR_UNKNOWN**	**Unknown problem** *(internal error)*

7.4.2 EZ-Serial GATT Database Validation Error Codes

The complete list of result/error codes generated by EZ-Serial during dynamic GATT database validation is contained in the table below. Refer to Section 3.4.1 (How to Define Custom Local GATT Services and Characteristics) and the documentation for the related GATT Server Group (ID=5) API command methods for detail.

Table 7-5. EZ-Serial GATT Validation Error Codes

Code (Hex)	Name	Description
0000	GATTS_DB_VALID_OK	Validation passed with no warnings or errors
0001	GATTS_DB_VALID_WARNING_NOT_ENOUGH_ATTRIBUTES	Structure is valid, but more attributes are required
0002	GATTS_DB_VALID_ERROR_ATTRIBUTE_LIMIT_EXCEEDED	Attribute count limit exceeded
0003	GATTS_DB_VALID_ERROR_ATTRIBUTE_DATA_EXCEEDED	Runtime attribute value data byte limit exceeded
0004	GATTS_DB_VALID_ERROR_CONSTANT_DATA_EXCEEDED	Constant default data byte limit exceeded
0005	GATTS_DB_VALID_ERROR_CCCD_LIMIT_EXCEEDED	CCCD attribute limit exceeded
0006	GATTS_DB_VALID_ERROR_SVC_DECL_REQUIRED	Service declaration required
0007	GATTS_DB_VALID_ERROR_UNEXPECTED_SVC_DECL	Unexpected service declaration
0008	GATTS_DB_VALID_ERROR_CHAR_DECL_REQUIRED	Characteristic declaration required
0009	GATTS_DB_VALID_ERROR_UNEXPECTED_CHAR_DECL	Unexpected characteristic declaration
000A	GATTS_DB_VALID_ERROR_CHAR_VALUE_REQUIRED	Characteristic value attribute required
000B	GATTS_DB_VALID_ERROR_UNEXPECTED_DESCRIPTOR	Specified descriptor not allowed at this position
000C	GATTS_DB_VALID_ERROR_INVALID_ATT_PROPERTIES	Attribute properties not compatible with type
000D	GATTS_DB_VALID_ERROR_INVALID_ATT_LENGTH	Invalid attribute length

Code (Hex)	Name	Description
`000E`	GATTS_DB_VALID_ERROR_INVALID_ATT_DATA	Attribute data not compatible with type

7.5 Macro Definitions

Macros in EZ-Serial are simple codes that result in text substitution within the parser. Macros may be used in either text mode or binary mode. Macros always begin with the '%' character and are followed by one or more alphanumeric characters (A-Z, 0-9). Macros are not case-sensitive.

Code	Description	Example Input	Example Output	Notes
`%M1`	Byte #1 of local public MAC address	`MyDevice %M1`	`MyDevice 00`	Examples assume that the local device has a public MAC address of `00:A0:50:E3:83:5F`.
`%M2`	Byte #2 of local public MAC address	`MyDevice %M2`	`MyDevice A0`	
`%M3`	Byte #3 of local public MAC address	`MyDevice %M3`	`MyDevice 50`	
`%M4`	Byte #4 of local public MAC address	`MyDevice %M4`	`MyDevice E3`	
`%M5`	Byte #5 of local public MAC address	`MyDevice %M5`	`MyDevice 83`	
`%M6`	Byte #6 of local public MAC address	`MyDevice %M6`	`MyDevice 5F`	

Macros may be used in series with or without special separators, as long as the entire macro code (including the '%' byte) remains intact. For example, to use the last three bytes of the MAC address in the same string, separated by the ':' byte, use the following:

```
MyDevice %M4:%M5:%M6
```

This string is particularly useful for setting a module-specific device name using the gap_set_device_name (SDN, ID=4/15) API command without needing to query or track the MAC address separately by hand.

8. GPIO Reference

This section describes the various GPIO connections provided by the EZ-Serial firmware on supported modules. It also provides details on the default boot state and what behavior to expect in different operational modes.

8.1 GPIO Pin Map for Supported Modules

The assignment of special functions for supported modules is described in Table 8-1.

Each pin is shown with its assigned module pin and the effective header pin when using the CYBLE-013025-EVAL board.

NOTE: The pins available on the EZ-BLE WICED module are similar to those on the EZ-BLE modules based on PSoC Creator (e.g., CYBLE-212019-00), but they are not identical due to hardware differences. The EZ-BLE WICED module has a single PWM channel, and does not have either the ATEN_SHDN pin or the CP_ROLE pins.

Table 8-1. GPIO Pin Map on Supported EZ-BLE WICED Modules

	Pin Name	**Pin Assignment**	
		CYBLE-013025-00	
		Module	Eval
DIGITAL FUNCTIONS	**UART_RX**	P2	J3.8
	UART_TX	P0	J3.7
	UART_RTS	P1	J5.6
	UART_CTS	P3	J5.5
	CONNECTION	P14	J5.9
	CYSPP	P27	J3.5
	DATA_READY	P15	J7.1
	LP_MODE	P24	J7.2
	LP_STATUS	P25	J7.3
PWM	**PWM0**	P26	J3.3
ADC	**ADC0**	P13	J3.2

8.2 GPIO Pin Functionality

WICED BLE EZ-Serial provides ten special-function digital GPIO pins, one optional PWM output pin for generating flexible PWM signals, and one optional analog input pin for ADC reads.

8.2.1 Digital Special-Function Pins

Table 8-2 below details the functionality of each digital function GPIO pin

Table 8-2. GPIO Pin Functionality Detail

Pin Name	Direction	Details
HCI_UART_RX	Input	UART Communication RX signal for incoming HCI commands or firmware from external host device
HCI_UART_TX	Output	UART Communication TX signal for outgoing HCI commands or firmware to external host device
PUART_RX	Input	UART Communication RX signal for incoming data from external host device
PUART_TX	Output	UART Communication TX signal for outgoing data to external host device
PUART_RTS	Output	UART Communication RTS signal signifying local receive permission (flow control)
PUART_CTS	Input	UART Communication CTS signal detecting remote receive permission (flow control)
CONNECTION	Output	**Description:** BLE connection or CYSPP data pipe readiness status. When the CYSPP pin is asserted, the external host can use this pin to detect whether the data sent to module will be transmitted to the remote peer device directly. **Status indicator logic (active-low):** • When CYSPP pin is de-asserted (API command mode active) o **LOW** – remote BLE peer device is connected. o **HIGH** – no remote BLE peer device is connected • When CYSPP pin is asserted (CYSPP mode active) o **LOW** – CYSPP data stream fully available (connected and ready) o **HIGH** – CYSPP data stream not available (disconnected or not ready) **Default boot state:** • **HIGH** (no connection)
CYSPP	Input	**Description:** CYSPP mode control. The external host can use this pin to begin automatic CYSPP operation without the need for any API commands. This pin is also internally pulled HIGH or LOW based on software-triggered entry or exit to and from CYSPP data mode. If connected to a high-impedance input pin (weaker than 5.6 kΩ, this pin may be used as a status indicator for software-based CYSPP mode changes. Otherwise, it should be driven externally to the desired state. **Control signal logic (active-low):** • **LOW** – module enters CYSPP data mode. • **HIGH** – module exits CYSPP data mode and returns to API command mode. **Status indicator logic (internally pulled, may be overridden by external signals):** • **LOW** – API commands or remote BLE Client GATT Client transactions have entered CYSPP data mode. • **HIGH** – API commands or remote BLE peer GATT Client transactions have exited CYSPP data mode. **Default boot state:** • Internally pulled **HIGH** (command mode active, CYSPP data mode inactive)

Pin Name	Direction	Details
DATA_READY	Output	**Description:** The external host can use this as an interrupt signal, which is especially useful if the host cannot wake up on UART activity. This signal is asserted when the available outgoing data is an API response or event (command mode) or serial data from a remote peer (CYSPP mode). When used in combination with flow control and the module's CTS pin, a host can efficiently manage the module's data flow in tandem with its own sleep requirements. **Status indicator logic (active LOW):** • **LOW** – data is ready to be sent to the external host. • **HIGH** – all data has been transmitted. **Default boot state:** • **HIGH**, but quickly goes **LOW** in command mode due to system boot event (**Note:** will remain **HIGH** right after boot if CYSPP pin is asserted)
LP_MODE	Input	**Description:** Low-power status control. The external host can use this pin to affect the sleep behavior of the module, specifically by either preventing or allowing entry into sleep modes. **Control signal logic (active LOW):** • **LOW** – CPU is allowed (but not forced) to sleep. • **HIGH** – CPU is kept in active mode. **Default boot state:** • Internally pulled **LOW** (sleep allowed)
LP_STATUS	Output	**Description:** Low-power status indicator. The external host can use this pin to understand the power state of the module. This is especially useful if the external microcontroller needs to know whether the module can communicate over UART (UART is disabled in Sleep states). **Status indicator logic (active LOW):** • **LOW** – CPU is in the sleep state. • **HIGH** – CPU is in active state. **Default boot state:** • **HIGH** (awake) until the boot process finishes; then **LOW** unless **LP_MODE** is asserted.

8.2.2 PWM Output Pins

EZ-Serial provides one dedicated PWM output pin (**PWM0**) on the EZ-BLE WICED module platform. You can enable PWM output on this channel using the gpio_set_pwm_mode (SPWM, ID=9/11) API command. PWM channels are controlled via an independent 24-MHz clock, and can use divider, prescaler, period, and compare settings for complete flexibility.

NOTE: Enabling PWM output automatically prevents the CPU from entering normal sleep under any circumstances. This happens because the high-frequency clock required to generate the PWM signal cannot operate while the CPU is in sleep. To allow sleep mode again, you must disable all PWM output. See Section 3.1.5 (How to Manage Sleep States).

8.2.3 Analog Input Pins (ADC)

EZ-Serial provides a single dedicated ADC input pin (**ADC0**) for reading analog voltages. The ADC supports an input voltage range of **0 V** minimum to VDD(usually **3.3 V**) maximum. To perform a single ADC conversion, use the gpio_query_adc (/QADC, ID=9/2) API command. Once the conversion completes, the module transmits the result in the response to this command.

8.3 Functional Capabilities

It is important to understand the intended use case for certain GPIO-related functions provided by the EZ-Serial firmware, especially analog-to-digital conversion (ADC). This helps ensure that your expectations are met.

8.3.1 Analog-to-Digital Conversion

The ADC operates very quickly but incurs significant processing overhead in order to transmit conversion results to an external host via API event packets. The EZ-Serial firmware platform provides a way to perform on-demand single ADC reads on individual analog channels, such as what might be involved in periodic battery voltage measurements or analog light, gas, or temperature sensor readings.

If your application requires rapid sequencing and data analysis, it may be necessary to use the WICED Smart SDK to create a custom firmware implementation.

9. Cypress GATT Profile Reference

The EZ-Serial platform makes use of a few custom GATT profiles defined by Cypress Semiconductor. The service UUIDs, characteristic UUIDs, special permissions, and overall structure are outlined here for quick reference. Much more detailed reference material can be found on the Cypress website here:

http://www.cypress.com/documentation/software-and-drivers/cypresss-custom-ble-profiles-and-services

9.1 CYSPP Profile

The Cypress Serial Port Profile (CYSPP) provides bidirectional serial data transfer between two remote devices, each of which passes data in through a single local hardware serial interface. It supports both acknowledged transfers and unacknowledged transfers, and provides a mechanism for virtual flow control in both the RX and TX direction.

The profile contains a single service ("CYSPP"), which contains three characteristics for data transfer and flow control ("Acknowledged Data", "Unacknowledged Data", and "RX Flow"). The structural outline of this profile is as follows:

- **CYSPP** Service: UUID **65333333-A115-11E2-9E9A-0800200CA100**

  - **Acknowledged Data** Characteristic: UUID **65333333-A115-11E2-9E9A-0800200CA101**
    (Write, Indicate)

    The Acknowledged Data Characteristic is used to send and receive data in an acknowledged fashion. The EZ-Serial firmware is able to fully track every transfer in both directions. This characteristic has a variable length, supporting transfers in each direction of up to 20 bytes per packet.

    - Configuration Descriptor: UUID **0x2902**

  - **Unacknowledged Data** Characteristic: UUID **65333333-A115-11E2-9E9A-0800200CA102**
    (Write without response, Notify)

    The Unacknowledged Data Characteristic is used to send and receive data in an unacknowledged fashion. The EZ-Serial firmware cannot track transfers using this mode once they have been accepted by the BLE stack. This provides less control, but the lack of acknowledgements also allows for much greater maximum throughput. This characteristic has a variable length, supporting transfers in each direction of up to 20 bytes per packet.

    - Configuration Descriptor: UUID **0x2902**

  - **RX Flow** Characteristic: UUID **65333333-A115-11E2-9E9A-0800200CA103**
    (Indicate)

    The RX Flow Characteristic is used to indicate to the Client that the Server can no longer safely receive new data. If the Client subscribes to indications from this characteristic, the Server will assume that the Client obeys flow control signals. This characteristic is one byte in length. An indicated value of "0" means that it is safe for the Client to send data, while a value of "1" means that the Client must refrain from sending data.

    - Configuration Descriptor: UUID **0x2902**

10. Configuration Example Reference

The configuration examples provided in this section are each designed to work independently, assuming in each case that the platform is initially configured using factory default settings. Applying all of the commands in one example and then immediately following this with the commands from another example may result in changes to the first set of behavior that are no longer in line with the expected results.

You can return a module to factory defaults as a baseline configuration at any time by using the system_factory_reset (/RFAC, ID=2/5) API command. This reset command is not explicitly included in any of the configuration snippets within this section.

10.1 Factory Default Settings

While you can return to the factory default settings on the module by performing a factory reset, it is also helpful to know what those settings are for comparison or to explicitly change one or more individual settings back to the default value without reverting all customizations at once. The following is a comprehensive list of commands that will return the EZ-Serial module to default behavior:

Text Content	Binary Content
`SPPM,M=00`	C0 01 01 01 00 5C
`SSLP,L=00`	C0 01 02 13 00 6F
`STXP,P=07`	C0 01 02 15 07 78
`STU,B=0001C200,A=00,C=00,F=00,D=08,P=00,S=01`	C0 0A 02 19 00 C2 01 00 00 00 00 08 00 01 4A
`SDN,N=EZ-Serial %M4:%M5:%M6`	C0 16 04 0F 15 45 5A 2D 53 65 72 69 61 6C 20 25 4D 34 3A 25 4D 35 3A 25 4D 36 4C
`SDA,A=0000`	C0 02 04 11 00 00 70
`SAD,D=`	C0 01 04 13 00 71
`SSRD,D=`	C0 01 04 15 00 73
`SAP,M=02,T=00,I=0030,C=07,L=00,O=0000,F=00`	C0 09 04 17 02 00 30 00 07 00 00 00 00 B6
`SGSP,F=01`	C0 01 05 0E 01 6E
`SPRV,M=00,I=012C`	C0 03 07 09 00 2C 01 99
`SSBP,M=11,B=01,K=10,P=00,I=03,F=01`	C0 06 07 0B 11 01 10 00 03 01 97
`.CYSPPSP,E=02,G=00,C=0131,L=00000000,R=00000000, M=00000000,P=01,S=00,F=02`	C0 13 0A 03 02 00 31 01 00 00 00 00 00 00 00 00 00 00 00 00 01 00 02 B0
`.CYSPPSK,M=01,W=05,L=14,E=0D`	C0 04 0A 07 01 05 14 0D 95

Remember that the above commands affect only RAM. To make them permanent, apply all settings to flash using the system_store_config (/SCFG, ID=2/4) API command.

10.2 Adopted Bluetooth SIG GATT Profile Structure Snippets

The snippets below demonstrate how to add various GATT service and characteristic structural elements in order to support official profiles defined by the Bluetooth SIG, and some other common services.

NOTE: These database structures concern only the **GATT Server** side of the profiles in question. GATT Client operations depend on the Client device.

NOTE: The information provided in this section only covers the basic GATT structure, but does not include any specific values which may be necessary or helpful for specific functionality. Many characteristics also have flexible **length** values which depend on application design. Refer to the official Bluetooth SIG documentation or other related resources linked under each service for further detail.

10.2.1 Generic Access Service (0x1800)

Official documentation for this service can be found on the Bluetooth SIG Developer website.

NOTE: This service is included in the EZ-Serial application. It is always present in the fixed, non-removable part of the GATT structure. Do not add another instance of this service to the EZ-Serial application.

```
/CAC,T=0,P=02,L=04,D=00280018
/CAC,T=0,P=02,L=07,D=0328020300002A
/CAC,T=1,P=0B,L=40,D=
/CAC,T=0,P=02,L=07,D=0328020500012A
/CAC,T=1,P=02,L=02,D=
/CAC,T=0,P=02,L=07,D=0328020700042A
/CAC,T=1,P=02,L=08,D=
/CAC,T=0,P=02,L=07,D=0328020900A62A
/CAC,T=1,P=02,L=01,D=
```

NOTE: EZ-Serial assumes that the attribute handle is starting from 1. Data item of characteristic attribute include the attribute handle (0x0003, 0x0005,0x0007 and 0x0009 respectively in this example) which corresponding to the characteristic value attribute.

10.2.2 Generic Attribute Service (0x1801)

Official documentation for this service can be found on the Bluetooth SIG Developer website.

NOTE: This service is included in the EZ-Serial application. It is always present in the fixed, non-removable part of the GATT structure. Do not add another instance of this service to the EZ-Serial application.

```
/CAC,T=0,P=02,L=04,D=00280018
/CAC,T=0,P=02,L=07,D=0328200300052A
/CAC,T=1,P=02,L=04,D=
/CAC,T=0,P=0A,L=04,D=0229
```

NOTE: EZ-Serial assumes that the attribute is handled starting from 1. Attribute handle (0x0003) corresponding to the value attribute

10.2.3 Immediate Alert Service (0x1802)

Official documentation for this service can be found on the Bluetooth SIG Developer website.

```
/CAC,T=0,P=02,L=04,D=00280218
/CAC,T=0,P=02,L=07,D=0328041800062A
/CAC,T=1,P=0A,L=01,D=
```

10.2.4 Link Loss Service (0x1803)

Official documentation for this service can be found on the Bluetooth SIG Developer website.

```
/CAC,T=0,P=02,L=04,D=00280318
/CAC,T=0,P=02,L=07,D=03280A1800062A
/CAC,T=1,P=0A,L=01,D=
```

10.2.5 TX Power Service (0x1804)

Official documentation for this service can be found on the Bluetooth SIG Developer website.

```
/CAC,T=0,P=02,L=04,D=00280418
/CAC,T=0,P=02,L=07,D=0328021800072A
/CAC,T=1,P=02,L=01,D=
/CAC,T=0,P=0A,L=04,D=0229
```

11. EZ-Serial Mac Address

The EZ-Serial firmware platform for EZ-BLE WICED Modules includes a static random MAC address when shipped from Cypress. The static random MAC address is configured during Cypress manufacturing programming process, and this address does not change for the life of the programmed image.

During the Cypress programming process, the EZ-Serial firmware generates a static random address and stores it in EEPROM. The address format follows the *Bluetooth Core Specification 5.0 Volume 6, part B, Section 1.3.2.1 Static Device Address*. This address is persistent during module power cycle or reset operations.

Note: The EZ-Serial firmware internally controls the address type by using smp_set_privacy_mode (SPRV, ID=7/9). If this mode bit is set to 0, it advertises as a public address type. If this mode bit set to 1, it advertises as a random address type. The default for the EZ-Serial address is 1 (random address).

If you want to use your own public address (using an assigned IEEE OUI), use the system_set_bluetooth_address (SBA, ID=2/13) command to configure the address to your OUI plus three additional random bytes, and then use the smp_set_privacy_mode (SPRV, ID=7/9) command to change the address type to public.

If you modify the type and format of the address and then want to revert to the EZ-Serial initial static random address, use the system_set_bluetooth_address (SBA, ID=2/13) command with the parameter address equal to 0. Using this command reverts the advertising address to the factory-provided static random address.

In all cases, you should be familiar with the rules set forth by the Bluetooth SIG for MAC address generation, format and usage as documented in the Bluetooth Core Specification 5.0, Volume 6, section 1.3.

12. Binary/Text Conversion Python Script

Due to internal memory limitations, CYBLE-013025-00 supports only the binary command/event mode. Cypress provides a simple Python script to help convert between text and binary API packets. You can use it to test commands and see both formats, and optionally copy it into to your code.

1. Install a Python 2.x or 3.x environment (3.x preferred).
2. Install pyserial from https://pypi.python.org/pypi/pyserial.
3. Copy the files *ezslib.py* and *textbin_console.py* to a folder of your choice.
4. Open a command line window, and start the script with a command such as the following:

   ```
   python textbin_console.py -p COM34 -s input -l wiced_ble
   ```

   Note: COM34 is a placeholder; replace it with the correct COM port in your environment.

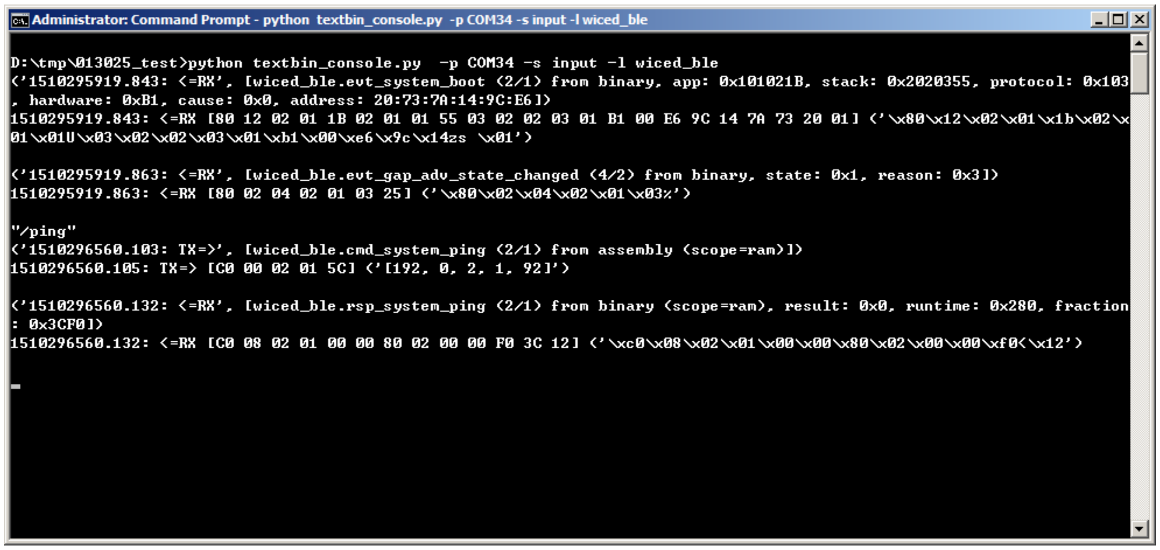
   You can run `python textbin_console.py` to get help information.

5. Reboot the CYBLE-013025-00 module.

   You should see the system boot and advertisement state changed events appear in binary format and the converted text format.

6. Send commands such as "/PING" by typing it directly in the script console. They are converted to binary format as `[C0 00 02 01 5C]`; a binary response similar to `[C0 08 02 01 00 00 80 02 00 00 F0 3C 12]` is generated, which is then converted back into text format.

Figure 12-1. Commands to Run *textbin_console.py*



Administrator: Command Prompt - python textbin_console.py -p COM34 -s input -l wiced_ble

```
D:\tmp\013025_test>python textbin_console.py  -p COM34 -s input -l wiced_ble
('1510295919.843: <=RX', [wiced_ble.evt_system_boot (2/1) from binary, app: 0x101021B, stack: 0x2020355, protocol: 0x103
, hardware: 0xB1, cause: 0x0, address: 20:73:7A:14:9C:E6])
1510295919.843: <=RX [80 12 02 01 1B 02 01 01 55 03 02 02 03 01 B1 00 E6 9C 14 7A 73 20 01] ('\x80\x12\x02\x01\x1b\x02\x
01\x01U\x03\x02\x02\x03\x01\xb1\x00\xe6\x9c\x14zs \x01')

('1510295919.863: <=RX', [wiced_ble.evt_gap_adv_state_changed (4/2) from binary, state: 0x1, reason: 0x3])
1510295919.863: <=RX [80 02 04 02 01 03 25] ('\x80\x02\x04\x02\x01\x03%')

"/ping"
('1510296560.103: TX=>', [wiced_ble.cmd_system_ping (2/1) from assembly (scope=ram)])
1510296560.105: TX=> [C0 00 02 01 5C] ('[192, 0, 2, 1, 92]')

('1510296560.132: <=RX', [wiced_ble.rsp_system_ping (2/1) from binary (scope=ram), result: 0x0, runtime: 0x280, fraction
: 0x3CF0])
1510296560.132: <=RX [C0 08 02 01 00 00 80 02 00 00 F0 3C 12] ('\xc0\x08\x02\x01\x00\x00\x80\x02\x00\x00\xf0<\x12')
```

Revision History

Document Title: EZ-Serial WICED BLE Firmware Platform User Guide **Document Number: 002-21887**			
Revision	**Issue Date**	**Origin of Change**	**Description of Change**
**	01/25/2018	DUDL	New user guide for EZ-BLE WICED Module EZ-Serial platform.