PSoC 6 MCU - Firmware Design for BLE Applications

About this document

Scope and purpose

AN215671 introduces the Bluetooth Low energy (BLE) stack used with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity, the BLE Component architecture, and the use of a dual-CPU processor for BLE applications. This application note shows how to design firmware for BLE applications. This application note uses the associated code examples to demonstrate the Multi-Master Multi-Slave feature of PSoC 6 BLE and validate the application using the CySmart™ BLE host emulation tool and Cypress' iOS/Android CySmart application.

To access an ever-growing list of hundreds of PSoC code examples, please visit our **code examples web page**. You can also explore the Cypress video training library **here**.

Table of contents

1 Introduction

Bluetooth Low Energy (BLE) is a low-power wireless standard introduced by the Bluetooth Special Interest Group (SIG) for short-range communication. The BLE physical layer, protocol stack, and profile architecture are designed and optimized to minimize power consumption. Similar to Classic Bluetooth, BLE operates in the 2.4-GHz ISM band with a maximum bandwidth of 2 Mbps.

PSoC 6 MCU with BLE Connectivity (PSoC 6 BLE) is Cypress' ultra-low-power PSoC device with dual CPUs specifically designed for wearables and Internet of Things (IoT) products. PSoC 6 BLE integrates a BLE 4.2 radio and a royalty-free protocol stack with enhanced security and privacy; it also has throughput compliant with the BLE 5.0 specification.

This application note introduces you to the BLE protocol stack in PSoC 6 BLE and discusses how to develop BLE applications using the PSoC Creator BLE Component. The BLE Component has built-in support for the standard profiles defined by Bluetooth SIG, which simplifies application development. In addition, this application note uses the code examples **CE223508** and **CE224714** to demonstrate the Multi-Master Multi-Slave feature of PSoC 6 BLE and validate the application using the CySmart BLE host emulation tool and Cypress' iOS/Android CySmart application.

This application note assumes that you are familiar with the basics of BLE, PSoC 6 BLE, and the PSoC Creator IDE. If you are new to PSoC 6 BLE, refer to **AN210781** – *Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity*. To get acquainted with PSoC Creator IDE, see the **PSoC Creator home page**.

2 PSoC Resources

Cypress provides a wealth of data at **www.cypress.com** to help you to select the right PSoC device and quickly and effectively integrate it into your design. The following is an abbreviated list of resources for PSoC 6 MCU:

- **Overview:** PSoC Portfolio, PSoC Roadmap
- **Product Selectors: PSoC 6 MCU**
- **Datasheets** describe and provide electrical specifications for each device family.
- **Application Notes** and **Code Examples** cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples. You can also browse our collection of code examples from directly inside PSoC Creator
- **Technical Reference Manuals (TRMs)** provide detailed descriptions of the architecture and registers in each device family.
- **PSoC 6 MCU Programming Specifications** provide the information necessary to program the nonvolatile memory of PSoC 6 MCU devices.
- **CapSense Design Guides:** Learn how to design capacitive touch-sensing applications with PSoC devices.
- **Development Tools**

  CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit is an easy-to-use and inexpensive development platform for PSoC 6 BLE.
- **Training Videos:** Cypress provides **video training** on our products and tools, including a dedicated series on **PSoC 6 MCU.**

See **KBA223067** for a comprehensive list of PSoC 6 MCU resources.

3 BLE Protocol Implementation in PSoC 6 BLE

The BLE protocol stack used with PSoC 6 BLE integrates a low-energy controller and host, and provides a full and flexible API for developing BLE applications. The BLE stack consists of two major blocks: BLE Host and BLE Controller. Partitioning the stack into host and controller provides flexibility of implementing the protocol stack using the dual-CPU architecture of PSoC 6 BLE. **Figure 1** shows the dual-CPU and single-CPU BLE implementation in PSoC 6 BLE.

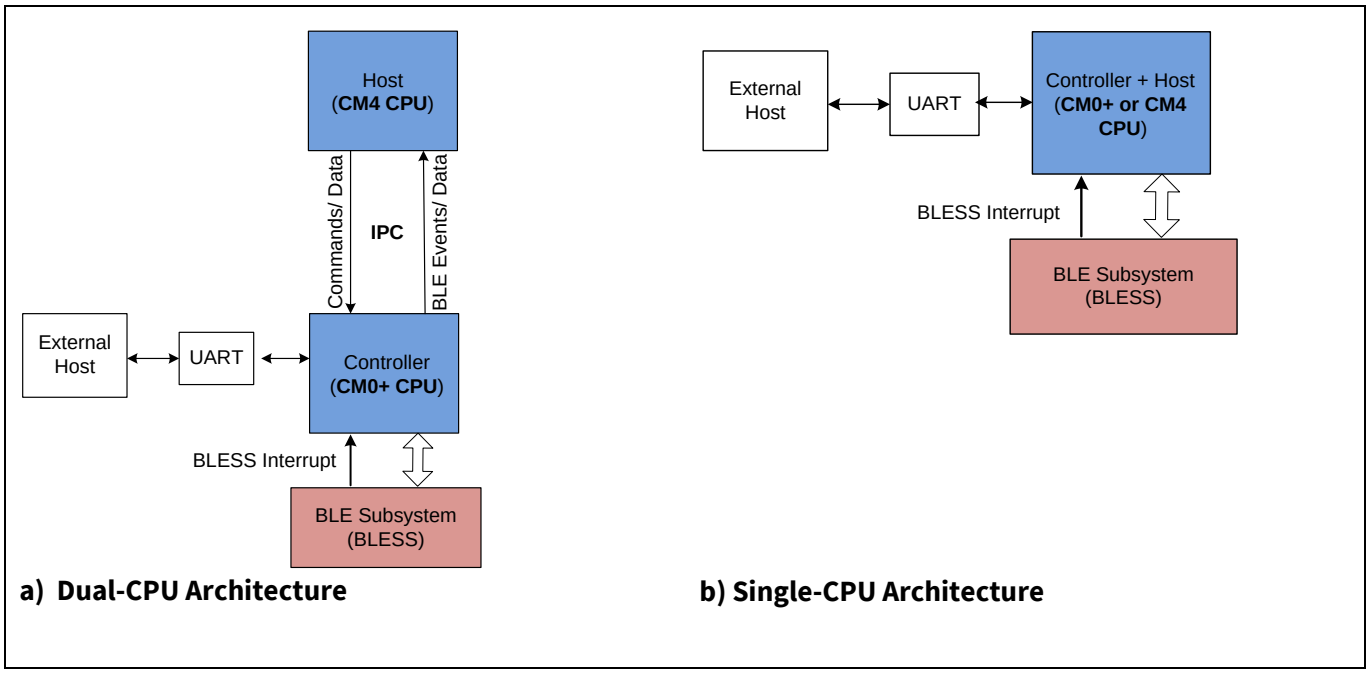


Figure 1 BLE Architecture in PSoC 6

The BLE stack is provided as a precompiled library along with the Cypress BLE Middleware Library, and packaged with Cypress Peripheral Driver Library (PDL) 3.x. The BLE middleware library contains a comprehensive API that allows you to configure the BLE stack and the underlying hardware.

The BLE stack has the following features:

- Operates in 2.4-GHz ISM band with a data rate of 2 Mbps, compliant with the BLE 5.0 specification
- Multi-Master Multi-Slave (MMMS) feature: supports up to four simultaneous connections in any combination of roles (Central/Peripheral)
- Simultaneous support for all Generic Access Profile roles – Peripheral, Central, Broadcaster, and Observer
- Attribute Protocol (ATT) that defines how the application data is organized and accessed
- Generic Attribute Profile (GATT) that defines methods to access data defined by the ATT layer (GATT Server and GATT Client)
- Support for BLE Special Interest Group (SIG)-adopted GATT-based Profiles and Services
- Security Manager Protocol (SMP) that provides a toolbox for secure data exchange over the BLE link. This toolbox includes:
  - Pairing methods: Just Works, Passkey Entry, Out of Band, Numeric Comparison
  - Authenticated Man-In-The-Middle (MITM) protection and data signing
- Logical Link Control and Adaptation Protocol (L2CAP) Connection-Oriented Channel
- Link Layer (LL) features which include:
  - Master and Slave role

- 128-bit AES Encryption
- Low Duty Cycle Advertising
- LE (Low Energy) Ping

The BLE stack implements a layered architecture of the BLE protocol as shown in **Figure 2**

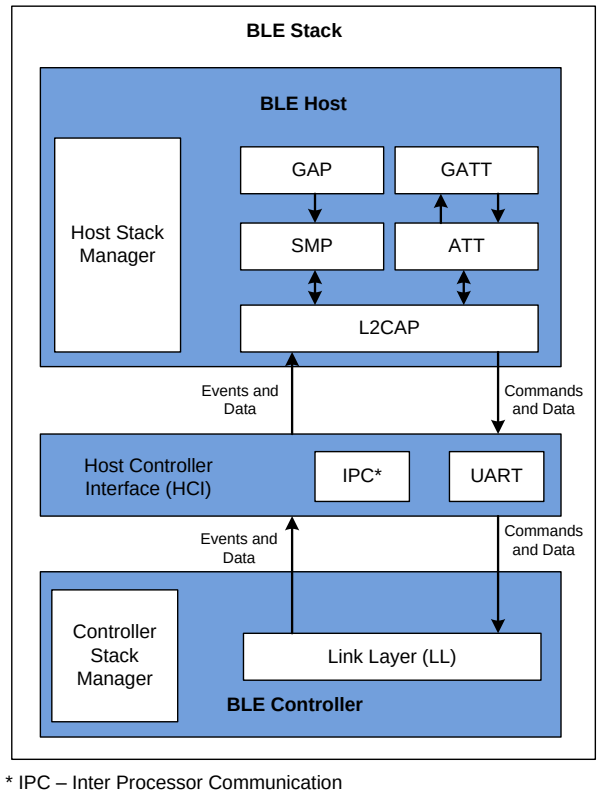


Figure 2 BLE Protocol Stack

3.1 BLE Host

The BLE Host implements the following protocol layers:

- Generic Access Profile (GAP)
- Generic Attribute Profile (GATT)
- Attribute Protocol
- Security Manager Protocol (SMP)
- Logical Link Control and Adaptation Protocol (L2CAP)
- Host Controller Interface (HCI)

You do not need a full working knowledge of the complex BLE protocol to develop a BLE application. From an application point of view, it is sufficient to have a good understanding of the following:

- **GAP Layer:** This layer deals with how the BLE link is established between two devices
- **GATT Layer:** This layer deals with how the data is represented. GATT is the layer where the data values are transferred.

In the remainder of the section, you will get an overview of GATT and GAP protocols and the related terms. You may refer to the **Bluetooth Core Specification** for a detailed description of the protocol.

3.1.1 Generic Access Profile (GAP)

GAP is the lowest layer of the BLE stack that an application interface with. It includes parameters that govern advertising and connection, among other things. To establish a BLE link between two devices (Cypress BLE Pioneer Kit and a smartphone, for example), you need to understand the two GAP device roles:

- **GAP Central**: The GAP Central is the device that initiates the connection with a peer device (GAP Peripheral). A GAP Central device scans for advertisements from GAP Peripherals and establishes a connection with them. A smartphone that connects to a heart-rate measurement device is an example of a GAP Central device.
- **GAP Peripheral**: The GAP Peripheral is the device that is connected to a GAP Central device. A GAP Peripheral advertises its presence and accepts the connection from a GAP Central device.

The Bluetooth Core Specification defines Observer and Broadcaster roles in addition to the Central and Peripheral roles. Observers listen to what's happening on the air; Broadcasters send but don't receive information.

Note: Irrespective of the GAP role, the device at either end of a BLE link is referred to as a peer device.

Depending upon the GAP role of the device, the device can be in any of the following states:

- **Advertising**: To establish a BLE link between two peer devices, the GAP Peripheral must advertise to let Central devices know of its presence. It sends out advertising packets at timed intervals, known as the advertising interval, that ranges from 20 ms to 10.24 s. The advertisement interval determines how fast a link is established.
- An advertisement packet can contain up to 31 bytes of data, which consist of the local device name, information pertaining to the services the device contains, etc. When a Central device receives an advertisement packet, if the Central device is configured as an active scanner, it may optionally send a request for more device-related data, called a Scan Request. The Peripheral responds to the request by sending a Scan Response that can contain an additional 31 bytes.
- **Scanning:** Scanning is used by the GAP Central to listen to advertisement packets from peer devices. In this context, you need to consider two parameters: Scan Window and Scan Interval. The Central device starts a scan once per Scan Interval. Within the interval, it will scan for the duration of the Scan Window. If the Scan Window is equal to the Scan Interval, the Central device does a continuous scan.
- **Initiating:** When the Central receives an advertisement packet, if it wants to establish a connection, it sends a connection request. This procedure is called "initiating".
- **Connected:** The Central and Peripheral are in a connected state from the first data exchange. When connected, the Central requests data from the Peripheral at a specifically defined interval called the "Connection interval". The connection interval is dictated by the Central when the link is established. A Peripheral can send Connection Parameter Update Requests to the Central. The connection interval must be between 7.5 ms and 4 s per the *Bluetooth Core Specification*.

  The Link Layer (LL) maintains the BLE link by sending at least one BLE packet in every connection interval. If the Peripheral doesn't respond to packets from the Central within the time frame, called "Connection Supervision Timeout", the link is considered lost. If the Peripheral has no data to send and power consumption is of concern in your design, then it can choose to ignore a certain number of intervals. The number of ignored intervals is called the "Slave Latency".

  For a proper operation, it is recommended to have connection supervision timeout as follows:

  $$\text{Connection supervision timeout} \geq (1+ \text{Slave latency}) * \text{Connection Interval}$$

3.1.2 Generic Attribute Profile (GATT)

After the Central device establishes a connection with the Peripheral, both devices are said to be connected over a BLE link. On a connected BLE link, independent of the GAP role, the Generic Attribute Profile (GATT) defines two profile roles based on the source and destination of the data:

- **GATT Server:** A GATT Server is a device that contains the data or state. When configured by a GATT Client, it sends data to the GATT Client or modifies its local state. For example, a heart-rate measurement device is a GATT Server that sends heart-rate data to a smartphone, which acts as the GATT Client. Similarly, a smart bulb is a GATT Server that contains the state of the bulb (ON/OFF) that can be configured by a smart switch, which acts as the GATT Client.
- **GATT Client:** A GATT Client is a device that configures the state of a GATT Server or receives data from a GATT Server. For example, a smartphone that receives heart-rate information from the heart-rate measurement device is a GATT Client.

3.1.3 ATT Protocol – Organizing the Data

The GATT Server uses the ATT protocol, which organizes the data systematically in the form of an attribute table called GATT database. A GATT Server uses Attributes, Characteristics, and Services to represent and abstract data in a BLE device. A Service contains one or more Characteristics; each Characteristic is composed of multiple Attributes that contain the actual data. **Figure 3** shows the GATT database hierarchy.

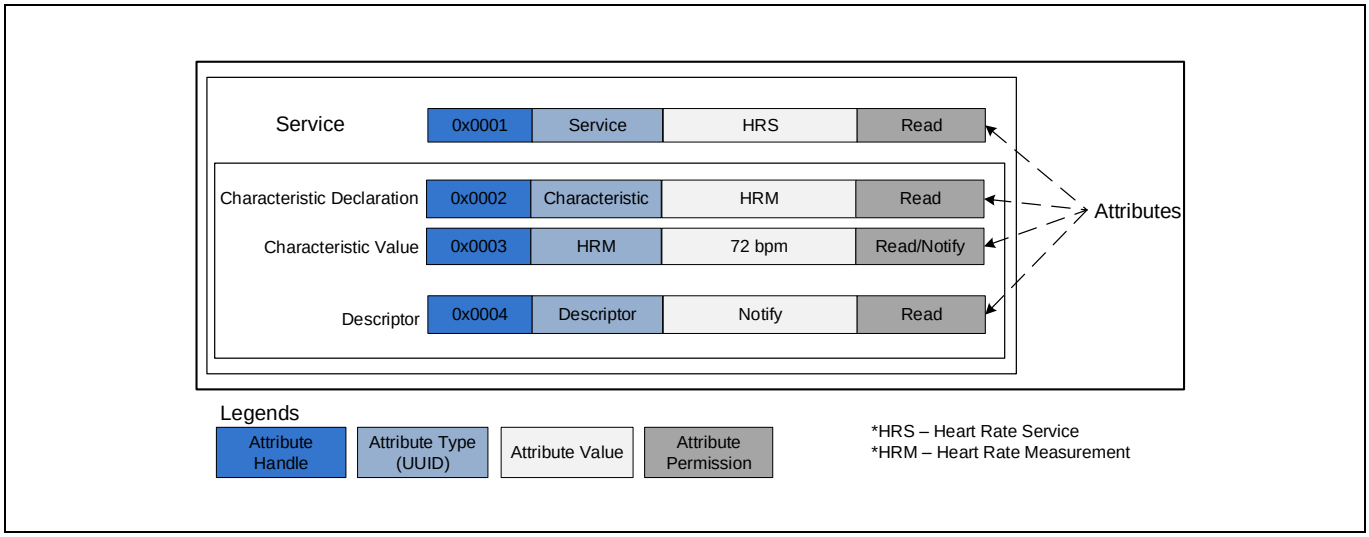


Figure 3 GATT Database Overview

Attribute: An Attribute is the fundamental data container of the GATT layer that represents a discrete piece of information. The structure of an Attribute consists of the following:

- Attribute Handle: Used to address the Attribute. A handle is a unique index of the Attribute in the GATT database.
- Attribute Type: A 16-bit Universally Unique Identifier (UUID) assigned by the Bluetooth SIG that specifies the data contained in the Attribute.
- Attribute Value: Contains the actual data.
- Attribute Permission: Specifies read/write permissions and security requirements for the Attribute.

Characteristic: A characteristic consists of at least two Attributes: a Characteristic declaration and an Attribute that holds the value for the Characteristic. All data that will be transferred through a GATT Service must be mapped to a set of Characteristics. It is a good idea to bundle the data so that each Characteristic is a self-

contained, single-instance data point. For example, if some pieces of data always change together, it will often make sense to collect them in one Characteristic. The Characteristic illustrated in **Figure 3** has three attributes: Characteristic Declaration, Characteristic Value, and the Descriptor Attribute.

Descriptors: Any Attribute within a Characteristic definition other than Characteristic Declaration Attribute and Characteristic Value Attribute is a Descriptor. A Descriptor is an additional Attribute that provides more information about a Characteristic, for instance, a human-readable description of the Characteristic.

However, there is one special Descriptor that is worth mentioning in particular: the Client Characteristic Configuration Descriptor (CCCD). This Descriptor is added for any Characteristic that supports the Notify or Indicate properties. Writing a '1' to the CCCD enables notifications, while writing a '2' enables indications. Writing a '0' disables both notifications and indications. Notify and Indicate are the attribute properties that allow a GATT Server to make the GATT Client aware of the changes to a Characteristic. The Indicate property has application-level acknowledgement while Notify does not have application-level acknowledgement.

Service: A Service is composed of one or more related Characteristics that define a function or feature of a device. GATT Services typically include pieces of related functionality such as a sensor's readings and settings, or the inputs and outputs of a Human Interface Device.

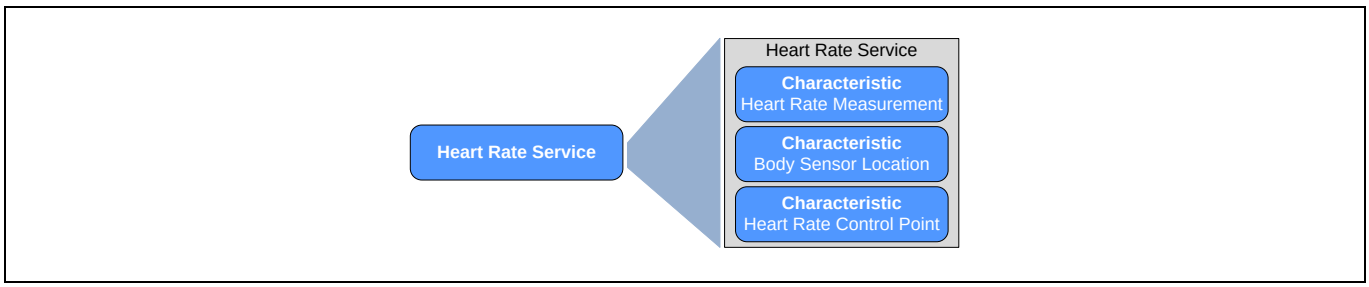


Figure 4 GATT Service Example

Profile: A BLE Profile is a specification that guarantees application-level interoperability between Profile-compliant devices. It defines the role and configuration of different BLE layers and GATT Service(s) to be supported to create a specific end application or use case. For example, in the case of a heart-rate monitoring device, the BLE Heart Rate Profile defines the required GAP and GATT roles, and the GATT Services to be supported by the heart-rate monitoring device to create an interoperable heart-rate monitoring device. The Bluetooth SIG offers a set of predefined standard Profiles for commonly used BLE end applications. In addition, you can create your own custom Profiles that consist of standard or custom Services.

The Profile defines two application roles:

- **Sensor or Server:** The Sensor Profile role is supported by the application that has data. The Sensor Profile specification defines the required roles (for example, GATT/GAP roles) and behavior (for example, advertisement interval, GATT Services to be supported) of the BLE device to support a Sensor application use case. Following the Sensor Profile specification guarantees interoperability of the Sensor application with any other device that implements the corresponding Collector Profile. For example, a heart-rate monitoring device that implements the Heart Rate Sensor Profile will be interoperable with all smartphones that implement the Heart Rate Collector Profile.
- **Collector or Client:** The Collector Profile role is supported by the application that wants data. The Collector Profile specification defines the required BLE device roles and behavior to interoperate with and collect information from any device that implements the corresponding Sensor Profile specification.

Figure 5 illustrates the data abstraction and hierarchy in a BLE device.

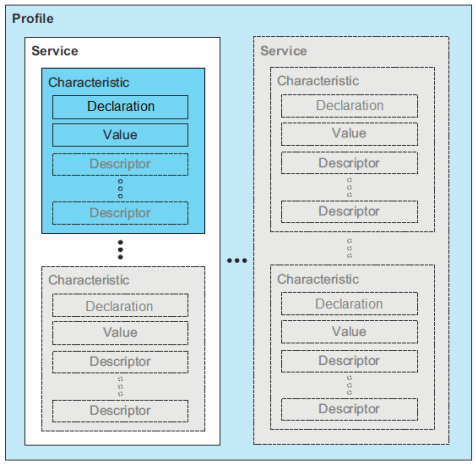


Figure 5 BLE Data Hierarchy[1]

Figure 6 illustrates the application-level interaction of two connected BLE devices using the GAP and GATT protocol layers.

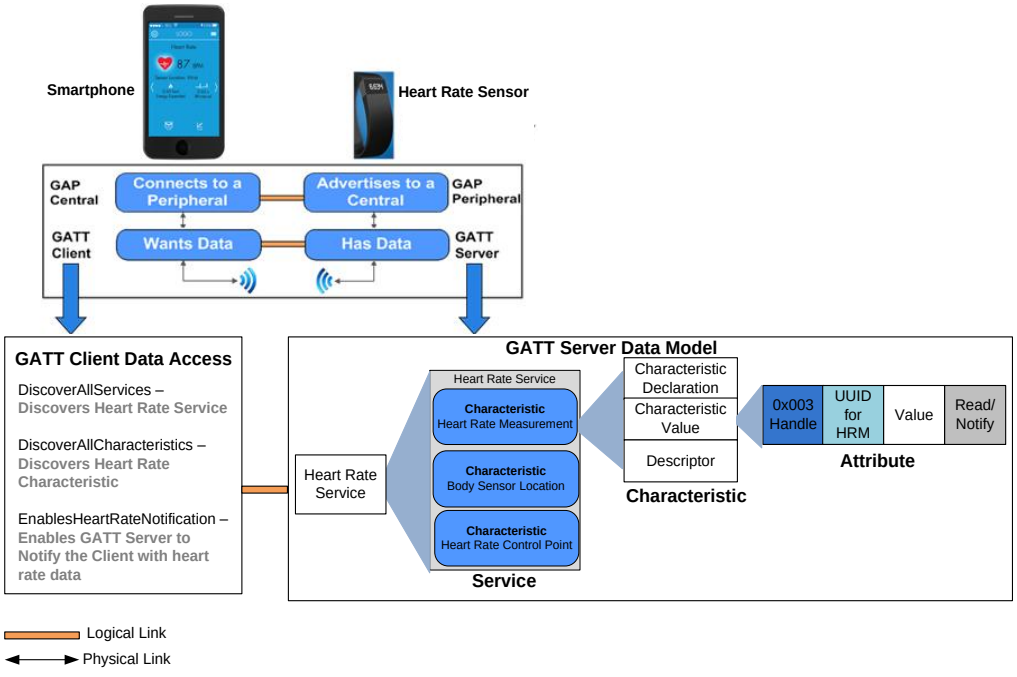


Figure 6 BLE Application Level Overview

3.2 BLE Controller

The BLE Controller implements the Link Layer (LL) of the BLE protocol. In PSoC 6 BLE, LL is a hardware-firmware co-implementation. The LL firmware implements a state machine that manages and controls the physical BLE connections between devices. It supports all LL states such as Advertising, Scanning, Initiating,

[1] Image courtesy of Bluetooth SIG

and Connecting. It implements all the key link control procedures such as LE Encryption, LE Connection Update, LE Channel Update, and LE Ping.

3.2.1 Managing Multiple Connections

The PSoC BLE stack may have up to four instances of the LL to support the multiple connection feature. Each LL instance caters to a particular BLE link and manages the LL functionality such as Advertising, Scanning, Initiating, and Connecting, independent of other BLE connection links.

During a connection event, the BLE stack assigns an identification number called “bdHandle” to the connecting peer device. A BLE event called `CY_BLE_EVT_GAP_DEVICE_CONNECTED` is triggered with an event parameter containing the bdHandle. During an active BLE connection, the application identifies the connected peer device using the bdHandle. **Figure 7** illustrates a simplified block diagram on managing multiple BLE connections using PSoC 6 BLE.

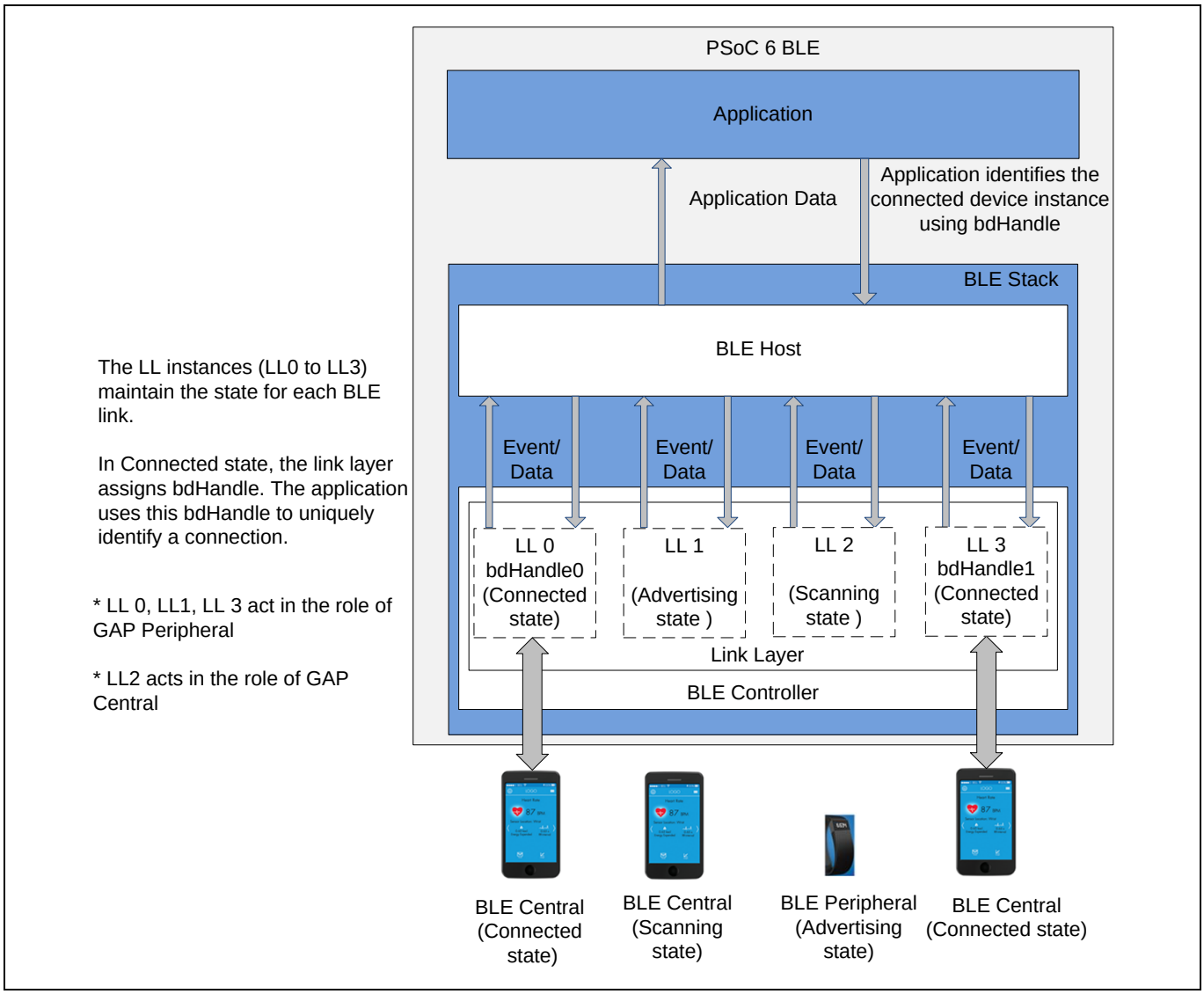


Figure 7 Multiple BLE Connections Using PSoC 6 BLE

This section described how a BLE link is established using the GAP layer and how the data is represented using the GATT layer. With this background, let us examine the typical firmware flow for developing a BLE application.

4 Developing a BLE Application: Firmware Flow

This section provides an overview of the minimal steps to be followed while developing BLE applications using PSoC 6 BLE. The BLE Component in PSoC Creator abstracts the BLE protocol into a simple and easy-to-use GUI. **Figure 8** shows a high-level architecture of the BLE Component, which illustrates the relationship between the BLE stack layer and the route in which the application interacts with the Component.

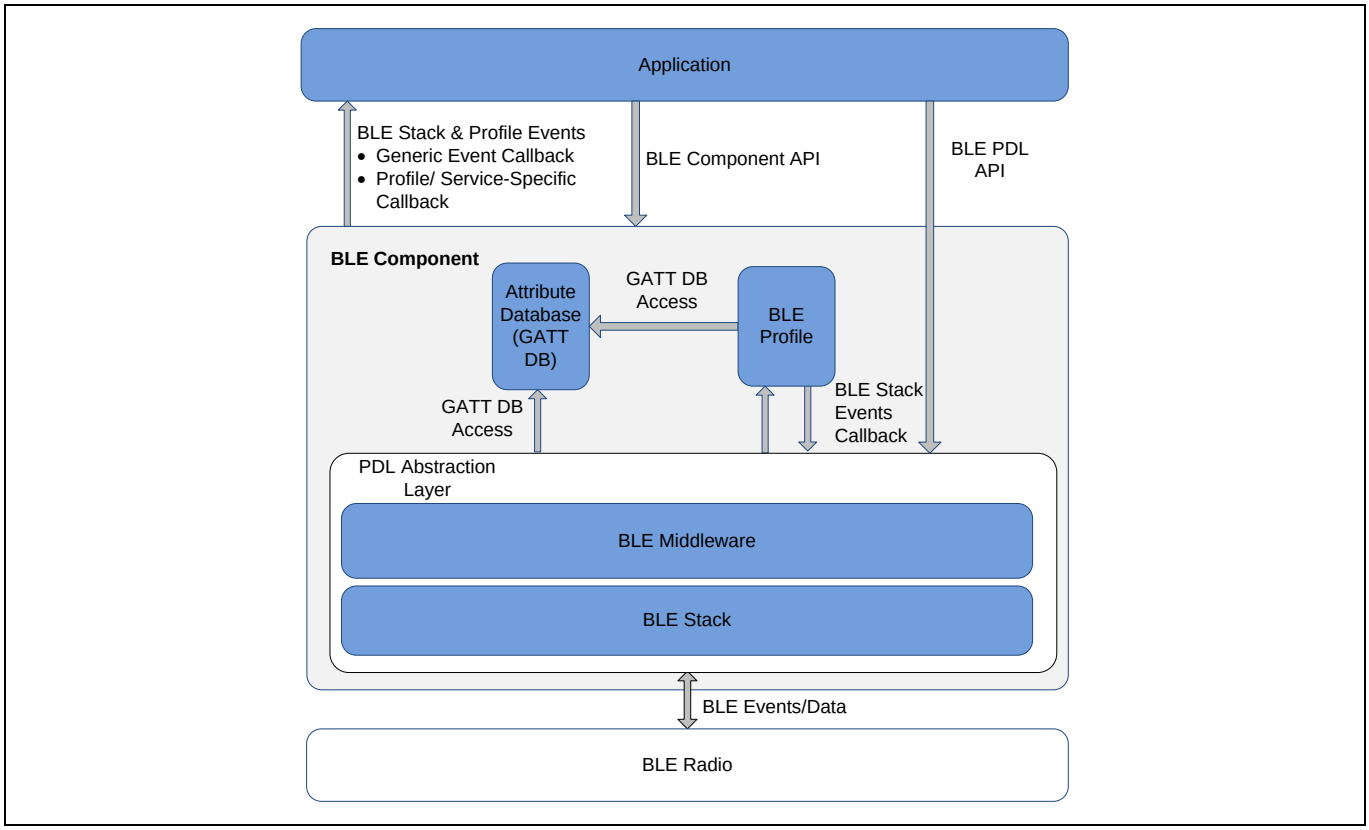


Figure 8 BLE Component Architecture

There are two major steps for developing a BLE application with PSoC 6 BLE:

- Configure the BLE Component.
- Write the firmware.

The following sections provide the details on how to accomplish each task.

- **Configure the BLE Component**
  - **Select the GAP Role and Number of Simultaneous Connections**

    PSoC 6 BLE supports up to four simultaneous connections in any combinations of GAP roles (Central, Peripheral, Observer, Broadcaster). In this step, you configure the desired GAP roles and the number of simultaneous BLE connections.
  - **Select the BLE Architecture**

    PSoC 6 BLE offers Dual CPU architecture, where the Controller runs on the CM0+ CPU and Host runs on CM4 CPU, and Single CPU Architecture, where both Controller and Host are run only on one CPU (either CM0+ or CM4). **Figure 9** illustrates this sub steps A and B.

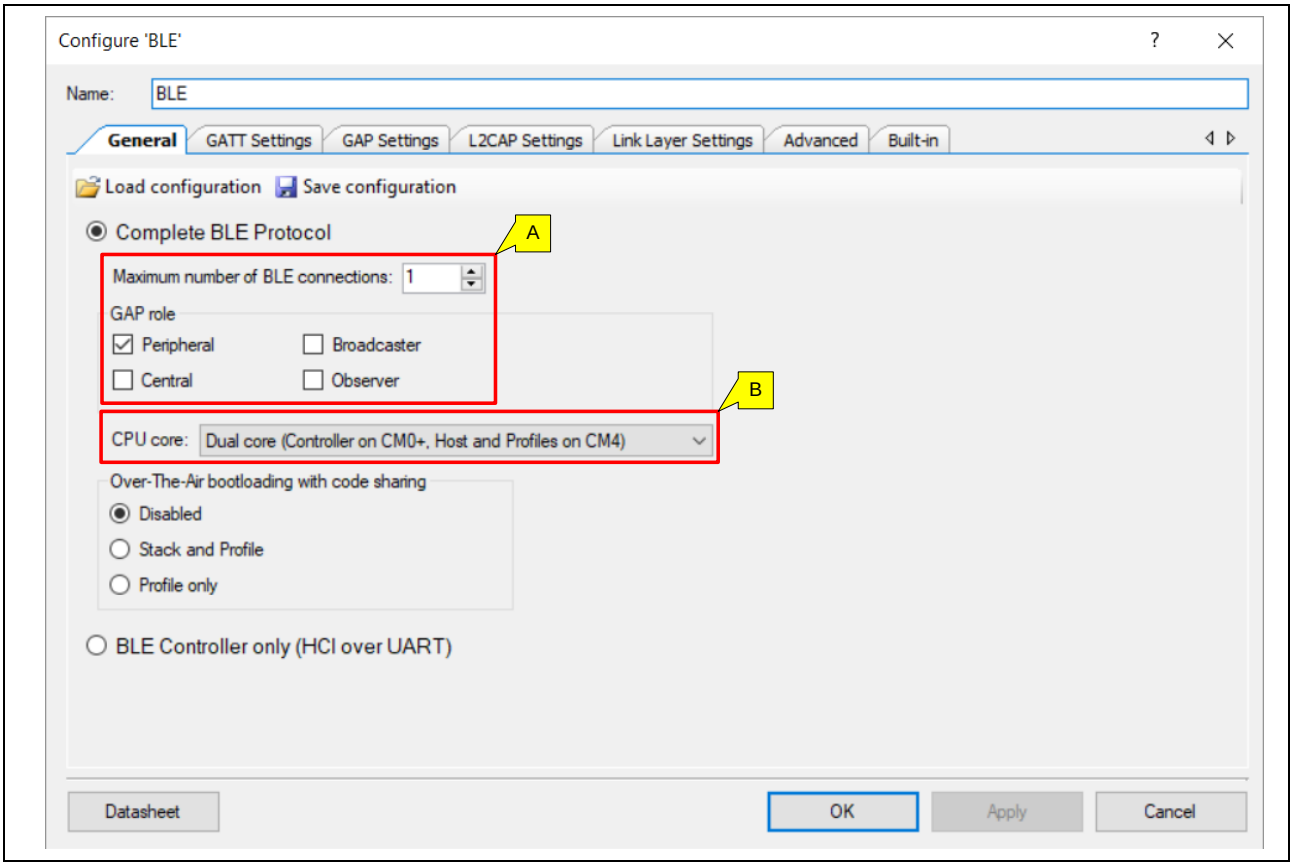


Figure 9 General BLE Settings

Assign the BLE Subsystem (BLESS) interrupt to the CPU on which the BLE Controller runs. In PSoC Creator, assign the BLESS interrupt to the Controller CPU in the **Interrupts** tab of the **Design Wide Resources** window as shown in **Figure 10**. By default, the BLESS interrupt is assigned to the CM0+ CPU.

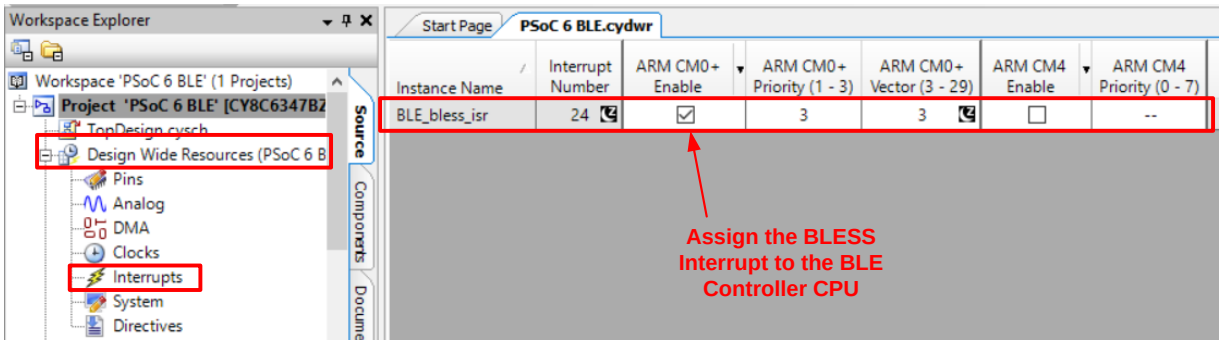


Figure 10 Assigning the BLESS Interrupt to the BLE Controller CPU

Note: The rest of the document assumes dual-CPU architecture for the BLE application.

– Configure GAP Settings

Depending upon the GAP role of the device, you must configure a few settings specific to the role. For example, if the device is a GAP Peripheral, then you must provide the Advertisement settings, Advertisement packet structure, etc. Similarly, if it is GAP Central, you need to provide the scan setting and connection parameters. If the device has multiple roles, you need to provide the configuration settings for each GAP role. **Figure 11** to **Figure 15** show GAP configuration.

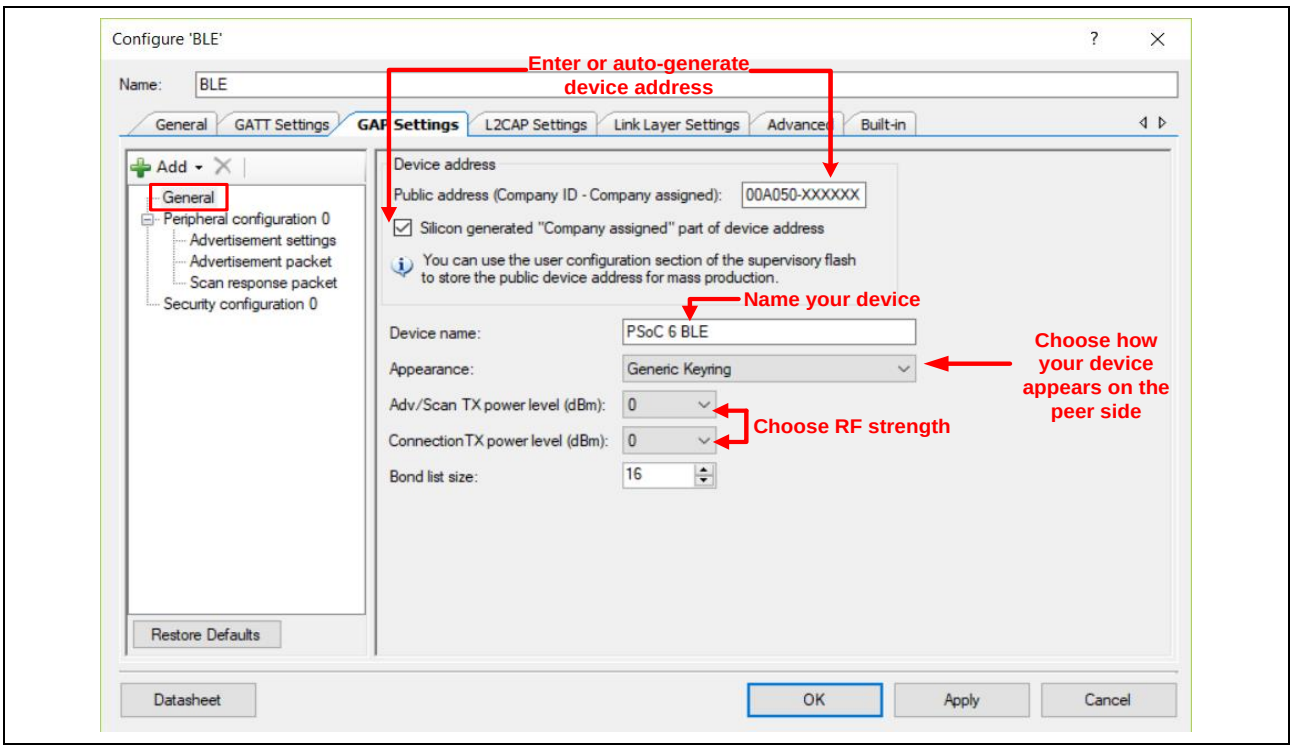


Figure 11 GAP General Settings

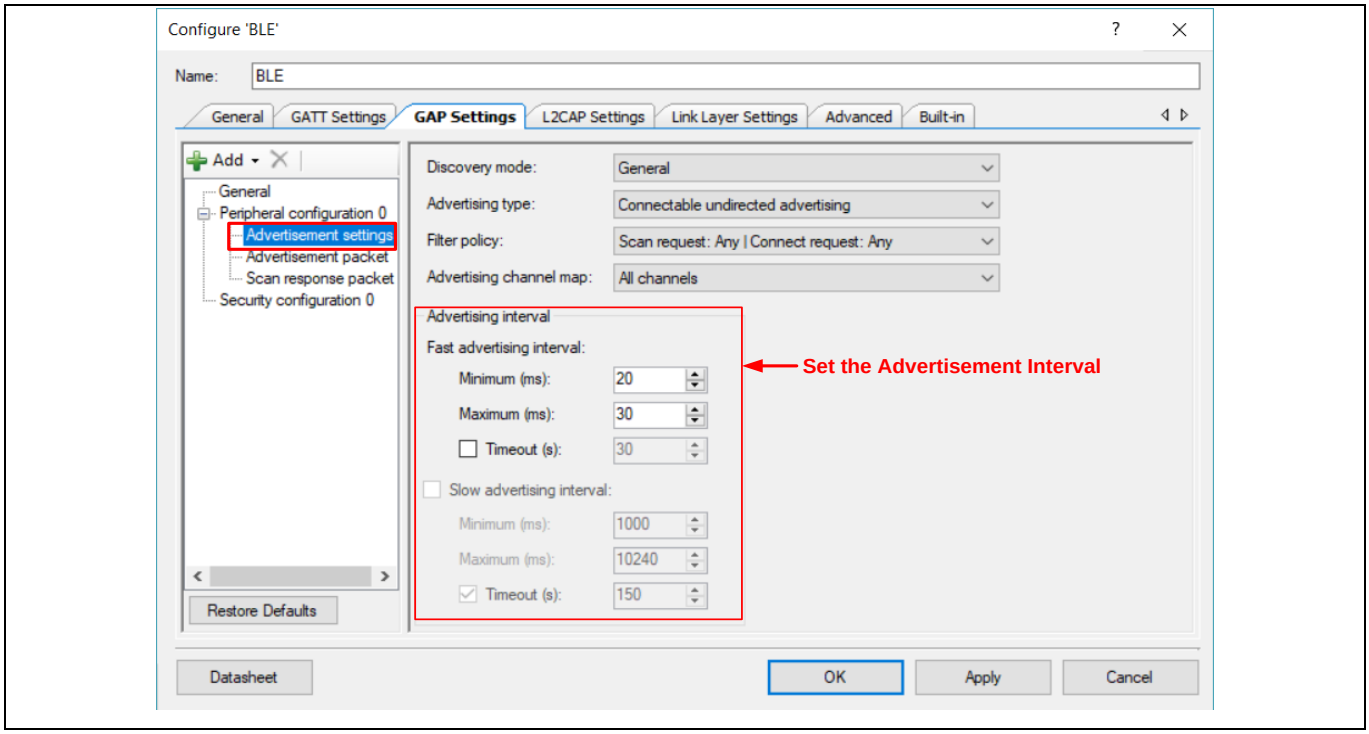


Figure 12 GAP Peripheral Configuration: Advertisement Settings

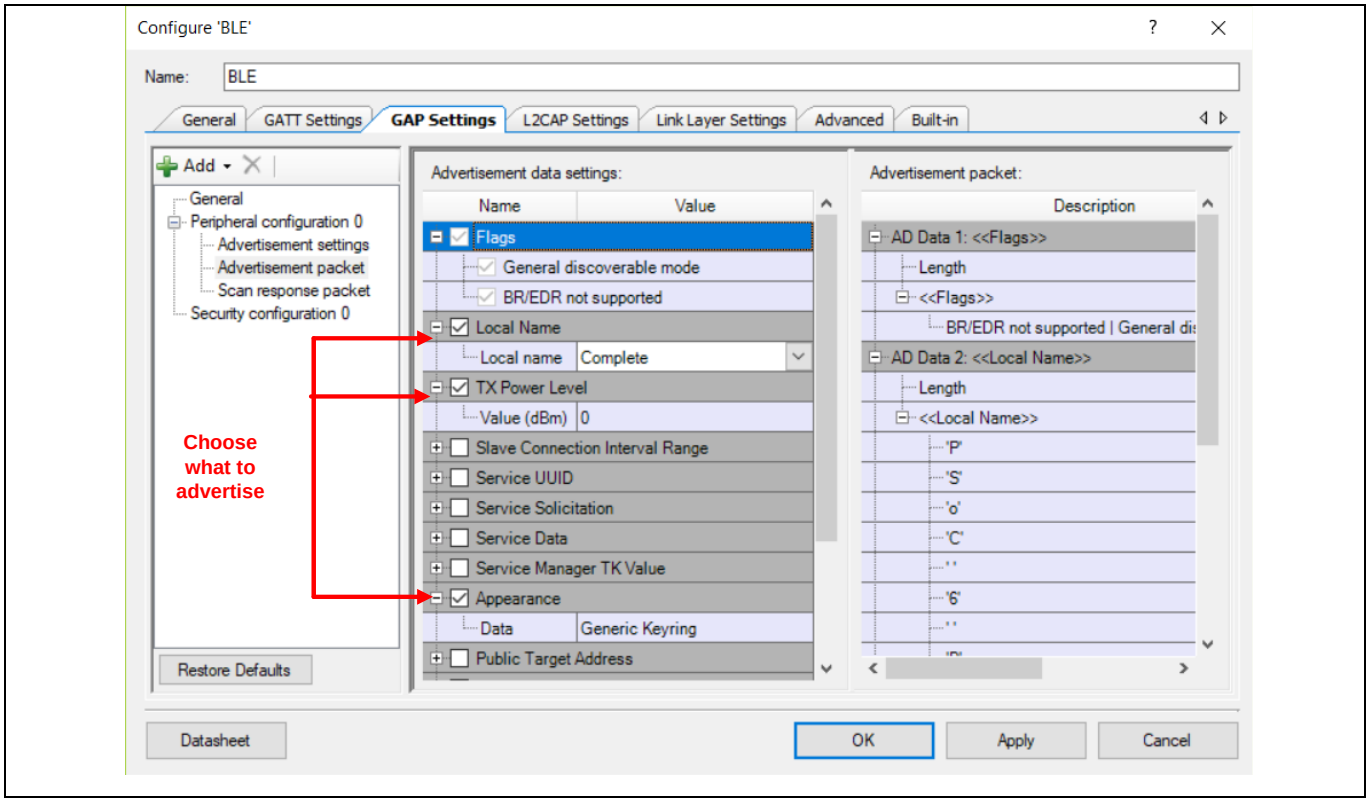


Figure 13 GAP Peripheral Configuration: Advertisement Packet

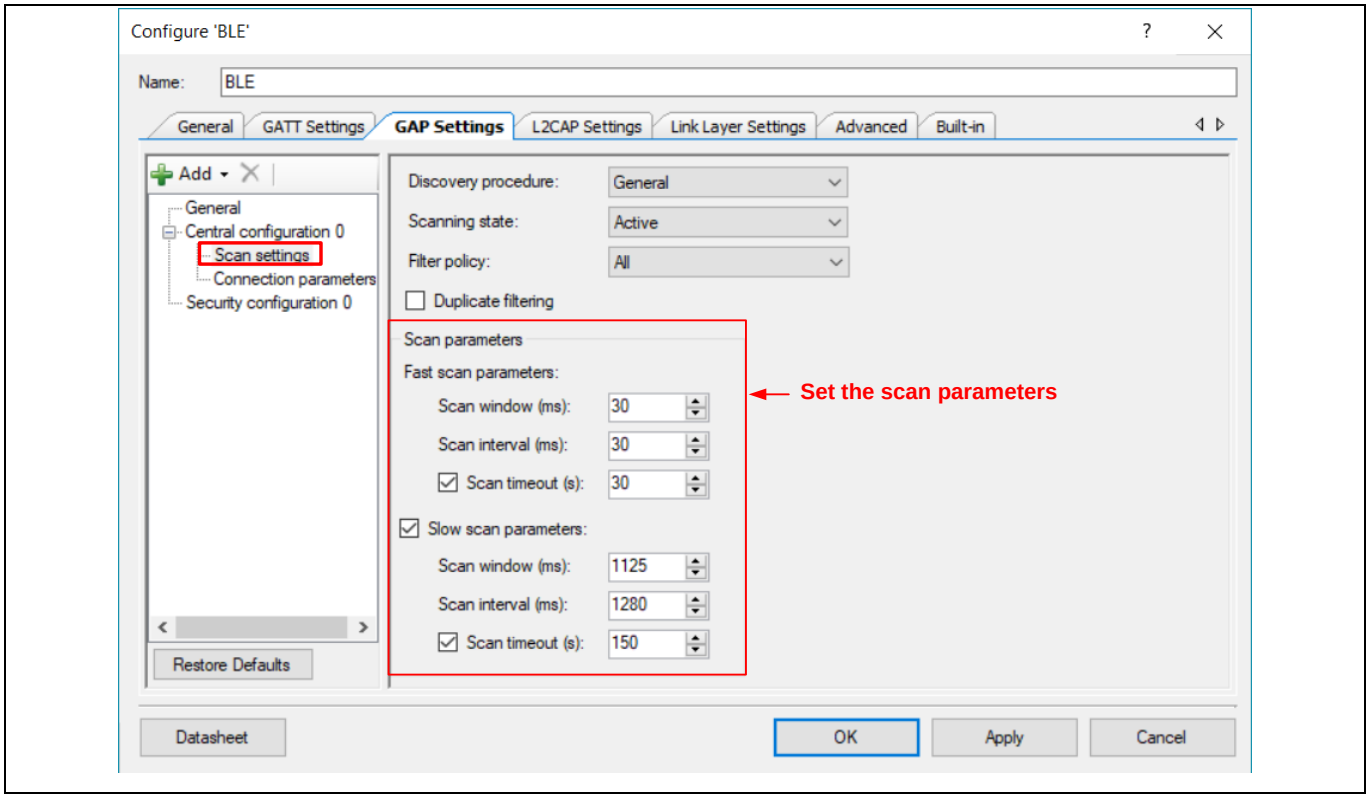


Figure 14 GAP Central Configuration: Scan Settings

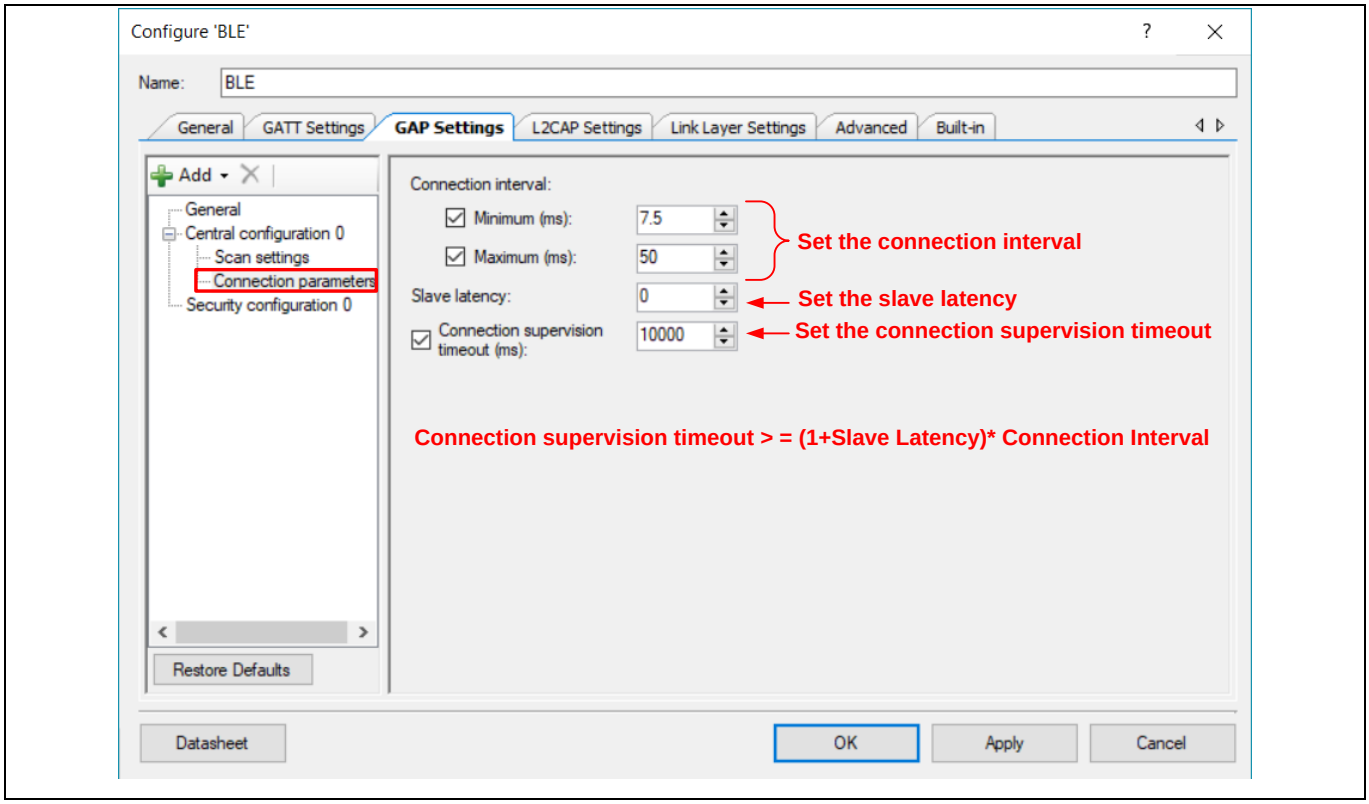


Figure 15 GAP Central Configuration: Connection Parameters

The parameters in GAP configuration settings are explained in **Section 3.1**. For a detailed description of each parameter, see the **BLE Component datasheet**. Further, examples provided in **Section 5: BLE Design Examples** illustrate how these parameters are set.

- **Configure GATT Settings**

  In this step, you include the services that you need in your application. The BLE Component supports numerous Bluetooth SIG-adopted GATT-based Profiles and Services. The Component generates all the necessary code for a Profile/Service operation as configured in the Component Customizer. **Figure 16** shows an example for configuring a standard Heart Rate Service (HRS).

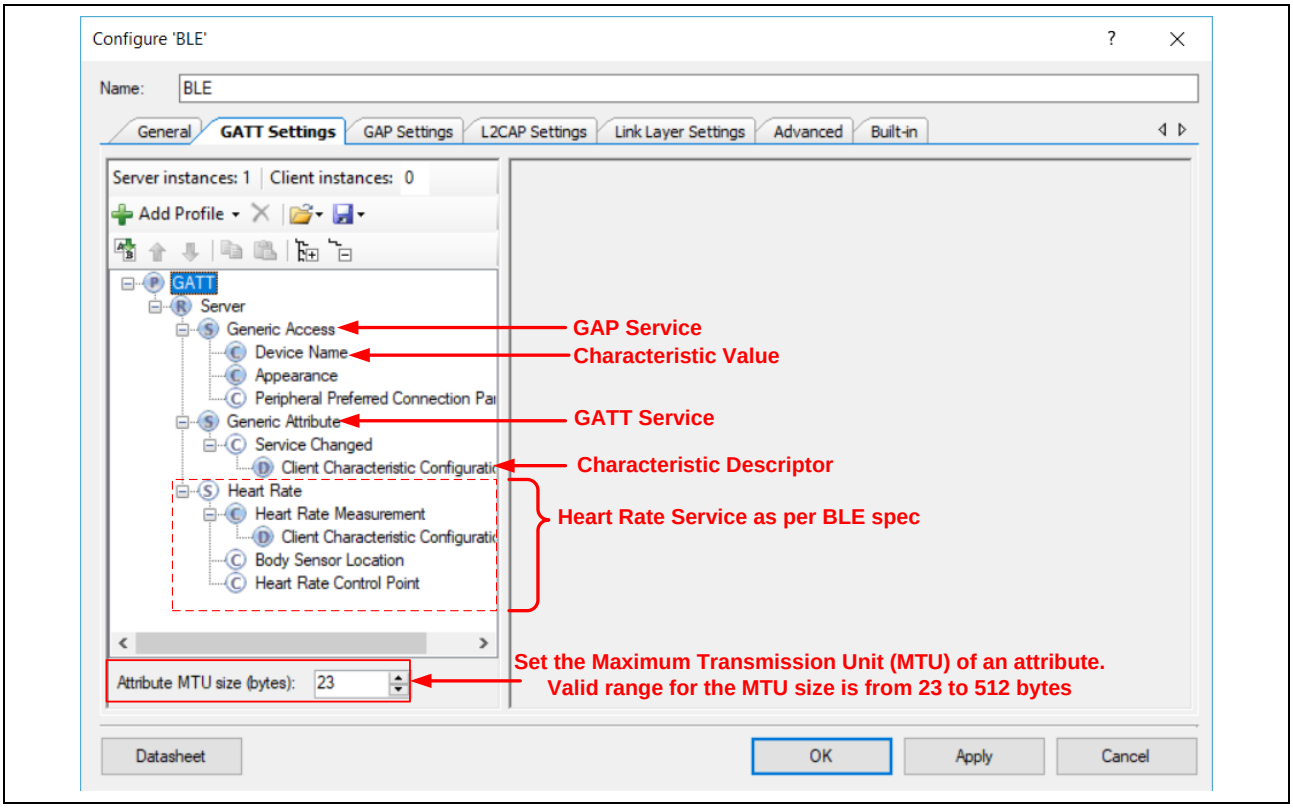


Figure 16 BLE Component GATT Configuration

This completes all the necessary steps for GAP and GATT profile-based settings. After you configure the BLE Component, the next step is to write the firmware to initialize the design and register event handler functions to process the BLE events and data as required by your application.

- **Write the Firmware**

This section discusses the typical steps required by a BLE application firmware. For each major task, specific steps are explained along with sample firmware code. For detailed firmware steps, refer to the code examples referenced in **Section 5: BLE Design Examples**.

- Firmware flow for the Controller (CM0+ CPU for Dual-CPU Architecture)

  The typical firmware flow for the controller is as follows:

  1. Start the BLE Controller.
  2. Process BLE events using the Cy_BLE_ProcessEvents()function.

The `Cy_BLE_ProcessEvents()` function checks the internal task queue in the BLE stack and processes any pending BLE operations. As shown in **Figure 17**, this function must be called at least once every 'T' intervals, where 'T' is equal to the Connection Interval or Scan Interval (if the device is a GAP Central), or the Connection Interval or Advertisement Interval (if the device is a GAP Peripheral) whichever is smaller.

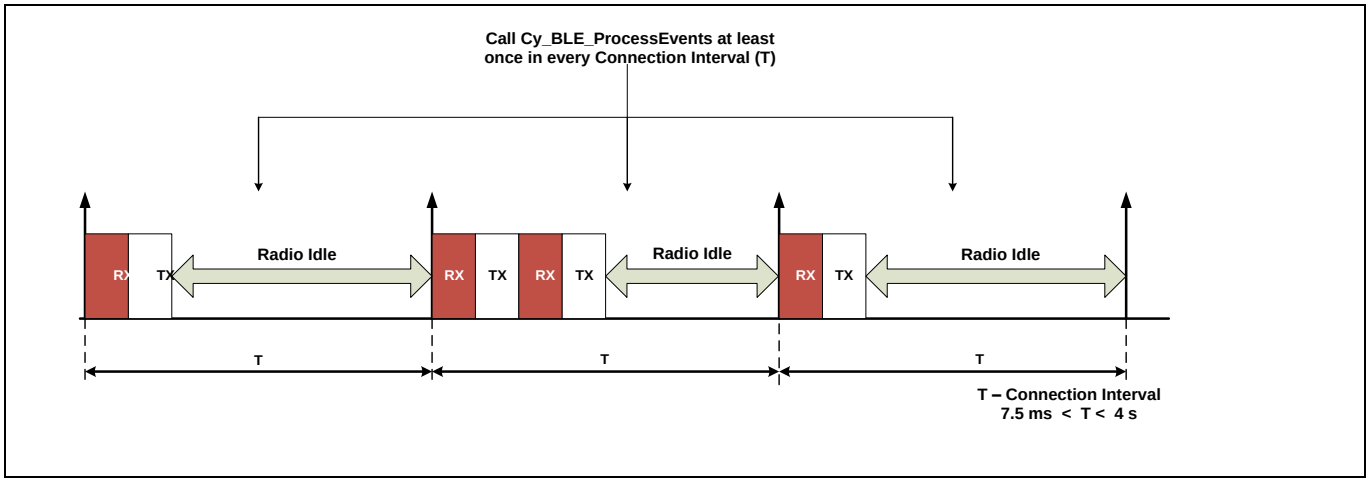


Figure 17 BLE Connection Interval

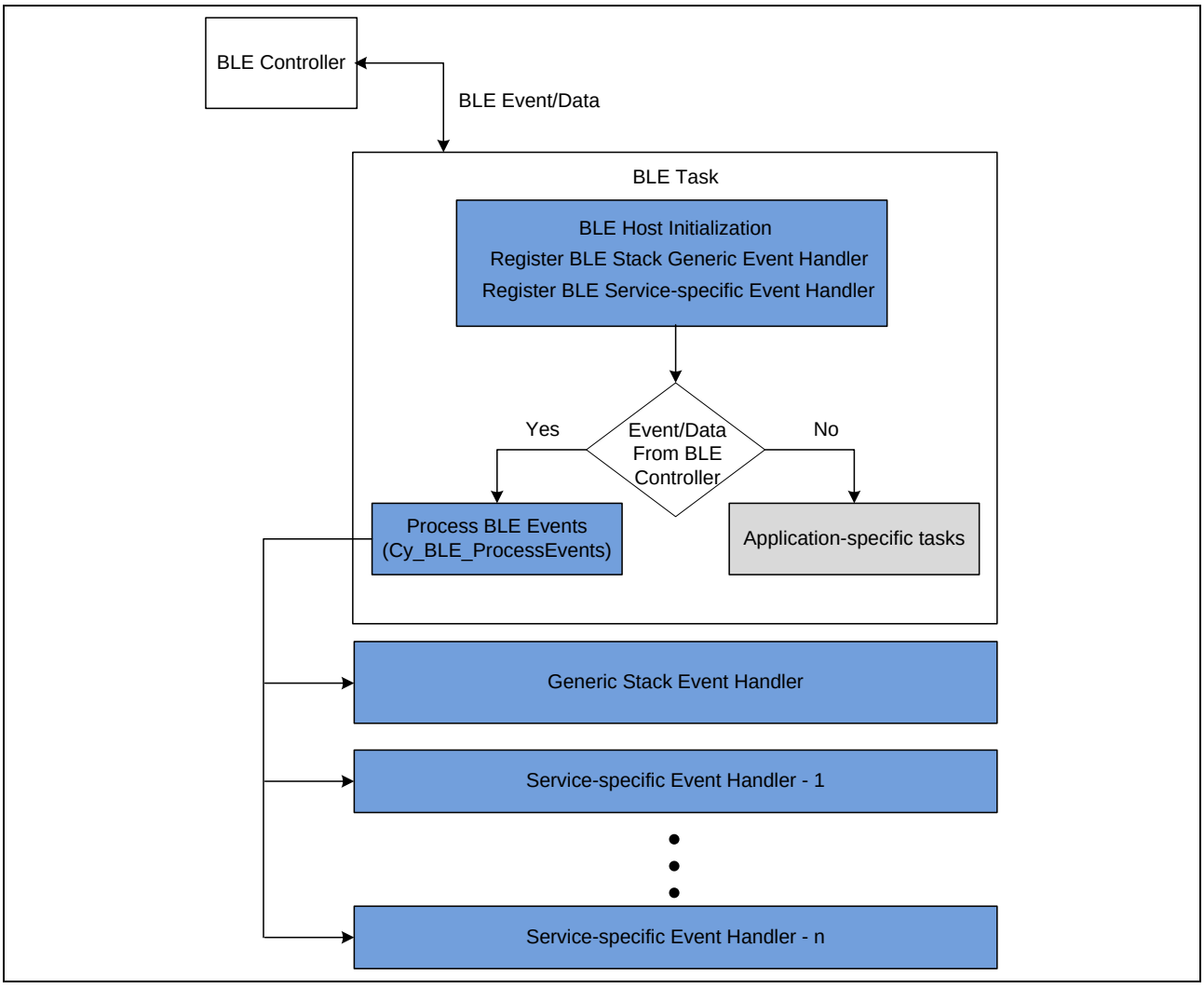
The following code snippet shows the basic controller firmware flow:

```
#include "project.h"

int main(void)
{
    /* Enable global interrupts */
    __enable_irq();

    /* Start the controller portion of BLE. Host runs on the
CM4 */
    if(Cy_BLE_Start(NULL) == CY_BLE_SUCCESS)
    {
            /* Enable CM4 only if BLE Controller started
successfully.
            CY_CORTEX_M4_APPL_ADDR must be updated if CM4
memory layout
            is changed. */
        Cy_SysEnableCM4(CY_CORTEX_M4_APPL_ADDR);
    }
    else
    {
        /* Halt the CPU */
        CY_ASSERT(0u);
    }

    for(;;)
    {
```

```
        /* Process the controller portion of the BLE events
and wake up the host
          (CM4) and send data to the host via IPC if necessary
*/
        Cy_BLE_ProcessEvents();

        /* Put CM0+ to Deep Sleep mode. The BLE hardware
automatically wakes
           up the CPU if processing is required */
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
    }
}/*End of main function */
```

- Firmware flow for the Host (CM4 CPU for Dual-CPU Architecture)

  The interaction between BLE Controller and the BLE Host are event-driven. The BLE stack generates events that provide status and data to the application firmware running on the BLE Host.



Developing a BLE Application: Firmware Flow

The typical firmware flow for the Host is as follows:

- Register the application Host callback function.
  The application Host callback function is called when the BLE Host needs to process pending stack events. The application Host callback function processes the pending events by calling `Cy_BLE_ProcessEvents()`. Note that the application Host callback function executes from within an ISR and must be very short.
- Start the BLE Host by providing the generic BLE generic event handler function.
- Register BLE service-specific event handlers.

```
void BleAppHost_Callback(void)
{
    /* On every interrupt from the BLE Controller, process the
pending
       Host events propagated from the BLE Controller */
    Cy_BLE_ProcessEvents();
}

int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

    /* Start BLE component and register generic event handler
*/
    if(Cy_BLE_Start(GenericStackEventHandler) ==
CY_BLE_SUCCESS)
    {

        /* Register the Host application callback function */
        Cy_BLE_RegisterAppHostCallback(BleAppHost_Callback);

        /* Register Service Specific event handler*/
        Cy_BLE_HRS_RegisterAttrCallback(HRSEventHandler);
    }
    else
    {
        /* Halt the CPU */
        CY_ASSERT(0u);
    }

    for(;;)
    {
```

```
        /* Code specific to your application */
    }
}
```

BLE events are handled with the user-defined generic BLE stack event handler. In the above code snippet, `GenericStackEventHandler()` is a user-defined function to handle BLE events. The generic stack event handler must handle a few basic events from the stack. **Table 1** lists these events.

Table 1 Basic BLE Stack Events

BLE Stack Event Name	Event Description	Event Handler Action
CY_BLE_EVT_STACK_ON	BLE stack initialization is completed successfully.	GAP Central: Discover GAP Peripherals using Cy_BLE_GAPC_StartScan(). GAP Peripheral: Start the advertisement using Cy_BLE_GAPP_StartAdvertisement().
CY_BLE_EVT_GAP_DEVICE_CONNECTED	BLE link with the peer device is established.	Application-specific action.
CY_BLE_EVT_GAPP_ADVERTISEMENT_START_STOP	BLE stack advertisement start/stop event.	Application-specific action.
CY_BLE_EVT_GAP_DEVICE_DISCONNECTED	BLE link with the peer device is disconnected.	GAP Central: Discover GAP Peripherals using Cy_BLE_GAPC_StartScan(). GAP Peripheral: Start the advertisement using Cy_BLE_GAPP_StartAdvertisement().
CY_BLE_EVT_HARDWARE_ERROR	BLE hardware error.	Application-specific action.
CY_BLE_EVT_STACK_SHUTDOWN_COMPLETE	BLE stack has been shut down.	Application-specific action.

The BLE Middleware Library provides functions for registering event handlers specific to standard BLE services. For example, `Cy_BLE_HRS_RegisterAttrCallback(HRSEventHandler)` registers the `HRSEventHandler` function to handle events specific to the Heart Rate Service.

Code examples described in the **Section 5: BLE Design Examples** explain how to define and implement the generic event handler function and BLE service-specific event handlers. **Table 2** lists commonly used functions in a typical BLE design.

For a comprehensive list of BLE stack events and functions, see the Cypress BLE Middleware Library. To open the documentation, from PSoC Creator, navigate to **Help** > **Documentation** > **Peripheral Driver Library.** Locate the **Cypress BLE Middleware Library** under the **Middleware** menu option**.**

Table 2 Basic BLE API Functions

BLE API Function	Description
Cy_BLE_Start()	Initializes the BLE stack and initializes the Profile layer, schedulers, timers, and other platform-related resources required by the BLE Component.
Cy_BLE_GAPP_StartAdvertisement()	GAP Peripherals use this function to start the advertisement using the advertisement data set in the BLE Component.
Cy_BLE_GAPP_StopAdvertisement()	GAP Peripherals use this function to stop advertisement.
Cy_BLE_GAPC_StartScan()	GAP Central uses this function to discover GAP Peripherals that are available for connection.
Cy_BLE_GAPC_StopScan()	GAP Central uses this function to stop the discovery of GAP Peripherals.
Cy_BLE_GAPC_ConnectDevice()	GAP Central uses this function to send connection requests to the GAP Peripheral with the connection parameters set in the BLE Component.
Cy_BLE_ProcessEvents()	Checks the internal task queue in the BLE stack and processes pending BLE events.

4.1 Implementing Low-Power BLE Design

Low-power operation is one of the major considerations when it comes to connected devices. PSoC 6 BLE provides low-power modes without sacrificing the performance. PSoC 6 BLE has several power modes that can affect either the whole system or just a single CPU. CPU power modes are active, sleep, and deep sleep as defined by Arm®. Device system power modes are Low Power (LP), Ultra Low Power (ULP), deep sleep, and hibernate.

AN219528 – PSoC 6 MCU Low-Power /modes and Power Reduction Techniques describes the low-power design considerations and the low-power modes in PSoC 6 MCU. In this section, we shall see how to implement a low-power design involving BLE connectivity. For low-power support, configure the BLE Component as shown in **Figure 18**.

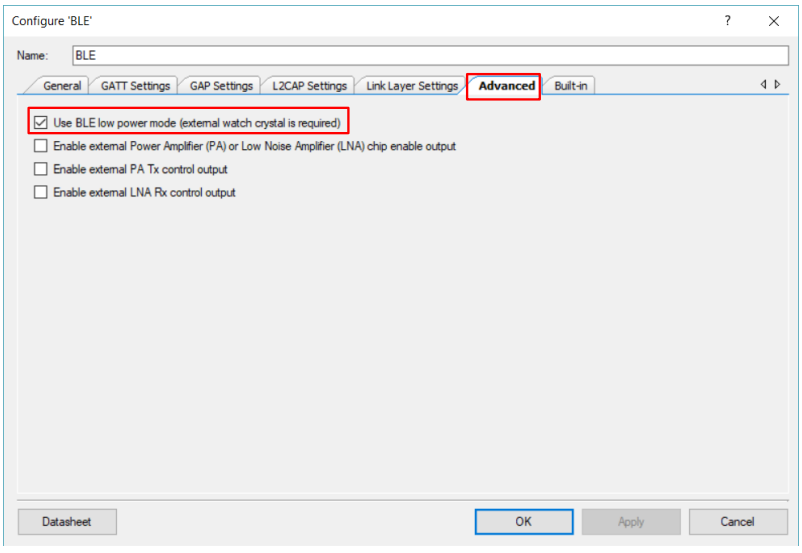


Figure 18 Configuring BLE Low-Power Support

To operate the BLE Component in low-power mode, it is mandatory to use an external watch crystal oscillator (WCO) as the source to the low-frequency clock (LFCLK) of PSoC 6 BLE. **Figure 19** shows configuring the WCO as the LFCLK source in the Design Wide Resources.

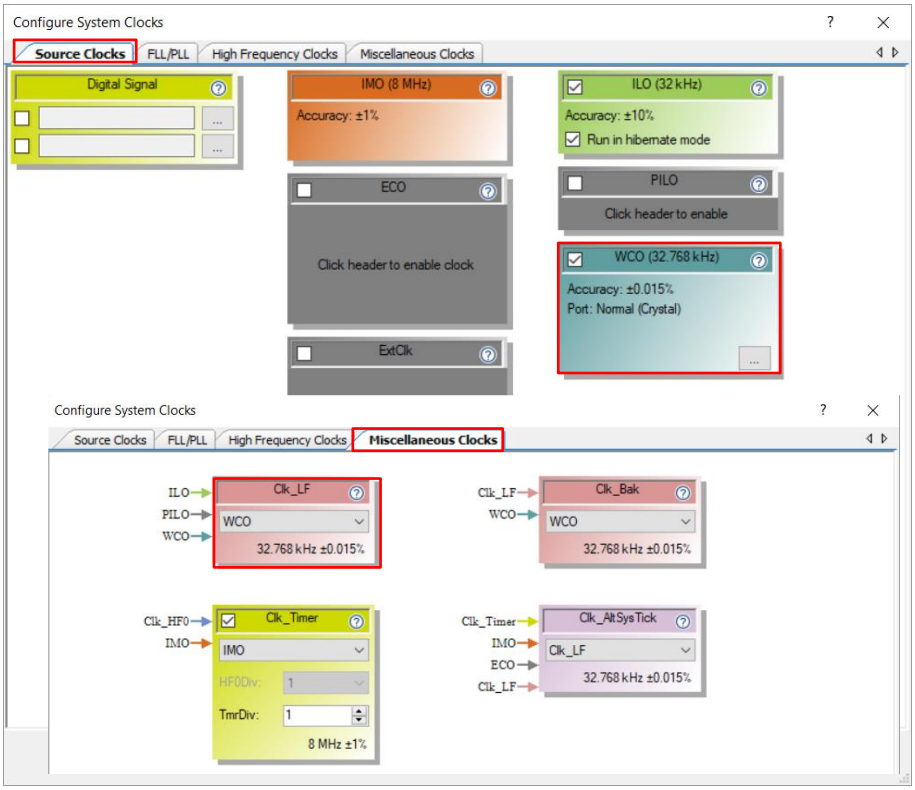


Figure 19 Enabling WCO for Low-Power Operation

In PSoC 6 BLE, wakeup from the low-power mode operation is possible with a set of interrupts handled by the Wakeup Interrupt Controller (WIC) block of PSoC 6 BLE. The WIC block supports up to 41 interrupts that can wake up a CPU from the CPU deep sleep power mode. Refer to **AN217666 – PSoC 6 MCU Interrupts** for more details on PSoC 6 MCU interrupts.

The BLESS interrupt is a CPU deep sleep-capable interrupt source. As described previously, BLE is implemented based on a Host-Controller architecture. The discussion that follows is split into two parts based on the BLE architecture: the Dual-CPU architecture and the Single-CPU architecture:

Dual-CPU Architecture: The BLESS interrupt is mapped to the BLE Controller CPU. The Host CPU and the Controller CPU can independently go into low-power mode if there are no BLE events to be processed by them. The following sequence of events is involved in a typical low-power BLE design for dual-CPU architecture:

- The BLESS interrupt wakes up the Controller during every connection interval.
- The controller CPU then services the BLE event with `Cy_BLE_ProcessEvents()` in firmware.
- If the Host must process a BLE event or data, the Controller wakes up the Host CPU using the Inter-Processor Communication (IPC) interrupt.
- Upon receiving the BLE event or data through the IPC interrupt, the Host CPU processes it using `Cy_BLE_ProcessEvents()` and registered event handler functions. If the Host CPU requires the service of the Controller CPU, the Host sends the BLE data/event using IPC.

Note: *The BLE Component uses system IPC pipes for communication between the Host and the Controller. System IPC pipes and associated interrupts are automatically configured by PSoC Creator.*

1. The Host CPU goes back to the low-power mode and waits for the IPC interrupt from the controller with the Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT) function.
2. The Controller processes the BLE events/data from the Host (if any). The Controller CPU goes back to the low- power mode and waits for the BLESS interrupt using the Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT) function.

Single-CPU Architecture: In single-CPU architecture, both the Controller and the Host runs on the same CPU (CM0+ or CM4). The CPU used by the BLE Component is hereafter referred to as the BLE CPU for the sake of explanation. The following sequence of events is involved in a typical low-power BLE design for single-CPU architecture:

- The BLESS interrupt wakes up the BLE CPU.
- The `Cy_BLE_ProcessEvents()` function processes both the Controller and the Host BLE events/data.
- The BLE CPU goes back to the low power mode and wait for the BLESS interrupt using the `Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT)` function.

Note: *When the design enables BLE low-power mode as shown in **Figure 18**, the BLE Component registers the low-power callback function. When the low-power mode transition is about to occur using the `Cy_SysPm_DeepSleep()` function, the registered callback function is executed. The CPU enters low-power mode only if the BLE stack is ready for low-power operation. If the design involves other blocks capable of low-power operation, your application must define how power mode transitions occur. See 'System Power Management' in the PDL API Reference Manual.*

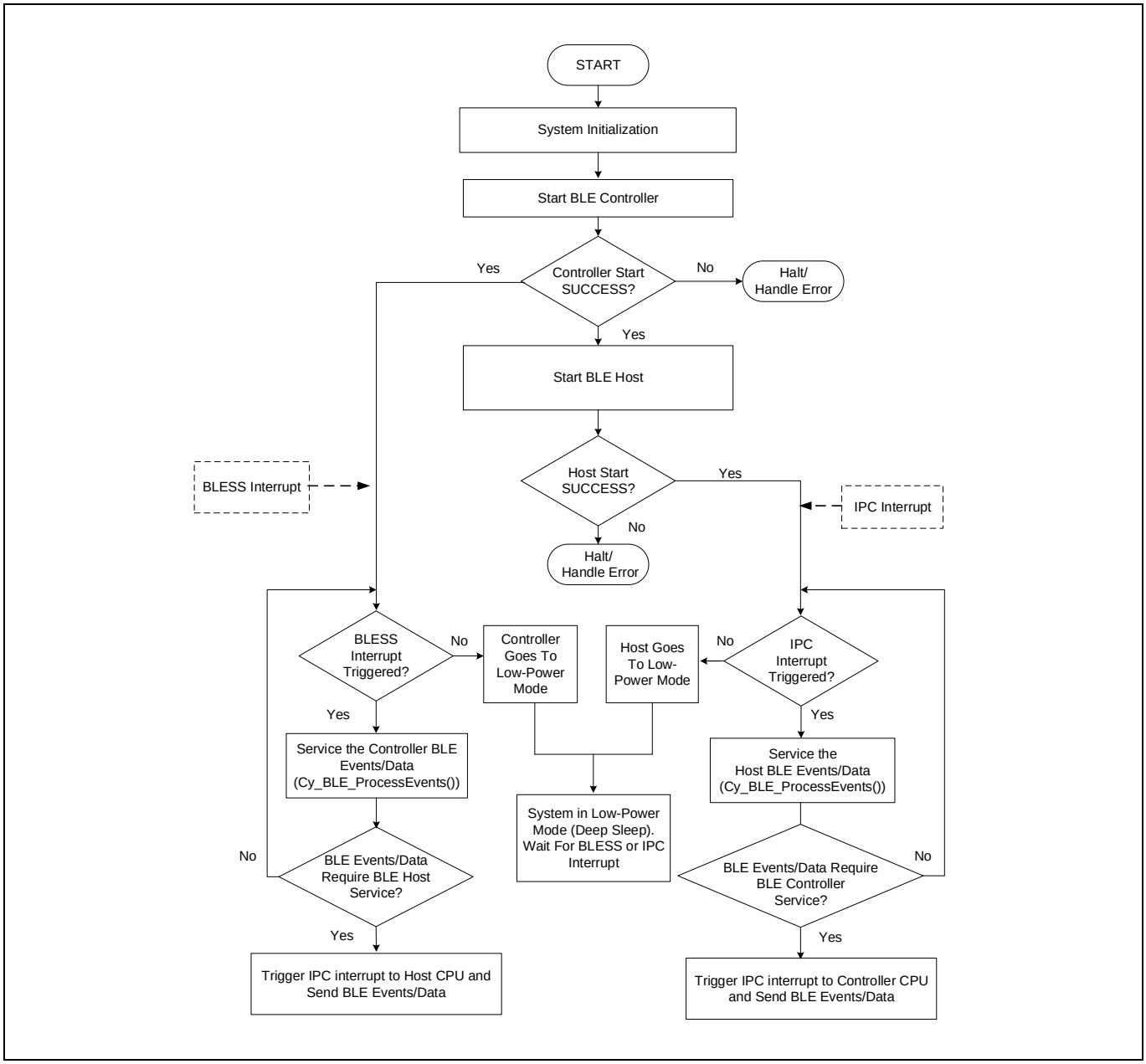


Figure 20 Typical Firmware Flow for Low-Power BLE Design

4.2 Implementing a Secure BLE Design

For every wireless device, protecting a user's private data is of paramount significance. PSoC 6 BLE is compliant with the Bluetooth Low Energy (BLE) 4.2 security features. The BLE stack in PSoC 6 BLE supports the Security Manager Protocol (SMP) with the following features:

- Encryption and authentication of user data
- Authenticated man-in-the-middle (MITM) protection and data signing
- Support for device bonding and different pairing methods such as Just Works, Passkey Entry, Out of Band, and Numeric Comparison
- Pairing method selection based on the I/O capability of the GAP Central and the GAP Peripheral devices

This section discusses how to configure the BLE Component to incorporate the security features into your design and handle them in firmware. For technical details on the BLE 4.2 security features, see the Bluetooth

Core Specification version 4.2 Volume 3 Part H. For details on developing secure embedded system with PSoC 6 MCU, see **AN221111 – PSoC 6 MCU: Creating a Secure System**.

Before proceeding further on the BLE security features, two new terms need to be introduced. The Security Manager Protocol (SMP) layer of the BLE stack defines two roles for a BLE device in establishing a secure connection. They are:

Initiator: The Link Layer Master always initiates a secure connection. Therefore, a GAP Central device is the Initiator.

Responder: The Link Layer Slave responds to the secure connection request from the Link Layer Master; therefore, a GAP Peripheral device is the Responder.

4.2.1 Configuring Security Features Using the BLE Component Security Mode and Security Level

PSoC 6 BLE supports two modes of security with multiple security levels within each mode as **Table 3** shows.

Table 3 BLE Security Modes and Levels

Security Mode	Security Level	Remark
Mode 1	No Security (No authentication, no encryption)	This mode is used in designs where data encryption is required.
	Unauthenticated pairing with encryption	
	Authenticated pairing with encryption	
	Authenticated LE Secure Connections pairing with encryption	
Mode 2	Unauthenticated pairing with data signing	This mode is used in designs where data signing is required.
	Authenticated pairing with data signing	

I/O Capabilities

Security levels described in **Error! Reference source not found.** take effect based on the device's input and output capability. Every BLE device determines its peer device input and output capabilities during the pairing request phase explained in **Establishing Secure BLE Link: Firmware Flow**. **Table 4** summarizes the device capabilities configurable from the BLE Component:

Table 4 Device I/O Capability

I/O Capabilities	Description	GAP Authentication Required (Yes/No)
Display	Used in devices with display capability and may display authentication data.	Yes
Display Yes/No	Used in devices with display and at least two input keys for the Yes/No action.	Yes
Keyboard	Used in devices with numeric keypad.	Yes
No Input No Output	Used in devices that don't have any capability to enter or display the authentication key data to the user.	No
Keyboard and Display	Used in devices like PCs and tablets.	Yes

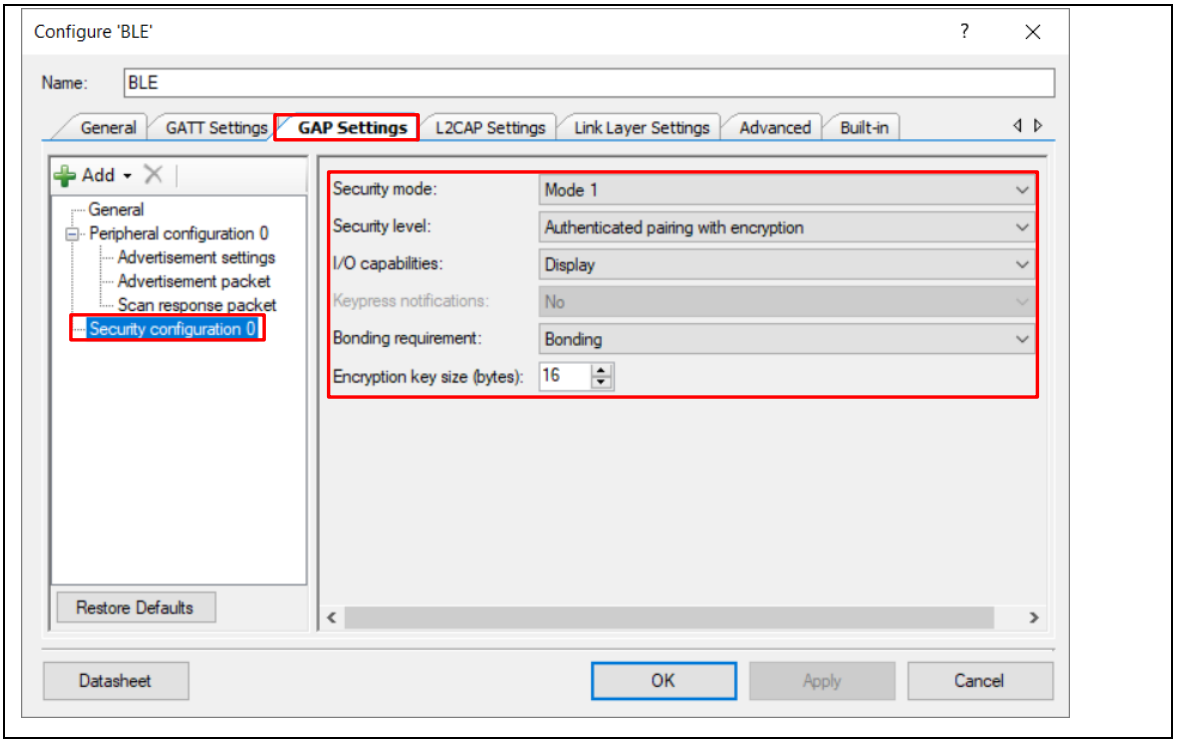


Figure 21 Configuring BLE Security Settings

When two devices want to communicate securely with BLE, they need to follow authentication procedure called “pairing”. PSoC 6 BLE supports four pairing methods compliant with BLE 4.2 feature as follows:

- **Just Works:** When the I/O capability of the device is set to **No Input No Output** and no MITM protection is required, the BLE device uses this pairing method.
- **Passkey Entry:** The user either inputs an identical passkey into both devices, or one device displays the passkey and the user enters that passkey into the other device.
- **Numeric Comparison:** Both devices display a six-digit number and the user authenticates by selecting ‘Yes’ if both devices display the same number.
- **Out of Band (OOB):** This pairing method is used when both devices have access to an out-of-band mechanism to discover the devices as well as exchange the information used in the pairing procedure.

See **AN99209 – PSoC 4 BLE and PRoC BLE: Bluetooth LE 4.2 Features** for an elaborate description on BLE pairing process and establishing a secure connection.

The pairing method is determined based on the device security mode and the I/O capability of the interacting BLE devices as summarized in **Table 5**.

Table 5 Pairing Method Based on the I/O Capabilities of Interacting BLE Devices

GAP Role	Initiator					
Responder	I/O Capabilities	Display Only	Display (Yes/No)	Keyboard only	No Input No Output	Keyboard and Display
	Display Only	Just Works	Just Works	Passkey Entry	Just Works	Passkey Entry
	Display (Yes/No)	Just Works	Numeric Comparison	Passkey Entry	Just Works	Numeric Comparison
	Keyboard only	Passkey Entry	Passkey Entry	Passkey Entry	Just Works	Passkey Entry
	No Input No Output	Just Works	Just Works	Just Works	Just Works	Just Works
	Keyboard and Display	Passkey Entry	Numeric Comparison	Passkey Entry	Just Works	Numeric Comparison

4.2.2 Establishing Secure BLE Link: Firmware Flow

The following steps illustrate the firmware flow for establishing a secure BLE link:

- The pairing procedure requires generating a set of security keys. These keys need to be generated irrespective of the GAP role. The BLE device exchanges the generated key with the peer device during the key exchange stage of the authentication procedure. On the `CY_BLE_EVT_STACK_ON` BLE event, generate the keys using the `Cy_BLE_GAP_GenerateKeys()` function.
- Upon successful generation of security keys, the `CY_BLE_EVT_GAP_KEYS_GEN_COMPLETE` event is triggered. Set the device identity address using the `Cy_BLE_GAP_SetIdAddress()` function.
- On `CY_BLE_EVT_GAP_DEVICE_CONNECTED` or `CY_BLE_EVT_GAP_ENHANCE_CONN_COMPLETE` event, the BLE device sets the security keys using the `Cy_BLE_GAP_SetSecurityKeys()` function. The Link Layer of the BLE stack uses these keys for encrypting the data.
- The Initiator sends a pairing request using `Cy_BLE_GAP_AuthReq()`. The parameter passed through this function contains the security mode/level, I/O capability, encryption key size, etc. This triggers the `CY_BLE_EVT_GAP_AUTH_REQ` BLE event at the GAP Peripheral.
- The GAP Peripheral (Responder) sends a request to the GAP Central device (Initiator) to initiate the pairing procedure using the `Cy_BLE_GAP_AuthReq()` function. This triggers the `CY_BLE_EVT_GAP_AUTH_REQ` BLE event at the GAP Central device.
- On receiving the `CY_BLE_EVT_GAP_AUTH_REQ` event, the GAP Peripheral responds with `Cy_BLE_GAPP_AuthReqReply()`. The pairing response contains much of the same information as the Initiator pairing request parameter.
- On successful pairing information exchange, the BLE stack generates the `CY_BLE_EVT_GAP_SMP_NEGOTIATED_AUTH_INFO` event at both the GAP Central and GAP Peripheral devices.
- During the authentication procedure, the BLE stack generates one of the following events based on the BLE device's I/O capabilities and security modes:
  - CY_BLE_EVT_GAP_PASSKEY_DISPLAY_REQUEST
  - CY_BLE_EVT_GAP_KEYPRESS_NOTIFICATION
  - CY_BLE_EVT_GAP_NUMERIC_COMPARISON_REQUEST

- On successful completion of the pairing procedure, the BLE stack generates the `CY_BLE_EVT_GAP_AUTH_COMPLETE` event.

  If authentication fails, `CY_BLE_EVT_GAP_AUTH_FAILED` is generated. Typically, this event is handled by disconnecting the BLE link using the `CY_BLE_GAP_Disconnect()` function.

Table 6 lists the sequence of BLE stack events and the action to be taken when using the LE Secure Connection feature.

Table 6 BLE Events and Corresponding Actions for LE Secure Connections

BLE Stack Event(s)	Event Handler Action
`CY_BLE_EVT_STACK_ON`	Generate the security keys using `Cy_BLE_GAP_GenerateKeys()`.
`CY_BLE_EVT_GAP_KEYS_GEN_COMPLETE`	Set the device identity address using `Cy_BLE_GAP_SetIdAddress()`.
`CY_BLE_EVT_GAP_DEVICE_CONNECTED` or `CY_BLE_EVT_GAP_ENHANCE_CONN_COMPLETE`	Set the security keys using `Cy_BLE_GAP_SetSecurityKeys()`. Additionally, if it is a GAP Peripheral, send a request to the GAP Central to initiate pairing using `Cy_BLE_GAP_AuthReq()`.
`CY_BLE_EVT_GAP_AUTH_REQ`	GAP Central: Initiate pairing using `Cy_BLE_GAP_AuthReq()`. GAP Peripheral: Respond to the pairing request using `Cy_BLE_GAPP_AuthReqReply()`.
`CY_BLE_EVT_GAP_SMP_NEGOTIATED_AUTH_INFO`	Initiate application-specific action.
`CY_BLE_EVT_GAP_PASSKEY_DISPLAY_REQUEST`	Display the 6-decimal-digit value extracted from the event parameter.
`CY_BLE_EVT_GAP_KEYPRESS_NOTIFICATION`	The BLE stack generates this event when a keypress (secure connections) is received from the peer device.
`CY_BLE_EVT_GAP_NUMERIC_COMPARISON_REQUEST`	This event indicates that the device must display a passkey during the secure connection pairing procedure. Call `Cy_BLE_GAP_AuthPassKeyReply()` with valid parameters on receiving this event.
`CY_BLE_EVT_GAP_AUTH_COMPLETE`	This event indicates that the authentication procedure is completed. Initiate application-specific action.
`CY_BLE_EVT_GAP_AUTH_FAILED`	This event indicates that the authentication process between two devices has failed. Typically, upon receiving this event, the BLE device disconnects the BLE link using `CY_BLE_GAP_Disconnect()`.

Table 7 list the API functions commonly used for establishing a secure BLE connection. For a comprehensive list of BLE events and API functions specific to LE secure connection, see Cypress BLE Middleware Library.

Table 7 API Functions for LE Secure Connection

API	Description
Cy_BLE_GAP_SetSecurityKeys()	This function sets the security keys that are to be exchanged with a peer device during the key exchange stage of the authentication procedure and sets it in the BLE stack.
Cy_BLE_GAP_GenerateKeys()	This function generates the security keys as per application requirement.
Cy_BLE_GAP_AuthReq()	This function starts the authentication/pairing procedure with the peer device.
Cy_BLE_GAPP_AuthReqReply()	The GAP Peripheral device uses this function to send the pairing response in the authentication/pairing procedure.
Cy_BLE_GAP_FixAuthPassKey()	This function sets or clears a fixed passkey to be used during the authenticated pairing procedure.

Device Bonding

The pairing procedure explained above is required to authenticate a peer device. If bonding is enabled, the device has the provision to store the link key of a previously authenticated link. If the bonding data is stored, and then if a connection is lost and re-established, bonded devices can resume the communication without the need of authentication again.

Table 8 lists the API functions to handle device bonding in the firmware. Figure 22 shows the flowchart for handling the pairing and bonding procedure in a secure BLE design.

Developing a BLE Application: Firmware Flow

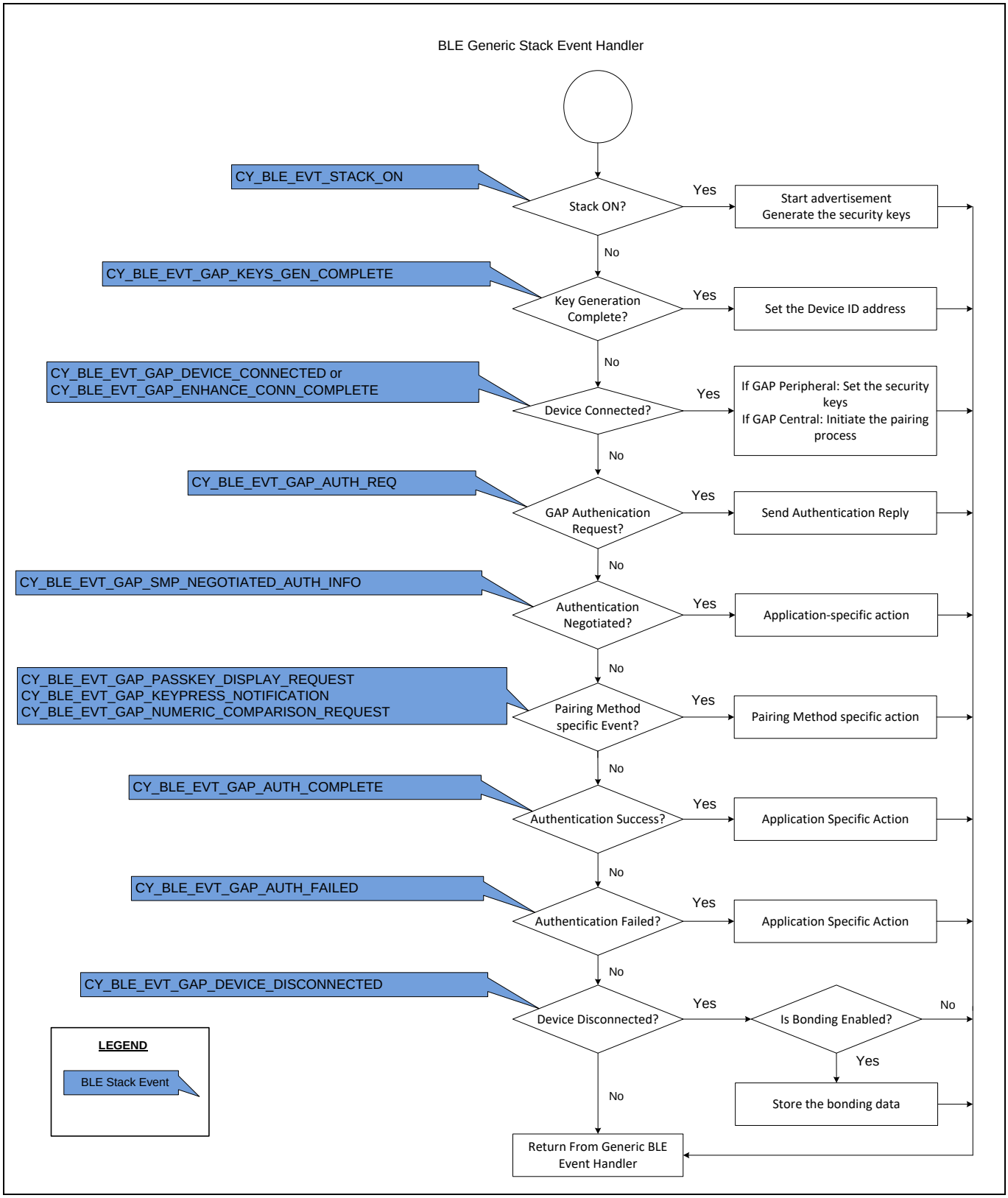


Figure 22 Firmware Flow for Secure BLE Design with Bonding

Table 8 API Functions for Bonding

API Function	Description
Cy_BLE_StoreBondingData()	This function writes the new bonding data from the RAM to the dedicated flash location as defined by the BLE Middleware.
Cy_BLE_GAP_GetBondList()	This function returns the count and list of bonded devices.
Cy_BLE_GAP_RemoveDeviceFromBondList()	This function removes the specified device from the bond list.

4.3 Additional BLE Design Considerations

Design considerations discussed in this section are optional steps that you can incorporate in your design based on your application requirements.

Link Layer Settings

The Link Layer (LL) is the part of the BLE protocol stack that handles advertising, scanning, creating, and maintaining connections. The LL of the BLE protocol stack supports BLE 4.2 features such as LE Data Packet Length Extension and Link Layer Privacy.

LE Data Packet Length Extension

The LE Data Packet Length Extension feature enables applications to get higher throughput, lower power consumption, etc. These benefits are available under the following conditions:

- Both BLE devices support LE Data Packet Length Extension.
- Higher-layer protocols use greater than the default (23 bytes) Maximum Transmission Unit (MTU) size.

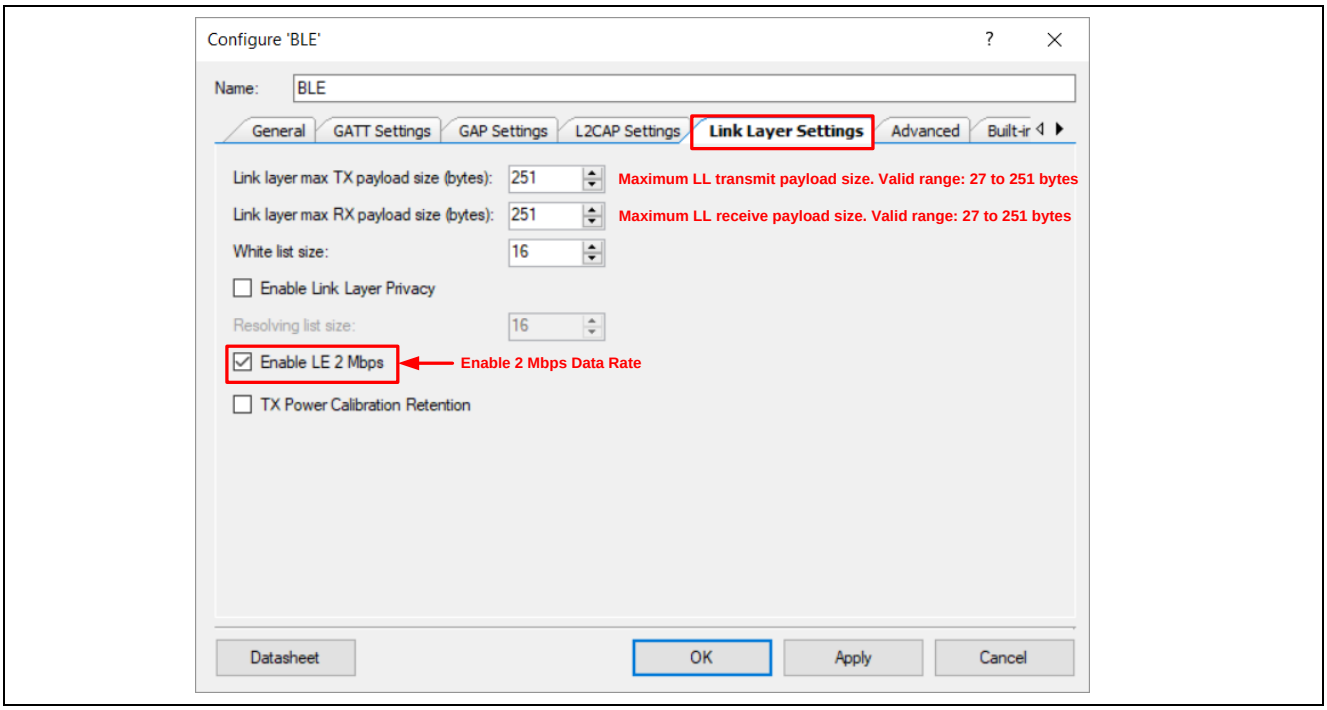


Figure 23 Configuring BLE Link Layer Settings

When a BLE connection is established, the BLE stack automatically negotiates TX and the RX payload sizes with the peer device. Negotiated maximum TX and RX payload sizes are reported to the application through the `CY_BLE_EVT_DATA_LENGTH_CHANGE` BLE stack event.

BLE Stack Event	Event Handler Action
CY_BLE_EVT_DATA_LENGTH_CHANGE	Informative event. This event reports the negotiated TX and RX payload sizes.

LE 2-Mbps Data Rate

The BLE protocol stack in PSoC 6 BLE supports the BLE 5.0-compliant data rate of 2 Mbps. To use the 2-Mbps data rate, you need to enable this feature under the Link Layer Settings tab of the BLE Component as shown in **Figure 23**. Note that the usage of the term PHY settings corresponds to the BLE data rate settings.

Table 9 lists the BLE stack events and the action to be taken while configuring PHY settings.

Table 9 BLE Events Associated with PHY Configuration

BLE Stack Event	Event Handler Action
CY_BLE_EVT_SET_PHY_COMPLETE	Indicates the completion of Cy_BLE_SetPhy().
CY_BLE_EVT_PHY_UPDATE_COMPLETE	Indicates that the Controller has changed the transmitter PHY or receiver PHY in use.
CY_BLE_EVT_GET_PHY_COMPLETE	Indicates the completion of Cy_BLE_GetPhy().

Table 10 lists the functions used for configuring the BLE data rate.

Table 10 API Functions for Handling PHY Settings

API	Description
Cy_BLE_SetPhy	Allows the application to set the PHY for the current connection.
Cy_BLE_GetPhy	Allows the application to read the current PHY setting for the specified connection.

See the **BLE Component datasheet** for more details on LL configuration parameters. For a comprehensive list of BLE events and API functions, refer to Cypress BLE Middleware Library.

See **AN99209 – PSoC 4 BLE and PRoC BLE: Bluetooth LE 4.2 Features** for an elaborate description on BLE 4.2 features. See **CE212742 – BLE 4.2 Data Length Security Privacy with PSoC 6 MCU with BLE Connectivity** for details on implementing LE secure connection and the data length extension (DLE) features of BLE.

5 BLE Design Examples

Now that you are familiar with basic firmware flow for BLE design using PSoC 6 BLE, it is time to get hands-on for designing BLE applications. This section discusses two BLE design examples illustrating the Multi-Master Multi-Slave (MMMS) capability of PSoC 6 BLE. Note that both the code examples discussed in this section are based on dual-CPU BLE architecture and the firmware is developed in an RTOS environment.

5.1 Multi-Master Multi-Slave: Implementing Four BLE Slaves

This design illustrates the connectivity between PSoC 6 BLE, acting as a GAP Peripheral and GATT Server, and four BLE-enabled devices (a personal computer running the CySmart BLE Host Emulation tool, or a mobile device running the CySmart mobile app) acting as a GAP Central and GATT Client. This design is available as a code example: **CE223508** – PSoC 6 MCU Implementing BLE Multi-connection (4 Slaves).

5.1.1 About the Design

In this design example, PSoC 6 MCU is configured as a GAP Peripheral and a GATT Server and can connect to as many as four GAP Central devices.

PSoC 6 BLE implements the BLE multi-slave functionality that consists of the following services as shown in **Figure 24**:

- Device Information Service
- Health Thermometer Service
- Custom service for RGB LED with color and intensity control
- 128-bit long Characteristic Read/Write custom service
- A custom notification service

Connected GAP Central devices can access the GATT database.

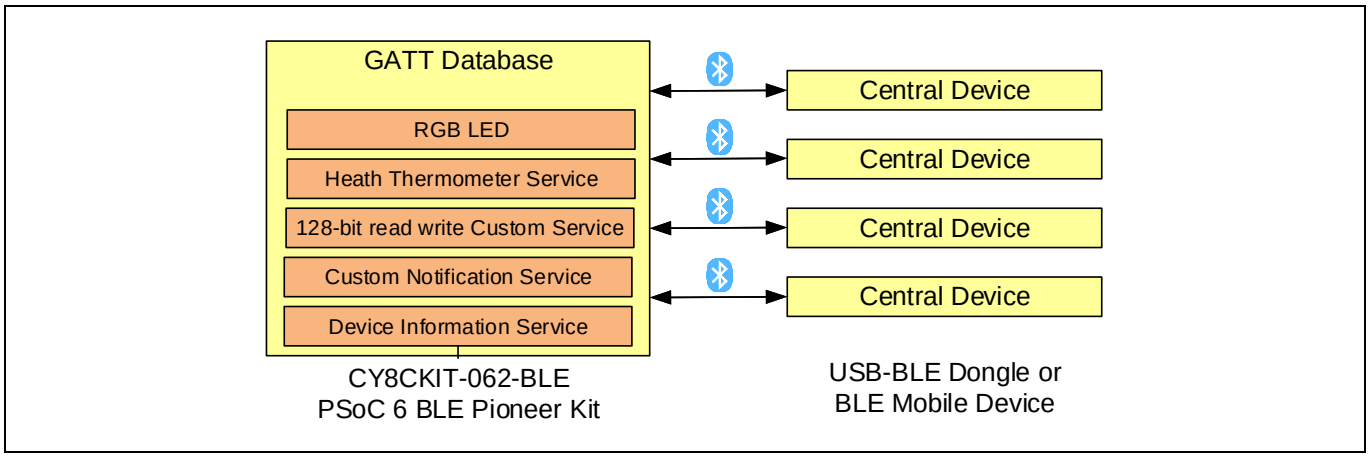


Figure 24 BLE Services Implemented by PSoC 6 BLE

The code example, **CE223508**, features the following:

- BLE connectivity:
  - Advertisement and connection with four GAP Central devices
  - Five services (RGB LED, Health Thermometer Service, Device Information, Read/Write 128-bit long custom service and custom notification service)
  - Data transfer over BLE using notifications, read, and write

- RGB LED color and intensity control using configurable digital blocks of PSoC 6 MCU
- The ADC in PSoC 6 MCU scans two differential channels and averages multiple samples without the need for CPU intervention for accurate temperature measurement from a thermistor circuit.
- Device Information Service gives manufacturer and/or vendor information about the device.
- 128-bit Read/Write custom service
- The custom notification sends notifications to connected devices about any changes to the GATT database. It sends a two-byte data: the first byte identifies the device that modified the data; the second byte identifies the Characteristic that has been modified.
- Low-power operation using the deep sleep mode with multi-counter watchdog timer (MCWDT) and GPIO interrupts
- The orange (LED8) and red (LED9) LEDs on the kit are used to show the status:
  - If the device is in hibernate mode, red LED will be ON.
  - The orange LED will blink if the device is advertising.

Figure 25 shows the firmware flow of **CE223508**.

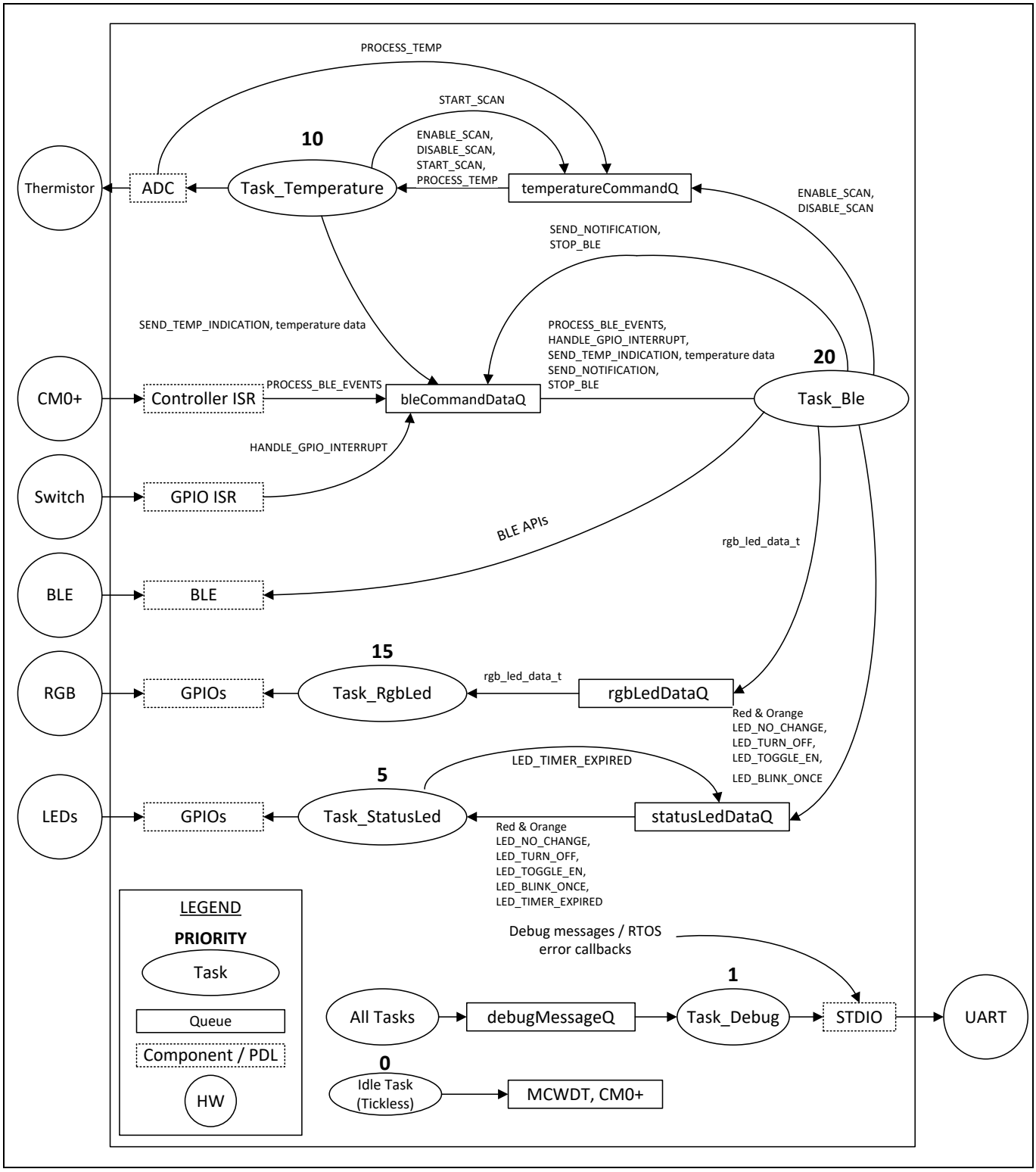


Figure 25 Firmware Flow Diagram for PSoC 6 MCU Implementing BLE Multi-Connection (4 Slaves)

5.2 Multi-Master Multi-Slave: Implementing Three BLE Masters and One BLE Slave

This design demonstrates how to configure PSoC 6 BLE in simultaneous Multiple Master and Single Slave mode of operation. This design configures PSoC 6 BLE as three BLE Central roles and one BLE Peripheral role (MMMS). The BLE Multi-Master Single Slave project is used in conjunction with the **CE215119** BLE Battery Level code example for PSoC 6 MCU or PSoC 4 devices to demonstrate the operation in simultaneous Multiple Master and Single Slave modes. For a detailed explanation on the design and implementation of the BLE design, see **CE224714 – PSoC 6 MCU Implementing BLE Multi-connection (3 Masters 1 Slave)**.

5.2.1 About the Design

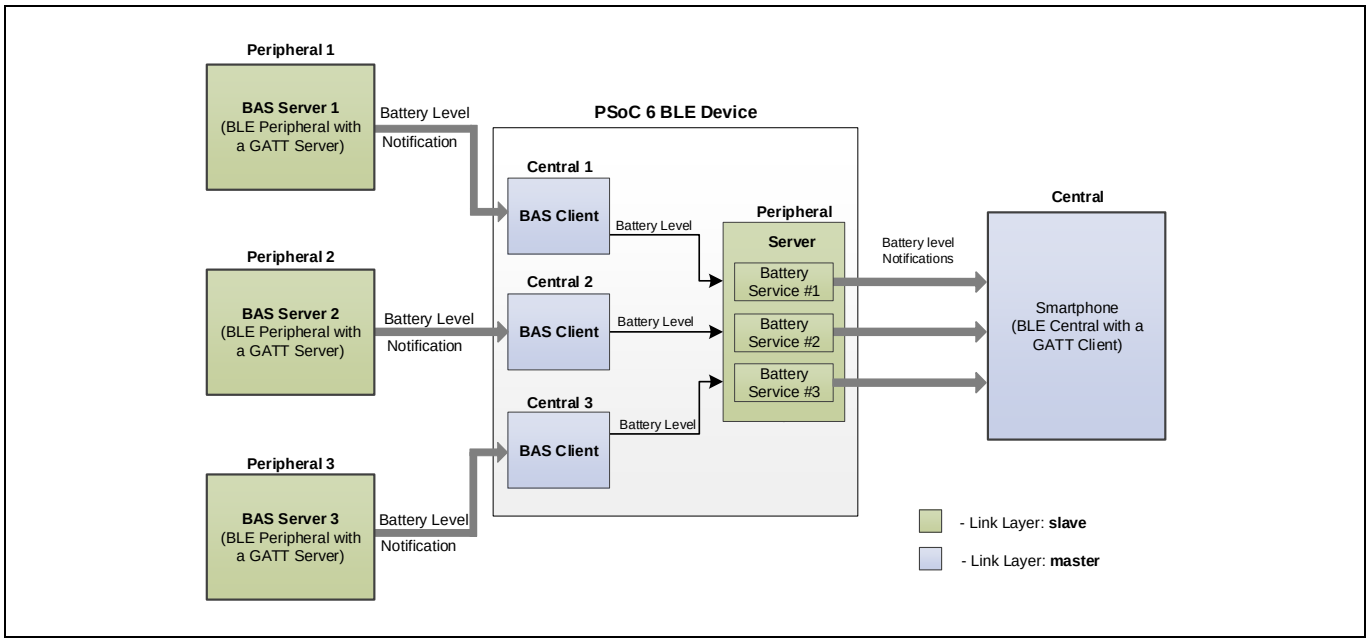


Figure 26 Multi-Master Single Slave

The Multi-Master Single Slave project uses three BLE Central connections and one Peripheral connection:

- The Central is configured as a Generic Attribute Profile (GATT) Client with a Battery Service that can communicate with a peer device in the Generic Access Profile (GAP) Peripheral and GATT Server roles. Use the existing *CE215119 – BLE Battery Level* code examples for PSoC 6 BLE/PSoC 4 devices or an application that can simulate a GATT Server with a Battery Service as a peer device.
- The Peripheral is configured as a GATT Server with three Battery services. This configuration represents the battery level of the three Peripherals that the device is connected to. **Figure 26** shows a block diagram of the Multi Master Single Slave example.

Figure 27 shows the firmware flow of **CE224714**.

Summary

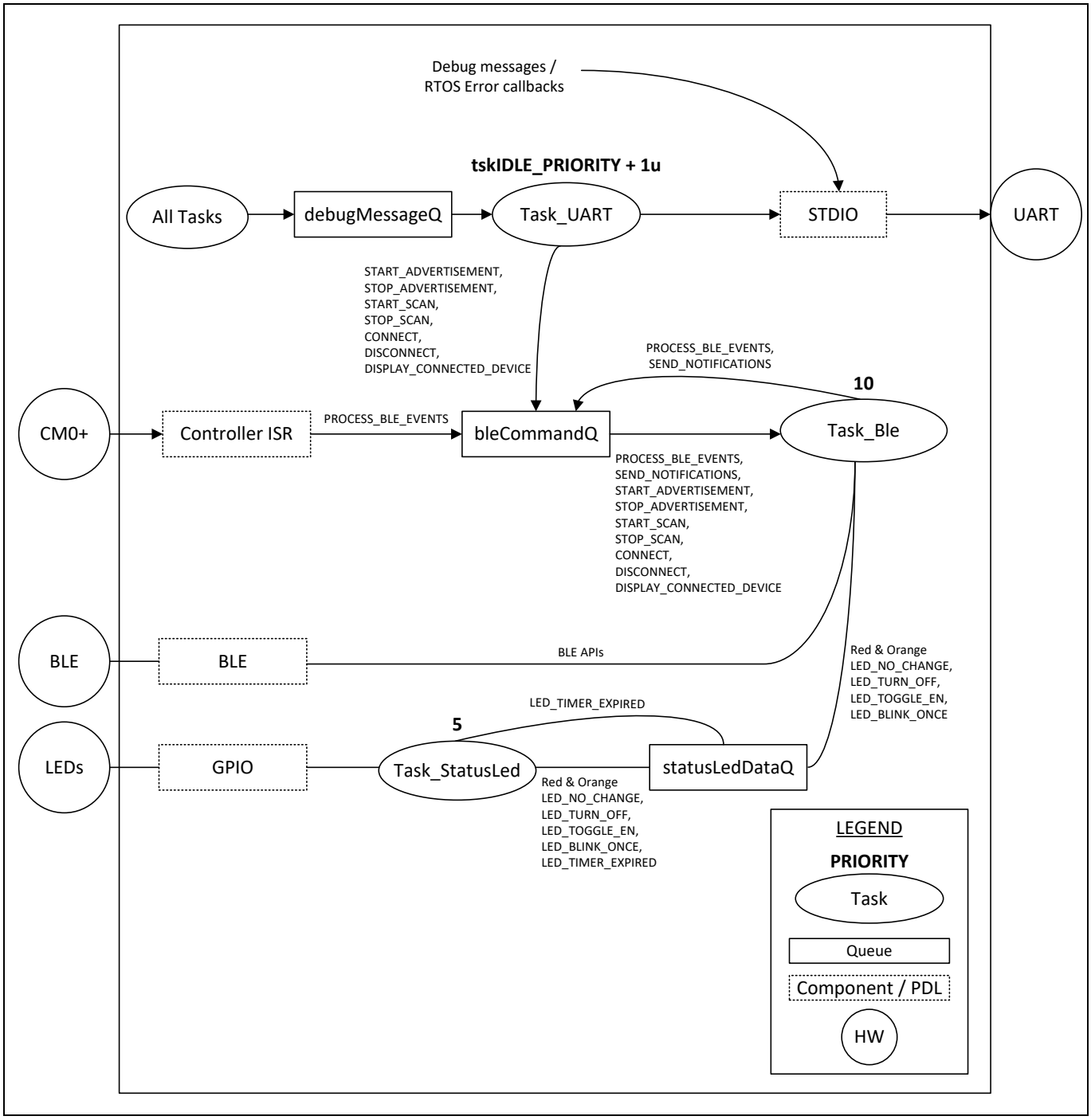


Figure 27 Firmware Flow Diagram for Multi-Master Single Slave

6 Summary

This application note introduced the BLE stack architecture and its implementation in PSoC 6 BLE. The application note further discussed the typical firmware flow and design considerations, such as low-power design and establishing BLE secure connection, for developing BLE applications. The code examples **CE223508** and **CE224714** discuss Multi-Master Multi-Slave (MMMS) feature of the BLE stack used with PSoC 6 BLE and illustrate the firmware flow for developing MMMS BLE applications.

7 Related Documents

Application Notes	
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC 6 MCU with BLE Connectivity devices and how to build your first PSoC Creator project
AN218241 – PSoC 6 MCU Hardware Design Considerations	Describes how to design a hardware system around a PSoC 6 MCU device.
AN215656 – PSoC 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC 6 MCU, and shows how to build a simple dual-CPU design
AN221111 – PSoC 6 MCU: Creating a Secure System	Describes how to create a secure embedded system with PSoC 6 MCU
AN213924 – PSoC 6 MCU Bootloader Software Development Kit (SDK) Guide	Provides comprehensive information on how to use the Bootloader Software Development Kit (SDK) to develop bootloadable systems for PSoC 6 MCU products
AN219434 – Importing PSoC Creator Code into an IDE for a PSoC 6 MCU Project	Describes how to import the code generated by PSoC Creator into your preferred IDE
AN99209 – PSoC 4 BLE and PRoC BLE: Bluetooth LE 4.2 Features	Provides an overview of the Bluetooth low energy (BLE) 4.2 features and explains their usage at the application level.
PSoC Creator Code Examples	
CE223508 – PSoC 6 MCU Implementing BLE Multi-connection (4 Slaves)	Demonstrates the implementation of multi-slave functionality of the PSoC 6 MCU with BLE Connectivity (PSoC 6 BLE) device.
CE224714 – PSoC 6 MCU Implementing BLE Multi-connection (3 Masters 1 Slave)	Demonstrates how to configure the PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity device in simultaneous Multiple Master and Single Slave modes of operation.
CE212742 – BLE 4.2 Data Length Security Privacy with PSoC 6 MCU with BLE Connectivity	Demonstrates the new BLE 4.2 and 5.0 features of the PSoC Creator BLE Component.
CE215119 – BLE Battery Level with PSoC 6 BLE	Demonstrates the operation of Bluetooth Low Energy (BLE) Battery Service (BAS) using the PSoC Creator BLE Component.
CE216767 – PSoC 6 MCU with Bluetooth Low energy (BLE) Connectivity Bootloader	Demonstrates simple over the air (OTA) bootloading with PSoC 6 MCU with BLE Connectivity
Device Documentation	
PSoC 6 MCU: PSoC 63 with BLE Datasheet	**PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual**
Development Kit Documentation	
CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit	

For more BLE code examples with PSoC 6 BLE, visit our GitHub repository: **PSoC 6 MCU BLE Connectivity Designs**

Revision history

Document version	Date of release	Description of changes
**	2018-09-21	New Application Note.
*A	2021-03-27	Migrated to Infineon template.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-03-27

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Go to www.cypress.com/support

Document reference

002-15671 Rev. *A

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.