AN91162

Creating a BLE Custom Profile

Author: Rohit Kumar

Associated Part Family: CY8C4XXX-BL, CYBL10XXX

Related Application Notes: For a complete list, click here.

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/AN91162.

AN91162 describes the methodology for developing a Bluetooth® Low Energy (BLE) application with PSoC 4 BLE or PRoC BLE devices using a custom BLE profile. It provides an overview of custom profiles and services and the procedure to build an application with PSoC 4 BLE using RGB LED control as an example. This application note also applies to the PRoC BLE part.

Content

1 Introduction

Bluetooth Low Energy (BLE) is an ultra-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 but with a lower bandwidth of 1 Mbps.

Cypress PSoC 4 BLE is a programmable embedded system-on-chip (SoC), integrating BLE along with programmable analog and digital peripheral functions, memory, and an ARM® Cortex®-M0 microcontroller on a single chip.

This application note demonstrates how to easily use the BLE Component GUI to create a custom BLE profile. You will define the structure of the custom profile. The tool will auto-generate APIs and event codes that are to be used. Similar steps can be followed to send or receive any type and length of data, as required by your custom profile. You will then test the custom profile on Cypress's CY8CKIT-042-BLE Pioneer Kit.

This application note assumes that you have a basic understanding of the BLE architecture and terms.

- If you are new to either BLE or PSoC, refer to the application note AN91267 - Getting Started with PSoC 4 BLE.
- For an understanding of the structure of the BLE Component in the PSoC Creator environment, and to learn how to develop applications based on standard BLE services, refer to the application note AN91184 - PSoC 4 BLE Designing BLE Applications.
- For complete details on the BLE specification, visit the BLE Developer Portal.

Install the latest BLE Pioneer Kit software from the kit webpage, which provides related tools for BLE application development and debugging. CY8CKIT-042-BLE or BLE Pioneer Kit is a BLE development kit from Cypress that supports both PSoC 4 BLE and PRoC BLE family of devices. This kit comprises pluggable PSoC 4 BLE (and PRoC BLE) modules that connect to a pioneer baseboard. This kit will be used for demonstrating the example project provided with this application note. The kit comprises a set of BLE example projects and documentation that help you get started with developing your own BLE applications.

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. For a comprehensive list of resources, see KBA86521, How to Design with PSoC 3, PSoC 4, and PSoC 5LP. Following is an abbreviated list for PSoC 4 BLE:

- **Overview:** PSoC Portfolio, PSoC Roadmap
- **Product Selectors:** PSoC 1, PSoC 3, PSoC 4, or PSoC 5LP. In addition, PSoC Creator includes a device selection tool.
- **Datasheets**: Describe and provide electrical specifications for the PSoC 41XX-BL and PSoC 42XX-BL device families.
- **Application Notes and Code Examples:** Cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples. PSoC Creator provides additional code examples; see Code Examples.

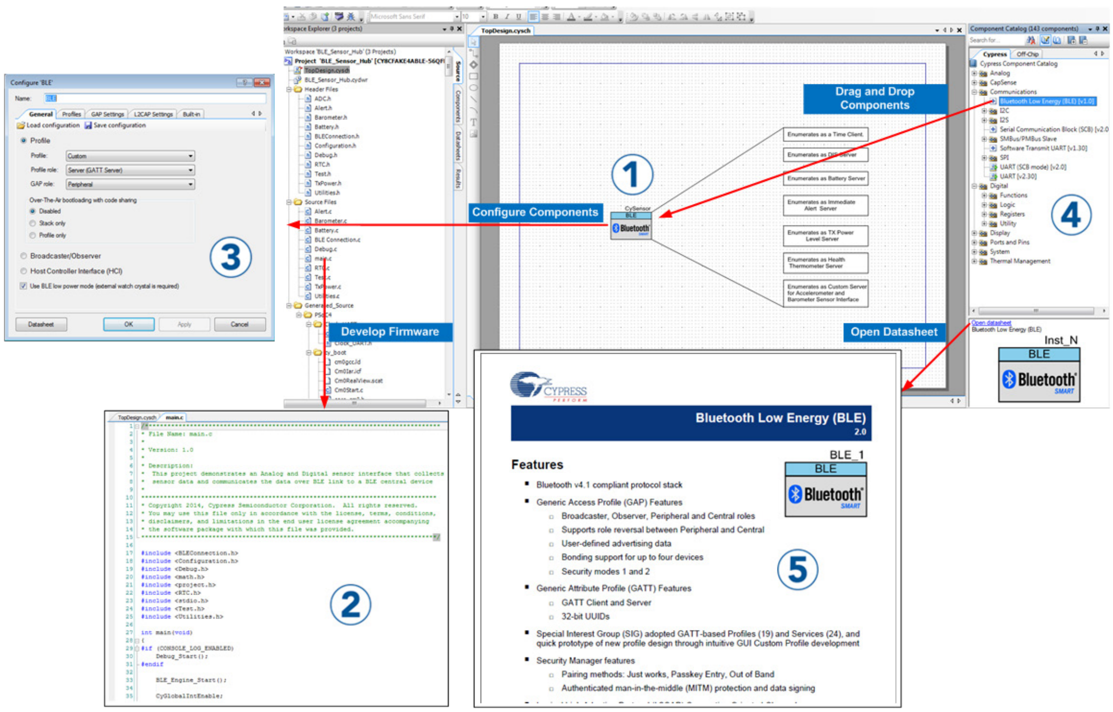
  Additionally, you can find code examples for PSoC devices and appropriate kits at PSoC 3/4/5 Code Examples webpage. For BLE, scroll to the table for CY8CKIT-042-BLE Pioneer Kit.
- **Technical Reference Manuals (TRMs):** Provide detailed descriptions of the architecture and registers in each PSoC 4 BLE device family.
- **CapSense Design Guide:** Learn how to design capacitive touch-sensing applications with the PSoC 4 BLE family of devices.
- **Development Tools**
  - CY8CKIT-042-BLE Bluetooth Low Energy (BLE) Pioneer Kit is an easy-to-use and inexpensive development platform for BLE. This kit includes connectors for Arduino™ compatible shields and Digilent® Pmod™ daughter cards.
  - CySmart BLE Host Emulation Tool for Windows, iOS, and Android is an easy-to-use GUI that enables you to test and debug your BLE Peripheral applications.
- **Technical Support**
  - Frequently Asked Questions (FAQs): Learn more about our BLE ecosystem
  - BLE Forum: See if your question is already answered by fellow developers on the PSoC 4 BLE and PRoC BLE forums.
  - Cypress support: Still no luck? Visit our support page and create a technical support case or contact a local sales representative. If you are in the United States, you can talk to our technical support team by calling our toll-free number: +1-800-541-4736. Select option 2 at the prompt.

2.1 PSoC Creator

PSoC Creator is a free Windows-based Integrated Design Environment (IDE). It enables you to design hardware and firmware systems concurrently, based on PSoC 4 BLE and PRoC BLE. As Figure 1 shows, with PSoC Creator, you can:

1. Drag and drop Components to build your hardware system design in the main design workspace.
2. Co-design your application firmware with the PSoC hardware.
3. Configure the Components using configuration tools.
4. Explore the library of more than 100 Components.
5. Review the Component datasheets.

Figure 1. PSoC Creator Schematic Entry and Components



2.2 PSoC Creator Help

Visit the PSoC Creator home page to download and install the latest version of PSoC Creator. Then launch PSoC Creator and navigate to the following items:

- **Quick Start Guide**: Choose **Help** > **Documentation** > **Quick Start Guide**. This guide gives you the basics for developing PSoC Creator projects.
- **Simple Component example projects**: Choose **File** > **Open** > **Example projects**. These example projects demonstrate how to configure and use PSoC Creator Components.
- **Starter designs**: Choose **File** > **New** > **Project** > **PSoC 4 Starter Designs**. These starter designs demonstrate the unique features of PSoC 4 BLE.
- **System Reference Guide**: Choose **Help** > **System Reference** > **System Reference Guide**. This guide lists and describes the system functions provided by PSoC Creator.
- **Component datasheets**: Right-click a Component and select “Open Datasheet.” Visit the PSoC 4 BLE Component Datasheets page for a list of all PSoC 4 BLE Component datasheets.
- **Document Manager**: PSoC Creator provides a document manager to help you to easily find and review document resources. To open the document manager, choose the menu item **Help** > **Document Manager**.

2.3 Code Examples

PSoC Creator includes a large number of code example projects. These projects are available from the PSoC Creator Start Page, as Figure 2 shows.

Example projects can speed up your design process by starting you off with a complete design, instead of a blank page. The example projects also show how PSoC Creator Components can be used for various applications. Code examples and datasheets are included, as Figure 3 shows.

In the **Find Example Project** dialog shown in Figure 3, you have several options:

- Filter for examples based on architecture or device family, that is, PSoC 4, PSoC 4 BLE, PRoC BLE, and so on; category; or keyword.
- Select from the menu of examples offered based on the **Filter Options**. There are more than 20 BLE example projects for you to get started, as shown in Figure 3.
- Review the datasheet for the selection (on the **Documentation** tab)
- Review the code example for the selection. You can copy and paste code from this window to your project, which can help speed up code development.
- Or create a new project (and a new workspace if needed) based on the selection. This can speed up your design process by starting you off with a complete basic design. You can then adapt that design to your application.

Figure 2. Code Examples in PSoC Creator

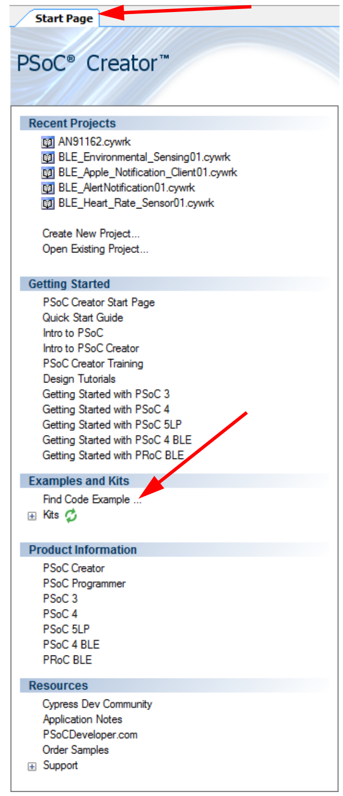
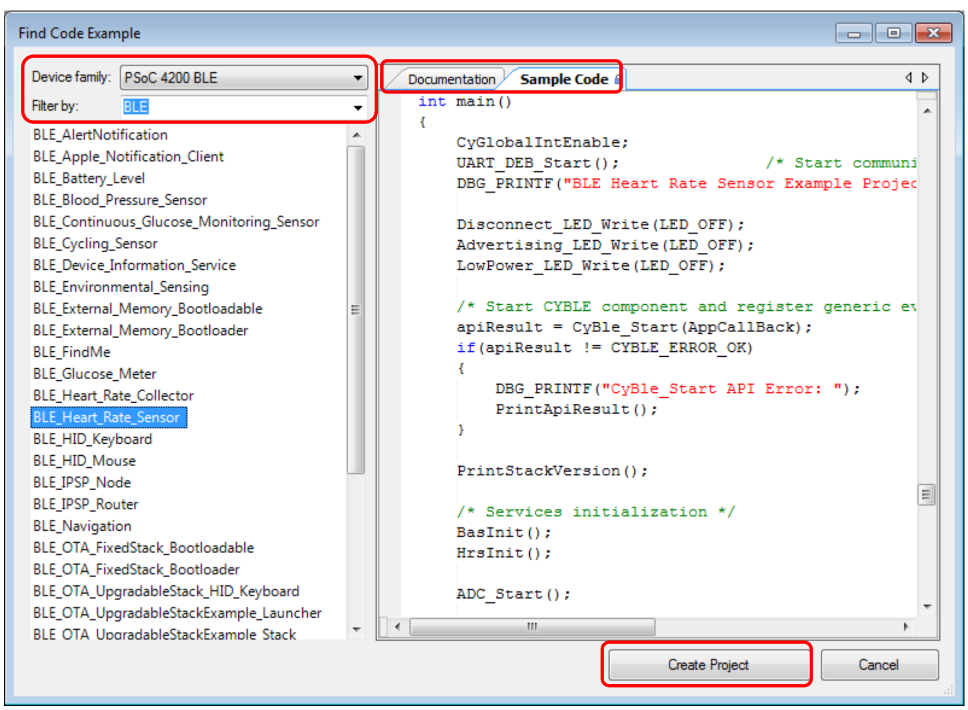


Figure 3. Code Example Projects with Sample Code



3 Standard Service versus Custom Service

A Service is a group of characteristics that defines a particular function. There are two types of services. The first is the Standard Service, which has been defined by the Bluetooth SIG for some common applications of BLE. Some examples are Heart Rate, Health Thermometer, Blood Pressure, and Alert Notifications. The complete list of standard services can be found in the Bluetooth Developer Portal. Refer to the application note AN91184 - PSoC 4 BLE Designing BLE Applications to learn how to design a standard application using PSoC 4 BLE.

The second type of service is the Custom Service. This type of service, as the name suggests, is defined for custom applications and not universally recognized. These services allow you to deploy BLE devices that can have custom applications beyond the limited set of services defined by the BLE SIG but still utilize the BLE framework. Custom services can be formulated by anyone developing a BLE application. The example project with this application note will demonstrate custom services that will allow you to transfer custom RGB LED data between the BLE Pioneer Kit and a BLE-capable mobile phone or PC.

4 Defining a Custom BLE Profile

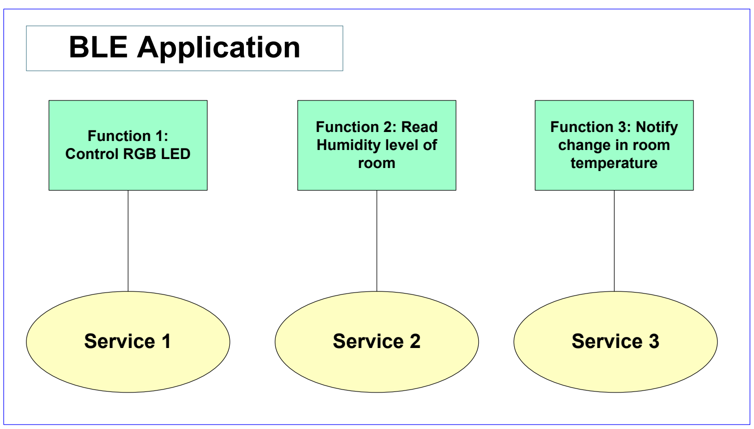
A custom BLE profile incorporates custom services and characteristics. It can also include standard services and characteristics.

4.1 Defining Services

The first thing to analyze while creating a custom BLE application is the set of functions that the application requires. Each of these functions is represented by a custom service, which can then be used to obtain any data required.

For example, one function can be controlling the red, green, and blue color intensity of an RGB LED. This function can be represented by a custom service, named “RGB LED Control”. Other functions could read the room humidity level or room temperature. Figure 4 shows one such instance of an application, which defines custom services to implement three functions.

Figure 4. Define Custom Services



Functions that differ only in the type of values they provide can be grouped under one service. In the RGB LED Control example, you do not need to create four different custom services for controlling the four RGB LED color values (red, green, blue, and intensity). As the function is to control the RGB LED values, one service will suffice. After the services have been defined, allocate universally unique IDs (UUIDs) to each of these services that will uniquely identify them. These UUIDs should be 128-bit values for custom services.

4.2 Defining Characteristics

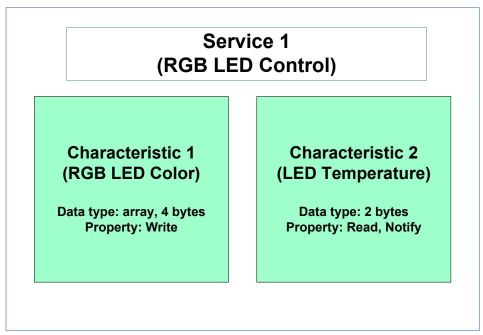
Next, you need to define characteristics for each service. This definition contains the following:

- Data Value: The data value describes the type and the length of the data transferred. Supported data types include unsigned byte, signed byte, word, character string, and array.
- Property: The property describes how the data value is accessed. Available choices are Broadcast, Read, Write, WriteWithoutResponse, Notify, Indicate, SignedWrite, and WritableAuxiliaries.
- Permissions: Permissions describe the access permissions for the data. Permission settings are provided for Encryption, Authentication, and Authorization.
- UUID: The UUID value (128-bit) uniquely identifies the characteristic.

In the RGB LED Control example, the defined characteristic sends an array of four bytes, one byte defining each of the color values of the RGB LED, and one byte to control intensity. The definition of the characteristic depends on how the application interprets the data. The property of this characteristic is “Write” because the GATT client writes the new RGB LED values to the GATT server.

Similarly, you can add another characteristic that will provide the 2-byte temperature information from an onboard heat sensor that monitors LED overheating. Figure 5 provides an overview of the characteristics described above.

Figure 5. Define Characteristic in Service

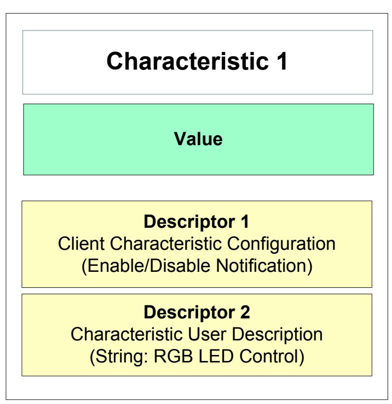


4.3 Defining Descriptors

Depending on the characteristics, you may add descriptors. These descriptors provide information to the user about characteristics. They can also be used by the GATT client device to enable or disable notifications and indications.

An example of descriptors under a custom characteristic is shown in Figure 6. In this example, a descriptor, termed Client Characteristic Configuration, is used by the GATT client to enable and disable notifications or indications. This is under the characteristic that supports notification or indication. Another example descriptor is the Characteristic User Description, which provides a string through which the characteristic can be recognized in a human-readable format.

Figure 6. Define Descriptor in Characteristics



4.4 Generate Custom UUIDs

All the BLE custom services and characteristic must use 128-bit UUIDs for identification and ensure that the base UUID is different from the BLE defined base UUID (00000000-0000-1000-8000-00805F9B34FB). Base UUID is a 128-bit value on which the standard UUIDs (16-bit and 32-bit) are defined.

The BLE specification does not define a method to generate custom UUIDs for BLE services and characteristics. It is up to the user to generate their own 128-bit UUID, which is different from the BLE defined base UUID. There can be various ways one can generate a UUID for custom services and characteristics.

Cypress uses the following mechanism to generate UUIDs for custom services and characteristic. Similar methodology can be used by you to create your own UUID.

Custom UUID value: XXXXYYYY-0000-1000-8000-00805F9B0131

Table 1. Cypress's method to generate custom 128-bit UUID from BLE defined based UUID

UUID Part	Description
XXXX	16-bit value identifying the device/product
YYYY	16-bit UUID for the specific service or characteristic
00805F9B**0131**	Base UUID for all Cypress's custom services and characteristics. This is the last 6 bytes of the BLE SIG defined base UUID with last 2 bytes replaced by Cypress's Bluetooth assigned company identifier (0x0131).

The XXXX for PSoC 4 BLE has been set to 0x0003.

For example, we use the RGB LED custom service in this project. Its 128-bit custom UUID is set as **0003CBBB-0000-1000-8000-00805F9B0131**. Here, the device identifier is set to 0x0003, the last 6 bytes of the base UUID is set to 0x00805F9B0131 and the RGB LED service specific 16-bit UUID is set to 0xCBBB.

Alternately, you can refer to the webpage http://www.itu.int/en/ITU-T/asn1/Pages/UUID/uuids.aspx to generate a unique 128-bit UUID.

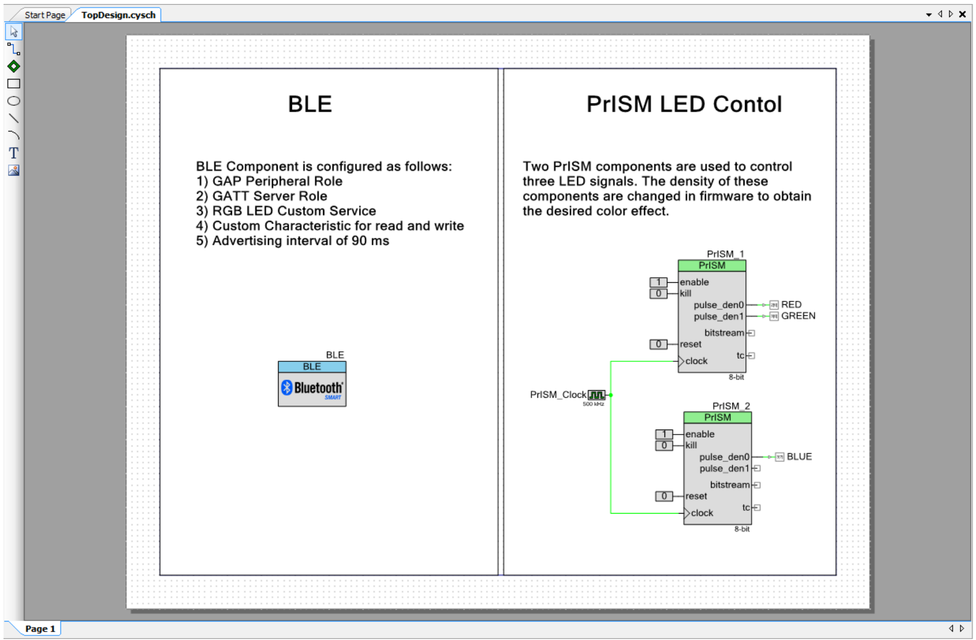
5 PSoC Creator Project: RGB LED Custom Profile

To create and verify this project, ensure that you have the following prerequisites:

- PSoC Creator 3.3 SP1 (or later) along with PSoC Programmer 3.24 (or later)
- CySmart™ Central Emulation Tool
- CySmart iOS App or CySmart Android App
- CY8CKIT-042-BLE Pioneer Kit

This project will use the following PSoC Creator Components: BLE, PrISM™, Clock, and Digital Output Pins. The project schematic in PSoC Creator looks as shown in Figure 7.

Figure 7. PSoC Creator Project Schematic



Do the following to implement the project:

1. Create a PSoC Creator project.
2. Configure Components in PSoC Creator.
3. Write the firmware to handle BLE events and other Components.
4. Build the project and program the BLE Pioneer Kit.
5. Test the project using the CySmart tool or app.

This example project contains a custom service for RGB LED control that will be used to control the color and brightness of an onboard RGB LED on the BLE Pioneer Kit.

For RGB LED control, you will define the data format as a 4-byte array of type uint8, as shown in Figure 8. Both Write and Read properties will be supported.

Figure 8. RGB LED Data Format

RED (0-255)	**GREEN 0-255**	**BLUE 0-255**	**INTENSITY 0-255**

5.1 Create a PSoC Creator Project

1. Open PSoC Creator from **Start** > **All Programs** > **Cypress** > **PSoC Creator 3.3** > **PSoC Creator 3.3**.
2. Create a new project (**File** > **New** > **Project**). Select the PSoC 4100 BLE / PSoC 4200 BLE Design template, and then select CY8C4247LQI-BL483 as the device. Name the project AN91162 and save the workspace in the desired location.

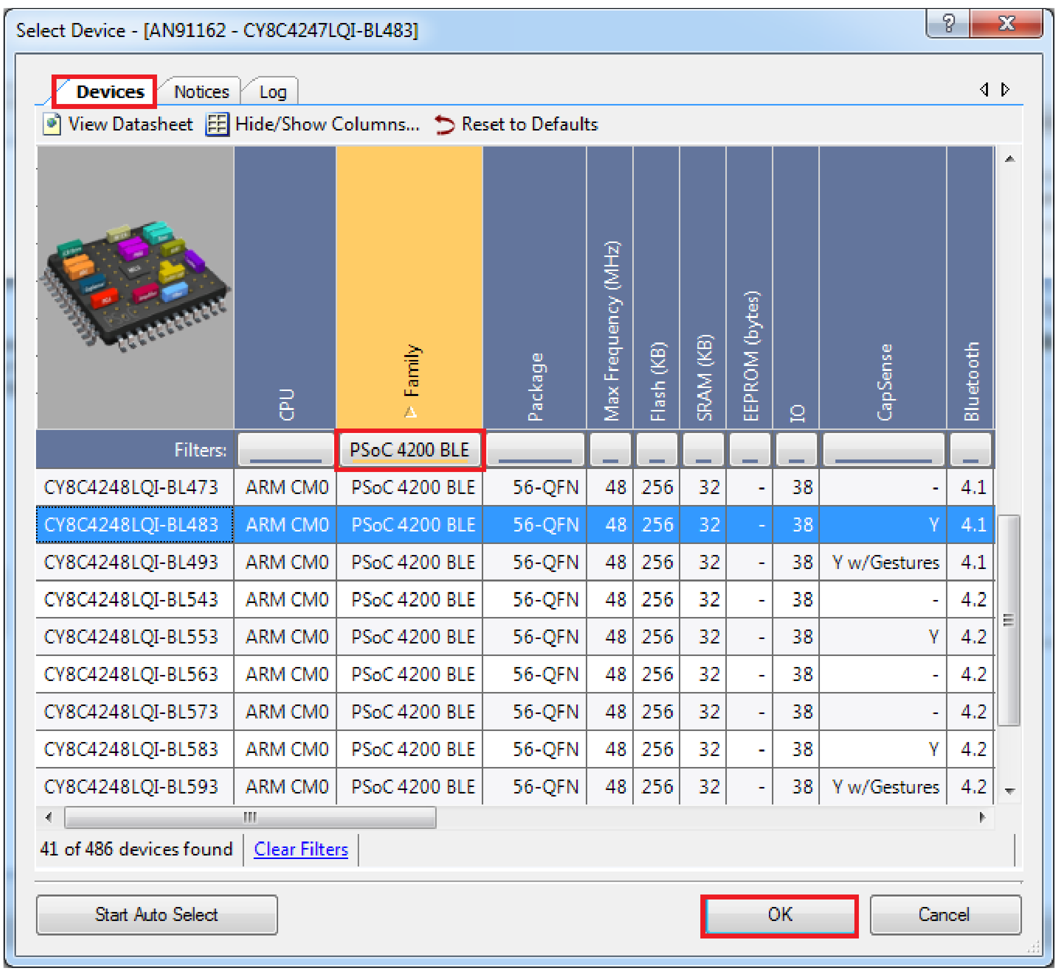
   Note CY8C4XX**7**-BL parts have 128K FLASH and 16K SRAM. Select CY8C4XX**8**-BL part if using PSoC 4 BLE device with 256K FLASH and 32K SRAM. These devices are supported on BLE component 2.3.

   CY8C4248LQI-**BL583** is a 256K FLASH and 32K SRAM device that supports BLE 4.2. Change the project to this part number from Device Selector settings if you have the BLE 4.2 capable device and update the BLE component to version 3.0 or greater.

   To change the part number between various PSoC 4 BLE devices, perform these steps:

   - Right-click on the project name in **Workspace Explorer**.
   - Select **Device Selector…**.
   - Set the **Family** to *PSoC 4200 BLE*.
   - Select your device number from the list and click **OK** (see Figure 9).

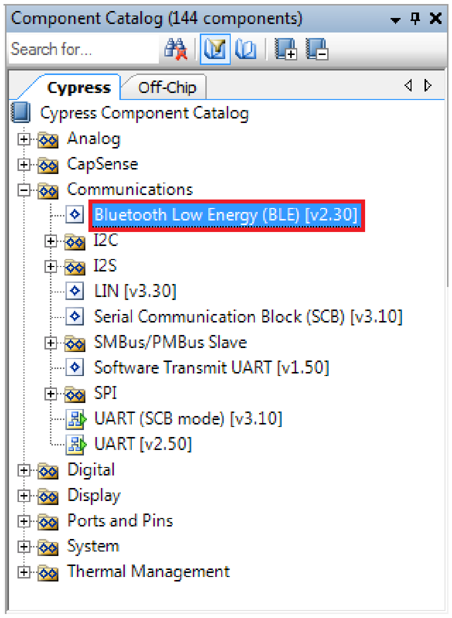
Figure 9. Device Selector settings



5.2 Configure Components

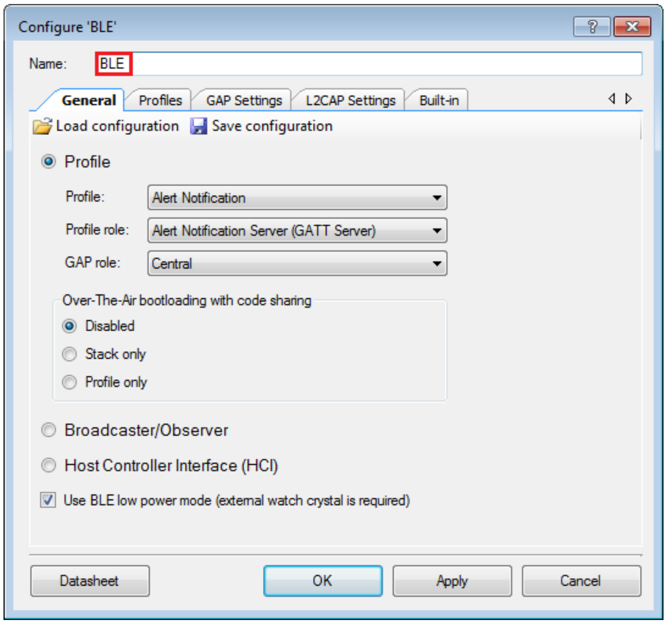
1. Drag and drop a BLE Component from the Component Catalog (on the right-hand side of the PSoC Creator IDE) onto Top Design, as shown in Figure 10.

Figure 10. BLE Component in Component Catalog



2. Double-click the Component to open its configuration window. Change the instance name of the Component to **BLE**, as shown in Figure 11.

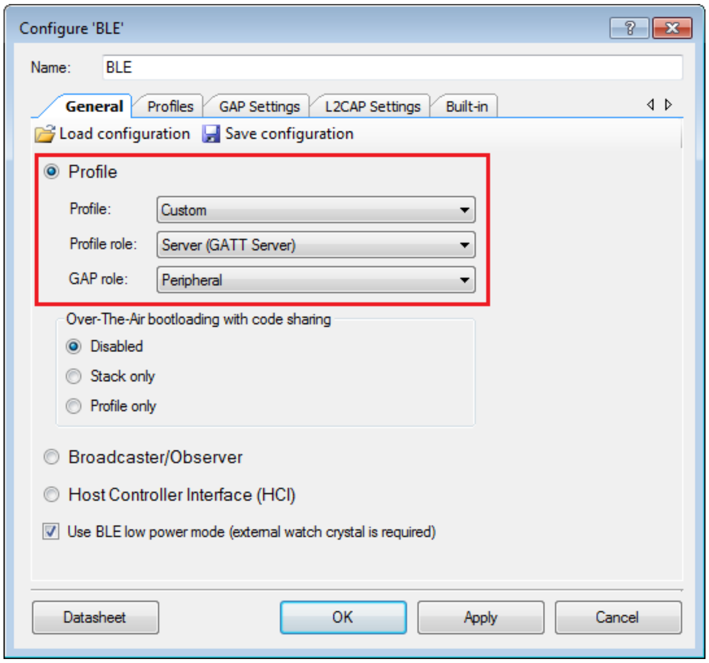
Figure 11. Instance Name of the BLE Component



Note Unlike other PSoC Creator Components, the instance name set in the BLE Component does not change the API naming convention. As BLE libraries are closed, the BLE Component APIs always start with "*CyBle_*" and not the instance name. The instance name only changes the name of the generated files.

3. On the **General** tab, select **Profile** and set **Custom** as the **Profile** option as shown in Figure 12. The other two options, Profile role and GAP role, are automatically set to **GATT Server** and **Peripheral**, respectively.

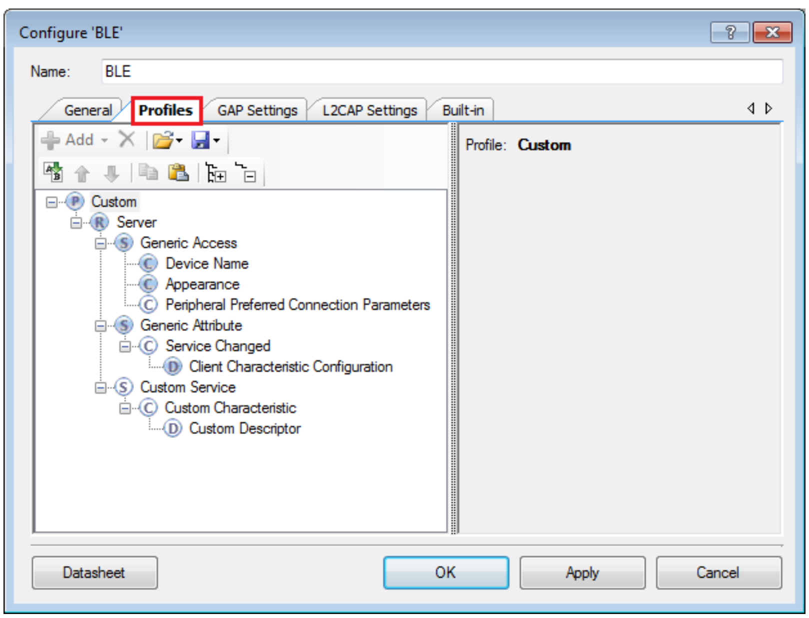
Figure 12. Set Profile Role as Custom



4. On the **Profiles** tab, configure profile-specific parameters. The Component exposes services, characteristics, and descriptors in the form of a profile tree, as shown in Figure 13.

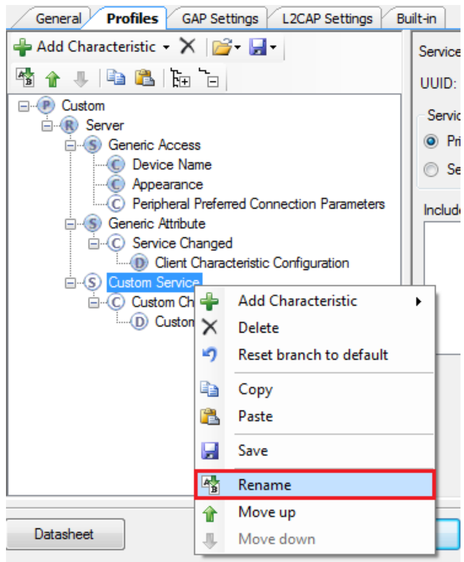
   No changes are required in the **Generic Access** and **Generic Attribute** services.

Figure 13. Default Custom Profile Tree in BLE Component



5. Right-click **Custom Service** and select **Rename**. Rename the service as **RGB LED** as shown in Figure 14.

Figure 14. Rename Custom Service

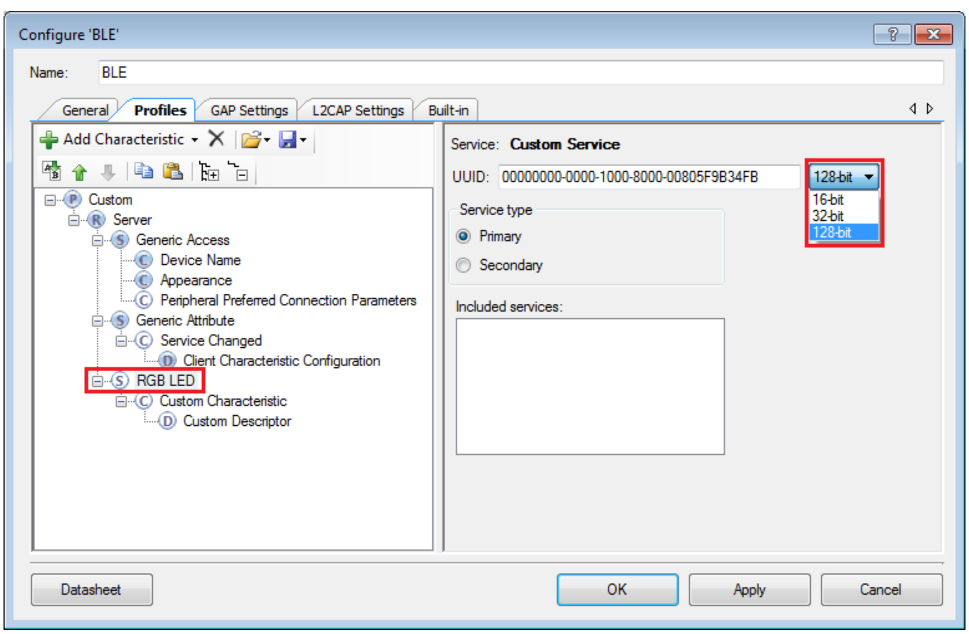


6. Click the **RGB LED** service on the service tree and set the UUID format for your custom service as **128-bit**, as shown in Figure 15. This UUID is used by the GATT client device to recognize the attribute present within the GATT server device.

   Note The default 128-bit UUID value seen in the Component (00000000-0000-1000-8000-00805F9B34FB) is the base UUID defined by the BLE SIG that is used to calculate the complete 128-bit UUIDs from 16-bit and 32-bit UUIDs.

   The BLE SIG recommends using a custom 128-bit UUID, different from the base UUID, for custom attributes to ensure that it does not conflict with existing UUIDs for standard services. Refer to Generate Custom UUIDs for methods to generate custom UUIDs.

Figure 15. Select 128-bit UUID Format

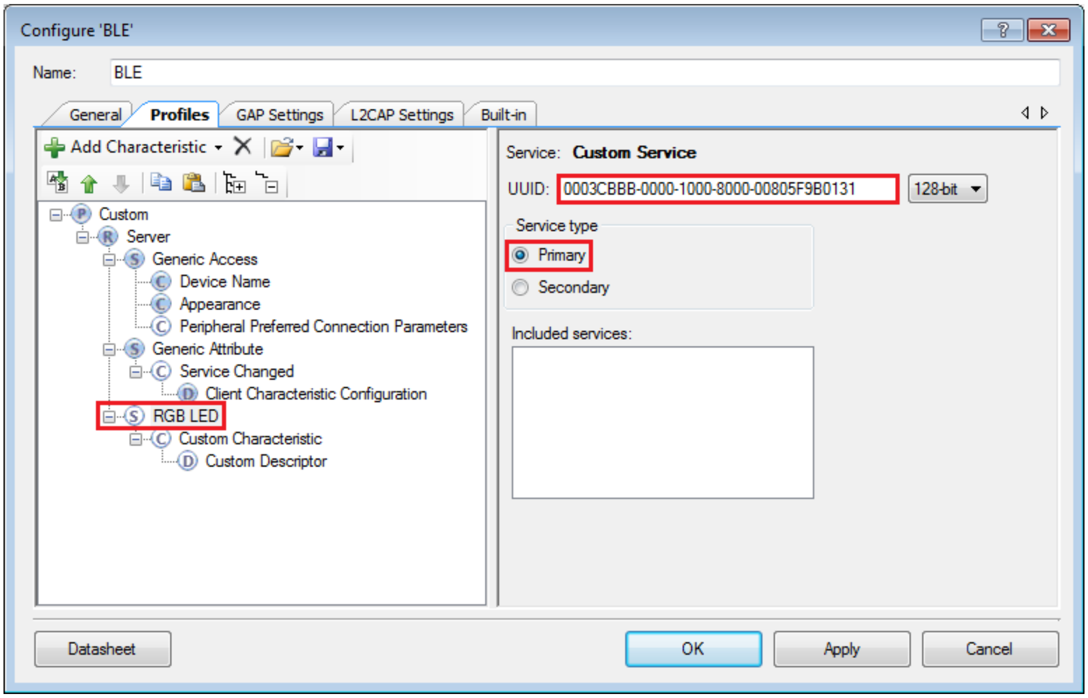


7. Modify the 128-bit UUID value to the hexadecimal value **0003CBBB-0000-1000-8000-00805F9B0131**. Select the service type as **Primary**, as shown in Figure 16.

   Note You must specify the UUID as **0003CBBB-0000-1000-8000-00805F9B0131**. Cypress defines this as the UUID for the RGB LED service. The CySmart app uses this UUID to display the RGB LED GUI page.

   For any other customer service/characteristic, you need to generate your own 128-bit UUID and add it in box.

Figure 16. Set UUID for RGB LED Service

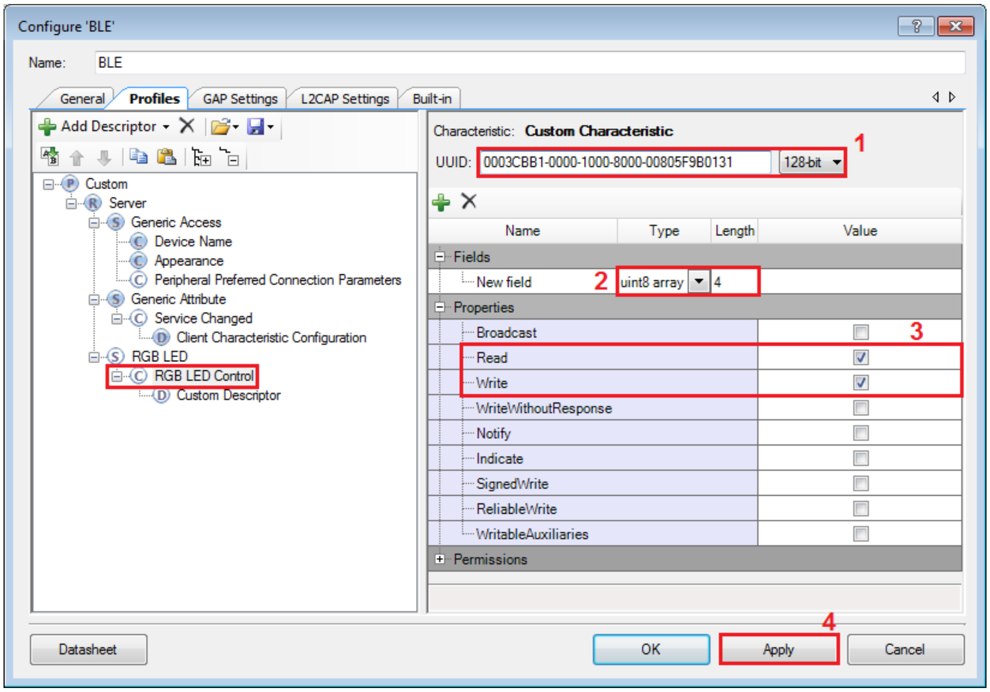


8. Right-click **Custom Characteristic** under the **RGB LED** service, rename it to **RGB LED Control**, and then edit its parameters per Table 1. These changes are shown in Figure 17.

Table 1. RGB LED Characteristic Parameters

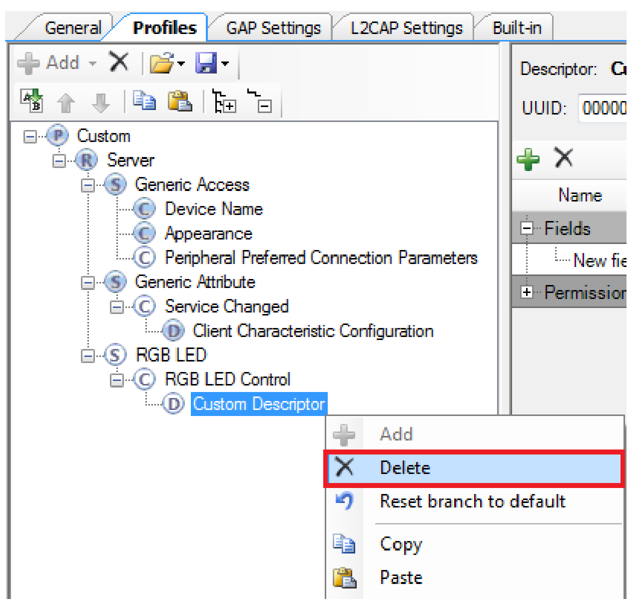
Parameter	Value	Description
UUID	0003CBB1-0000-1000-8000-00805F9B0131	Specifies the 128-bit UUID for the RGB LED characteristic. Use this value as the UUID to allow mobile applications to present the correct GUI page for RGB LED.
Fields	Type: uint8 array Length: 4	Specifies the type of data that will be transferred.
Properties (checkbox)	Read, Write	Specifies that the GATT client device can both read from and write to this characteristic.

Figure 17. RGB LED Characteristic Values



9. This characteristic does not require a custom descriptor, so right-click **Custom Descriptor** and select **Delete** as shown in Figure 18. Custom descriptors may be added to a characteristic if you want to append custom information to it.

Figure 18. Delete Custom Descriptor



10. Select the **RGB LED Control** characteristic, and then click the **Add Descriptor** option in the toolbar. From the pull-down list, select **Characteristic User Description** as shown in Figure 19.

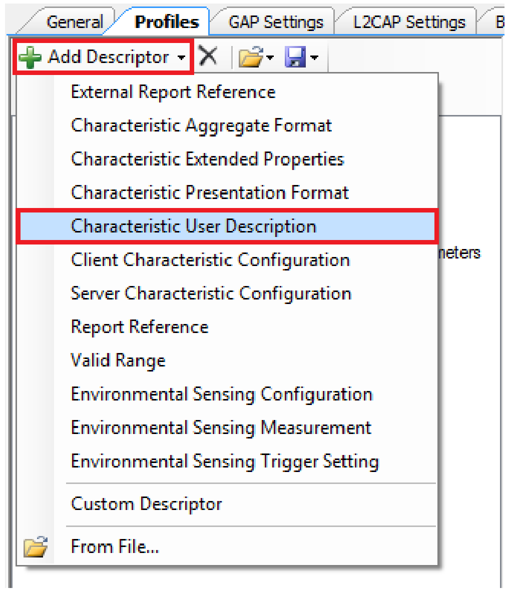


Figure 19. Add Characteristic User Description to RGB LED Characteristic

11. Click the **Characteristic User Description** descriptor. On the right, under **Fields**, click the **Value** field and type in the name as **RGB LED Control** as the user description of this characteristic, and set **Permissions** to 'Read', as shown in Figure 20. This will allow the client to read the name of the RGB LED Control characteristic.

    Note The Characteristic User Description descriptor is a BLE SIG-defined standard descriptor. Its 16-bit UUID has the value 0x2901 per the BLE specification. The BLE Component adds this descriptor with the correct UUID value; no change is required in it.

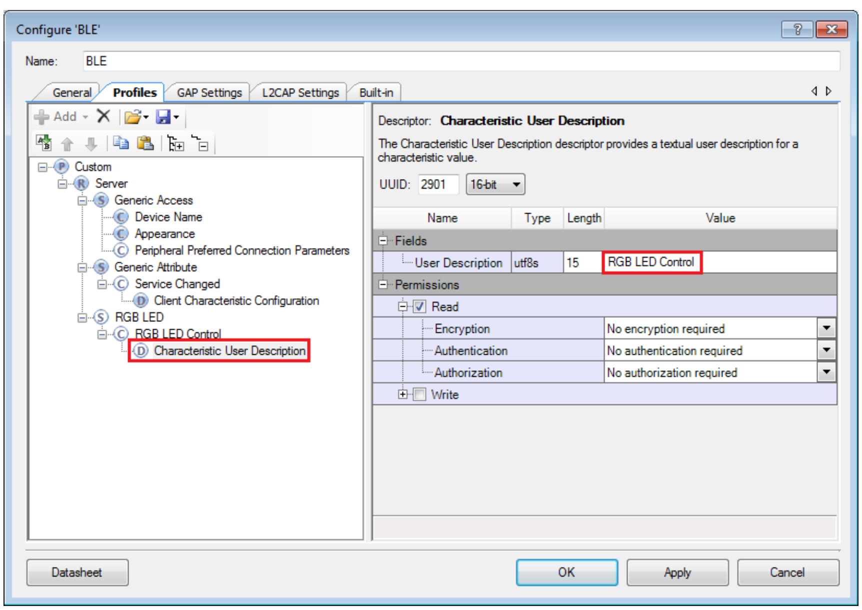


Figure 20. Characteristic User Description for RGB LED Characteristic

5.3 Configure the BLE Peripheral

1. On the **GAP Settings** tab in the BLE Component configuration window, configure the parameters under **General** settings per Table 2, and then click **Apply**.

Table 2. General GAP Settings for the Peripheral Device

Parameter	Value	Description
Public Address	00A050-XXXXXX	Specifies the 6-byte Bluetooth device address. This address is used during advertising. The last three bytes of the address are silicon-generated if the Silicon generated option is selected. This must be a non-zero value per the BLE SIG. Note that 00A050 is the Company ID of Cypress Semiconductor.
Silicon generated “Company assigned” part of device address	Checked	Allows the company-assigned part of the public address to be generated from the silicon. With this setting, each device will have a unique public address.
Device Name	CY Custom BLE	Specifies the name of the device that you want the GATT client to see while scanning.
Appearance	Unknown	Specifies the appearance of the device. For this custom service, leave it as Unknown.
MTU size (bytes)	23 (default)	Specifies the size of the protocol data unit (PDU) that can be transferred on the attribute level.
Adv/San TX Power level (dBm)	0 (default)	Specifies the radio TX power level while advertising data.
Connection TX power level (dBm)	0 (default)	Specifies the radio TX power level while sending data during connection.

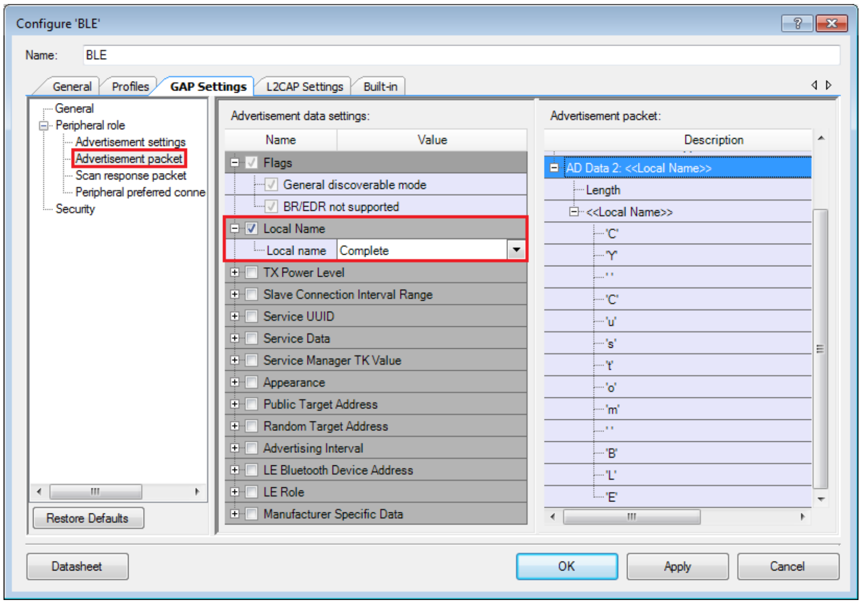
2. Click **Advertisement settings** under **Peripheral role** and configure the parameters per Table 3, and then click **Apply**.

Table 3. Advertisement Settings Configuration

Parameter	Value	Description
Discovery Mode	General	Sets the device to advertise in general mode so that it can be found by any Central device.
Advertising type	Connectable undirected advertising	Sets the Peripheral to advertise without any preference for a Central device and to receive connection requests from any Central device scanning it.
Filter policy	Scan request: Any Connect request: Any	Sets the Peripheral to choose whether to receive scan and connect requests from a particular device or any Central device. In this project, it is set to receive requests from any Central device for both scan requests and connection requests.
Advertising channel Map	All channels	Sets the Peripheral to advertise in all three advertising channels (37, 38, and 39).
Fast advertising interval: Minimum (ms)	80 ms	Specifies the minimum interval for advertising data. Actual advertising interval is calculated using both minimum and maximum intervals.
Fast advertising interval: Maximum (ms)	100 ms	Specifies the maximum interval for advertising data. Actual advertising interval is calculated using both minimum and maximum intervals.
Fast advertising interval: Timeout (s)	Uncheck	Specifies the time for which the Peripheral device will continue advertising before timing out and ceasing to advertise further. If unchecked, the advertisement happens continuously without any timeout.
Slow advertising interval	Uncheck	Disables slow advertising. If this setting is enabled, the Peripheral device will go into the slow advertising mode after a fast advertising timeout. In the slow advertising mode, the interval between advertisements is longer but saves power during advertisement. This project does not use this feature.

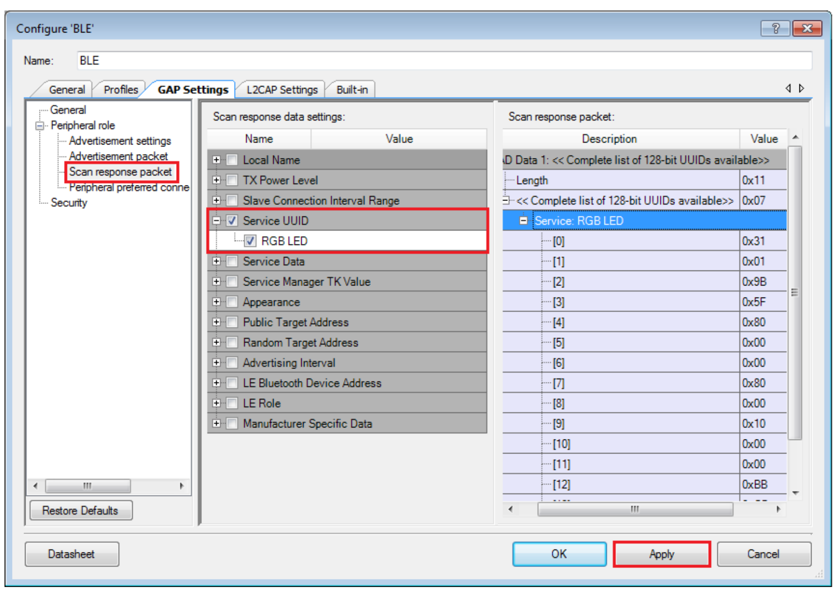
3. Click **Advertisement packet** under **Peripheral role** settings to specify the information in advertisement packets that Central devices receives. For this project, select the complete **Local name** to be sent as part of advertisement packet, as shown in Figure 21.

Figure 21. Advertisement Packet Settings



4. Click **Scan response packet** under **Peripheral role** to specify the data that the Peripheral should send in response to requests from Central devices during scanning. For this project, select the **Service UUID** > **RGB LED** service data as shown in Figure 22, and click **Apply**.

Figure 22. Scan Response Packet Settings for Peripheral



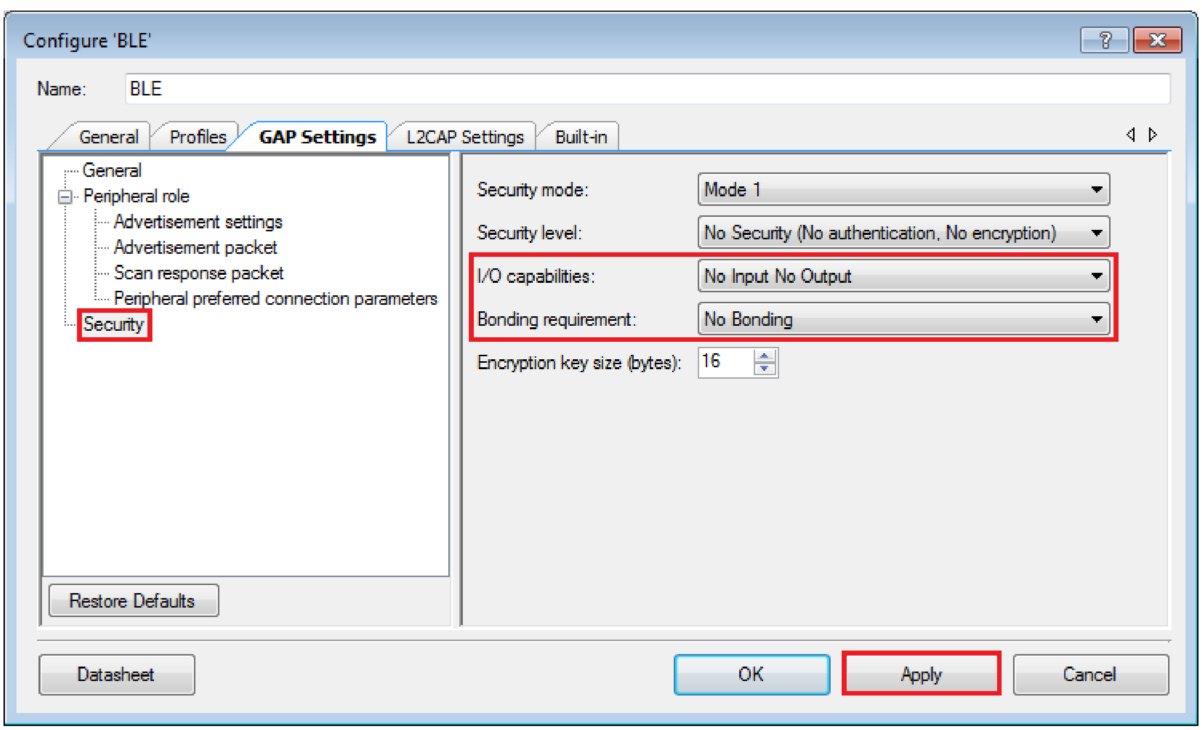
5. Click Peripheral preferred connection parameters and set the parameters as shown in Table 4.

Table 4. Peripheral Preferred Connection Parameters

Parameter	Value	Description
Connection interval: Minimum (ms)	75 ms	Sets the minimum interval in which the Peripheral and Central device will go into the transmission mode to communicate data after the Peripheral device is connected. A lower minimum interval means faster data rate but more power consumption.
Connection interval: Maximum (ms)	80 ms	Specifies the maximum connection interval that the Peripheral device supports. Central and Peripheral devices must agree upon the connection interval to have a successful connection. The actual connection interval is negotiated with the Central device during connection.
Slave latency	0	Sets the maximum number of times the Peripheral device can choose not to answer when the Central device requests data. It is useful for devices that want to send data at a faster rate but also want to remain in the low-power mode when no data is present to be sent. For this project, no slave latency is required.
Connection supervision timeout (ms)	2000 (2 seconds)	Sets the total time after the last successful connection event for which the Peripheral or Central device will consider the connection alive. If no connection event happens during this time, then the link is assumed broken, so devices disconnect.

6. Click **Security** under **Peripheral role** to configure the security level of the BLE communication. This project does not require security settings, so set the I/O capabilities to **No Input No Output** and Bonding requirement to **No Bonding**. Retain rest of the settings at their default values and click **Apply**, as shown in Figure 23.

Figure 23. Security Settings



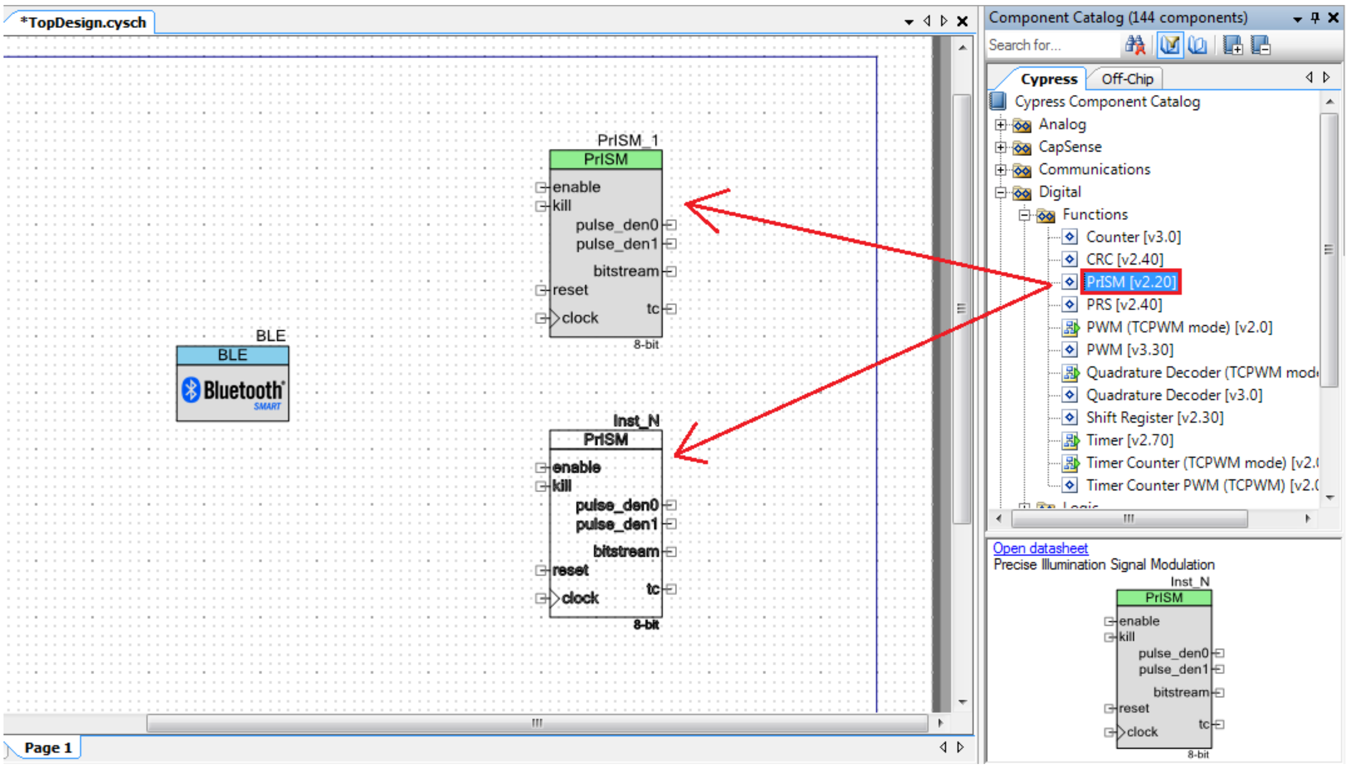
7. Click **OK** to save the changes and close the BLE Component configuration window.

5.4 RGB LED Control

For RGB LED control, this project uses a PrISM™ Component based on Cypress's propriety technology for LED intensity control. This Component utilizes stochastic signal density modulation to control the intensity of individual LEDs. Combining multiple LEDs allows for both color and intensity control. For more information, see AN47372 - PrISM™ Technology for LED Dimming.

1. Drag two **PrISM** Components from the Component catalog (Cypress > Digital > PrISM) as shown in Figure 24. Each Component supports two outputs, so for controlling three LEDs, two PrISM Components are required.

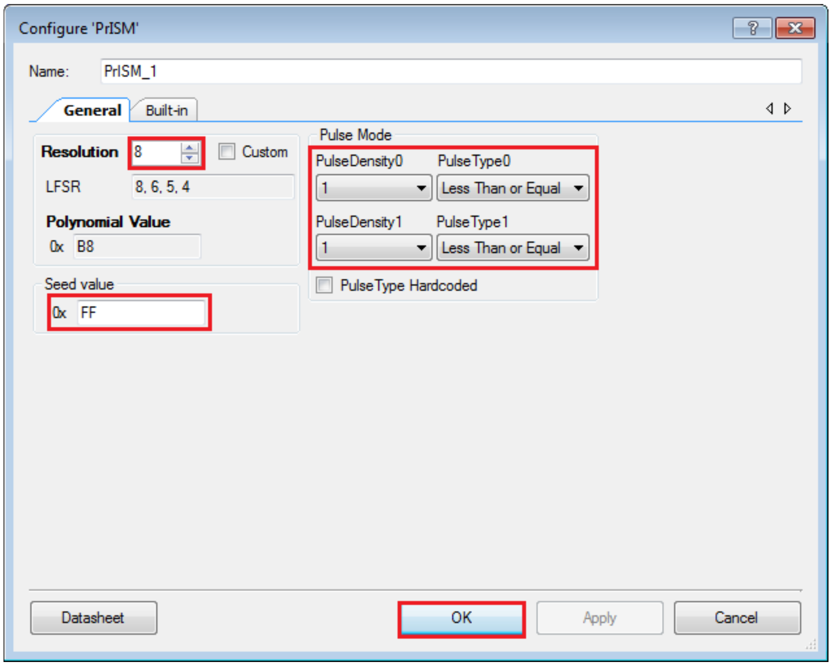
Figure 24. Place PrISM Components on Top Design



2. Double-click the first PrISM Component. On the Configuration window, perform the following on the **General** tab, as shown in Figure 25, and click **OK**:

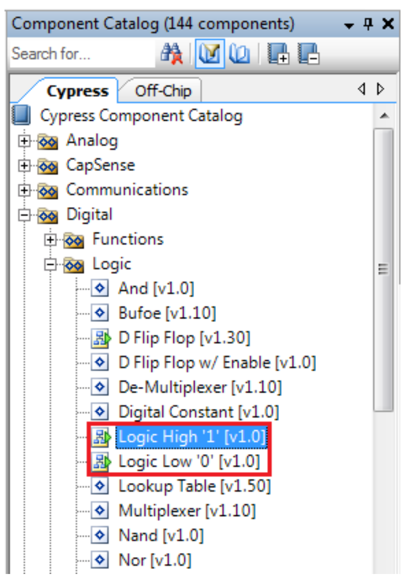
   - Set the **Resolution** as 8 bits and **Seed value** to the full range of 0xFF.
   - Under **Pulse Mode**, keep **PulseDensity0** and **PulseDensity1** at the default value of '1'. The generated random number is compared to this value.
   - Set both **PulseType0** and **PulseType1** as **Less than or Equal**. This implies that whenever a random number generated by the Component is less than or equal to the set Pulse Density value, the Component output at pulse_den0 and pulse_den1 will be HIGH; otherwise, it will be LOW.

Figure 25. PrISM Component Settings



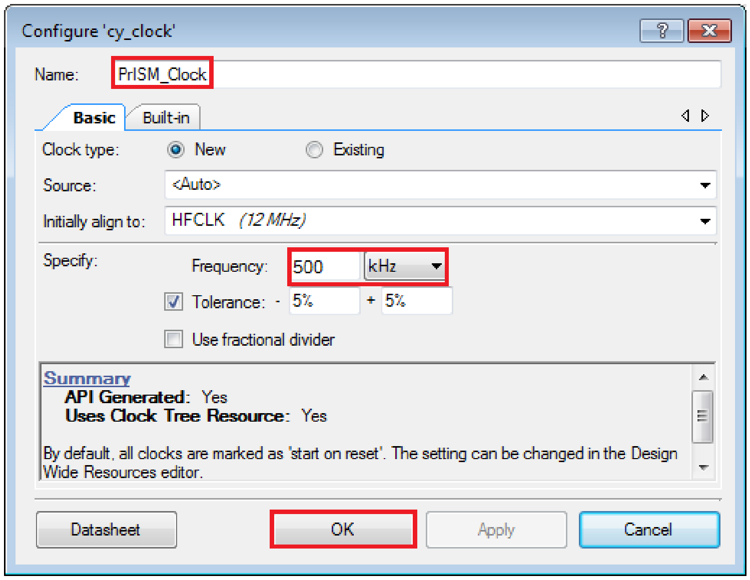
3. Configure the **PrISM_2** Component with identical settings. For this Component, only one output, **pulse_den0**, is used for the third LED. The other output will not be connected.
4. Add the following Components to the input connections of both PrISM Components from the Component Catalog:
   - A Logic High (1) Component on the enable input pins to enable the Component by default.
   - Logic Low (0) Component on the kill and reset input pins to disable the hardware reset and kill options of the Component. These two options are not needed in this project.

Figure 26. Logic High and Logic Low Components



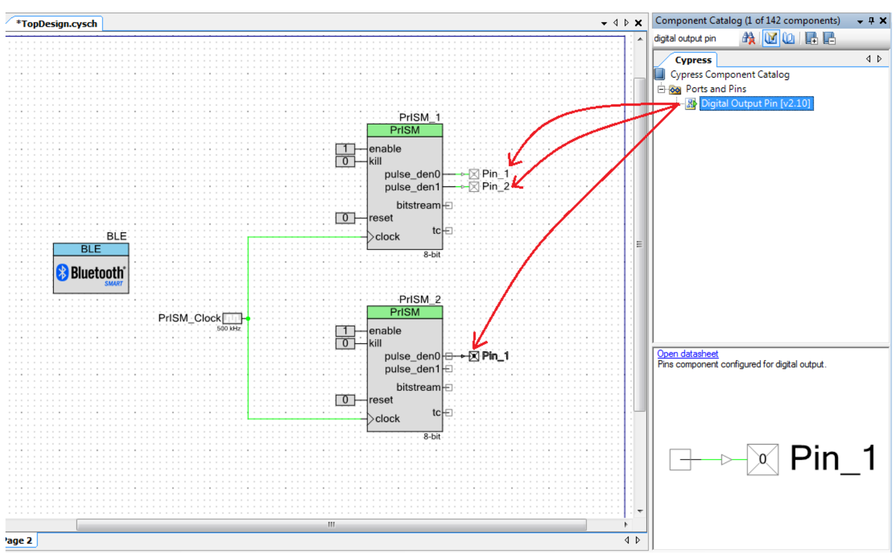
5. Drag and drop a **Clock** Component from the **System** group on the Component Catalog, and configure it as shown in Figure 27, and click **OK**:
   a. Rename the instance to **PrISM_Clock**.
   b. Set the frequency as **500 kHz**.

Figure 27. PrISM Clock Configuration



6. Connect the **Clock** Component to the **clock** input of both PrISM Components using the wire tool (press 'w' anywhere on Top Design to enable the wire functionality and then click the connecting points).
7. Drag and drop three **Digital Output Pin** Components from the **Ports and Pins** group in the Component Catalog. Connect them to the **pulse_den0** and **pulse_den1** pins of **PrISM_1** and the **pulse_den0** pin of **PrISM_2** as shown in Figure 28. These pins will be driven by the PrISM Components and will control the RGB LEDs.

Figure 28. Connect Digital Output Pins to PrISM

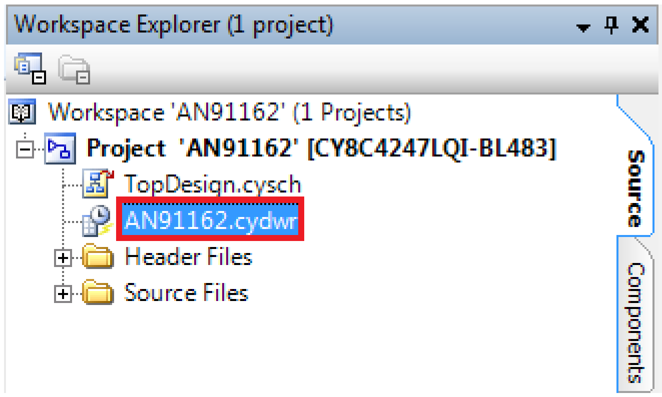


8. Double-click Pin_1, Pin_2, and Pin_3 Components and name them **RED**, **GREEN,** and **BLUE**, respectively. Set the drive mode to **High impedance Analog**. This is done because the RGB LED on BLE Pioneer Kit is active low and the initial strong drive of RGB LED pins will cause the RGB LED to show white light for a short duration.

5.5 Configure Project's Design-Wide Resources

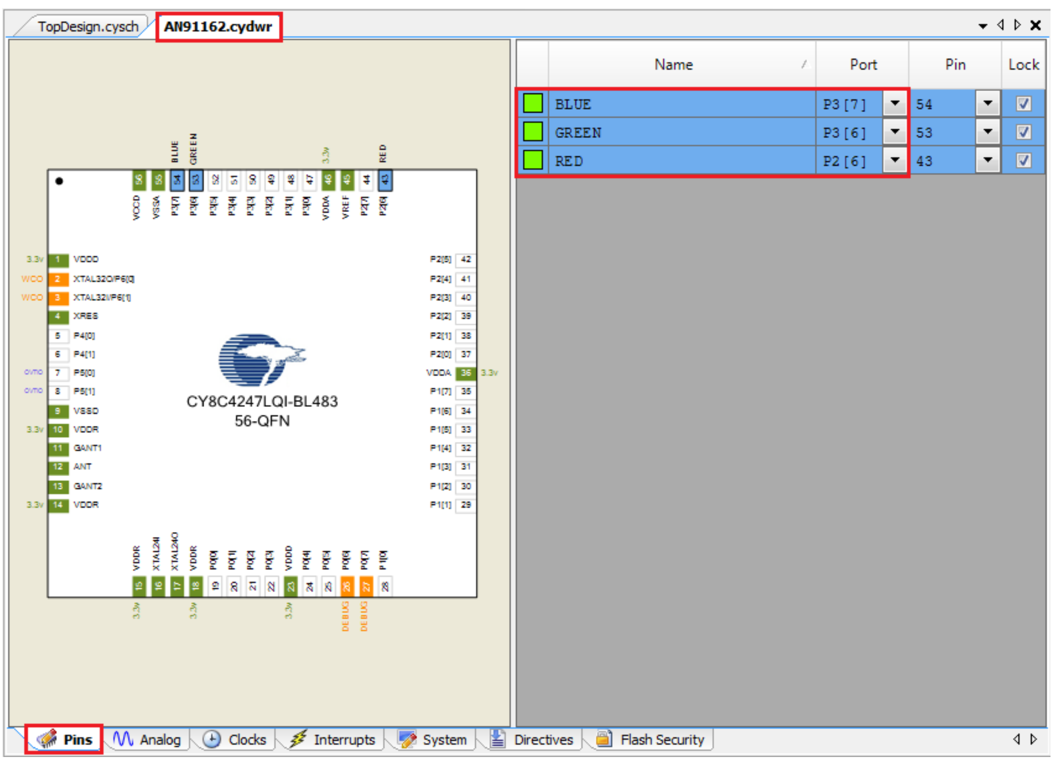
1. To assign ports to Components, double-click on the project CYDWR in Workspace Explorer as shown in Figure 29.

Figure 29. Open Project's CYDWR



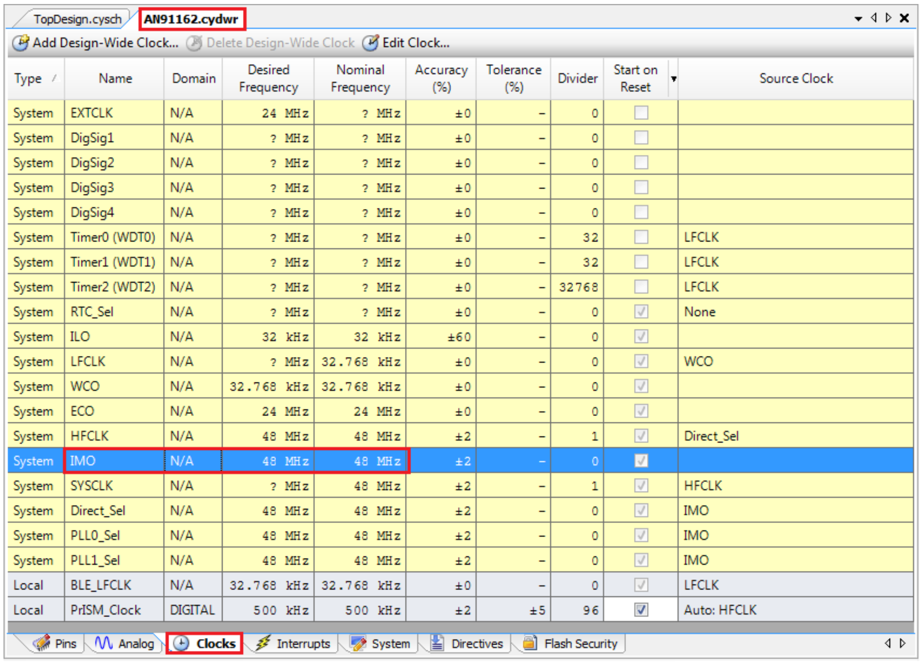
2. On the **Pins** tab, configure the pin numbers for each Component as shown in Figure 30. You can use the drop-down menu, enter the port name (for example, P3[7]), or drag the pin name to the desired location in the figure to assign the ports.

Figure 30. Pins Configuration in CYDWR



3. On the **Clocks** tab, double-click on the **IMO** clock to open the System Clocks configuration window, as shown in Figure 31.

Figure 31. CYDWR Clock Settings

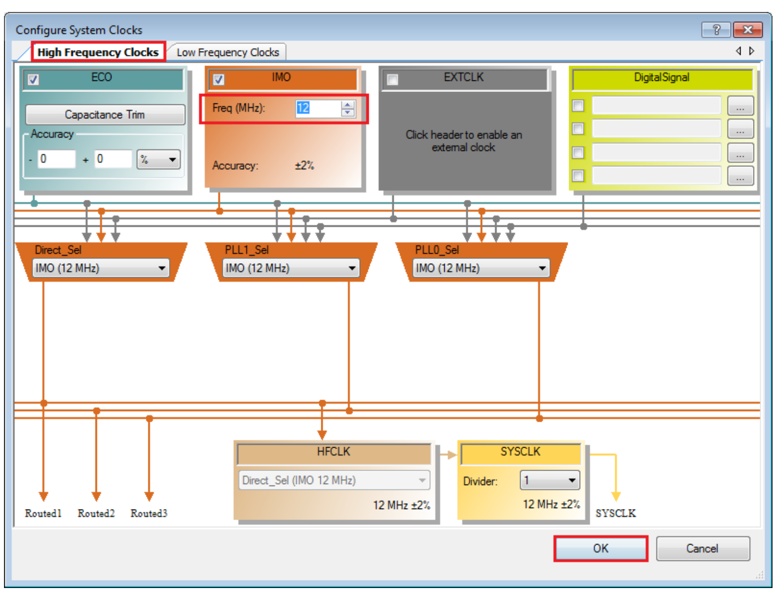


Type	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	Source Clock
System	EXTCLK	N/A	24 MHz	? MHz	±0	–	0	☐	
System	DigSig1	N/A	? MHz	? MHz	±0	–	0	☐	
System	DigSig2	N/A	? MHz	? MHz	±0	–	0	☐	
System	DigSig3	N/A	? MHz	? MHz	±0	–	0	☐	
System	DigSig4	N/A	? MHz	? MHz	±0	–	0	☐	
System	Timer0 (WDT0)	N/A	? MHz	? MHz	±0	–	32	☐	LFCLK
System	Timer1 (WDT1)	N/A	? MHz	? MHz	±0	–	32	☐	LFCLK
System	Timer2 (WDT2)	N/A	? MHz	? MHz	±0	–	32768	☐	LFCLK
System	RTC_Sel	N/A	? MHz	? MHz	±0	–	0	☑	None
System	ILO	N/A	32 kHz	32 kHz	±60	–	0	☑	
System	LFCLK	N/A	? MHz	32.768 kHz	±0	–	0	☑	WCO
System	WCO	N/A	32.768 kHz	32.768 kHz	±0	–	0	☑	
System	ECO	N/A	24 MHz	24 MHz	±0	–	0	☑	
System	HFCLK	N/A	48 MHz	48 MHz	±2	–	1	☑	Direct_Sel
System	IMO	N/A	48 MHz	48 MHz	±2	–	0	☑	
System	SYSCLK	N/A	? MHz	48 MHz	±2	–	1	☑	HFCLK
System	Direct_Sel	N/A	48 MHz	48 MHz	±2	–	0	☑	IMO
System	PLL0_Sel	N/A	48 MHz	48 MHz	±2	–	0	☑	IMO
System	PLL1_Sel	N/A	48 MHz	48 MHz	±2	–	0	☑	IMO
Local	BLE_LFCLK	N/A	32.768 kHz	32.768 kHz	±0	–	0	☑	LFCLK
Local	PrISM_Clock	DIGITAL	500 kHz	500 kHz	±2	±5	96	☑	Auto: HFCLK

4. For this project, the IMO frequency (High-Frequency Clock) is reduced from the default 48 MHz to 12 MHz for a lower power consumption, as shown in Figure 32. Keep other clock configurations at their default values, and click **OK**.

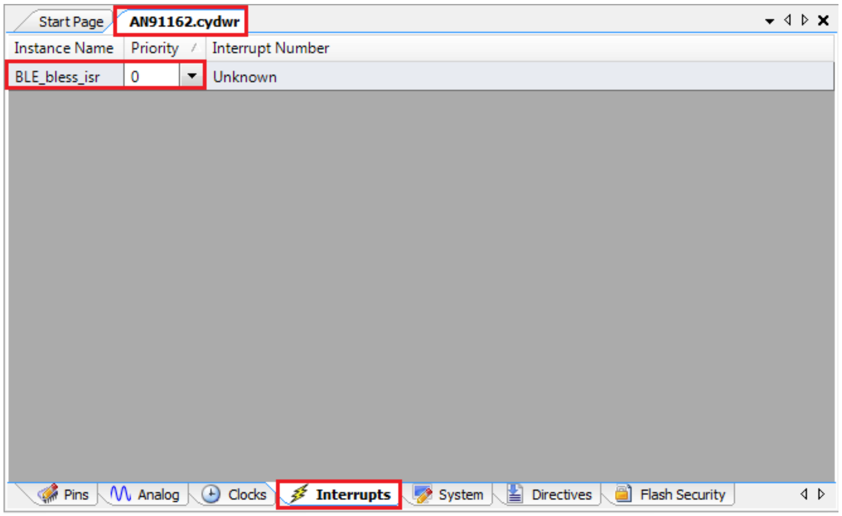
   Note The CPU clock frequency set here will affect the overall power consumption of the device. In addition, some peripherals require a minimum clock frequency to work correctly. Choose a CPU clock frequency to keep the power consumption low while not hindering the operation of the project.

Figure 32. IMO Clock Settings



5. On the **Interrupts** tab, set the priority for the BLE interrupt as '0', as shown in Figure 33. This will ensure that any other low-priority interrupt added will not affect the BLE operation.

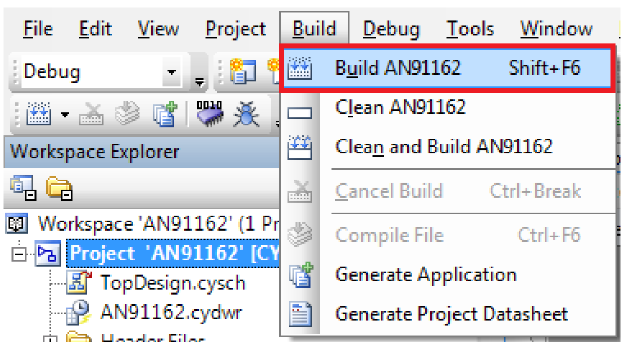
Figure 33. CYDWR Interrupts Setting



5.6 Build the Project

Select **Build** > **Build AN91162** [Shift+F6] to build the complete project, as shown in Figure 34.

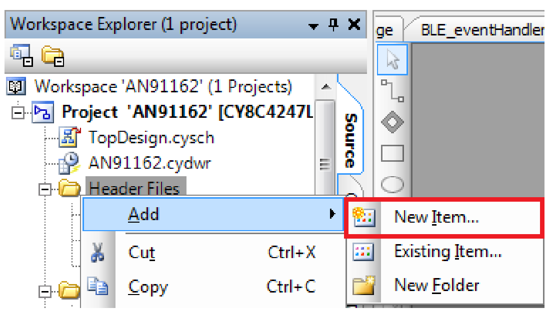
Figure 34. Build Project



5.7 Add a Source/Header File to Project

To add a new header (H) or source (C) file to the project, right-click **Header Files** or **Source Files** and then select **Add** > **New Item** as shown in Figure 35. Select the file type to be added, enter the desired name, and then click **OK**.

Figure 35. Add a Source/Header File



5.8 Project Files

The associated project has the following files:

- **main.c/.h**: These files contain the main function that acts as the entry point for the system. It initializes the system, including BLE, and regularly calls the function to process BLE events.
- **BLEProcess.c/.h**: These files contain the definitions of the functions for handling BLE event callbacks and updating the RGB LED characteristic value in the GATT database.
- **led.c/.h**: These files contain the definitions of the function that handles the Component for displaying RGB LED colors and intensity.

5.9 Configure the Firmware

This project's firmware handles the following processes:

- Initializes the Components and enable interrupts.
- Processes the BLE events that are generated by the BLE stack, such as BLE start, connection request, and write command.
- Displays the color on the RGB LED when a new color data is received from the GATT client.

This project uses the following BLE APIs:

API	Description
CyBle_Start(CYBLE_CALLBACK_T)	Starts the BLE Component and registers a function as the event handler for events coming from the BLE stack. The argument to this function is the name of the event-handler function.
CyBle_ProcessEvents(void)	Processes the BLE events between the BLE stack and the application. This should be continuously called in the main function. This function has no argument.
CyBle_GappStartAdvertisement(uint8)	Starts BLE Peripheral advertising with the interval set in the BLE Component (as listed in Table 3). The argument defines if the advertisement is fast, slow, or custom.
CyBle_GattsWriteRsp(CYBLE_CONN_HANDLE_T)	Sends a write response back to the GATT client device whenever the GATT client device sends a write request. This function has the connection handle as the argument.
CyBle_GattsWriteAttributeValue(CYBLE_GATT_HANDLE_VALUE_PAIR_T *, uint16, CYBLE_CONN_HANDLE_T *, uint8)	Updates the data value of an attribute (such as a characteristic) so that the value is available for read by the GATT client device. This function has four arguments to receive the updated data, offset, connection handle, and flags related to the data to be communicated.

5.9.1 Macro Definitions

Each header file contains macros for constants used in the code. Macros from each file are shown below:

main.h

```
#define TRUE                                      0x01
#define FALSE                                     0x00
```

BLEProcess.h

```
/* RGB LED Characteristic data length*/
#define RGB_CHAR_DATA_LEN                         4
```

led.h

```
/* LED Color and status related Macros */
#define RGB_LED_MAX_VAL                           0xFF
#define RGB_LED_OFF                               0xFF
#define RGB_LED_ON                                0x00

/* Index values in array where respective color coordinates
* are saved */
#define RED_INDEX                                 0x00
#define GREEN_INDEX                               0x01
#define BLUE_INDEX                                0x02
#define INTENSITY_INDEX                           0x03
```

5.9.2 System Initialization

The first step in firmware configuration is to initialize the Components in the system. The following function is called first after entering *main.c*. Open *main.c* by double-clicking on it in the Workspace Explorer window on the left-hand side of the PSoC Creator window. Add the following function definition in *main.c*:

```
void InitializeSystem(void)
{
	/* Enable Global Interrupt Mask */
	CyGlobalIntEnable;

	/* Start BLE stack and register the event callback function. */
	CyBle_Start(GeneralEventHandler);

	/* Start PrISM modules for LED control */
	PrISM_1_Start();
	PrISM_2_Start();

	/* Switch off the RGB LEDs through PrISM modules */
	PrISM_1_WritePulse0(RGB_LED_OFF);
	PrISM_1_WritePulse1(RGB_LED_OFF);
	PrISM_2_WritePulse0(RGB_LED_OFF);

	/* Set Drive modes of the output pins to Strong */
	RED_SetDriveMode(RED_DM_STRONG);
	GREEN_SetDriveMode(GREEN_DM_STRONG);
	BLUE_SetDriveMode(BLUE_DM_STRONG);
}
```

5.9.3 Event Handler Registration

Unlike other Components' startup, the BLE Component requires the registering of an **event callback function** while starting the Component. This function is called to handle BLE events, including general events such as stack ON and events at the GAP/GATT layer such as connection, disconnection, and write command. The General event handler function is defined in *BLEProcess.c* in the example project. You can either place it in a separate file or in *main.c*. See

Table 5 for a description of the events that are included in the switch statement. In the function definition shown below, each case is empty. We will add code to handle each event in the next section.

```
void GeneralEventHandler(uint32 event, void * eventParam)
{
        /* Structure to store data written by Client */
        CYBLE_GATTS_WRITE_REQ_PARAM_T *wrReqParam;

        /* 'RGBledData[]' is an array to store 4 bytes of RGB LED data*/
        uint8 RGBledData[RGB_CHAR_DATA_LEN];

        switch(event)
       {
                case CYBLE_EVT_STACK_ON:
                /* This event is generated when BLE stack is ON */

                break;

                case CYBLE_EVT_GAP_DEVICE_DISCONNECTED:
                /* This event is generated at GAP disconnection. */

                break;

               case CYBLE_EVT_GATTS_WRITE_REQ:
                /* This event is generated when the connected Central */
                /* device sends a Write request. */
                /* The parameter 'eventParam' contains the data written */

                break;

               default:

                break;
     }
}
```

These events are the basic events to be handled in the application to allow a successful BLE connection. These events are explained in Table 5. Other events that can be generated by the BLE Component are described in the *BLE_Stack.h* file in "`CYBLE_EVENT_T`" `enum`.

Table 5. BLE Events

Event Name	Event Description	Event Handling
CYBLE_EVT_STACK_ON	BLE stack is initialized successfully after calling CyBle_Start().	When BLE stack is ON, start the advertisement.
CYBLE_EVT_GAPP_ADVERTISEMENT_START_STOP	Peripheral advertising starts or stops.	Go into a low-power mode or restart the advertisement.
CYBLE_EVT_GAP_DEVICE_DISCONNECTED	The BLE connection between the Peripheral and Central device is disconnected.	Go into a low-power mode or restart the advertisement.
CYBLE_EVT_GATT_CONNECT_IND	A connection has been established between the Peripheral and a Central device.	Update the connection handle variable. Not used in this project.
CYBLE_EVT_GATT_DISCONNECT_IND	The connection with the Central device has been disconnected.	Reset the GATT database values.
CYBLE_EVT_GATTS_WRITE_REQ	A write request has been sent from the GATT client device.	Extract the data sent by the GATT client and send the Write response.

5.9.4 Start Advertisement

As the project is a GAP Peripheral, it needs to start advertisement to allow a GAP Central device to connect to it. There are two events where advertisement will be started. Place the respective code in the general event callback function for the following events:

- When the system powers up and the BLE Stack is ON (event `CYBLE_EVT_STACK_ON`)

```
case CYBLE_EVT_STACK_ON:
       /* BLE stack is on. Start BLE advertisement */
       CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
break;
```

- When the existing connection with a Central device has been disconnected (event `CYBLE_EVT_GAP_DEVICE_DISCONNECTED`)

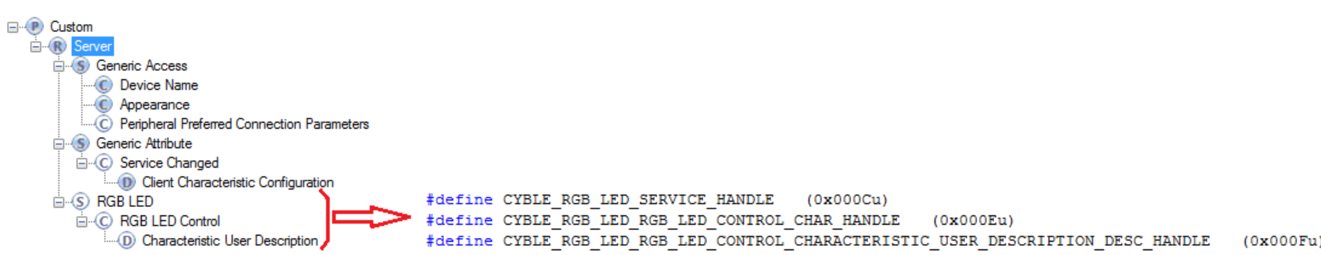
```
case CYBLE_EVT_GAP_DEVICE_DISCONNECTED:
       /* This event is generated at GAP disconnection. */
       /* Restart advertisement */
       CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
break;
```

5.9.5 Attribute Handles for Custom Service

In BLE communication, both the GATT client and the GATT server access data on attributes (services, characteristics, or descriptors) by using an **attribute handle**. This attribute handle is a 16-bit value that uniquely identifies the attribute after establishing a connection.

For custom services and characteristics added to the BLE Component, the value of these handles is generated by the Component and can be found in the generated file *BLE_custom.h* as #defines. For the BLE custom services added in this project (RGB LED), the handles generated are as shown in Figure 36.

Figure 36. Attribute Handle Data Structure for Custom Services



The RGB LED service supports both reads and writes on the same characteristic with the attribute handle of value *0x000E*.

5.9.6 Handle Write Requests

For the RGB LED characteristic, the GATT client sends a Write request with four bytes of data. This data is received as part of the `CYBLE_EVT_GATTS_WRITE_REQ` event in the general event callback function. The attribute handle of the received data is compared with that of the RGB LED Control characteristic. If they match, the following actions are taken:

1. The four bytes of data are extracted and stored in an array.
2. The RGB LED update function (`UpdateRGBLED`) is called to update the onboard LED color.
3. The RGB Control characteristic update function (`UpdateRGBcharacteristic`) is called to update the internal GATT database value.
4. Irrespective of whether the attribute handle matches RGB LED Control Characteristic handle, a write response is sent back to the Client device using the BLE function `CyBle_GattsWriteRsp`, so that the client knows that the data was received.

Place the following code under `CYBLE_EVT_GATTS_WRITE_REQ` event:

```
case CYBLE_EVT_GATTS_WRITE_REQ:
       /* Extract the Write data sent by Client */
      wrReqParam = (CYBLE_GATTS_WRITE_REQ_PARAM_T *) eventParam;

       /* If the attribute handle of the characteristic written to
       * is equal to that of RGB_LED characteristic, then extract
       * the RGB LED data */
      if(CYBLE_RGB_LED_RGB_LED_CONTROL_CHAR_HANDLE ==
                              wrReqParam->handleValPair.attrHandle)
      {
               /* Store RGB LED data in local array */
                RGBledData[RED_INDEX] =
                              wrReqParam->handleValPair.value.val[RED_INDEX];
                RGBledData[GREEN_INDEX] =
                              wrReqParam->handleValPair.value.val[GREEN_INDEX];
                RGBledData[BLUE_INDEX] =
                              wrReqParam->handleValPair.value.val[BLUE_INDEX];
                RGBledData[INTENSITY_INDEX] =
                              wrReqParam->handleValPair.value.val[INTENSITY_INDEX];

              /* Update the PrISM component density value to represent color */
             UpdateRGBLED(RGBledData, sizeof(RGBledData));

              /* Update the GATT DB for RGB LED read characteristics*/
              UpdateRGBcharacteristic(RGBledData,
                                         sizeof(RGBledData),
                                         CYBLE_RGB_LED_RGB_LED_CONTROL_CHAR_HANDLE);
      }
       /* Send the response to the write request received. */
       CyBle_GattsWriteRsp(cyBle_connHandle);
break;
```

The `UpdateRGBLED` function calculates the brightness of each of the RGB LEDs using the four-byte (red, green, blue, intensity) values received. It then updates the density value of the PrISM Components to achieve the desired color. Place the following function in the project (defined in the *led.c* file of the associated project).

```
void UpdateRGBLED(uint8* ledData, uint8 len)
{
       /* Local variables to store calculated color components */
       uint8 calc_red;
       uint8 calc_green;
       uint8 calc_blue;

       /* Check if the array has length equal to expected length for
       * RGB LED data */
       if(len == RGB_CHAR_DATA_LEN)
       {
              /* True color to be displayed is calculated on basis of color
              * and intensity value received */
              calc_red = (uint8)
       (((uint16)ledData[RED_INDEX]*ledData[INTENSITY_INDEX])/RGB_LED_MAX_VAL);
              calc_green = (uint8)
       (((uint16)ledData[GREEN_INDEX]*ledData[INTENSITY_INDEX])/RGB_LED_MAX_VAL);
              calc_blue = (uint8)
       (((uint16)ledData[BLUE_INDEX]*ledData[INTENSITY_INDEX])/RGB_LED_MAX_VAL);

              /* Update the density value of the PrISM module */
              PrISM_1_WritePulse0(RGB_LED_MAX_VAL - calc_red);
              PrISM_1_WritePulse1(RGB_LED_MAX_VAL - calc_green);
              PrISM_2_WritePulse0(RGB_LED_MAX_VAL - calc_blue);
       }
}
```

When the LED color is set, the GATT database has to be updated for the RGB LED characteristic so that the Client receives the latest RGB color set when it sends a Read request. The `UpdateRGBcharacteristic` function updates the attribute value for RGB LED color control. Add the following function in the project (defined in *BLEProcess.c* file of the associated project).

```
void UpdateRGBcharacteristic(uint8* ledData, uint8 len, uint16 attrHandle)
{
       /* 'rgbHandle' stores RGB control data parameters */
       CYBLE_GATT_HANDLE_VALUE_PAIR_T          rgbHandle;

       /* Update RGB control handle with new values */
       rgbHandle.attrHandle = attrHandle;
       rgbHandle.value.val = ledData;
       rgbHandle.value.len = len;

       /* Update the RGB LED attribute value. This will allow
       * Client device to read the existing color values over
       * RGB LED characteristic */
      CyBle_GattsWriteAttributeValue(&rgbHandle,
                                    FALSE,
                                    &cyBle_connHandle,
                                    CYBLE_GATT_DB_PEER_INITIATED);
}
```

5.9.7 Handle BLE Disconnection

When the device is disconnected from the Central device, the RGB LED and GATT database should be reset before next connection. Place the following code snippet under the `CYBLE_EVT_GAP_DEVICE_DISCONNECTED` event in the general event callback function, along with the start advertisement API call:

```
case CYBLE_EVT_GAP_DEVICE_DISCONNECTED:
        /* This event is generated at GAP disconnection. */

        /* Reset the color values*/
        RGBledData[RED_INDEX] = FALSE;
        RGBledData[GREEN_INDEX] = FALSE;
        RGBledData[BLUE_INDEX] = FALSE;
        RGBledData[INTENSITY_INDEX] = FALSE;

        /* Switch off LEDs */
        UpdateRGBLED(RGBledData, sizeof(RGBledData));

        /* Register the new color in GATT DB*/
        UpdateRGBcharacteristic(RGBledData,
                                        sizeof(RGBledData),

        CYBLE_RGB_LED_RGB_LED_CONTROL_CHAR_HANDLE);

        /* Restart advertisement */
        CyBle_GappStartAdvertisement(CYBLE_ADVERTISING_FAST);
break;
```

5.9.8 Main Function

With the general event callback function complete, we now modify the main function to initialize the Components in the project and process the BLE events. Modify the existing main function in *main.c* as provided below:

```
int main()
{
    /* Start the components */
       InitializeSystem();

    for(;;)
    {
        /* Process BLE Events. This generates events in the callback function */
              CyBle_ProcessEvents();
    }
}
```

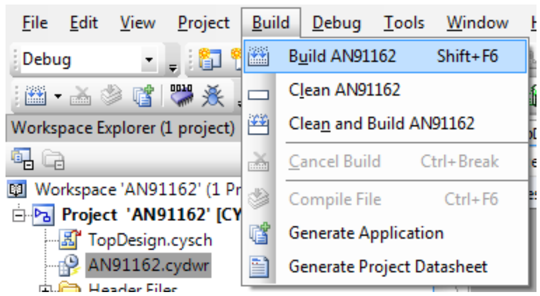
CyBle_ProcessEvents() should be called periodically, and at least once between each BLE connection interval, to process the BLE events successfully.

Refer to the associated project for the complete firmware.

5.10 Build and Program

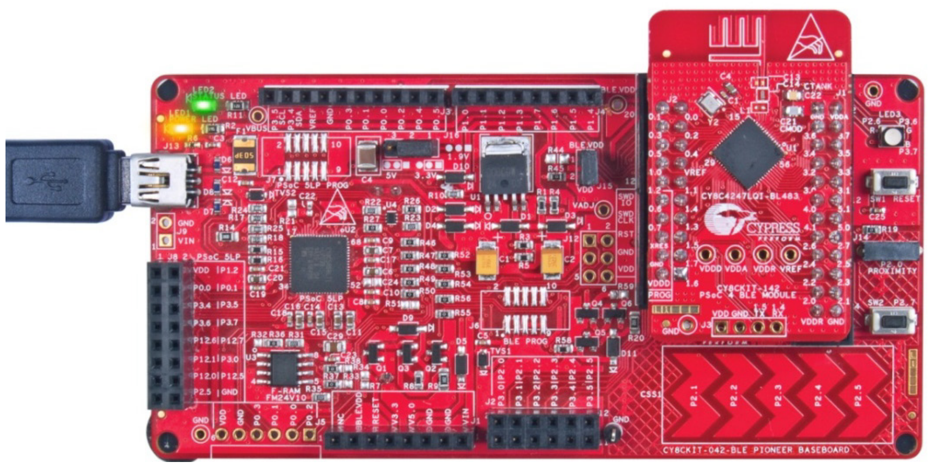
1. Select **Build** > **Build AN91162** to build and compile the firmware, as shown in Figure 37. The project should build without warnings or errors.

Figure 37. Build Project



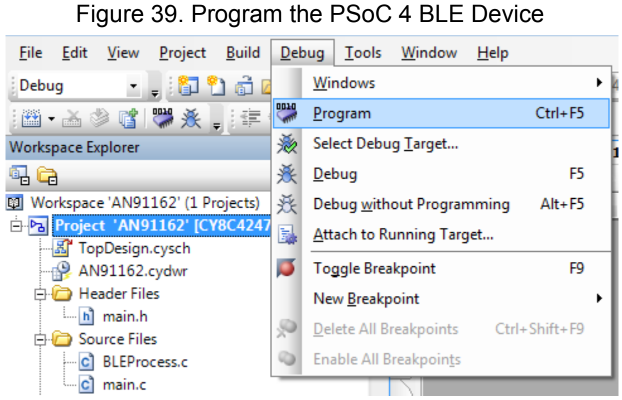
2. Plug the PSoC 4 BLE module (red module) to the BLE Pioneer baseboard, and then connect the kit to your PC using the USB Standard-A to Mini-B cable (see Figure 38). Allow the USB enumeration to complete on the PC.

Figure 38. Connect to PC Using a USB Cable



3. Select **Debug** > **Program** (see Figure 39). If there is only one kit connected to the PC, programming will start automatically. If multiple kits are present, PSoC Creator will prompt you to choose the kit to be programmed.

Figure 39. Program the PSoC 4 BLE Device



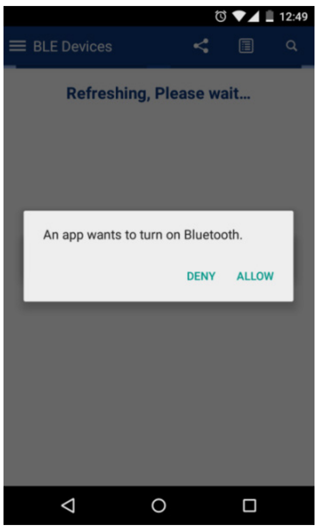
After programming is complete, the BLE Pioneer Kit will start advertising.

5.11 Testing with CySmart Mobile App

1. Download the CySmart mobile app on your BLE-enabled phone. For iOS devices (iPhone 4S or later), download the app from App Store. For Android devices (Android 4.3 or later), download the app from Play Store.
2. Start the app on your mobile. If Bluetooth on your mobile is not enabled, the app will prompt you to enable it, as shown in Figure 40.

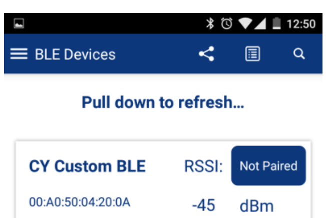
   Note The screenshots are for the CySmart Android app. The look and feel of the CySmart iOS app may differ slightly.

Figure 40. Enable Bluetooth on Mobile



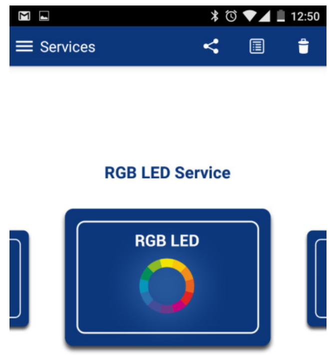
3. After enabling Bluetooth, the device screen will be displayed. Swipe down to list all the BLE devices present in the vicinity, including the PSoC 4 BLE Custom Service project "**CY Custom BLE**" (see Figure 41).

Figure 41. BLE Devices Listed



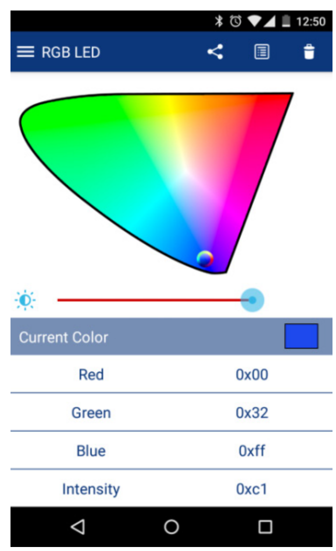
4. Select the **CY Custom BLE** device. The connection procedure should be initiated and device connected. The screen will display profile/service pages that the connected device supports (see Figure 42).

Figure 42. Service Page



5. If RGB LED icon is not displayed, Swipe left or right. Select the RGB LED icon when it appears.
6. On the RGB LED GUI screen, the Color Gamut (see Figure 43) controls the value of Red, Green, and Blue components whereas the linear slider controls the intensity. Increase the intensity using slider and then swipe on the Color Gamut to see the same color being set on the BLE Pioneer Kit RGB LED.

Figure 43. RGB LED Color Control



7. To disconnect the device, tap the **Back** button in the app until you reach the Device Search page.

5.12 Testing with CySmart Central Emulation Tool

The CySmart Central Emulation Tool, along with a BLE Dongle, emulates a BLE GATT client device. This allows you to connect to any BLE device, discover its attributes, and communicate data over these attributes with a Peripheral device. Download the latest CySmart Central Emulation Tool from www.cypress.com/cysmart and the latest firmware HEX file for the BLE Dongle from www.cypress.com/CY8CKIT-042-BLE.

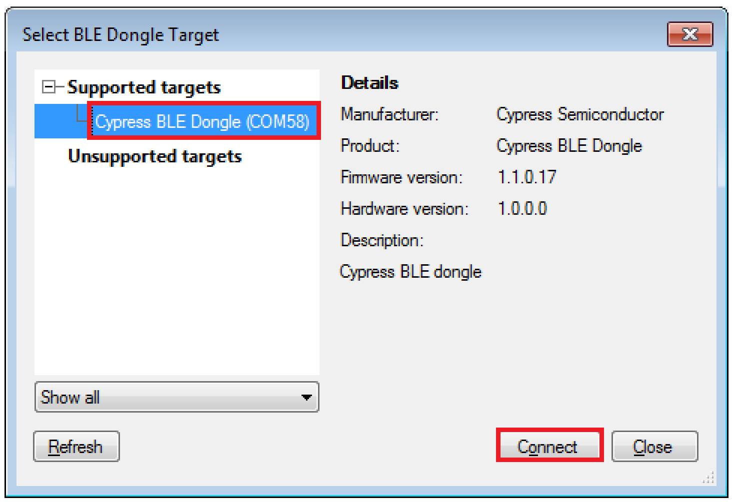
Note The CySmart Central Emulation Tool is currently supported only on Windows PCs.

To test the project with the CySmart Central Emulation Tool, follow these steps:

1. Connect the BLE Dongle to your PC. Allow the USB enumeration to complete.
2. Launch the CySmart tool: click **Start** > **All Programs** > **Cypress** > **CySmart *<version>*** > **CySmart *<version>***.
3. On the CySmart Central Emulation Tool, select the **Cypress BLE Dongle** from the **Supported targets** list, and click **Connect**, as shown in Figure 44.

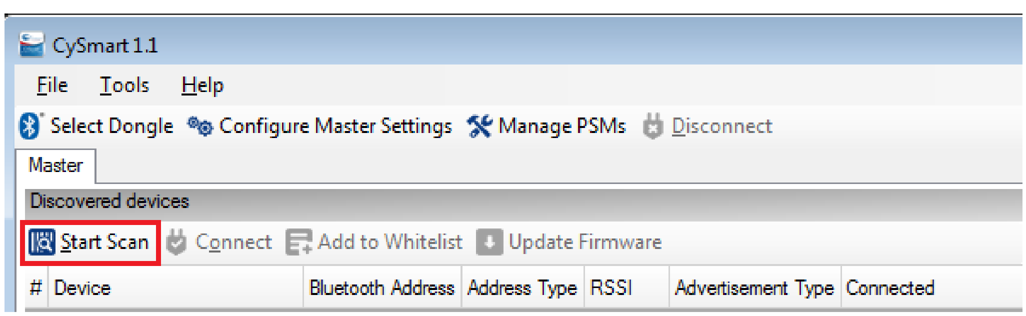
   Note If the BLE Dongle is not listed, press the reset button on the BLE Dongle and then click **Refresh**.

Figure 44. Select BLE Dongle Target



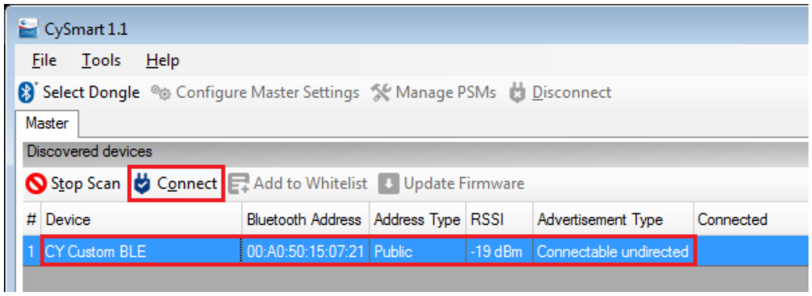
4. After the BLE Dongle is selected, click **Start Scan** at top left to start scanning for BLE Peripheral devices, as shown in Figure 45.

Figure 45. Start Scan on CySmart Central Emulation Tool



5. Select your device with the name **CY Custom BLE** and click **Connect**, as shown in Figure 46.

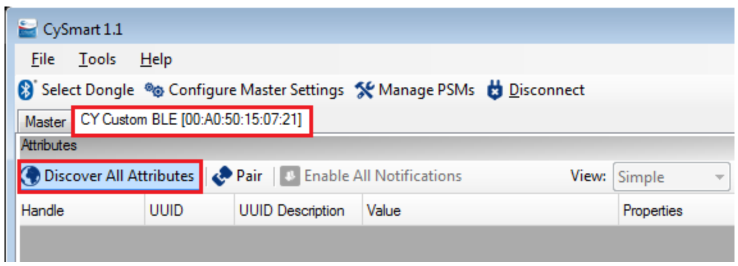
Figure 46. Connect to BLE_Custom Device



After the device is connected, on the CySmart Central Emulation Tool, a new tab opens beside the Master tab, with the name of the device it is connected to, as shown in Figure 47.

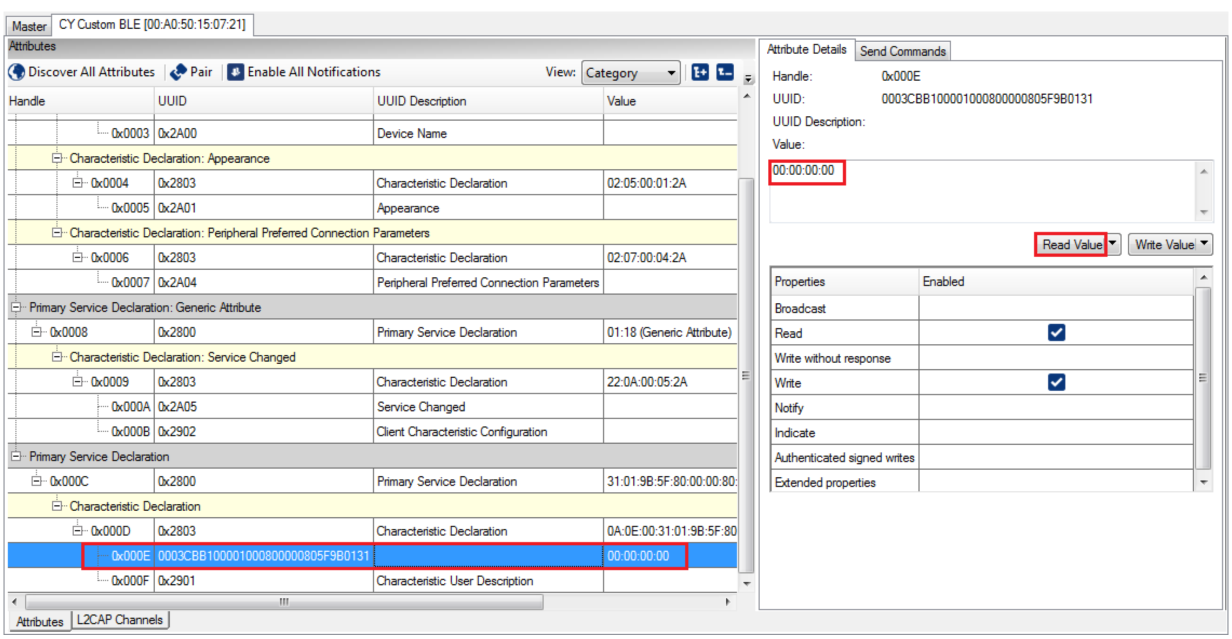
6. Select **Discover All Attributes** to initiate the CySmart tool querying for supported attributes by the **CY Custom BLE** device, as shown in Figure 47.

Figure 47. Discover All Attributes



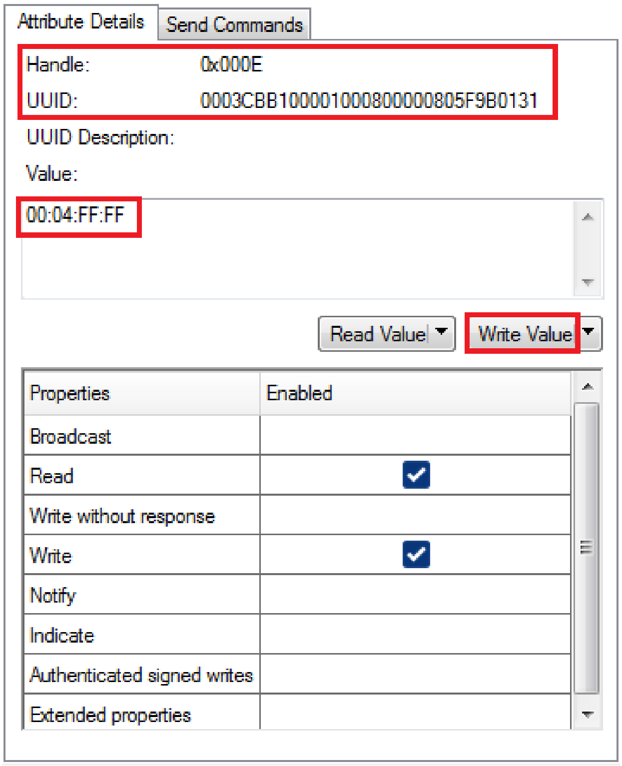
Scroll down the attribute list and click on the RGB LED custom characteristic (UUID **0003CBB1-0000-1000-8000-00805F9B0131**). This characteristic supports both read and write, as indicated by Attribute details on the right part of CySmart window. Click on **Read Value** to read the existing color values, as shown in Figure 48. A 4-byte value will be displayed in the Value field.

Figure 48. RGB LED Custom Characteristic



7. Write non-zero values in the four bytes in the **Value** field and click **Write Value** to send the new color values, as shown in Figure 49. The format of the 4-byte value is **Red:Green:Blue:Intensity**, with '0' being the lowest value and 'FF' being the highest value.

Figure 49. Write New Color Values to Custom Characteristic

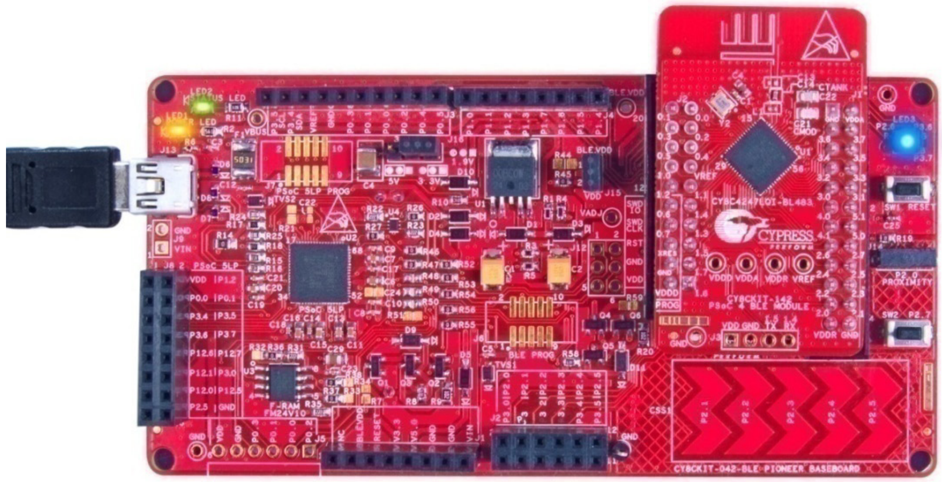


Send any other 4-byte data and observe the corresponding colors.

RGB Data	Color Observed
00:00:00:00	No Color
FF:00:00:FF	Full Red
00:FF:00:FF	Full Green
00:00:FF:FF	Full Blue
FF:00:FF:22	Purple, low intensity
FF:FF:00:55	Yellow, medium intensity

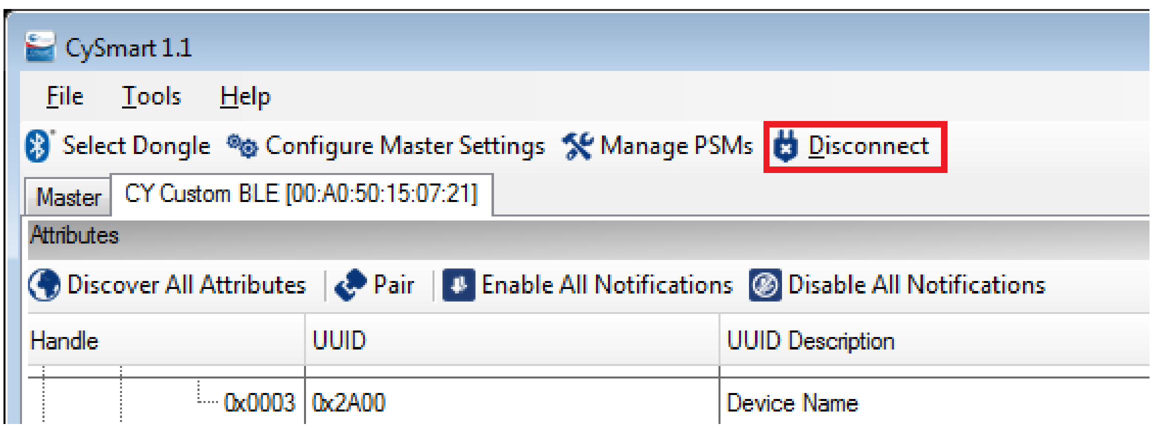
8. Observe the new color and intensity on the RGB LED of the BLE Pioneer Kit, as shown in Figure 50.

Figure 50. RGB LED Control on BLE Pioneer Kit



9. To disconnect the device, click **Disconnect**, as shown in Figure 51.

Figure 51. Disconnect Device



6 Summary

This application note demonstrated the steps to add custom BLE services in a PSoC 4 BLE project using the BLE Component, configuring the services, and reading and writing data from and to a BLE GATT client device. The method described here can be easily extended to any type and any number of BLE custom services in your PSoC 4 BLE project.

7 Related Information

- AN91267 - Getting Started with PSoC 4 BLE
- AN91184 - PSoC 4 BLE Designing BLE Applications
- CY8CKIT-042-BLE Pioneer Kit
- BLE Developer Portal
- CySmart iOS App
- CySmart Android App

About the Author

Name: Rohit Kumar

Title: Senior Applications Engineer

A Appendix

A.1 Send Notifications

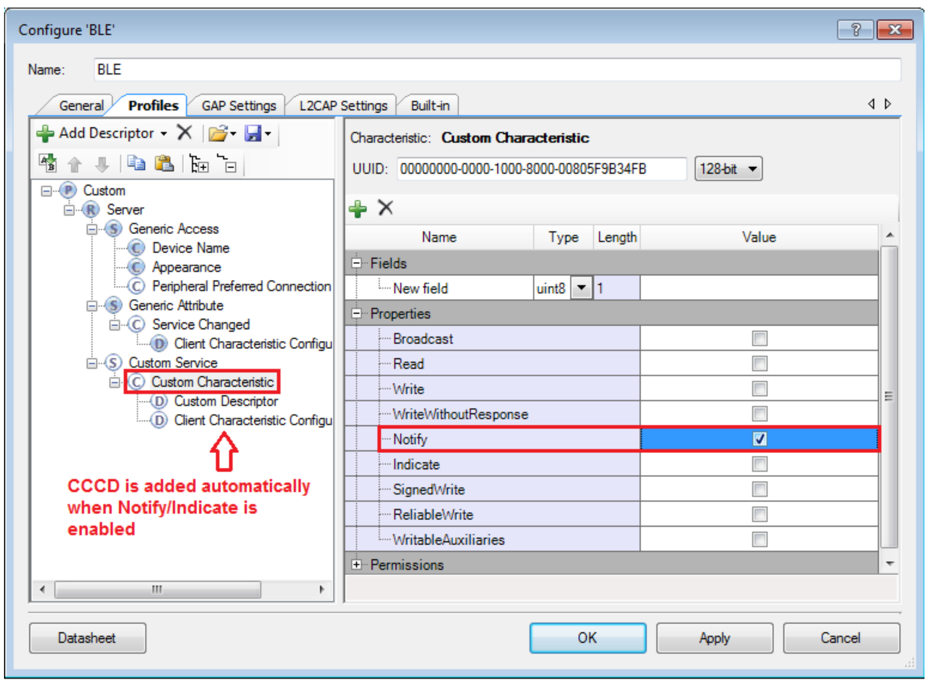
In addition to reading and writing from a characteristic, another important access that is commonly required is notifications. Using notifications, a GATT server can send new data to a GATT client without having the GATT client continuously poll for it.

Every characteristic that supports notifications has an associated descriptor, called Client Characteristic Configuration Descriptor (CCCD). The GATT client enables and disables notifications on the GATT server by writing to this CCCD. Until the GATT client has enabled notifications on the GATT server, the GATT server cannot send data through notifications.

To allow the notification access to a custom characteristic and send the data to GATT client device, follow these steps in your project. Similar steps are valid for Indicate support:

1. In the BLE Component configuration window, select the characteristic on which notifications are to be enabled. Select the checkbox against **Notify** as shown in Figure 52. The Client Characteristic Configuration Descriptor is automatically added in the attribute list, below the characteristic. Click **OK**.

Figure 52. Select Notify Access in Component



2. To enable notifications on the GATT server, the GATT client will write a value of **0x0001** to the CCCD. When '`CYBLE_EVT_GATTS_WRITE_REQ`' event occurs, do following things:
   a. Check whether the Write request is for CCCD's attribute handle
   b. If yes, then check if the value sent only has either of the lowest two bits set and no other bits are set. These bits are the only allowed values that can be sent as part of write request on CCCD.
   c. If yes, then record the CCCD value in the GATT server.
   d. Send a write response or error response back to client, depending on whether the CCCD write was successful or not.

```
case CYBLE_EVT_GATTS_WRITE_REQ:

wrReqParam = (CYBLE_GATTS_WRITE_REQ_PARAM_T *) eventParam;

/* Check if the returned handle is matching to CCCD attribute */
if(CYBLE_CUSTOM_CLIENT_CHARACTERISTIC_CONFIGURATION_DESC_HANDLE ==
                                        wrReqParam->handleValPair.attrHandle)
{
       /* Only the first and second lowest significant bit can be
       * set when writing on CCCD. If any other bit is set, then
       * send error code */
       if(FALSE ==
              (wrReqParam->handleValPair.value.val
       [CYBLE_CUSTOM_CLIENT_CHARACTERISTIC_CONFIGURATION_DESC_INDEX] &
       (~CCCD_VALID_BIT_MASK)))
       {
              /* Set flag for application to know status of notifications.
              * Only one byte is read as it contains the set value. */
              startNotification =
                            wrReqParam->handleValPair.value.val
              [CYBLE_CUSTOM_CLIENT_CHARACTERISTIC_CONFIGURATION_DESC_INDEX];

              /* Update GATT DB with latest CCCD value */
              CyBle_GattsWriteAttributeValue(&wrReqParam->handleValPair,
                                                  FALSE,
                                                  &cyBle_connHandle,
                                                  CYBLE_GATT_DB_PEER_INITIATED);
       }
       else
       {
              /* Send error response for Invalid PDU against Write
              * request */
              CYBLE_GATTS_ERR_PARAM_T err_param;

              err_param.opcode = CYBLE_GATT_WRITE_REQ;
              err_param.attrHandle = wrReqParam->handleValPair.attrHandle;
              err_param.errorCode = ERR_INVALID_PDU;

              /* Send Error Response */
              (void)CyBle_GattsErrorRsp(cyBle_connHandle, &err_param);

              /* Return to main loop */
              return;
       }
}

/* Send response to the Write request */
CyBle_GattsWriteRsp(connectionHandle);
break;
```

The error code '`ERR_INVALID_PDU`' has a value of 0x04, according to the "BLE Core specification, Vol 3, Part F, section 3.4.1".

Define the following in your application code:

```
/****************GATT Error code******************/
#define ERR_INVALID_PDU                                   0x04
#define CCCD_VALID_BIT_MASK                               0x03
#define NOTIFY_BIT_MASK                                   0x01
```

3. In the main application, send the data through a notification whenever data is available and notifications have been enabled from the GATT client.

```
/* 'notificationHandle' is handle to store notification data parameters */
CYBLE_GATTS_HANDLE_VALUE_NTF_T    notificationHandle;

/* Check if the notification bit is set or not */
if(startNotification & NOTIFY_BIT_MASK)
{
        /* Update Notification handle with new data*/
        notificationHandle.attrHandle = CYBLE_CUSTOM_CHAR_HANDLE;
        notificationHandle.value.val = &data;
        notificationHandle.value.len = dataLength;

        /* Report data to BLE component for sending data by notifications*/
        CyBle_GattsNotification(connectionHandle, &notificationHandle);
}
```

4. For an example project implementing BLE notifications over custom profile, refer to CapSense_Proximity or CapSense_Slider_and_LED projects of CY8CKIT-042-BLE Pioneer Kit.

Document History

Document Title: AN91162 – Creating a BLE Custom Profile

Document Number: 001-91162

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4606922	ROIT	03/20/2015	New Application note
*A	4767190	ROIT	05/20/2015	Updated project to PSoC Creator 3.2. Added support for PSoC 4 BLE 256K parts, CY8C4XX8-BL. Updated BLE component v2.0 screenshots. Updated CySmart Android App screenshots. Removed HandleStatusLed() function and usage. Updated UpdateRGBcharacteristic() function definition. Added GATT DB update and error response code for handling Notification in Appendix A1. Renamed UpdateRGBled() function to UpdateRGBLED() function.
*B	4905597	ROIT	09/02/2015	Fixed broken links in the document. Updated template to add numbered headings.
*C	5094468	ROIT	02/11/2016	Updated associated project to PSoC Creator 3.3 SP1. Updated Figure 1 through Figure 3, Figure 10 through Figure 27, Figure 31 through Figure 33, Figure 44 through Figure 49 and Figure 51. Updated PSoC Creator Project: RGB LED Custom Profile, Macro Definitions. Changed the parameter of API in Handle Write Requests. Removed events CYBLE_EVT_GAPP_ADVERTISEMENT_START_STOP and CYBLE_EVT_GATT_DISCONNECT_IND events from BLEProcess.c. Added information on various PSoC 4 BLE devices and BLE component compatibility in Create a PSoC Creator Project. Added Generate Custom UUIDs.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com