CE220960 – PSoC 6 MCU BLE Upgradable Stack Bootloader

Objective

This example demonstrates over-the-air (OTA) bootloading with a PSoC® 6 MCU with Bluetooth Low Energy connectivity (PSoC 6 BLE). The BLE stack code is shared between applications to reduce flash usage. The bootloader may download updates to the BLE stack or to the application.

Overview

This example demonstrates how to use the “Stack and Profile” and “Profile only” options of the PSoC Creator™ BLE Component to enable code sharing of the BLE stack between the stack and the user applications, considerably reducing the amount of flash memory used. Additionally, it demonstrates an architecture that allows upgrading the BLE stack.

Requirements

Tool: PSoC Creator 4.2, Peripheral Driver Library (PDL) 3.0.1 with Bootloader SDK 2.10, CySmart™ 1.2.1.711

Programming Language: C (Arm® GCC 5.4.1 and Arm MDK 5.22)

Associated Parts: All PSoC 63 MCU BLE parts

Related Hardware: CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit

Hardware Setup

Set the VDD Select Switch (SW5) of the CY8CKIT-062-BLE kit to 3.3 V to fully use the RGB LED.

For BLE communications, the BLE USB dongle (CY5677) provided with the CY8CKIT-062-BLE kit is required.

Software Setup

Install the latest CySmart software on your computer to use the BLE USB dongle.

Operation

The bootloader can download a user application (App2), a stack application (App1) update, or both. Additionally, the bootloader can transfer control to a previously downloaded user application. Table 1 shows a list of the steps to be taken depending on the development phase of your project. When developing, it is faster to reprogram the device than to bootload an update.

Table 1. Operation Section Order

Development and Testing	Field Update
Develop applications: 1. Program the PSoC 6 MCU Device 1.1. Test applications Test Bootloader functionalities: 2. Configure CySmart 3. Switch to the Bootloader 4. Update the Stack and User Applications 5. Switch to the Bootloader 6. Switch to the User Application	Deploy an update: 1. Configure CySmart 2. Switch to the Bootloader 3. Update the Stack and User Applications 3.1. Updating the User Application 3.2. Updating the Stack and User Application

Program the PSoC 6 MCU Device

1. Plug the CY8CKIT-062-BLE kit board into your computer's USB port.
2. Build the projects in the following order: App0, App1, and App2. Any change to App0 or App1 requires subsequent projects to be rebuilt. For more information on how to build a project or program a device, see PSoC Creator Help.

Note: In some cases, during the build process you may be prompted to replace files from your project with files from the PDL. The PDL files are templates. Do not replace the customized files for the project. Click **Cancel**.

3. Set App2 as the active project and program it into PSoC 6 MCU. When App2 is built, App0 and App1 are merged into App2. This results in all apps being programmed. Similarly, programming App1 results in App0 and App1 being programmed.
4. Confirm that the kit LED blinks green, indicating that App2 is running.

Configure CySmart

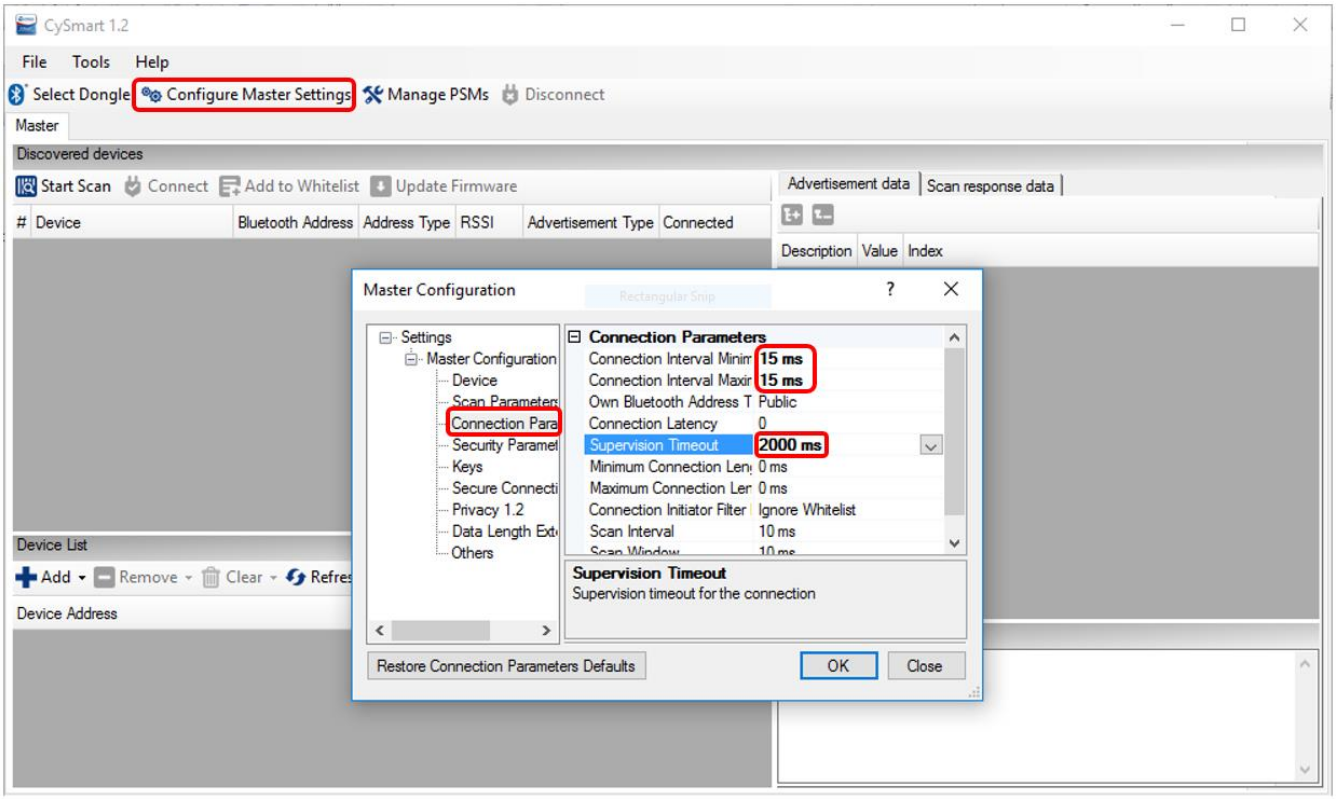
Default Bluetooth connection interval settings of the CySmart PC tool must be increased to allow enough time for flash memory write operations during the bootloading process. This section explains how to change the default settings.

1. Connect the BLE USB dongle (CY5677) provided with the CY8CKIT-062-BLE kit to your computer.
2. Run the CySmart tool on your computer and connect to the BLE USB dongle.
3. Click **Configure Master Settings**.
4. Go to **Connection Parameters**. Change the **Connection Interval Minimum**, **Connection Interval Maximum**, and **Supervision Timeout** to 15, 15, and 2000 ms, respectively, as Figure 1 shows. Click **OK**.

Note: Using a lower connection interval speeds up the application transmission at an increased risk of losing the connection.

These steps must be redone every time CySmart is reopened.

Figure 1. CySmart Connection Interval Settings



Switch to the Bootloader

To receive an update to either the user application or the stack application, the user application must transfer control to the stack application, which contains the bootloader. The following methods are provided to switch to the stack application.

Switching Using IAS

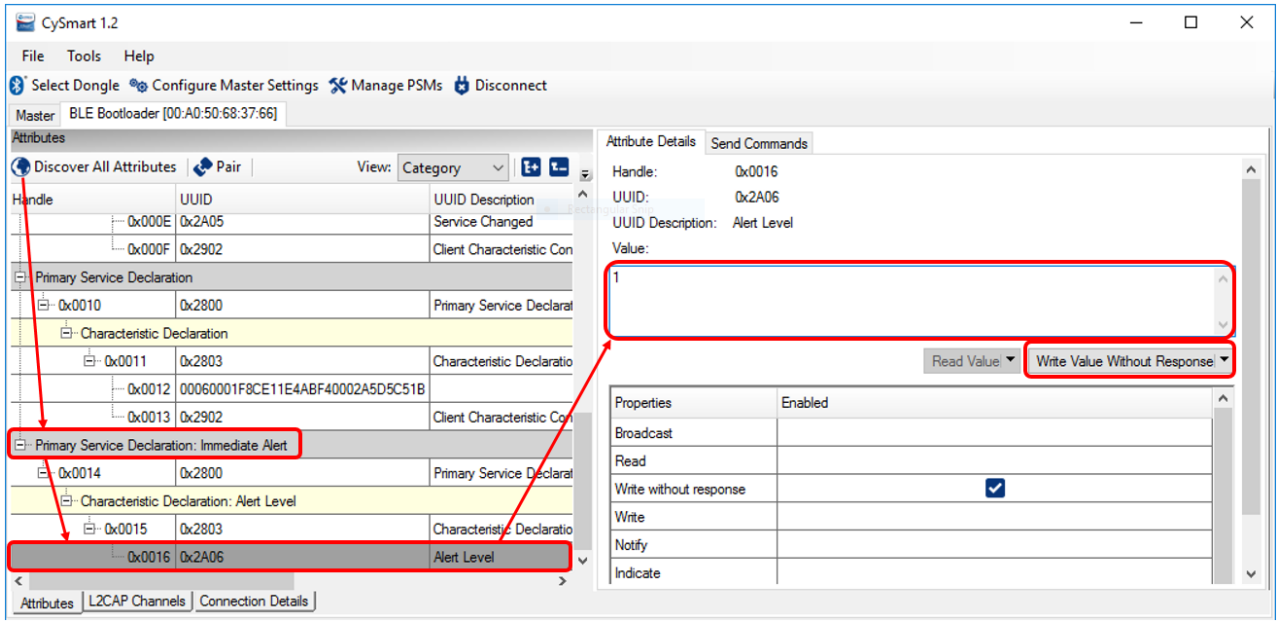
The user application switches to the stack application when it receives a nonzero Immediate Alert Service (IAS) alert level. If the CySmart PC tool is not already running, do the steps in Configure CySmart.

1. Press and release the kit user button (SW2) if the PSoC 6 MCU device is hibernating (indicated by a steady red LED).
2. In CySmart, click **Start Scan** to start scanning for the user application. In this example, the user application is the "BLE Keyboard" device. When the device is listed, select it.
3. Click **Connect** to connect to the device.
4. Click **Pair** to pair with the device. The Pair button may be hidden if the CySmart window size is small.

Note: If pairing fails, disconnect from the device and clear the device list in the CySmart tool. Go back to Step 2.

5. Click **No** when prompted to add the device to the resolving list.
6. Click **Discover All Attributes**.
7. Navigate to the **Immediate Alert** service at the bottom of the attributes list. Click **Alert Level**. Enter '1' in the **Value** text box and click **Write Value Without Response**, as Figure 2 shows.

Figure 2. Using the IAS to Switch Between Applications



8. Confirm that the LED blinks white once every two seconds, indicating that the stack application is running.

Switching Using the User Button and a Hardware Reset

App0 starts after a hardware reset. It supervises the user button and switches to the stack application if the button is pressed.

1. Press the reset and user buttons at the same time.
2. Release the reset button while holding the user button down.
3. Wait until the LED starts blinking white before releasing the user button. The stack application is now running.

Switch to the User Application

The stack application switches automatically to the user application after 300 seconds of Bluetooth inactivity. The following methods describe how to immediately transfer control to the user application.

Switching Using IAS

The stack application switches to the user application when it receives a nonzero Immediate Alert Service (IAS) alert level. If the CySmart PC tool is not already running, perform the steps described in Configure CySmart.

1. In CySmart, click **Start Scan** to start scanning for the bootloader device. Select the “Bootloader BLE” device when listed.
2. Click **Connect** to connect to the device.
3. Click **Pair** to pair with the device. The Pair button may be hidden if the window size is small.
4. Click **Discover All Attributes**.
5. Navigate to the **Immediate Alert** service at the bottom and click **Alert Level**. Enter ‘1’ in the **Value** textbox and click **Write Value Without Response**, as Figure 2 on page 3 shows.
6. Confirm that the LED is blinking green, indicating that the user application is running.

Switching to the User Application Using the User Button

1. Press and hold the kit user button for 0.5 seconds.
2. Wait until the LED starts blinking green. The user application is now running.

Update the Stack and User Applications

The stack and user applications may be updated to fix bugs or add new features. To update either application, the stack application, which contains the bootloader, must be running and the CySmart tool must be correctly configured. See Switch to the Bootloader and Configure CySmart.

Updating the User Application

In most cases, only the user application needs updating. The following steps describe how to update the user application.

1. In CySmart, click **Start Scan** to start scanning for the bootloader device. Select the “Bootloader BLE” device when listed.
2. Click **Stop Scan** to stop scanning.
3. Click **Update Firmware**.
4. Select the **Application only update** option and click **Next**.
5. Select the new user application firmware image file (*Bootloader_BLE_Upgradable_Stack_App2.cyacd2*) located in the path **Bootloader_BLE_Upgradable_Stack_App2.cydsn** > **CortexM4** > **[compiler name]** > **[Debug | Release]**. This file is generated when App2 is built.
6. Click **Update**.
7. Wait for the application firmware to be downloaded. While the firmware is downloaded, the white LED blinks twice every two seconds.
8. Confirm that the LED is blinking green, indicating that the updated user application is installed and running.

Updating the Stack and User Applications

A stack update may bring new features for the BLE Component. After a stack update, the user application must also be updated. The following steps describe how to update the stack and user applications in a single operation. Note that updating the stack application is a critical phase because it contains the bootloader.

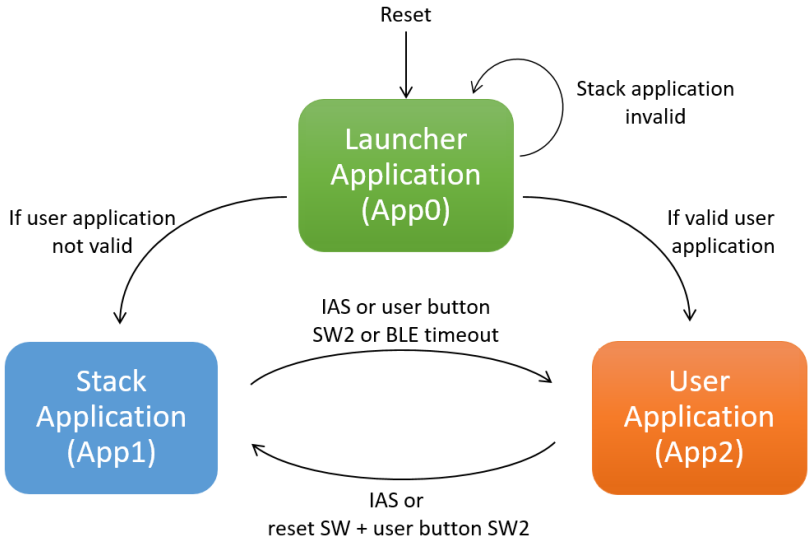
1. In CySmart, click **Start Scan** to start scanning for the bootloader device. Select the “Bootloader BLE” device when listed.
2. Click **Stop Scan** to stop scanning.
3. Click **Update Firmware**.
4. Select the **Application and Stack update** option and click **Next**.
5. Select the new stack application firmware image file (*Bootloader_BLE_Upgradable_Stack_App1.cyacd2*) located in the path **Bootloader_BLE_Upgradable_Stack_App1.cydsn** > **CortexM4** > **[compiler name]** > **[Debug | Release]**. This file is generated when App1 is built.
6. Select the new user application firmware image file (*Bootloader_BLE_Upgradable_Stack_App2.cyacd2*) located in the path **Bootloader_BLE_Upgradable_Stack_App2.cydsn** > **CortexM4** > **[compiler name]** > **[Debug | Release]**. This file is generated when App2 is built.
7. Click **Update**.
8. Wait for the stack application to be downloaded. While it is being downloaded, the white LED flashes twice every two seconds.
9. Confirm that the LED is ON in a purple color, indicating that App0 is copying the stack application from temporary storage to its final location. This operation may take several seconds.
10. Confirm that the LED flashes white twice every two seconds indicating that the updated stack application is installed, and the bootloader is downloading the user application.
11. Confirm that the LED is blinking green, indicating that the updated user application is installed and running.

Application Switching and LED Status Overview

Figure 3 shows an overview of how applications transfer control to one another.

The launcher application (App0) is located at the start of the user flash. Its purpose is to copy a stack update from temporary storage to a final location. The stack application (App1) contains the bootloader. For more information, see Design and Implementation.

Figure 3. Application Switching Overview



An overview of the LED status for each application is shown in Table 2.

Table 2. LED Status Overview

Application	Color	State	Description
Launcher Application (App0)	Purple	Steady	Copying a stack application update from temporary storage to its final location.
	Yellow	Steady	Stack application is invalid. Device must be reprogrammed.
Stack Application (App1)	White	Blinks once every 2 seconds	Bootloader is advertising.
	White	Blinks twice every 2 seconds	Bootloading is taking place; an application is being received.
	None	OFF	Bootloader is connected but not receiving an application.
	Red	Steady	Hibernating.
User Application (App2)	Green	Blinking	BLE Keyboard is advertising.
	None	OFF	BLE Keyboard is connected.
	Blue	Steady	Caps Lock is ON.
	Red	Steady	Hibernating.

Note: If the specified V_{DDD} in the PSoC Creator project **Design Wide Resources** > **System** window is less than 2.7 V, only the red LED is used in the stack and user applications. Status indicators in Table 2 that use the blue and green LEDs are not shown.

Note: To enable the blue LED in App2, connect to PSoC 6 MCU via Bluetooth without using the BLE dongle. Press the **Caps Lock** key on your keyboard or press the user button on the kit.

Design and Implementation

This example has three applications: “App0”, “App1”, and “App2”. Each application is a separate PSoC Creator project. The projects have the following features:

- App0 is the launcher application. It copies a stack application update from temporary storage to its final location. It also validates and starts either the stack or the user application. App0 cannot be updated by OTA bootloading.
- App1 is the stack application. It contains the bootloader and the BLE stack. The bootloader can download an update to the stack or user application.
- Updating the stack application requires placing the update in temporary storage and switching to the launcher for copying, because the stack cannot overwrite itself.
- App2 is the user application. It includes a BLE Component, but that Component is configured to contain only BLE profiles without the supporting stack code. Required stack code and variables are shared from the stack application, considerably reducing the size of the user application.
- Modifying the stack application requires recompiling the user application because the location of the stack application functions and variables may change.
- A RAM region is reserved for stack variables; the user application cannot use this.
- The stack and the user applications transfer control between each other using either a BLE event or by pressing the kit user and reset buttons. See Figure 3 on page 6.
- App2 demonstrates several Bluetooth services. It is based on CE215121, BLE HID Keyboard.

Comparison with the Standard BLE Bootloader

The Upgradable Stack Bootloader demonstrated in this code example has the following advantages when compared to the standard BLE Bootloader (see CE216767):

- Lower total flash memory usage, because the BLE stack is shared between applications.
- Faster update when updating only the user application, because of the reduced size.

It also has the following disadvantages:

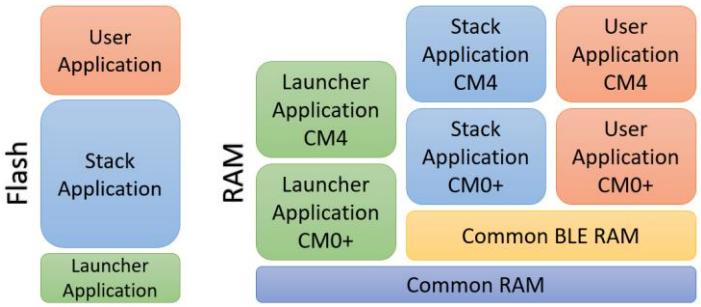
- The stack and user application projects both include an instance of the BLE Component. The Component **CPU core** setting – Single core (Complete Component on CM0+), Single core (Complete Component on CM4), or Dual Core (Controller on CM0+, Host and Profiles on CM4) – must be the same in both instances.
- Updating the stack and the user application takes longer than the standard BLE Bootloader update.
- The user application RAM is slightly reduced due to the memory reserved for the stack.
- Updating the stack requires the user application to also be updated, because the location of the stack functions and variables may change.
- Updating the stack is a critical phase because it contains the bootloader. Receiving a non-functional stack application update renders the device unusable.

Common Files

The three projects in this code example contain common linker configuration files that control how the two CPUs in each of the applications use flash and RAM memory. They are heavily customized versions of the Bootloader Source Development Kit (SDK) linker files. For more information on customizing PSoC Creator projects for the Bootloader (SDK), see the PSoC 6 MCU Bootloader SDK Guide.

Figure 4 shows an overview of how the RAM and flash memory are distributed. The actual distribution depends on the compiler used and on the CPU core configuration of the BLE Component.

Figure 4. RAM and Flash Memory Overview



The common linker configuration files support the single-core BLE Component configuration “Complete Component on CM4” as the default in this code example. The files include configurations for each CPU core configuration of the BLE Component. For more information on how to change the CPU core configuration, see Appendix A.

Common linker configuration files are provided for GCC, MDK, and IAR compilers, as Table 3 shows:

Table 3. Common Linker Configuration Files

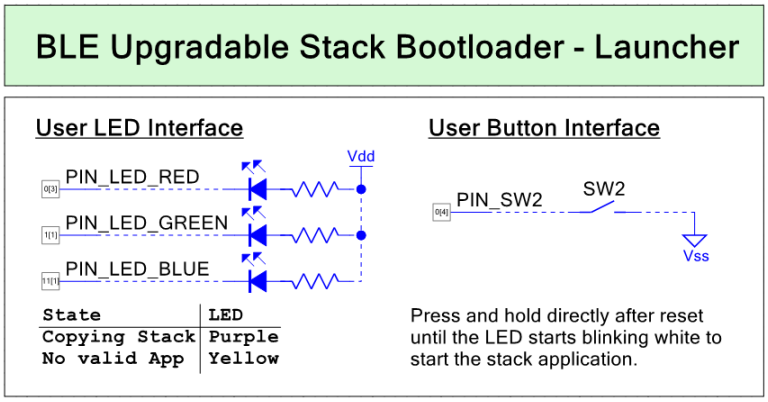
File	Description
bootload_common.ld	GCC linker configuration file. It is included in the *bootload_cm0p.ld* and *bootload_cm4.ld* linker script files.
bootload_mdk_common.h	MDK linker configuration file. MDK linker script files cannot include other linker script files so the necessary symbols are defined in a C header file.
bootload_common.icf	IAR linker configuration file. It is included in the *bootload_cm0p.icf* and *bootload_cm4.icf* linker script files.

Launcher Application Firmware Design

The launcher application (App0) can start the stack application (App1) or the user application (App2). Additionally, it copies a stack application update from a temporary location to its final location if a flag is set. If the launcher is the only valid application on PSoC 6 MCU, the device must be reprogrammed.

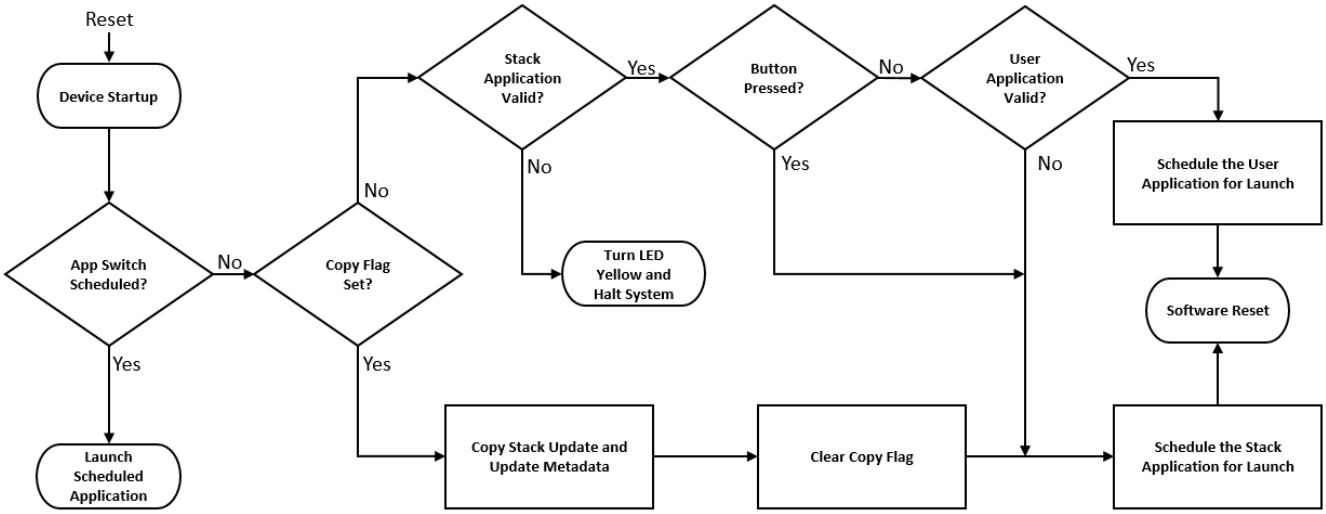
Figure 5 shows the launcher’s PSoC Creator project schematic.

Figure 5. Launcher Application (App0) Project Schematic



The CM0+ CPU executes the launcher application. The CM4 CPU remains in Deep Sleep. Figure 6 shows the firmware flow of the launcher application.

Figure 6. Firmware Flow of the Launcher Application (App0)



Memory Layout

Figure 7 shows the layout of the RAM and flash memory for the launcher application. The memory layout of the launcher application is independent of the compiler used and the BLE Component configuration. The metadata row is defined in the *bootload_user.c* file of the launcher and is populated with the data defined in the common linker configuration files.

Figure 7. Memory Layout of the Launcher Application (App0)

RAM

Address	App0 (Launcher)	Size
0x0804 7FFF	Unused	272 KB
0x0800 4000		
0x0800 2000	App0 CM4 Application Data	8 KB
0x0800 0100	App0 CM0+ Application Data	7.75 KB
0x0800 0000	Common RAM	256 B

Flash

Description	Size
Metadata flash row	512 B
0x100F FA00	
Copy flag flash row	512 B
0x100F F800	
Empty	1002 KB
0x1000 5000	
App0, CM4	8 KB
0x1000 3000	
App0, CM0+	12 KB
0x1000 0000	

Design Files

Table 4 lists the files used in the launcher application and describes their functionality.

Table 4. Design Firmware Files of the Launcher Application (App0)

File	Description
main_cm4.c, main_cm0p.c	Contains the `main()` function for each CPU. The launcher functionality is implemented in the main function of the CM0+ CPU.
cy_bootload.c / .h	Bootloader software development kit (SDK) files.
bootload_user.h	Contains user-editable `#define` statements that control the operation and enable features in the SDK.
bootload_user.c	Defines the metadata initial values. Contains two functions (`Cy_Bootload_ReadData` and `Cy_Bootload_WriteData`) that control access to the internal memory for validating and copying applications.
bootload_cm0p.ld, bootload_cm4.ld	Custom GCC linker scripts. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.scat, bootload_cm4.scat	MDK scatter files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.icf, bootload_cm4.icf	IAR linker configuration files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
post_build_core1.bat	Copies the resulting ELF file into the project's root folder for merging with App1 and App2.

Stack Application Firmware Design

The stack application (App1) consists of the BLE stack and a BLE bootloader. The BLE stack code is shared with the user application via symbol extraction from the output file of the linker using a post-build command.

The *BLE_Symbols.txt* file contains a list of all the BLE-related symbols to extract. It also includes symbols to enable reinitializing the stack's variables from the user application. An assembler file, which defines the BLE symbols and their addresses, is generated for each CPU and placed in the user application project.

The bootloader in App1 can download an update for just the user application or for the stack and user applications. Figure 8 shows the flow of the application update process. To customize the bootload operation and enable Bootloader SDK features, update the `#define` statements as needed in the *bootload_user.h* file. For more information on the SDK, see the Bootloader SDK Guide.

When downloading an update for the stack project, the bootloader receives new metadata containing the final location of the update. The update is placed in a temporary location in flash, the address of which is stored in another metadata entry. The launcher reads both metadata entries and copies the stack from the temporary location to the final location. The metadata of the stack application is not updated until the launcher has successfully copied it from the temporary location.

The default temporary location for the stack update is set so as to overwrite the user application. To change the temporary location, modify the `temporaryLocation` variable located in the `Cy_Bootload_WriteData()` function of the *bootload_user.c* file.

Figure 8. Stack and User Application Update Process

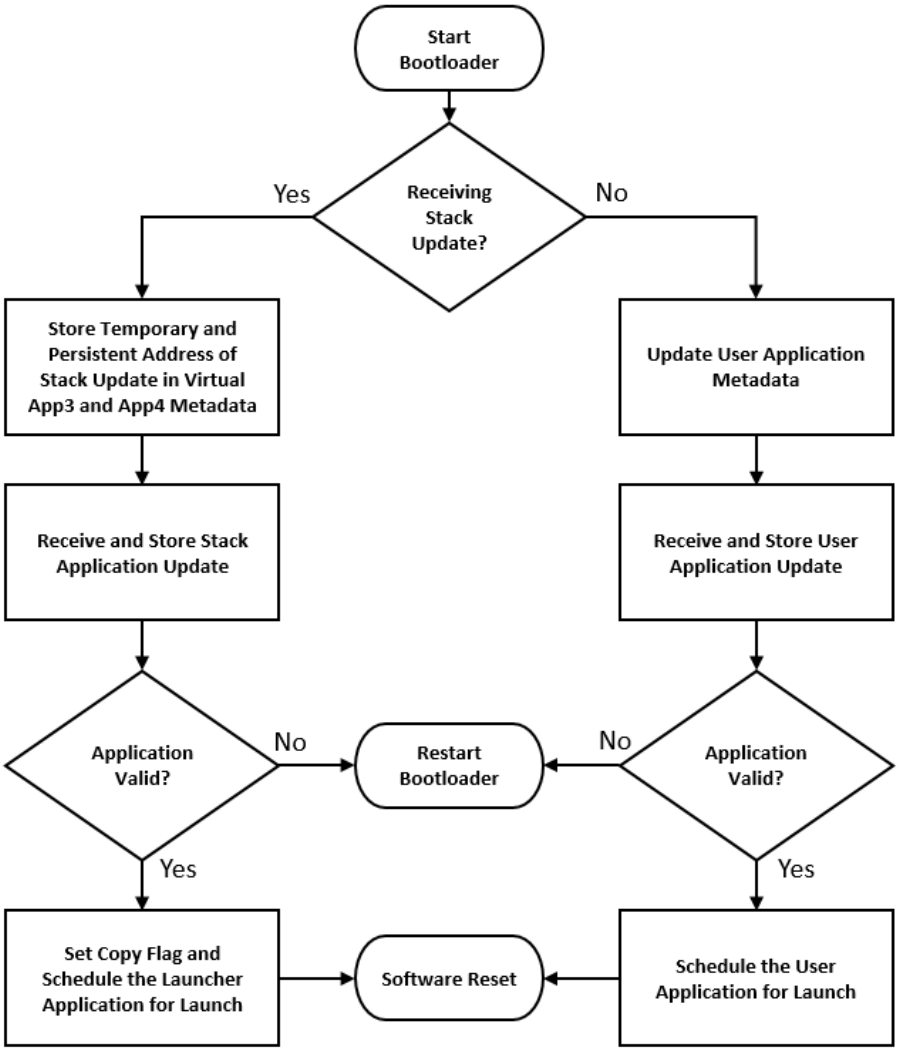
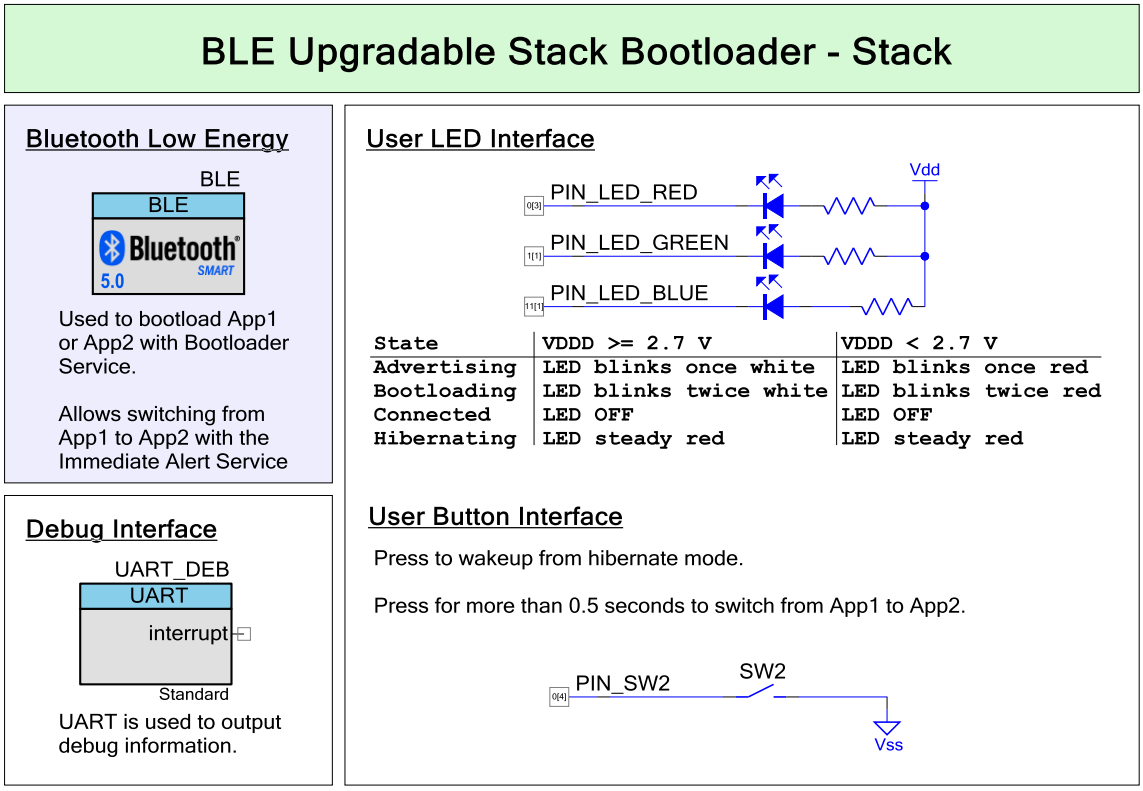


Figure 9 shows the stack application's project schematic.

Figure 9. Stack Application (App1) PSoC Creator Project Schematic



The RAM is partitioned to reduce the reserved space for the BLE stack. The reserved memory size depends on the compiler used and the CPU core configuration of the BLE Component. Common BLE RAM sections are defined to store the variables required by the stack that must be updated with the application's settings (Clock, IPC, etc.).

Table 5 shows the tasks executed by each CPU depending on the configuration of the BLE Component.

Table 5. App1 CPU Tasks per BLE Configuration

CPU Core Configuration of BLE Component	Cortex-M0+	Cortex-M4
Complete Component on CM0+	Bootloads a stack update or a user application update. Supervises the user button and IAS value for application switching.	Does nothing.
Complete Component on CM4	Does nothing.	Bootloads a stack update or a user application update. Supervises the user button and IAS value for application switching.
Controller on CM0+, Host and Profiles on CM4	Services the BLE subsystem (BLESS) controller interrupt.	

Stack Code Sharing Implementation

GCC is the default compiler used by PSoC Creator. The following section explains the implementation of the code sharing functionality for GCC. For MDK and IAR compilers, see Appendix B: Implementing Stack Code Sharing with MDK and IAR.

The CPU-specific linker scripts of the Bootloader SDK (*bootload_cm0p.ld* and *bootload_cm4.ld*) were modified to do the following:

- Explicitly keep the BLE stack functions from being removed.
- Place the required RAM data of the BLE stack in specific sections to be reinitialized by the user application.
- Expand the copy and zero initialization tables to initialize stack-related variables.
- Provide memory configurations for the CPU mode of each BLE Component.

Unused functions and other elements are removed by the linker as a default; however these elements may be used by the user application and must be explicitly retained, as shown partially in Figure 10.

Note: In this example all BLE functions are retained. Functions may be removed as needed to reduce flash usage.

Figure 10. App1 GCC Linker Script Excerpt Showing Retention of Unused BLE Stack Elements

```
144         /* Make sure that the BLE stack is not optimized out by the linker */
145         /* General BLE Stack functions */
146         KEEP(*(*Cy_BLE_GetStackLibraryVersion*))
147         KEEP(*(*Cy_BLE_StackInit*))
148         KEEP(*(*Cy_BLE_StackShutdown*))
149         KEEP(*(*Cy_BLE_StackSoftReset*))
150         KEEP(*(*Cy_BLE_ProcessEvents*))
```

The BLE stack requires common variables such as clock settings, values of which depend on the running application and that are updated by the firmware. These variables must be overwritten by the user application and therefore are placed in specific sections and order in RAM. Other BLE stack-related variables and RAM functions are placed directly after them.

The common BLE RAM sections are defined only for the CPU in BLE Component configuration. Figure 11 shows the definitions of the common BLE RAM sections (*.data* and *.bss*) for the CM4 CPU. Highlighted entries contain variables that are overwritten when executing the user application.

Figure 11. App1 Common BLE RAM Definition for GCC

```
398     __ble_core1_data_at_flash = __etext;
399     .cy_boot_ble_data : AT (__etext)
400     {
401         __ble_core1_data_start__ = .;
402         *system_psoc63_cm4.o(.data*)
403         *cy_ipc_sema.o(.data*)
404         *cy_ipc_config.o(.data*)
405         *cy_ipc_pipe.o(.data*)
406         *cy_syspm.o(.data*)
407         *cy_flash.o(.data*)
408         *cy_ble.o(.data*)
409         *cy_ble_hal*(.data*)
410         *cy_ble_stack_gcc_host_ipc_cm4.a:*(.data*)
411         *cy_ble_stack_gcc_radio_max_cm4.a:*(.data*)
412         *cy_ble_stack_gcc_soc_cm4.a:*(.data*)
413         . = ALIGN(4);
414         KEEP(*(.cy_ramfunc*))
415         __ble_core1_data_end__ = .;
416     } > ram_ble_core1
```

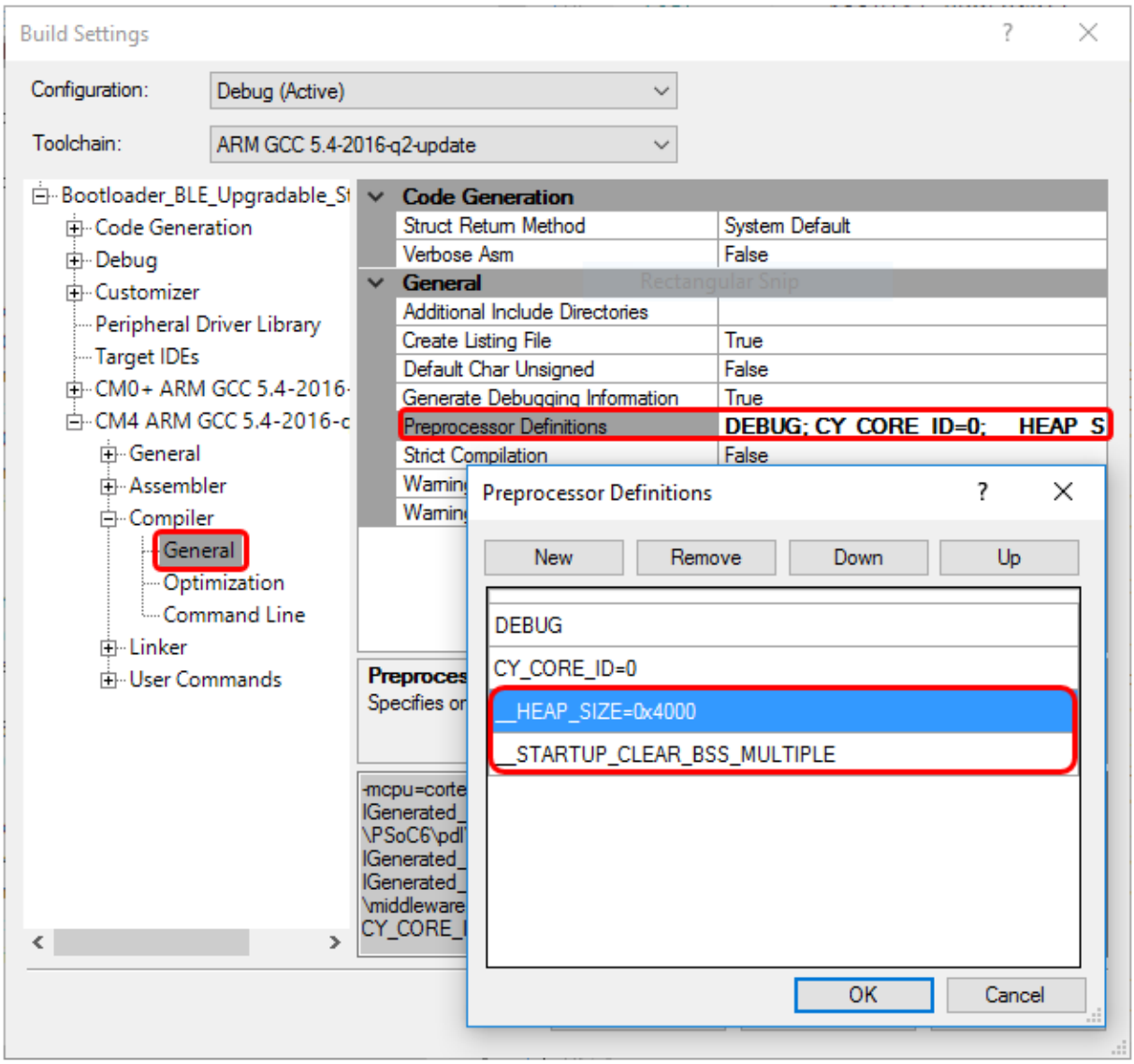
Data which is overwritten

```
418     .cy_boot_ble_bss (NOLOAD):
419     {
420         __ble_core1_bss_start__ = .;
421         *system_psoc63_cm4.o(.bss*)
422         *cy_ipc_sema.o(.bss*)
423         *cy_ipc_config.o(.bss*)
424         *cy_ipc_pipe.o(.bss*)
425         *cy_syspm.o(.bss*)
426         *cy_flash.o(.bss*)
427         *cy_ble.o(.bss*)
428         *cy_ble_hal*(.bss*)
429         *cy_ble_stack_gcc_host_ipc_cm4.a:*(.bss*)
430         *cy_ble_stack_gcc_radio_max_cm4.a:*(.bss*)
431         *cy_ble_stack_gcc_soc_cm4.a:*(.bss*)
432         __ble_core1_bss_end__ = .;
433     } > ram_ble_core1
```

The stack application must provide a way for the user application to reinitialize its BLE stack variables. This is realized by expanding the copy and zero initialization tables with symbols to initialize the BLE stack-related data. Afterwards, these symbols are exported to the user application and placed in its copy and zero initialization tables.

By default, the zero initialization table is disabled; it must be enabled. Additionally, The BLE Component requires an increased amount of heap memory when configured with the “Stack and Profile” option. Figure 12 shows the project configuration to increase the heap size of the stack application and enable the zero initialization table.

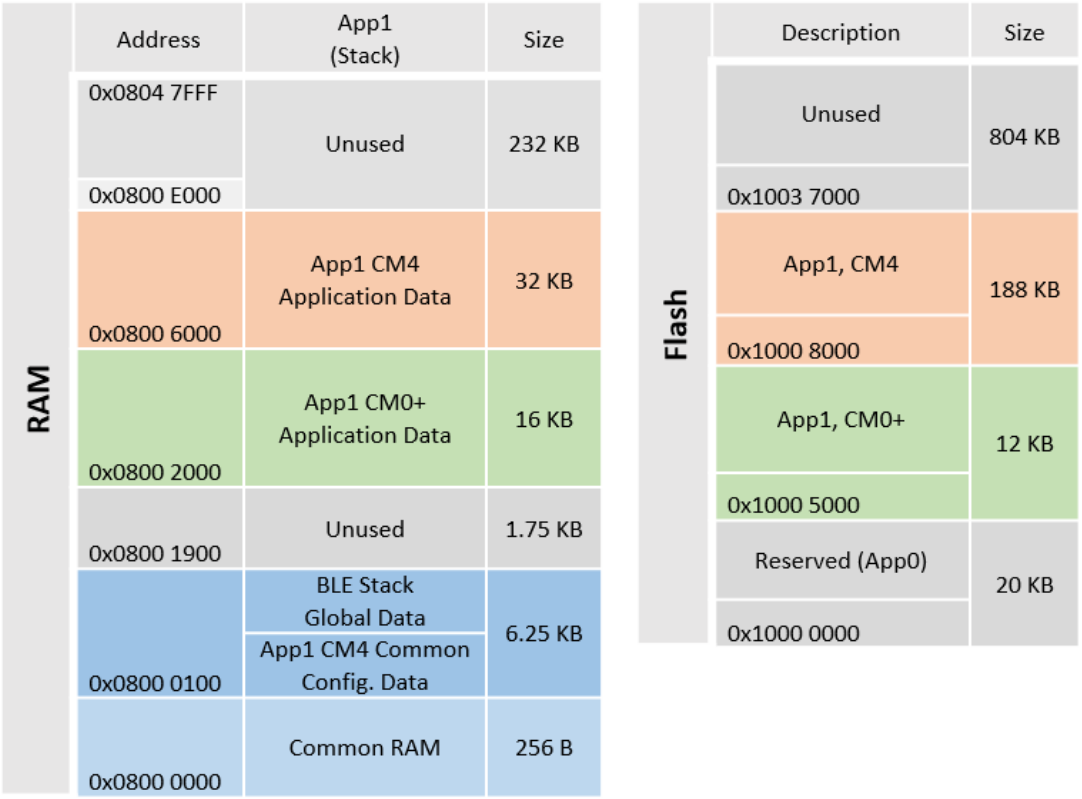
Figure 12. App1 Heap Size and Zero Table Definitions



Memory Layout

The memory layout of the stack application depends on the compiler used and the CPU core configuration of the BLE Component. Figure 13 shows the layout of the RAM and flash memory using the GCC compiler and the BLE Component running on the CM4 CPU. The BLE stack reserves 6.25 KB of RAM with this configuration.

Figure 13. Memory Layout of the Stack Application (App1)



Design Files

Table 6 lists the files used in the stack application and describes their functionality.

Table 6. Design Firmware Files of the Stack Application (App1)

File	Description
main_cm4.c,main_cm0p.c	Contains the `main()` function for each CPU and the required functions to reinitialize the stack's variables. Calls the bootloader main function or goes into the CPU Deep Sleep mode, depending on the BLE Component configuration.
bootloader.c	Contains the bootloader main and supporting functions.
ias.c / .h	Immediate Alert Service (IAS) files. Used to implement IAS for communication between BLE GAP Central and Peripheral. When the stack application receives a non-zero value with the IAS, it switches to the user application.
debug.c / .h	UART `printf` implementation and LED status notification.
cy_bootload.c / .h	Bootloader software development kit (SDK) files.
bootload_user.h	Contains user-editable `#define` statements that control the operation and enable features in the SDK.
bootload_user.c	Contains user functions required by the SDK: ▪ Five functions that control communications with the bootloader host. These are also called *transport functions.* ▪ Two functions, `Cy_Bootload_ReadData()` and `Cy_Bootload_WriteData()`, that control access to the internal or external memory. ▪ Support functions for `Cy_Bootload_ReadData()` and `Cy_Bootload_WriteData()`, to perform checks on data prior to reading or writing.
transport_ble.c / .h	Contains bootloader transport functions for the BLE Component. These functions are typically called by the transport functions in *bootload_user.c.*
bootload_cm0p.ld, bootload_cm4.ld	Custom GCC linker scripts. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.scat, bootload_cm4.scat	MDK scatter files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.icf, bootload_cm4.icf	IAR linker configuration files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
post_build_core0.bat	Batch file to share code from the CM0+ CPU with the user application.
post_build_core1.bat	Batch file to share code from the CM4 CPU with the user application, create the bootloadable file, and merge App0 with App1 into a single hex file.

User Application Firmware Design

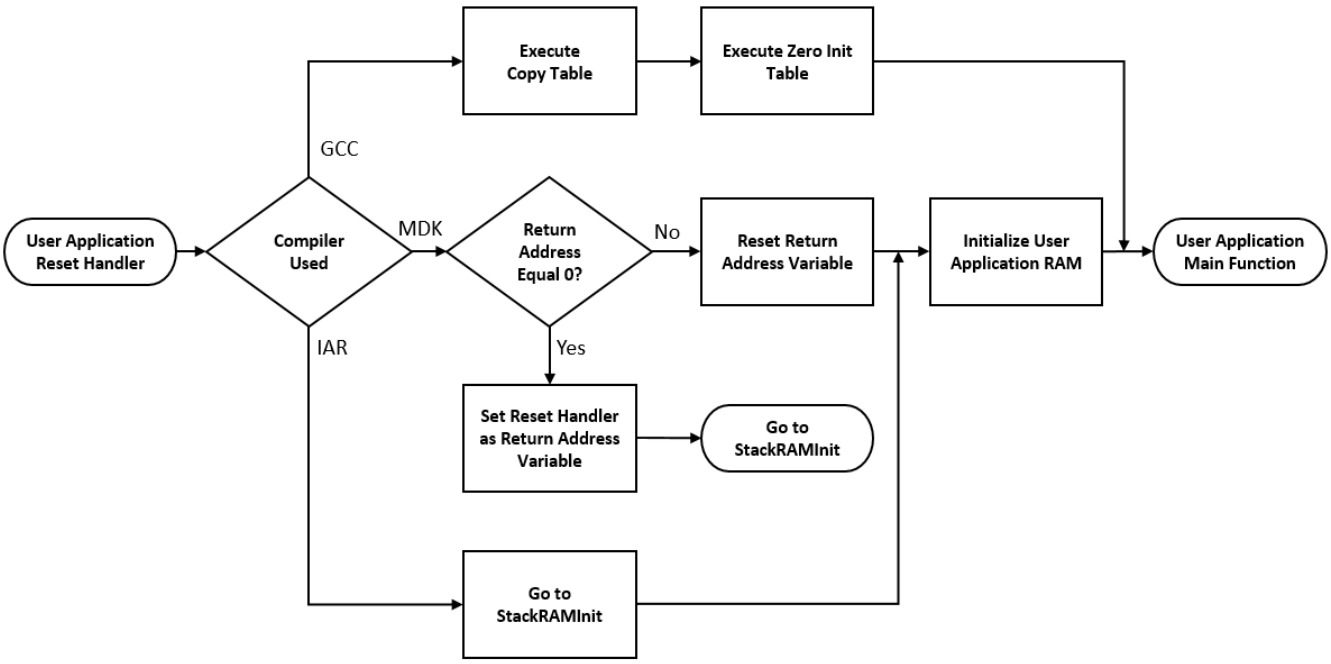
The user application (App2) demonstrates a BLE HID Keyboard. The application simulates keyboard presses and battery level. The BLE Component is configured as Host and Profiles only. Stack functions and variables are imported from the stack application into the *BLE _core0_shared.s* and *BLE_core1_shared.s* files, for the CM0+ and CM4 CPUs respectively.

Before using the BLE Component, the stack variables must be reinitialized. The method for initialization depends on the compiler used:

- GCC: The startup copy and zero init table were expanded to directly initialize stack variables at startup. No explicit function is called to reinitialize stack variables.
- MDK: The `StackRAMInit()` function is called in the reset handler with a return address as a parameter. This function is in the stack application and calls the `__main` initialization routine from the stack application. Afterwards, execution returns to the reset handler.
- IAR: The `StackRAMInit()` function is called in the reset handler. This function is in the stack application and calls the `__iar_data_init3()` initialization routine from the stack application.

Figure 14 shows a flowchart of the stack reinitialization routine for all three compilers.

Figure 14. RAM Reinitialization Flow



The BLE interrupt vector must be explicitly initialized by the CPU running the BLE controller before starting the BLE Component. The following function achieves this:

```
Cy_SysInt_SetVector(BLE_bless_isr_cfg.intrSrc, &Cy_BLE_BlessInterrupt);
```

Table 7 shows the tasks executed by each CPU depending on the configuration of the BLE Component.

Table 7. App2 CPU Tasks per BLE Configuration

CPU Core Configuration of BLE Component	Cortex-M0+	Cortex-M4
Complete Component on CM0+	Demonstrates BLE services, as documented in *CE215121 – BLE HID Keyboard*. Switches to the stack application if the IAS value is greater than zero.	Does nothing.
Complete Component on CM4	Does nothing.	Demonstrates BLE services, as documented in *CE215121 – BLE HID Keyboard*. Switches to the stack application if the IAS value is greater than zero.
Controller on CM0+, Host and Profiles on CM4	Services the BLESS controller interrupt.	

Stack Code Sharing Implementation

GCC is the default compiler used by PSoC Creator. The following section explains the implementation of the code sharing functionality for GCC. For MDK and IAR compilers, see Appendix B: Implementing Stack Code Sharing with MDK and IAR.

The Bootloader SDK CPU-specific linker scripts (*bootload_cm0p.ld* and *bootload_cm4.ld*) were modified to do the following:

- Place the common RAM data in specific sections to overwrite the stack application's variables.
- Expand the copy and zero init tables to reinitialize stack-related variables.
- Provide memory configurations for the CPU mode configuration of each BLE Component.

The BLE stack requires common variables, such as clock settings, values of which depend on the running application and that are updated by the firmware. These variables must be overwritten by the user application and therefore placed in a specific section and order in the RAM.

The placing of the common variables in the linker script is shown in Figure 15. The definition of the copy and zero init tables in the linker script is shown in Figure 16 on page 19. Symbols that were imported from the stack application are highlighted.

Figure 15. Placement of Common Variables

```
233     .cy_boot_ble_data : AT (__etext)
234     {
235         __common_config_start__ = .;
236         *system_psoc63_cm4.o(.data*)
237         *cy_ipc_sema.o(.data*)
238         *cy_ipc_config.o(.data*)
239         *cy_ipc_pipe.o(.data*)
240         __common_config_end__ = .;
241     } > ram_ble_core1
242
243     .cy_boot_ble_bss ABSOLUTE(__ble_core1_bss_start__) (NOLOAD):
244     {
245         *system_psoc63_cm4.o(.bss*)
246         *cy_ipc_sema.o(.bss*)
247         *cy_ipc_config.o(.bss*)
248         *cy_ipc_pipe.o(.bss*)
249     } > ram_ble_core1
```

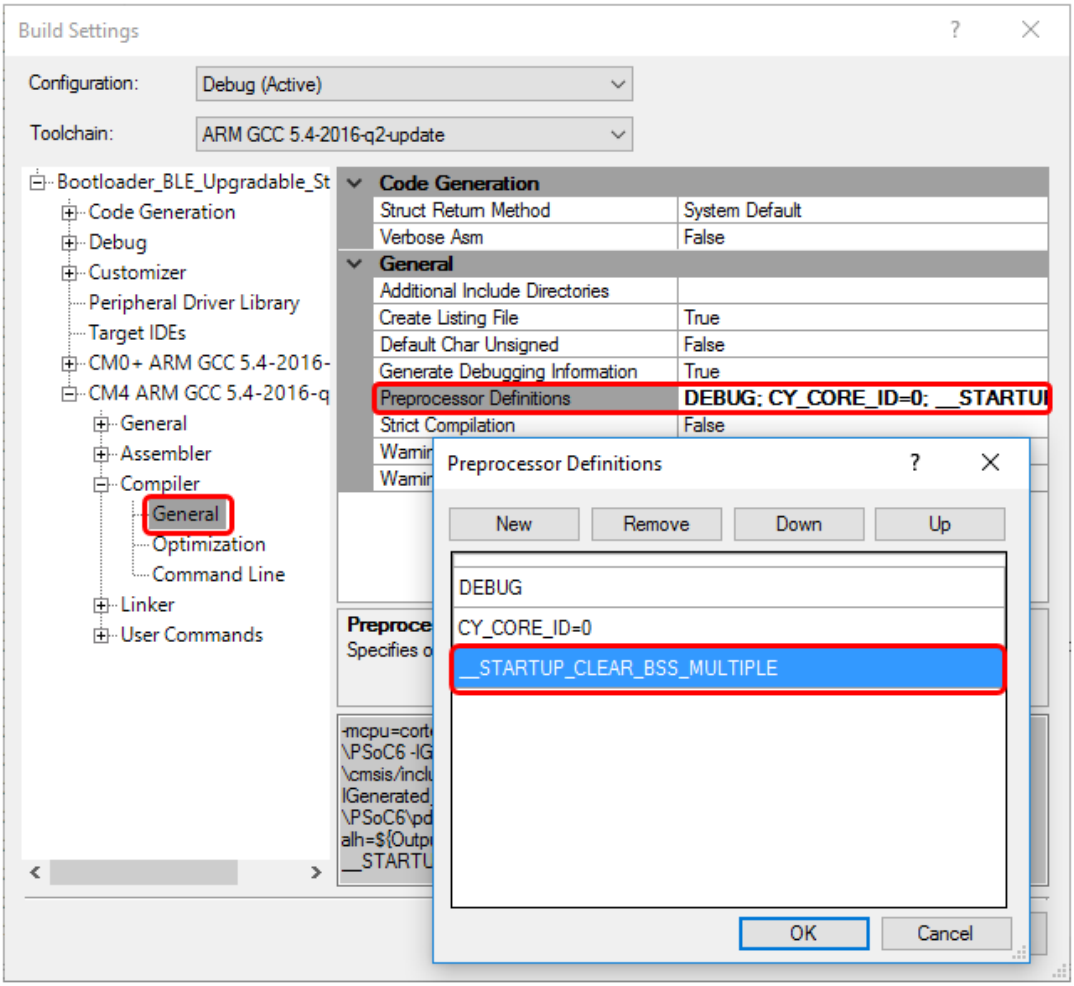
Figure 16. Definition of Copy and Zero Init Tables

```
185     .copy.table :
186     {
187         . = ALIGN(4);
188         __copy_table_start__ = .;
189
190         LONG (__Vectors)
191         LONG (__ram_vectors_start__)
192         LONG (__Vectors_End - __Vectors)
193
194         LONG (__ble_core1_data_at_flash)
195         LONG (__ble_core1_data_start__)
196         LONG (__ble_core1_data_end__ - __ble_core1_data_start__)
197
198         LONG (__etext)
199         LONG (__common_config_start__)
200         LONG (__common_config_end__ - __common_config_start__)
201
202         LONG (__etext + (__common_config_end__ - __common_config_start__))
203         LONG (__data_start__)
204         LONG (__data_end__ - __data_start__)
205
206         __copy_table_end__ = .;
207     } > flash
```

```
210     .zero.table :
211     {
212         . = ALIGN(4);
213         __zero_table_start__ = .;
214
215         LONG (__bss_start__)
216         LONG (__bss_end__ - __bss_start__)
217
218         LONG (__ble_core1_bss_start__)
219         LONG (__ble_core1_bss_end__ - __ble_core1_bss_start__)
220
221         __zero_table_end__ = .;
222     } > flash
```

The zero init table is disabled by default and must be enabled. Figure 17 shows the project configuration to enable the zero init table.

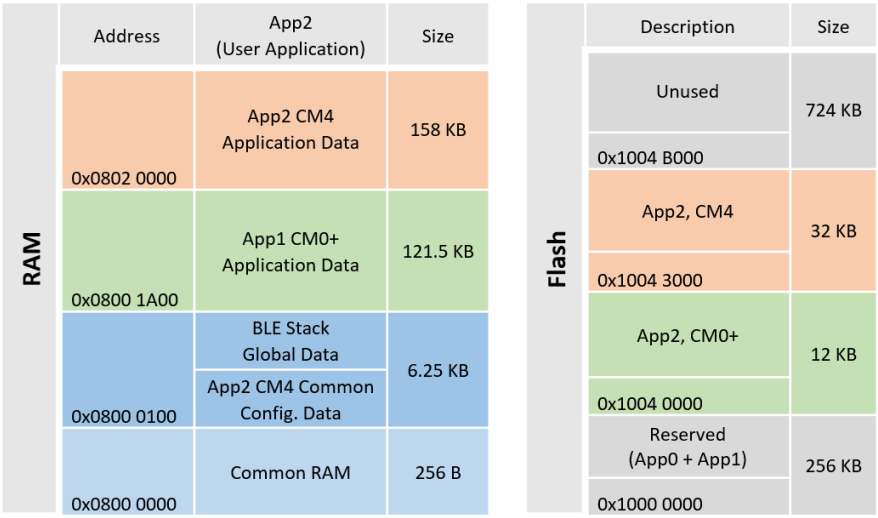
Figure 17. Zero Table Definition



Memory Layout

The memory layout of the stack application depends on the compiler used and the CPU core configuration of the BLE Component. Figure 18 shows the layout of the RAM and flash memory using the GCC compiler and the BLE Component running on the CM4 CPU. The BLE stack reserves 6.25 KB of RAM on this configuration. The rest of the RAM is divided between both CPUs.

Figure 18. Memory Layout for the User Application



Design Files

Table 8 lists the files used in the user application (App2) and describes their functionality. For more information on the implementation of the BLE HID Keyboard, see CE215121 – BLE HID Keyboard.

Table 8. App2 Design Firmware Files

File	Description
main_cm4.c, main_cm0p.c	Contains the `main()` function for each CPU and the required external functions to reinitialize the stack variables. Calls the host main function or goes into the CPU Deep Sleep mode depending on the CPU core configuration of the BLE Component.
ias.c / .h	Immediate Alert Service (IAS) files. Used to implement IAS for communication between BLE GAP Central and Peripheral. When the device receives a non-zero value with the IAS, it switches applications.
BLE_core0_shared.s, BLE_core1_shared.s	Assembly files that contain definitions from variables and functions shared from the stack application.
bas.c / .h, common.h, hids.c /.h, scps.c / .h user_interace.c / .h bond.c, debug.c, host_main.c	BLE Keyboard implementation files.
cy_bootload.c / .h	Bootloader SDK files.
bootload_user.h	Contains user-editable `#define` statements that control the operation and enable features in the SDK.
bootload_cm0p.ld, bootload_cm4.ld	Custom GCC linker scripts. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.scat, bootload_cm4.scat	MDK scatter files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
bootload_cm0p.icf, bootload_cm4.icf	IAR linker configuration files. These files place the code and data sections for each CPU as well as the bootloader and other regions.
post_build_core1.bat	Batch file to create the downloadable application and to merge App0, App1, and App2 into a single hex file.

Design Considerations

Software Reset

When transferring control from one application to another, the recommended method is through a device software reset. This enables each application to initialize device hardware blocks and signal routing from a known state.

You can freeze the state of I/O pins so that they are maintained through a software reset. Defined portions of SRAM are also maintained through a software reset. For more information, see the PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual.

Components and Settings

This section describes the PSoC Creator Components used by the launcher and stack applications, how they are used in the design, and the non-default settings required so they function as intended. Additionally, BLE Component settings for the user application are shown to enable code sharing.

For information on the hardware resources used by a Component, see the Component datasheet.

Launcher Application (App0)

Table 9 lists the PSoC Creator Components used in the launcher application.

Table 9. PSoC Creator Components for the Launcher Application

Component	Instance Name	Purpose	Non-default Settings
Pin	PIN_LED_RED	LED Status notification.	HW Connection: Unchecked External Terminal: Checked Initial drive state: High (1) Max Frequency: 1 MHz SW2 Pin Initial Drive Mode: Resistive Pull-Up
	PIN_LED_GREEN		
	PIN_LED_BLUE		
	PIN_SW2		

Stack Application (App1)

Table 10 lists the PSoC Creator Components used in the stack application.

Table 10. PSoC Creator Components for the Stack Application

Component	Instance Name	Purpose	Non-default Settings
Bluetooth Low Energy	BLE	Provides communication between the PSoC 6 MCU device and the Bluetooth Host for bootloading and app switching.	See Stack Application (App1) BLE Component Configuration.
UART	UART_DEB	Outputs Bluetooth-related debug information.	Interrupt mode external.
Pin	PIN_LED_RED	LED Status notification.	HW Connection: Unchecked External Terminal: Checked LED Pins Drive Mode: High Impedance Digital SW2 Pin Initial Drive Mode: Resistive Pull-Up
	PIN_LED_GREEN		
	PIN_LED_BLUE		
	PIN_SW2		

User Application (App2)

Table 11 shows the required PSoC 6 BLE Component configuration to enable code sharing. For more information on the PSoC Creator Components used in the user application, see CE215121 – BLE HID Keyboard.

Table 11. PSoC Creator BLE Component Required Setting

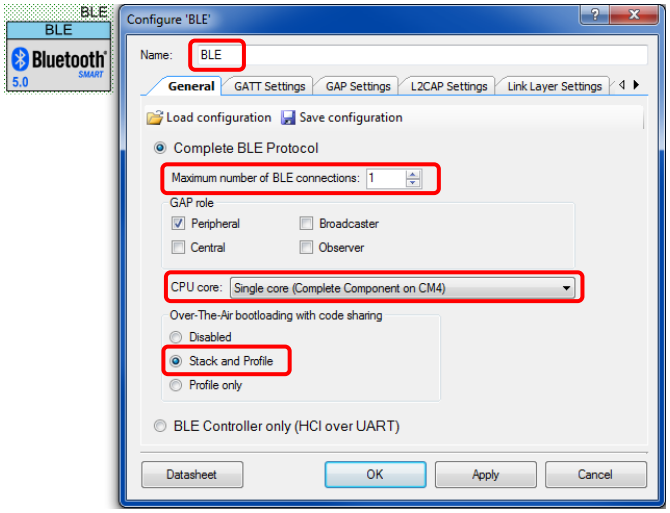
Component	Instance Name	Purpose	Non-default Settings
Bluetooth Low Energy	BLE	Provides BLE communication support.	OTA bootloading with code sharing: Profile Only

Stack Application (App1) BLE Component Configuration

General Tab (see Figure 19)**:**

- Maximum number of BLE connections: 1
- CPU core: Single core (Complete Component on CM4)
- Over-The-Air bootloading with code sharing: Stack and Profile

Figure 19. BLE Component, General Tab Configuration

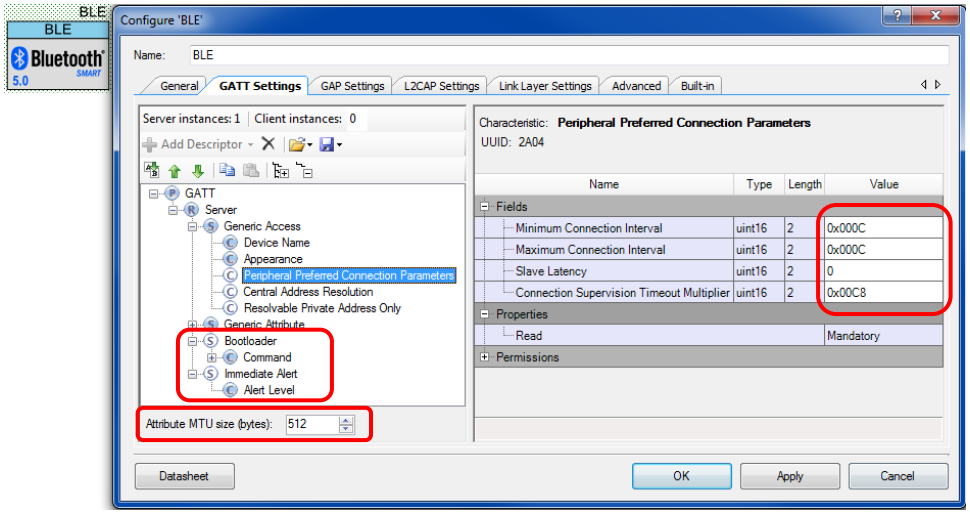


GATT Settings Tab (see Figure 20)**:**

- Generic Access, Peripheral Preferred Connection Parameters:
  - Minimum Connection Interval: 0x000C
  - Maximum Connection Interval: 0x000C
  - Connection Supervision Timeout Multiplier: 0x00C8

  The above intervals are selected to minimize bootloading time.
- Bootloader service for BLE bootloading
- Immediate Alert service for app switching
- Attribute MTU size (bytes): 512

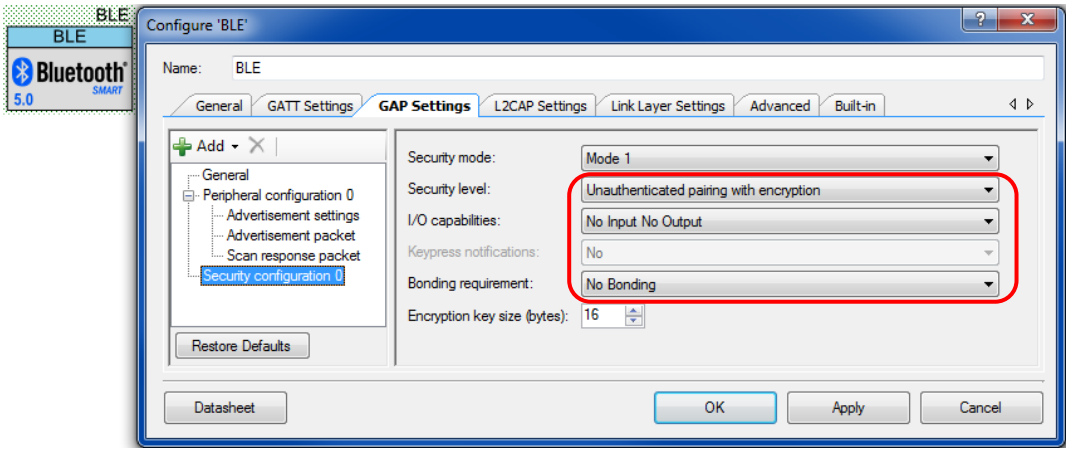
Figure 20. BLE Component, GATT Settings Tab Configuration



GAP Settings Tab:

- Device Name: “BLE Bootloader”
- Peripheral Configuration 0, Advertisement packet: Local Name checked and set to Complete
- Security configuration 0 (see Figure 21):
  - Security level: Unauthenticated pairing with encryption
  - I/O capabilities: No Input No Output
  - Bonding requirement: No Bonding

Figure 21. BLE Component, GAP Settings Tab Configuration



Link Layer Settings Tab:

- Link layer max TX and RX payload size (bytes): 251

Related Documents

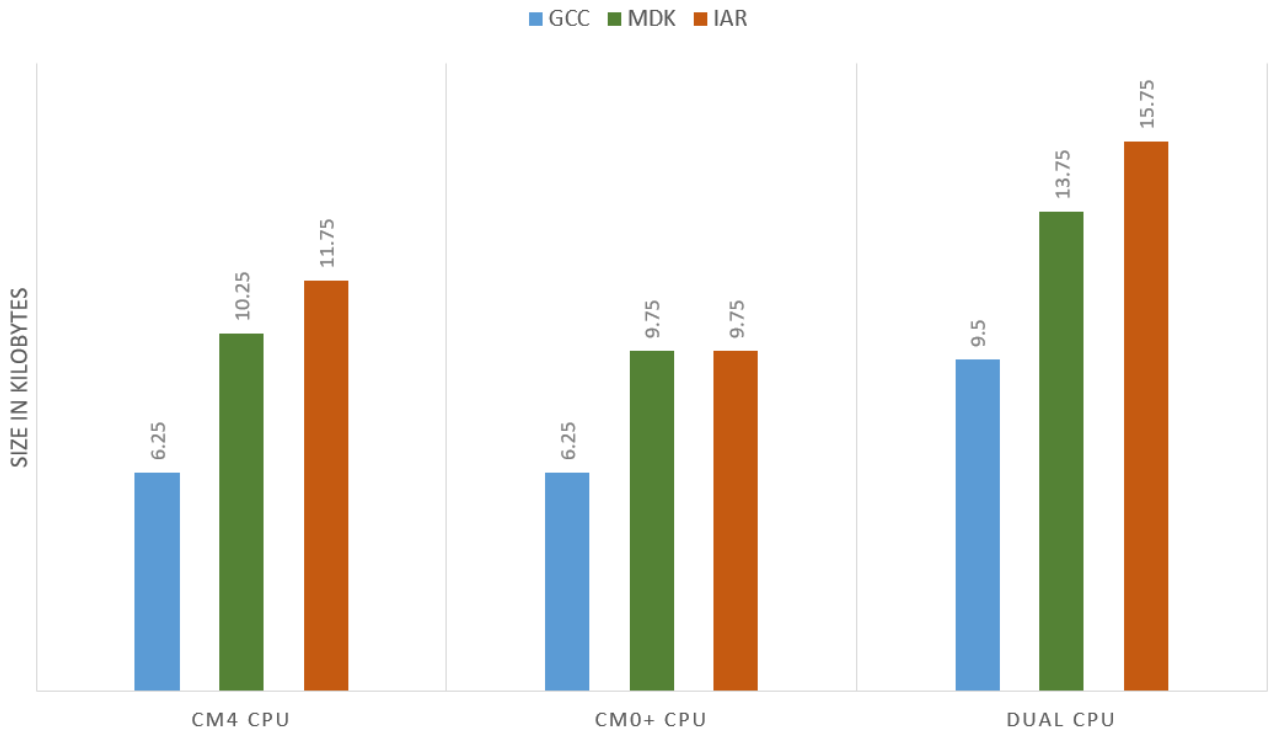
Application Notes	
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC 6 MCU with BLE Connectivity devices and how to build your first PSoC Creator project
AN213924 – PSoC 6 MCU Bootloader Software Development Kit (SDK) Guide	Provides information on how to use the Bootloader SDK, as well as information on bootloading in general
AN215671 – PSoC 6 MCU Firmware Design for BLE Applications	Shows how to design firmware for BLE applications on PSoC 6 MCU.
PSoC 6 MCU Bootloader-Related Code Examples	
CE213903 – Basic Bootloaders	Describes a basic bootloader using UART, I2C or SPI communication
CE221984 – Dual-Application Bootloader	Describes an I2C bootloader with two applications, with golden image mode.
CE216767 – BLE Bootloader	Describes a basic BLE bootloader
CE220959 – BLE Bootloader with External Memory	Describes a BLE bootloader that uses SMIF external memory
CE222802 – Encrypted Bootloader	Describes a UART bootloader with application encryption and signing
PSoC Creator Component Datasheets	
BLE	Provides information on Bluetooth Low Energy (BLE) settings and API
UART	Provides information on UART settings and API
Device Documentation	
PSoC 6 MCU: PSoC 63 with BLE Datasheets	PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual
Development Kit Documentation	
CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit	

Appendix A: CPU Core Configuration of the BLE Component

This example is configured to use the BLE Component on the CM4 CPU. Presets are available to easily switch to the CM0+ or dual-CPU configuration.

Figure 22 shows the comparison of the reserved RAM size for the BLE stack between the GCC, MDK, and IAR compilers depending on the BLE Component core configuration.

Figure 22. Size of Reserved RAM for the BLE Stack



The following steps describe how to change the CPU core configuration to CM0+ or dual-CPU mode.

1. Change the CPU core configuration of the BLE Component to the desired configuration in both the stack and user application schematics.
2. Set the interrupts to the appropriate CPUs in the user and stack applications.
    2.1. Set the BLE interrupt to the CM0+ CPU.
    2.2. Set all other interrupts of this example to the CM4 CPU if using the dual-CPU mode; else set them to the CM0+ CPU.
3. If the BLE Component is on the CM0+ CPU:
    3.1. Move the bootloader folder located in the CM4 folder of the stack application into the CM0+ folder.
    3.2. Move the host files folder located in the CM4 folder of the user application into the CM0+ folder.

The following steps depend on the compiler used:

GCC Compiler

4. Configure the linker scripts.

   4.1. Open the *bootload_common.ld* linker script of the launcher application project.

   4.2. Enable the desired configuration by opening/closing the appropriate comment blocks, as Figure 23 shows.

Figure 23. Linker Script BLE Configuration

```
/* ---------- BLE Stack CM4 Start ---------- */        Configuration Active

flash_app1_core0   (rx)   : ORIGIN = 0x10005000, LENGTH = 0x03000
flash_app1_core1   (rx)   : ORIGIN = 0x10008000, LENGTH = 0x2F000
flash_app2_core0   (rx)   : ORIGIN = 0x10040000, LENGTH = 0x03000
flash_app2_core1   (rx)   : ORIGIN = 0x10043000, LENGTH = 0x08000

ram_ble_core0      (rwx) : ORIGIN = 0x08000100, LENGTH = 0x0000
ram_ble_core1      (rwx) : ORIGIN = 0x08000100, LENGTH = 0x1900

ram_app1_core0     (rwx) : ORIGIN = 0x08002000, LENGTH = 0x4000
ram_app1_core1     (rwx) : ORIGIN = 0x08006000, LENGTH = 0x8000

ram_app2_core0     (rwx) : ORIGIN = 0x08001A00, LENGTH = 0x1E600
ram_app2_core1     (rwx) : ORIGIN = 0x08020000, LENGTH = 0x27800

/* ---------- BLE Stack CM4 End ---------- */

/* ---------- BLE Stack CM0 Start ----------      Configuration Inactive

flash_app1_core0   (rx)   : ORIGIN = 0x10005000, LENGTH = 0x03000
flash_app1_core1   (rx)   : ORIGIN = 0x10008000, LENGTH = 0x32000
flash_app2_core0   (rx)   : ORIGIN = 0x10040000, LENGTH = 0x03000
flash_app2_core1   (rx)   : ORIGIN = 0x10043000, LENGTH = 0x08000
```

   4.3. Copy the contents of the modified linker script and paste them in the *bootload_common.ld* linker scripts of the stack and user applications.

   4.4. Repeat Step 4.2 for the *bootload_cm0p.ld* and *bootload_cm4.ld* linker scripts of the stack and user application.

5. Set the heap size and enable the zero init table.

   5.1. Open the build settings of the stack application and set the preprocessor directives, as Table 12 shows.

Table 12. GCC Preprocessor Directives of the Stack Application

CPU Core Configuration of BLE Component	CM0+ Compiler Preprocessor Directives	CM4 Compiler Preprocessor Directives
BLE on CM0+	DEBUG; CY_CORE_ID=0; __HEAP_SIZE=0x4000; __STARTUP_CLEAR_BSS_MULTIPLE	DEBUG; CY_CORE_ID=0;
BLE on Dual CPU	DEBUG; CY_CORE_ID=0; __HEAP_SIZE=0x2500; __STARTUP_CLEAR_BSS_MULTIPLE	DEBUG; CY_CORE_ID=0; __HEAP_SIZE=0x4000; __STARTUP_CLEAR_BSS_MULTIPLE

   5.2. Click **OK**.

5.3. Open the build settings of the user application and set the preprocessor directives, as Table 13 shows.

Table 13. GCC Preprocessor Directives of the User Application

CPU Core Configuration of BLE Component	CM0+ Compiler Preprocessor Directives	CM4 Compiler Preprocessor Directives
BLE on CM0+	DEBUG; CY_CORE_ID=0; __STARTUP_CLEAR_BSS_MULTIPLE	DEBUG; CY_CORE_ID=0;
BLE on Dual CPU	DEBUG; CY_CORE_ID=0; __STARTUP_CLEAR_BSS_MULTIPLE	DEBUG; CY_CORE_ID=0; __STARTUP_CLEAR_BSS_MULTIPLE

5.4. Click **OK**.

6. **Clean and Build** all projects.

MDK Compiler

4. Configure the scatter files.

4.1. Open the *bootload_mdk_common.h* header file of the launcher application project.

4.2. Change the `#define BLE_CM4` line to the desired BLE configuration.

4.3. Copy the contents of the modified header file and paste them in the *bootload_mdk_common.h* header files of the stack and user applications.

5. Set the heap size and disable the MicroLib library.

5.1. Open the build settings of the stack application and set the assembler command line flags, as Table 14 shows.

Table 14. MDK Assembler Command Line Custom Flags of the Stack Application

CPU Core Configuration of BLE Component	CM0+ Assembler Command Line Custom Flags	CM4 Assembler Command Line Custom Flags
BLE on CM0+	--pd "__HEAP_SIZE SETA 0x4000"	
BLE on Dual CPU	--pd "__HEAP_SIZE SETA 0x2500"	--pd "__HEAP_SIZE SETA 0x4000"

5.2. Go to **Linker settings** of the CM0+ CPU set the **Use MicroLib** option to ‘false’. Do the same for the CM4 CPU if the BLE Component is on dual-CPU mode.

5.3. Click **OK**.

5.4. Open the build settings of the user application and repeat Steps 5.2 and 5.3.

6. Update the linker options to keep the stack.

6.1. Open the build settings of the stack application.

6.2. Update the linker command line custom flags of both CPUs to keep the linker from optimizing the stack out. See the *BLE Stack Keep Command List.txt* file located in the stack application project for the required commands for the CPU core configuration of each BLE Component.

6.3. Click **OK**.

7. **Clean and Build** all projects.

IAR Compiler

4. Modify the linker configuration files.
   4.1. Open the *bootload_common.icf* linker configuration file of the launcher application project.
   4.2. Change the `define symbol BLE_CM4 = 1;` line to the desired BLE configuration.
   4.3. Copy the contents of the modified linker configuration file and paste them in the *bootload_common.icf* linker configuration files of the stack and user applications.
5. Set the heap size.
   5.1. Right-click the CM0+ stack application project and select **Options**.
   5.2. Go to the **Linker settings** and to the **Config** tab.
   5.3. Set the configuration file symbol definitions as shown in Table 15.

Table 15. IAR Configuration File Symbol Definitions of the Stack Application

CPU Core Configuration of BLE Component	CM0+ Configuration File Symbol Definitions	CM4 Configuration File Symbol Definitions
BLE on CM0+	__HEAP_SIZE=0x4000	
BLE on Dual CPU	__HEAP_SIZE=0x2500	__HEAP_SIZE=0x4000

   5.4. Click **OK**.
   5.5. Right-click the CM4 stack application project and select **Options**.
   5.6. Repeat Steps 5.2 to 0.
6. Update the linker options to keep the stack.
   6.1. Right-click the CM0+ stack application project and select **Options**.
   6.2. Go to the **Linker settings** and to the **Input** tab.
   6.3. Update the symbol list to keep the stack from being optimized out. See the *BLE Stack Keep Command List.txt* file located in the stack application project for the required commands for the CPU core configuration of each BLE Component.
   6.4. Click **OK**.
   6.5. Right-click the CM4 stack application project and select **Options**.
   6.6. Repeat Steps 6.2 to 6.3 to configure the linker settings for the CM4 CPU.
7. **Clean and Build** all projects.

Appendix B: Implementing Stack Code Sharing with MDK and IAR Compilers

Applications generated by IAR and MDK compilers use special routines to initialize the RAM. These routines limit the amount of reserved RAM that can be saved in comparison with GCC, and require the user application to call a function from the stack application to reinitialize the RAM, and afterwards return to the user application.

Figure 22 on page 25 shows the comparison of the reserved RAM size for the BLE stack between the GCC, MDK, and IAR compilers depending on the CPU core configuration of the BLE Component.

MDK Compiler

The CPU-specific scatter files of the Bootloader SDK (*bootload_cm0p.scat* and *bootload_cm4.scat*) were modified to do the following:

- Separate the stack and heap from the RAM data to reduce the reserved RAM size for the BLE stack.
- Place the common configuration variables in a specific section to be overwritten by the user application.

Unused sections are removed by the linker. Stack functions may be used by the user application and not by the stack application. These functions must be explicitly kept to not be removed. Command line custom flags on the linker are used to keep the stack from being removed. See the *BLE Stack Keep Command List.txt* file located in the stack application project for the required commands for the CPU core configuration of each BLE Component.

The BLE stack requires common variables, such as clock settings, values of which depend on the running application and that are updated by the firmware. These variables must be overwritten by the user application and therefore placed in a specific section and order in the RAM.

The separation of the stack and heap from the RAM data and the definition of the Common BLE RAM section is made only if the BLE Component runs on the specific CPU. Figure 24 shows the definition of the Common BLE RAM section and the separation of the stack and heap from the RAM data for CM4.

Figure 24. Common BLE RAM Definition for MDK

```
#if defined BLE_CM4 || defined BLE_DUAL
    ER_RAM_STACKHEAP STACKHEAP_START UNINIT STACKHEAP_SIZE
    {
        .ANY (HEAP)
        .ANY (STACK)
    }

    ER_RAM_COMMON_CONFIG CONFIGRAM_START CONFIGRAM_SIZE
    {
        system_psoc63_cm4.o(+RW, +ZI)
        cy_ipc_sema.o(+RW, +ZI)
        cy_ipc_config.o(+RW, +ZI)
        cy_ipc_pipe.o(+RW, +ZI)
    }
#endif
```

The stack application must provide a way for the user application to reinitialize its BLE stack variables. This is realized by providing the `StackRAMInit()` function, which requires a return address as a parameter. The MicroLib library must be disabled for RAM reinitialization to work.

The return address is stored in the `CyReturnToBootloadableAddress` global variable. This variable is in the common RAM and is set to '0' after every reset.

`StackRAMInit()` calls the scatterload function to initialize the RAM of the stack application. Afterwards, scatterload calls the `_platform_pre_stackheap_init` weak function, which has been redefined to return to the user application using `CyReturnToBootloadableAddress`.

IAR Compiler

The CPU-specific linker configuration files of the Bootloader SDK (*bootload_cm0p.icf* and *bootload_cm4.icf*) were modified to do the following:

- Separate the stack and heap from the RAM data to reduce the reserved RAM size for the BLE stack.
- Place common configuration variables in a specific section to be overwritten by the user application.

Unused sections are removed by the linker. Stack functions may be used by the user application and not by the stack application. These functions must be explicitly kept to not be removed. The Keep symbols list in the linker input settings is used to keep the stack from being removed. See the *BLE Stack Keep Command List.txt* file located in the stack application project for the required commands for the CPU core configuration of each BLE Component.

The BLE stack requires common variables, such as clock settings, values of which depend on the running application and that are updated by the firmware. These variables must be overwritten by the user application and therefore placed in a specific section and order in the RAM.

The separation of the stack and heap from the RAM data and the definition of the Common BLE RAM section is made only if the BLE Component runs on the specific CPU. Figure 25 shows the definition of the Common BLE RAM section and the separation of the stack and heap from the RAM data for CM4.

Figure 25. Common BLE RAM Definition for IAR

```
70  if(isdefinedsymbol(BLE_CM4) || isdefinedsymbol(BLE_DUAL)) {
71      define block CONFIG_DATA with fixed order
72      {
73          readwrite object system_psoc63_cm4.o,
74          readwrite object cy_ipc_sema.o,
75          readwrite object cy_ipc_config.o,
76          readwrite object cy_ipc_pipe.o,
77      };
78  }
```

The stack application must provide a way for the user application to reinitialize its BLE stack variables. This is realized by providing the `StackRAMInit()` function. In comparison with the MDK implementation, this variable requires no parameters.

`StackRAMInit()` calls the `__iar_data_init3()` function to initialize the RAM of the stack application. Afterwards, the function returns to the user application.

Document History

Document Title: CE220960 – PSoC 6 MCU BLE Upgradable Stack Bootloader

Document Number: 002-20960

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	6097169	CFMM	03/13/2018	New code example
*A	6317430	MKEA	09/30/2018	Corrected errors when changing BLE Component configurations. Minor document updates.

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

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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

© Cypress Semiconductor Corporation, 2018. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.