Chapter 9: Over the air firmware update

Updating firmware in the field is critical to many consumer devices. For devices that support Bluetooth®, it is natural for the user to expect updates to happen using the Bluetooth® link rather than requiring the device to be connected in some other way such as a USB connection.

This chapter discusses how firmware can be updated using the Bluetooth® connection. This process is called Over the air (OTA) firmware update.

Table of contents

Document conventions

Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

9.1 Over the air update firmware architecture

The OTA process involves both the CM0+ and the CM4 so you will need firmware for both MCUs. The CM0+ runs the MCUboot application while the CM4 runs the OTA agent and user application.

The project for each core is discussed separately below.

The OTA agent and MCUboot applications are responsible for:

- OTA agent identifies that updates are available (alternately, updates may be peer initiated – this will be the case for Bluetooth®)
- OTA agent downloads updated firmware to a secondary slot in the device
- MCUboot validates and then copies updated firmware from the secondary slot to the primary slot
- MCUboot starts the application running on the CM4

The firmware images may be stored in either internal or external flash memory.

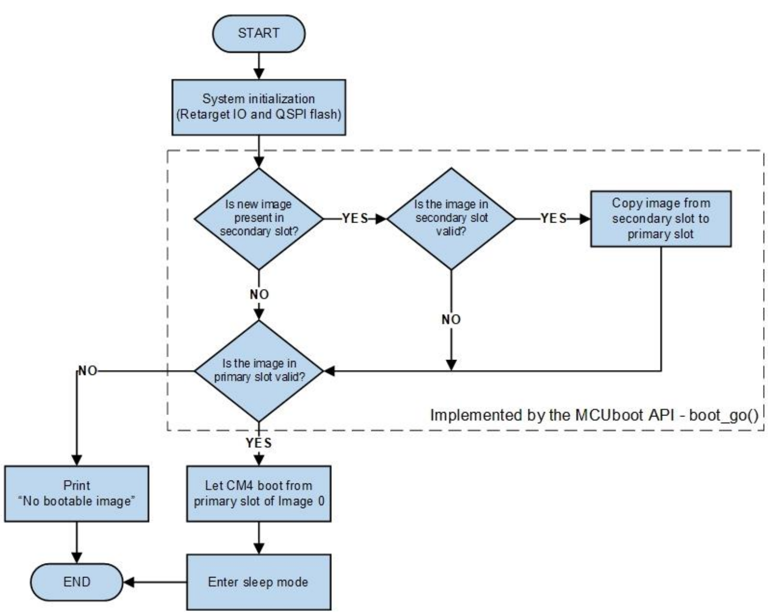
The OTA update process can use either encrypted or unencrypted firmware images and the images may be signed.

9.1.1 CM0+ project

The CM0+ project runs the MCUboot bootloader which handles validating, copying and starting the CM4 firmware images as shown below.

As mentioned above, the OTA update process uses two firmware images – a primary and a secondary. The user application always runs from the primary image. When an OTA update is performed, the new firmware is downloaded to the secondary image slot. On the next reboot, the firmware from the secondary slot is verified. If it is a valid image, it is copied to the primary slot and the new firmware is started.

Note that once the user application starts, the bootloader app running on the CM0+ puts it to sleep.



Note: Picture taken from the basic bootloader code example README.md file.

9.1.1.1 CM0+ MCUboot example

The easiest way to get the CM0+ application is to use the project from the MCUboot-Based Basic Bootloader (*mtb-example-psoc6-mcuboot-basic*) code example. It is a dual-core application that has MCUboot running on the CM0+ and a simple blinking LED app running on the CM4 along with the OTA agent. We will use just the CM0+ project (with minor modifications) in our Bluetooth® OTA exercises.

Note: The MCUboot-Based Basic Bootloader code example has a very detailed README.md file with information about how MCUboot works and the available options. We will use that code example in the exercises. You can either view the README.md file from inside the IDE, with a markdown viewer, or you directly on GitHub at: https://github.com/Infineon/mtb-example-psoc6-mcuboot-basic. The markdown viewer in Eclipse does not show tables correctly so it is recommended to use a different viewer (e.g. Chrome browser plugin or VSCode) or view it on GitHub.

The modifications required to the CM0+ project from the code example for use in our exercises are:

- Copy the file that specifies the memory organization used for the Bluetooth® project into the CM0+ project. This is necessary so that both projects have the same understanding of the memory layout.

  The file that is used for the CM4 project can be found in the *Makefile* for that project in the variable `OTA_FLASH_MAP`:

  ```
  OTA_FLASH_MAP?=$(SEARCH_ota-update)/configs/flashmap/psoc62_2m_ext_swap_single.json
  ```

  For the exercise, you will copy the file from *mtb_shared/ota-update/<release>/configs/flashmap/psoc62_2m_ext_swap_single.json* to the *flashmap* directory inside the CM0+ project.

Note: When you do this in the exercises, you will create the CM4 project first so that the ota-update library is available to copy the file from.

- Open the file *shared_config.mk* from the CM0+ project's directory and update the value of `FLASH_MAP` to specify the name of the file copied in the previous step:

  ```
  FLASH_MAP=psoc62_2m_ext_swap_single.json
  ```

Note: The JSON files in the ota-update library cover different cases of internal and external flash usage. See the README.md of that library for descriptions of the different options. In our case, we are using a file for a PSoC 62 device with 2 MB of internal flash and an external flash device. The memory layout places the primary image in the internal flash and the secondary image in external flash.

- Install the Python tools needed to create the bootloadable image.

  Open a command terminal window (e.g. modus-shell) and go to the directory *mtb_shared/mcuboot/<version>/scripts*

  Run the command `python -m pip install -r requirements.txt`

Once that's done, you can either program just the CM0+ or you can program the entire dual-core application. In either case, the CM4 project will be replaced with our Bluetooth® project in the next step.

Note: *To program just the CM0+, you can use* `make -j program_proj` *from the command line. Once programing completes, the bootloader messages in the UART will indicate that no image was found. If you program the entire application, the bootloader will launch the project from the code example which just blinks an LED.*

9.1.2 CM4 project

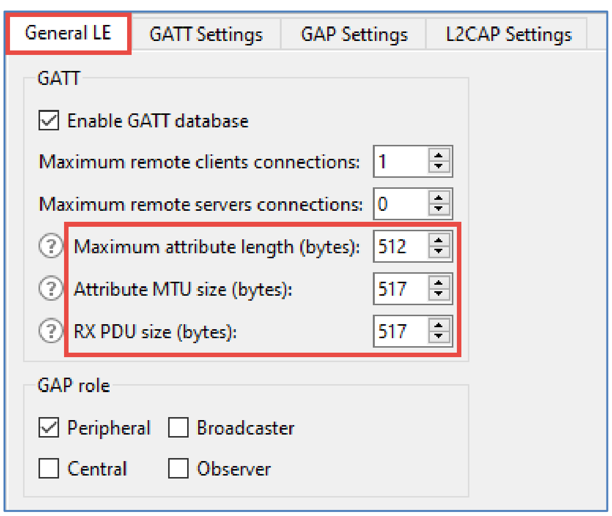
The CM4 project contains the end user application as well as the OTA agent that starts the upgrade process and downloads the new firmware image to the secondary slot. There is a special Bluetooth® service used to allow a peer to start the update process and to download the firmware. That is, the OTA agent is controlled by a custom Bluetooth® service.

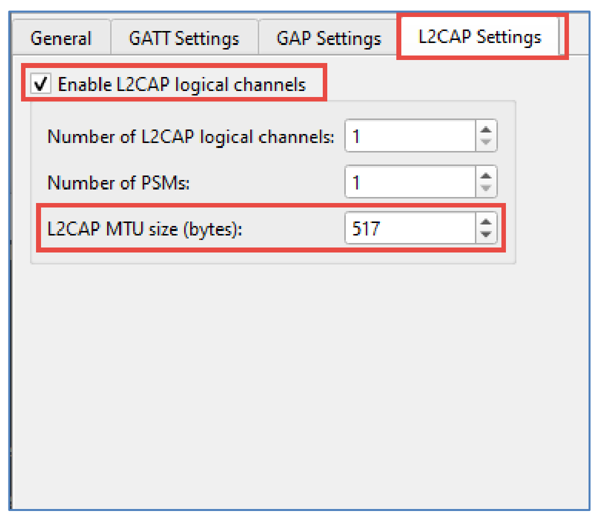
Note: *The project used in the exercises is based on the Bluetooth LE Batter Server code example which has OTA capability included.*

9.1.2.1 Bluetooth Configurator Settings

MTU sizes

The OTA process uses the largest packets it can to send the new firmware across the Bluetooth® link so the first change required is to set the Maximum attribute length to 512 and the Attribute MTU size to 517. It also requires that L2CAP is enabled and the L2CAP MTU size is set to 517.





Bluetooth® OTA service

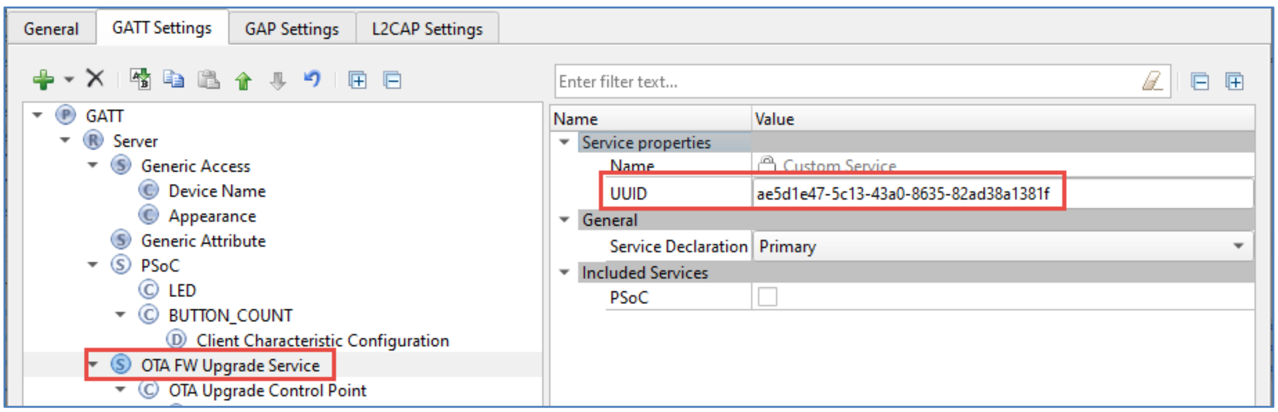
In order to load the new image, a custom Bluetooth® OTA service is included in the device firmware. This service takes care of loading the new firmware image into the secondary slot using the Bluetooth® link. Once the image has been loaded, the device is rebooted and MCUboot takes care of copying the firmware to the primary slot and then reboots again to execute the new firmware.

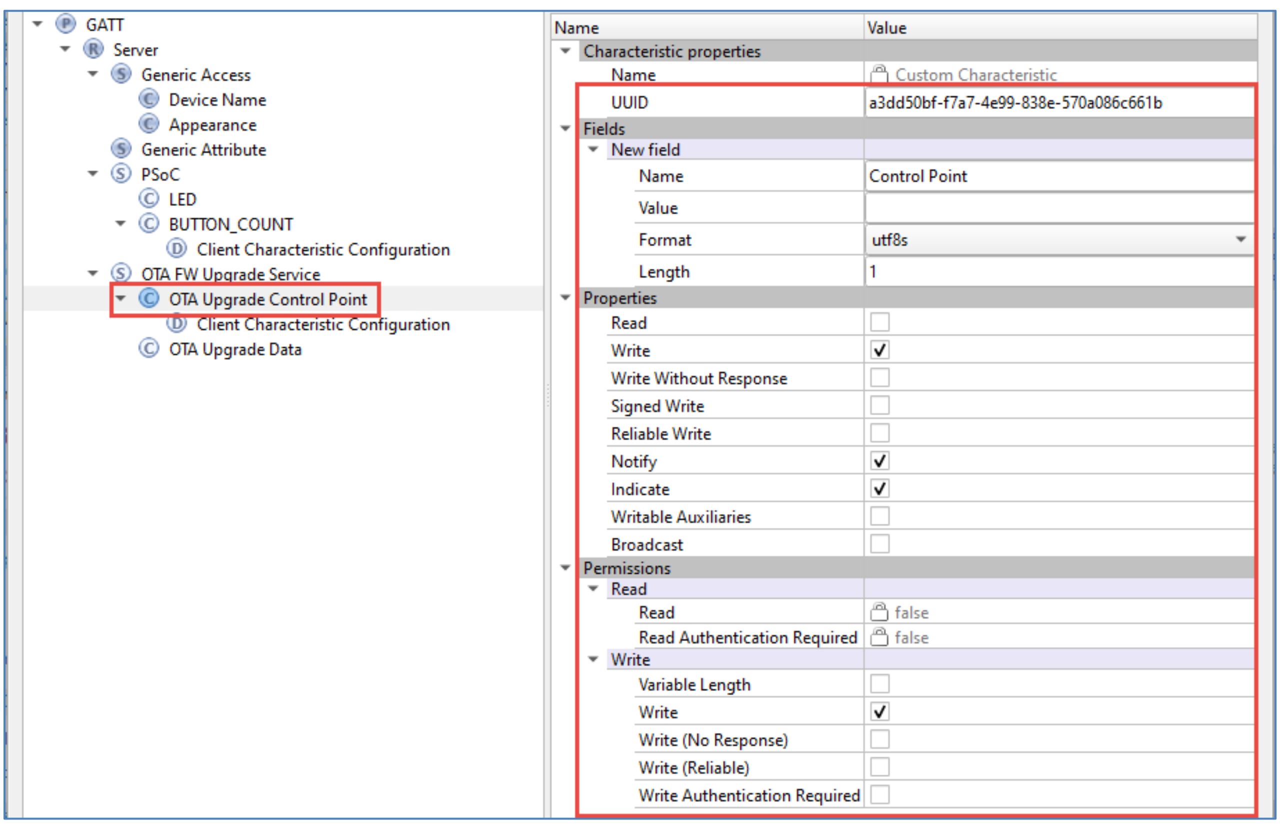
Note: *The OTA library relies on the service and characteristic names so you must match them exactly with what is shown below.*

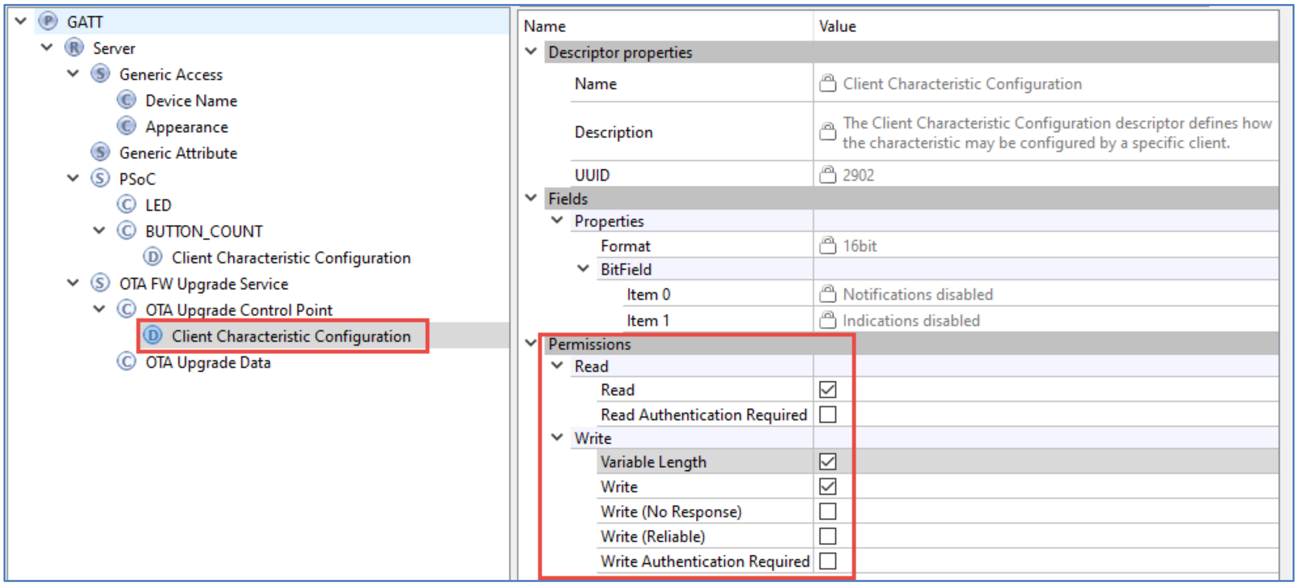
Note: *The UUIDs do not matter for the OTA service on the device, but the Windows application we will use to load new firmware uses hard-coded UUID values so you must match these in your Bluetooth® configuration to use the Windows application.*

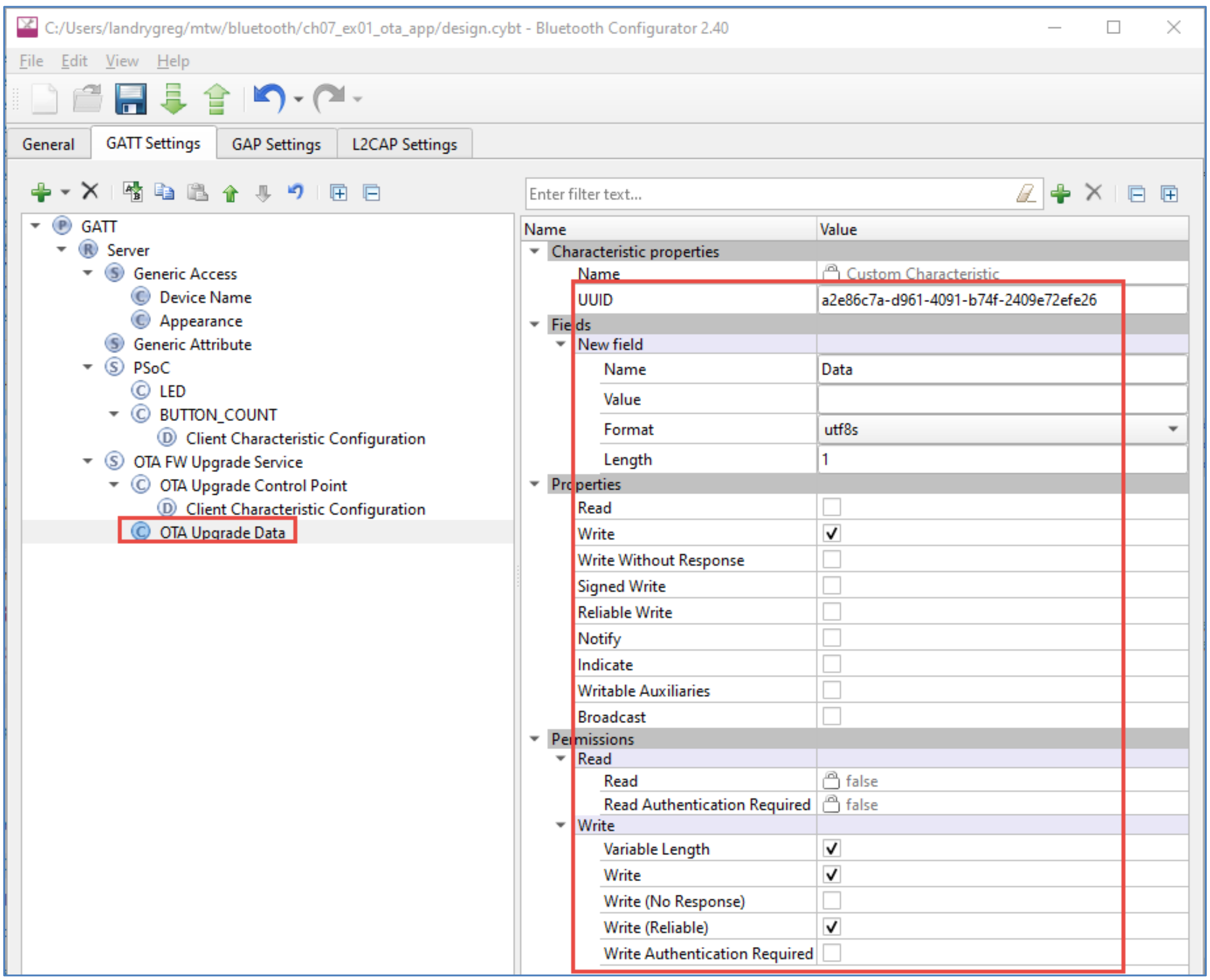
The service has two characteristics – a control point characteristic and a data characteristic. The control point has a CCCD to allow notify and indicate. The required properties and permissions for each characteristic and descriptor can be seen below. For easy copy/paste, the names and UUIDs for the service and characteristics are provided here:

Name	**UUID**
`OTA FW Upgrade Service`	ae5d1e47-5c13-43a0-8635-82ad38a1381f
`OTA Upgrade Control Point`	a3dd50bf-f7a7-4e99-838e-570a086c661b
`OTA Upgrade Data`	a2e86c7a-d961-4091-b74f-2409e72efe26






GATT
Server
Generic Access
Device Name
Appearance
Generic Attribute
PSoC
LED
BUTTON_COUNT
Client Characteristic Configuration
OTA FW Upgrade Service
OTA Upgrade Control Point
Client Characteristic Configuration
OTA Upgrade Data
Name
Value
Descriptor properties
Name
Client Characteristic Configuration
Description
The Client Characteristic Configuration descriptor defines how the characteristic may be configured by a specific client.
UUID
2902
Fields
Properties
Format
16bit
BitField
Item 0
Notifications disabled
Item 1
Indications disabled
Permissions
Read
Read
Read Authentication Required
Write
Variable Length
Write
Write (No Response)
Write (Reliable)
Write Authentication Required


C:/Users/landrygreg/mtw/bluetooth/ch07_ex01_ota_app/design.cybt - Bluetooth Configurator 2.40
File Edit View Help
General
GATT Settings
GAP Settings
L2CAP Settings
Enter filter text...
GATT
Server
Generic Access
Device Name
Appearance
Generic Attribute
PSoC
LED
BUTTON_COUNT
Client Characteristic Configuration
OTA FW Upgrade Service
OTA Upgrade Control Point
Client Characteristic Configuration
OTA Upgrade Data
Name
Value
Characteristic properties
Name
Custom Characteristic
UUID
a2e86c7a-d961-4091-b74f-2409e72efe26
Fields
New field
Name
Data
Value
Format
utf8s
Length
1
Properties
Read
Write
Write Without Response
Signed Write
Reliable Write
Notify
Indicate
Writable Auxiliaries
Broadcast
Permissions
Read
Read
false
Read Authentication Required
false
Write
Variable Length
Write
Write (No Response)
Write (Reliable)
Write Authentication Required

9.1.2.2 OTA Library

On PSoC 6 + 43012 devices, over the air updates are accomplished using the library *ota-update*. It must be included as a dependency by the CM4 project. The documentation for that library includes information on how to integrate the library into an application.

9.1.2.3 Config file

There is a configuration file provided in the *ota-update* library that should be copied into the CM4 application project's root directory:

mtb_shared/ota-update/<version>/configs/cy_ota_config.h

9.1.2.4 Makefile

The application Makefile must be modified to configure OTA. This includes changes to:

1. Specify the application version (optional, but useful in verifying the correct image is running). There are 3 levels of versioning: major, minor, and build.
2. Specify the CM4 as the processor for the project.
3. Define variables `OTA_SUPPORT` and `OTA_BT_SUPPORT`
4. Add `OTA_BLUETOOTH` and `OTA_PSOC_062` to the `COMPONENTS` variable.
5. Set the `CY_IGNORE` variable to exclude libraries that are included as dependencies of the *ota-upadte* library that are not needed for Bluetooth® OTA updates.
6. Set variables for MCUboot options. These must match the settings used in the MCUBoot project.
7. Define custom linker flags and specify a custom linker script.
8. Provide code that creates the boot loadable image file and encrypts/signs the image.

In the exercise, a Makefile with the required edits is provided. If you open this file, you will see that all of the OTA setup is done in two sections – one just after the POSTBUILD variable and before the Paths section, and the other at the end of the file.

9.1.2.5 Firmware

Updates are needed to the firmware to handle OTA setup and events are described below. There are a number of changes, but they are fairly straightforward. The exercise template will already have these changes done.

Includes

Three header files must be included. The first is from the *ota-update* library and the second is in the application. The third is from the *abstraction-rtos* library and it allows the use of the `cy_rtos_delay_milliseconds` function.

```
/* OTA related header files */
#include "cy_ota_api.h"  /* ota-update library header file */
#include "ota.h"         /* OTA init function and write handler for OTA BLE events */

#include "cyabs_rtos.h"
```

If the external flash is used to store the secondary image, an additional header file is required:

```
/* External flash API - used for secondary image storage */
#ifdef OTA_USE_EXTERNAL_FLASH
#include "ota_serial_flash.h"
#endif
```

Initialization in main

Several initialization steps are added in the main function before starting the RTOS scheduler:

1. The watchdog timer must be cleared and stopped so that the MCUboot application doesn't reboot the device.

```
/* Clear watchdog so the bootloader doesn't reboot on us */
cyhal_wdt_init(&wdt_obj, cyhal_wdt_get_max_timeout_ms());
cyhal_wdt_free(&wdt_obj);
```

2. The serial flash QSPI interface must be initialized.

```
/*Initialize QuadSPI if using external flash*/
#if defined(OTA_USE_EXTERNAL_FLASH)
       /* We need to init from every ext flash write
        * See ota_serial_flash.h
        */
       if (CY_RSLT_SUCCESS != ota_smif_initialize())
       {
              cy_log_msg(CYLF_OTA, CY_LOG_ERR,"QSPI initialization FAILED!!\r\n");
              CY_ASSERT(0 == 1);
       }
       else
       {
              cy_log_msg(CYLF_OTA, CY_LOG_ERR, "successfully Initialized QSPI \r\n");
       }

#endif /* OTA_USE_EXTERNAL_FLASH */
```

3. The `ota_config` structure holding the OTA configuration must be initialized:

```
/* Set default values for OTA configuration */
app_bt_initialize_default_values();
```

4. After the application is updated and MCUboot copies the new image into the primary slot, the application must specify that the new image should now become the permanent application. This is done during application startup so that each time a new image runs, it will be validated as the new permanent application.

```
/* Validate the update so we do not revert on reboot */
cy_ota_storage_validated();
```

Update GATT server event handler

In the GATT server event handler, the `GATT_HANDLE_VALUE_CONF` event needs to check to see if the OTA state is `CY_OTA_STATE_OTA_COMPLETE`. If so, it either reboots the device (the default) or stops the OTA agent depending on the value of `ota_context.reboot_at_end`.

```
case GATT_HANDLE_VALUE_CONF: /* Value confirmation */
   {
      cy_log_msg(CYLF_OTA, CY_LOG_DEBUG, "  %s() GATTS_REQ_TYPE_CONF\r\n", __func__);
      cy_ota_agent_state_t ota_lib_state;
      cy_ota_get_state(ota_config.ota_context, &ota_lib_state);
      if ((ota_lib_state == CY_OTA_STATE_OTA_COMPLETE) && /* Check if we completed the */
          (ota_config.reboot_at_end != 0))              /* download before rebooting */
      {
          cy_log_msg(CYLF_OTA, CY_LOG_NOTICE, "%s()   RESETTING IN 1 SECOND !!!!\r\n",
                      __func__);
          cy_rtos_delay_milliseconds(1000);
          NVIC_SystemReset();
      }
      else
      {
          cy_ota_agent_stop(&ota_config.ota_context); /* Stop OTA */
      }
       status = WICED_BT_GATT_SUCCESS;
   }
   break;
```

Update GATT write handler

The GATT write handler must be updated to take care of writes related to OTA. As you will recall, the write handler normally searches for the characteristic in the GATT database. If it finds it, the data provided is stored to the appropriate GATT database location.

For OTA, the data being sent is not stored in the GATT database. Rather, it is stored in external flash by the OTA functions. Therefore, the GATT write handler needs a separate section just for writes to one of the OTA characteristics. When such a write is done, the appropriate OTA API function is called.

The separate handling for OTA writes is added before the normal GATT write handling. When an OTA write is done, the GATT write function returns before it gets to the normal GATT write handling. The entire function is shown on the next page. The OTA section is shown in bold.

As you can see, the function that is called for OTA writes is called `app_bt_ota_write_handler`. It is defined in the file *ota.c.* You should review the `app_bt_ota_write_handler` function in *ota.c* to see how the OTA writes are accomplished using the *ota-update* library. You will see several functions with the prefix `cy_ota_ble_` being used at various times.

```
static wiced_bt_gatt_status_t app_bt_write_handler(wiced_bt_gatt_event_data_t *p_data)
{
    wiced_bt_gatt_write_req_t *p_write_req = &p_data->attribute_request.data.write_req;;
    wiced_bt_gatt_status_t status = WICED_BT_GATT_ERROR;

    /* OTA GATT write handling */
    switch ( p_write_req->handle )
    {
        case HDLD_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_CONTROL_POINT_CLIENT_CHAR_CONFIG:
        case HDLC_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_CONTROL_POINT_VALUE:
        case HDLC_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_DATA_VALUE:
            /*Call OTA write handler to handle OTA related writes*/
            status = app_bt_ota_write_handler(p_data);
            return status;
    } /* End of OTA write handling */

    /* Normal GATT write handling */
    for (int i = 0; i < app_gatt_db_ext_attr_tbl_size; i++)
    {
        /* Check for a matching handle entry */
        if (app_gatt_db_ext_attr_tbl[i].handle == p_write_req->handle)
        {
            /* Detected a matching handle in the external lookup table */
            if (app_gatt_db_ext_attr_tbl[i].max_len >= p_write_req->val_len)
            {
                /* Value fits within the supplied buffer; copy over the value */
                app_gatt_db_ext_attr_tbl[i].cur_len = p_write_req->val_len;
                memset(app_gatt_db_ext_attr_tbl[i].p_data, 0x00,
                       app_gatt_db_ext_attr_tbl[i].max_len);
                memcpy(app_gatt_db_ext_attr_tbl[i].p_data, p_write_req->p_val,
                       app_gatt_db_ext_attr_tbl[i].cur_len);

                if (memcmp(app_gatt_db_ext_attr_tbl[i].p_data, p_write_req->p_val,
                           app_gatt_db_ext_attr_tbl[i].cur_len) == 0)
                {
                    status = WICED_BT_GATT_SUCCESS;
                }

                 switch ( p_write_req->handle )
                {
                    // Add action when specified handle is written
                    case HDLC_PSOC_LED_VALUE:
                        cyhal_gpio_write(CYBSP_LED_RGB_BLUE, app_psoc_led[0] == 0 );
                        printf( "Turn the LED %s\r\n", app_psoc_led[0] ? "ON" : "OFF" );
                        break;
                 }
            }
            else
            {
                /* Value to write will not fit within the table */
                status = WICED_BT_GATT_INVALID_ATTR_LEN;
                printf("Invalid attribute length during GATT write\n");
            }
            break;
        }
    }

    if (WICED_BT_GATT_SUCCESS != status)
    {
       printf("GATT write failed: %d\n", status);
    }

    return status;

}
```

9.1.2.6 Log message utility

The template provided for the exercise uses the `cy_log` API to print debug messages. This API is part of the *connectivity_utilities* library which is a dependency of the *ota-update* library. The `cy_log` API has the advantage over `printf` in that it allows messages to have a specified severity level. A single configuration setting can be used to change which severity level's messages are printed. This is very useful to allow enabling/disabling of debug messages.

The available levels are:

```
CY_LOG_OFF = 0,      /**< Do not print log messages */
CY_LOG_ERR,          /**< Print log message if run-time level is <= CY_LOG_ERR       */
CY_LOG_WARNING,      /**< Print log message if run-time level is <= CY_LOG_WARNING   */
CY_LOG_NOTICE,       /**< Print log message if run-time level is <= CY_LOG_NOTICE    */
CY_LOG_INFO,         /**< Print log message if run-time level is <= CY_LOG_INFO      */
CY_LOG_DEBUG,        /**< Print log message if run-time level is <= CY_LOG_DEBUG     */
CY_LOG_DEBUG1,       /**< Print log message if run-time level is <= CY_LOG_DEBUG1    */
CY_LOG_DEBUG2,       /**< Print log message if run-time level is <= CY_LOG_DEBUG2    */
CY_LOG_DEBUG3,       /**< Print log message if run-time level is <= CY_LOG_DEBUG3    */
CY_LOG_DEBUG4,       /**< Print log message if run-time level is <= CY_LOG_DEBUG4    */

CY_LOG_PRINTF, /* Identifies log messages generated by cy_log_printf calls */
```

As you can see, there is a lot of control over what gets printed.

The system is initialized by calling the `cy_log_init` function and specifying the level of messages that should be printed:

```
cy_log_init(CY_LOG_DEBUG, NULL, NULL);
```

The level can be changed at any time during execution. For example, you can turn logging off for OTA messages:

```
cy_ota_set_log_level(CY_LOG_OFF);
```

Finally, to print a message, you use `cy_log_msg` (normal `printf` syntax is supported). For example, the following will print the value of var1 if the log level is debug or higher.

```
cy_log_msg(CYLF_DEF, CY_LOG_DEBUG, "Value is: %d\n, var1);
```

The first argument to `cy_log_msg` is an indication of where the message came from. The available values are:

```
CYLF_DEF = 0,                 /**< General log message */
CYLF_TEST,                    /**< Test Facility */
CYLF_DRIVER,                  /**< Driver Facility */
CYLF_LP,                      /**< Low Power Facility */
CYLF_MIDDLEWARE,              /**< Middleware Facility */
CYLF_AUDIO,                   /**< Audio Facility */
```

Note: *The template provided for the exercises repurposes `CYLF_AUDIO` to be `CYLF_OTA`.*

9.2 Secure OTA update

The MCUboot basic code example uses public key encryption so that the file that is downloaded to the device during OTA update is encrypted. The keys used are test keys that are located in the MCUboot basic code example CM0+ project and the exercise template project in the *keys* directories. The file *cypress-test-ec-p256.pem* is the private key and the file *cypress-test-ec-p356.pub* is the public key.

The public key is used to encrypt the build output of the CM4 project. That is done as a post-build step. The private key is included during the build of the CM0+ project in the MCUboot basic code example so that it can unencrypt the image when it is loaded during the OTA process.

The file names are specified in *shared_config.mk* for the CM0+ project and in *Makefile* for the CM4 exercise template project. Both projects have a copy of both keys, although each project uses only one key as described above.

9.2.1 Generate a public/private key pair

While the key pair from the MCUboot library is used for the exercises in this chapter, it would be an extremely bad idea to use it for a production design since the private key is widely available.

You can generate your own key pair using the python *imgtool* program or you can use another key generation utility.

The *imgtool utility can be found in the mcuboot library:*

<workspace>/mtb_shared/mcuboot/<version>/scripts

Once you are in that directory in a command terminal (modus-shell for Windows) use the following to generate the private key and then extract the public key in the form of a C array.

```
python imgtool.py keygen -k my_key.pem -t ecdsa-p256
python imgtool.py getpub -k my_key.pem >> my_key.pub
```

Note: *The name of the private and public keys should be the same except for the extension (pem for the private key and pub for the public key).*

9.2.2 Root of Trust for secured boot and secure key storage

The examples shown here demonstrate the image signing and validation features of MCUboot. They do NOT implement root of trust (RoT)-based secured services such as secured boot and secured storage. Those features are beyond the scope of this class, but if you are interested in them, check out the [PSoC™ 64 line of secured MCUs](#).

9.3 Bluetooth® OTA update Windows peer application

A Windows application is provided that can be used to perform the Bluetooth® OTA update process. This application is provided only as an example – for an end product, the product creator normally provides his/her own application either as a stand-alone app or as part of an existing app. Often these will be iOS or Android apps.

The example Windows peer application is called *WsOtaUpgrade*. Both the source code and executable are provided in the *btsdk-peer-apps-ota* Git repo located on the Infineon Github site at https://github.com/Infineon/btsdk-peer-apps-ota.

The easiest way to get access to it is to clone the repo locally. To do that, open modus-shell (on Windows) or a command terminal (on MacOS or Linux), change to the directory where you want the repo to be stored locally, and run the command:

```
git clone https://github.com/Infineon/btsdk-peer-apps-ota.git
```

Once you have the repo, the Windows executable can be found under the following directory:

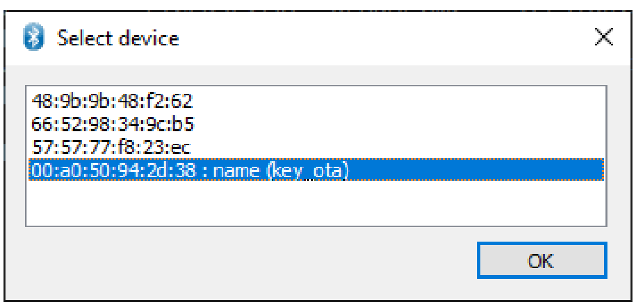
btsdk-peer-apps-ota/Windows/WsOtaUpgrade/Release/x64/WsOtaUpgrade.exe

To use the Windows peer application, you must first copy the **.bin* file from the build Debug directory of the application into the same directory as the Windows peer application. Then run the application with the **.bin* file provided as an argument. For example, from modus-shell:

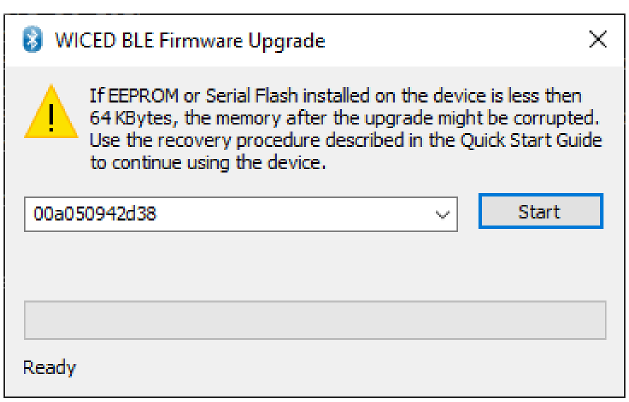
```
./WsOtaUpgrade.exe app.bin
```

Note: You can also just drag the .bin file onto WSOtaUpgrade.exe from a Windows file explorer window.

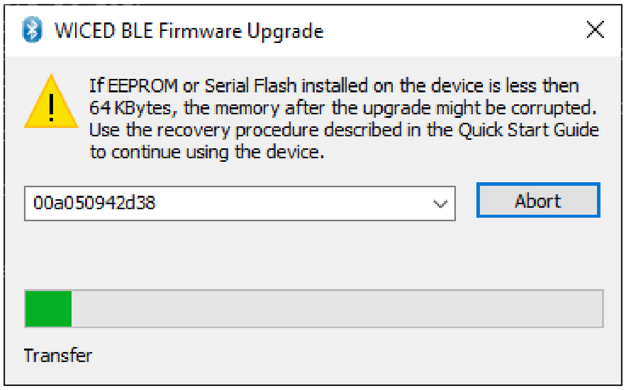
You will get a window that looks like the following. Select the device you want to update and click **OK**.



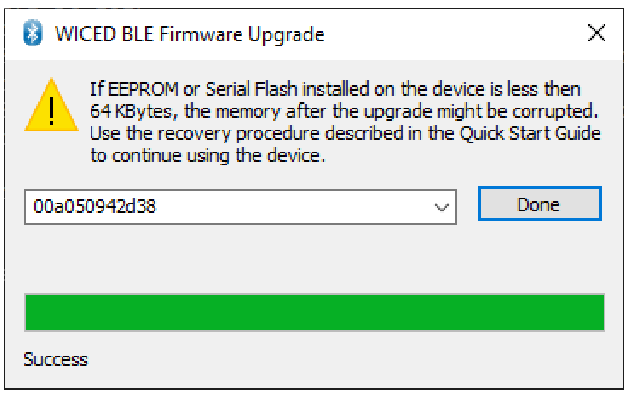
On the next window, click **Start**.



Once the update starts, you will see a progress bar. It may take up to a minute for the new firmware to be downloaded to the secondary flash location.



Once it finishes, the window will show "Success" at the bottom if the update worked. Click **Done** to close the window.



Once the update is done, the device is rebooted. MCUboot then verifies the new image and copies it to the primary slot. That process may also take up to a minute to complete. During that time, the UART window will look like this:

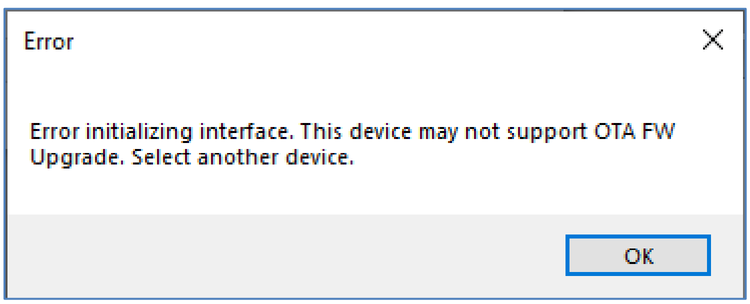
```
[INF] MCUBoot Bootloader Started
[INF] External Memory initialized w/ SFDP.
[INF] Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: none
[INF] boot_swap_type_multi: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] boot_swap_type_multi: Secondary image: magic=good, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: test
[INF] Erasing trailer; fa_id=1
[INF] Erasing trailer; fa_id=3
[INF] Erasing trailer; fa_id=2
```

Once the new image has been validated and copied, the new application will start as normal:

```
[INF] MCUboot Bootloader Started
[INF] External Memory initialized using SFDP
[INF] swap_status_source: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] Boot source: none
[INF] boot_swap_type_multi: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] boot_swap_type_multi: Secondary image: magic=good, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: test
[INF] Erasing trailer; fa_id=1
[INF] Erasing trailer; fa_id=2
[INF] User Application validated successfully
[INF] Starting User Application on CM4. Please wait...
**********Application Start*****************
**********Version: 1.0.0********************
********************************************
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:94:2D:38
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
```

9.3.1 Troubleshooting

You may see the following error message from the Windows OTA update application when you click the **Start** button:



This message can be caused by several different issues. The following items should resolve the issue:

1. Verify that the OTA service name, characteristic names, fields properties, permissions and UUIDs all match what was shown earlier. Remember that the Windows application uses hard-coded UUID values to identify the service and characteristics so them must match exactly.
2. Verify that the code for supporting OTA is correct.
3. Quit the application, restart the application, reset the kit, and try again.
4. Quit the application, turn off the PC's Bluetooth® radio, re-enable the Bluetooth® radio, reset the kit, and try again.
5. Reboot the PC, reset the kit, and try again.

9.4 Exercises

Exercise 1: Bluetooth® LE application with OTA update capability

This exercise will be done in several stages to create, program, and test the OTA update process:

1. Create a Bluetooth® project for the CM4 using a template that has been modified to support OTA firmware updates.
2. Create, update and program the MCUboot basic code example onto the device.
3. Program the Bluetooth® project to the CM4 to replace the project from the MCUboot basic CE and test the Bluetooth® functionality.
4. Modify the Bluetooth® project's functionality and load it onto the CM4 using a Bluetooth® OTA update.

Create the Bluetooth® app from the provided template and update project settings

☐ 1. Create a new ModusToolbox™ application for the CY8CKIT-062S2-43012 BSP.

On the application template page, use the **Browse** button to start from the template in *Templates/ch09_ex01_ota_app*. Keep the application name the same.

☐ 2. Open the Bluetooth® configurator and set the device name to **<inits>_ota** in the GAP Settings.

☐ 3. On the General tab, set **Maximum attribute length** to 512, **Attribute MTU size** to 517 and **RX PDU size** to 517.

☐ 4. On the L2CAP Settings tab, check the box for **Enable L2CAP logical channels** and set the **L2CAP MTU size** to 517.

☐ 5. On the GATT settings tab, add a new custom service and characteristics for OTA update support.

Use the table and images from the Bluetooth® configurator settings section of this chapter (9.1.2.1) to set the correct names, UUIDs, fields properties, and permissions.

Note: The table contains the exact values required for the service/characteristic names and UUIDs so it is easiest to copy/paste from the table.

☐ 6. Save changes and close the configurator.

☐ 7. Copy the file *cy_ota_config.h* file from the *ota-update* library to the application. The default settings in this file are OK so we won't need to change it.

Note: The file is in *mtb_shared/ ota-update/<version>/configs*.

☐ 8. The *Makefile* already has all of the changes needed for OTA. Review the file to understand was changed.

☐ 9. The *ota-update* library has already been added to the application as a dependency in the *deps* directory.

☐ 10. All changes required to the code are already done in *main.c*. Review the code to understand the changes that were described earlier in this chapter.

Create the MCUboot basic bootloader app, update the CM0+ project settings, and program the device

☐ 1. Create another new ModusToolbox™ application for the CY8CKIT-062S2-43012 BSP.

Use the *MCUboot-Based Basic Bootloader* code example as the template.
Name the new application *ch09_ex01_ota_bootloader*.

Note: *This is a dual-core application but we will replace the CM4 project with the one from the previous section later.*

Note: When you create the application, you will see 2 errors reported. These are intended to bring attention to the flashmap generation step and can be ignored as long as the end of the log says that the project was successfully created and exported. The error messages look like this:

```
=============================================================
ERROR:= Generating cy_flash_map.h and flashmap.mk using psoc62_swap_single.json
file (Look out for any warnings or errors in this step) =
=============================================================
python ./scripts/flashmap.py -p PSOC_062_2M -i ./flashmap/psoc62_swap_single.json
-o ./source/cy_flash_map.h > ./flashmap.mk
=============================================================
```

☐ 2. Go to the CM0+ project and make the updates described in 9.1.1.1. As a reminder, you must:

a. Copy the file that specifies the memory organization used for the Bluetooth® project into the CM0+ project.

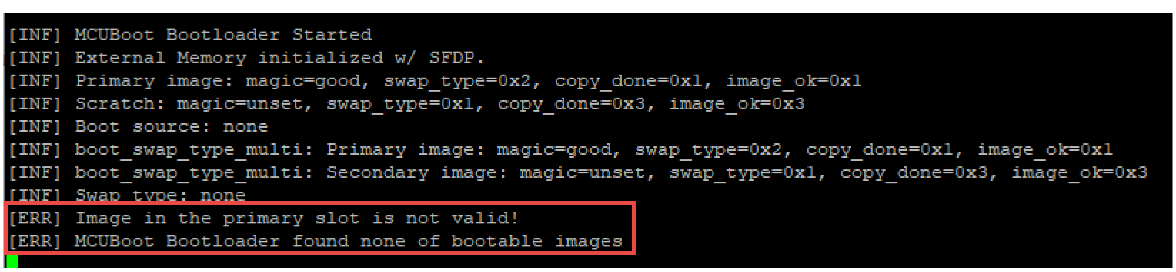
b. Update the value of `FLASH_MAP` in the file *shared_config.mk* from the CM0+ project's directory.

c. Ensure that the tools required for creating the bootloadable image are installed.

☐ 3. Build and program the application or just the CM0+ project to the kit.

You can program the application as normal, or if you prefer you can program just the CM0+ by using the command `make -j program_proj` from the CM0+ project directory on the command line.

Note: *If you program just the CM0+, you will see a message in the UART terminal window saying that a valid application was not found. In that case, the device will remain in bootloader mode as shown here:*



Note: *If you program the full application, the bootloader will start the CM4 project from the code example. It is a "blinking LED" application. In that case, the UART terminal window will look like this:*

```
[INF] MCUBoot Bootloader Started
[INF] External Memory initialized w/ SFDP.
[INF] Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: none
[INF] boot_swap_type_multi: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] boot_swap_type_multi: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: none
[INF] User Application validated successfully
[INF] Starting User Application (wait)...
[INF] Start slot Address: 0x10018400
[INF] MCUBoot Bootloader finished
[INF] Deinitializing hardware...

=========================================================
[BlinkyApp] Image Type: BOOT, Version: 1.0.0, CPU: CM4

=========================================================
[BlinkyApp] Watchdog timer started by the bootloader is now turned off to mark the successful start of Blinky app.
[BlinkyApp] User LED toggles at 1000 msec interval
```

Program the Bluetooth® project to the CM4 and test its functionality

☐ 1. Now that we have the bootloader programmed to the CM0+, we need to program the Bluetooth® project for the CM4. Program the *ch09_ex01_ota_app* application as normal.

☐ Note: *The programming step will only replace the CM4 project since its linker script (referenced from the ota-update library) knows where to program it in flash.*

☐ 2. With the application running and advertising, use AIROC™ Connect to connect to the device and test the Bluetooth® functionality.

The application has the same functionality as the ch05_ex01_pair exercise. There is one characteristic that allows you to turn the LED on/off and a second characteristic with notifications that counts how many times the button has been pressed.

Note: *You will also see a second service with two characteristics – one with Write & Indicate permissions and one with just Write permission. That is the OTA service. Don't write to those characteristics since they are intended for the OTA update process. You will get a chance to try that next.*

☐ 3. Disconnect when you are done so that the OTA update process will be able to connect to the device.

☐ 4. Remove the device from the paired Bluetooth device list on the phone.

Note: *This is necessary to be able to reconnect later because the application does not store bonding information.*

Modify firmware and use OTA update to load the new firmware onto the device and re-test

1. In the CM4 application (*ch09_ex01_ota_app*), open the *Makefile* and change OTA_*APP_VERSION_MINOR* to 1.

2. Open the CM4 application's *main.c* file. Change the line that increments the counter on each button press to count down instead of up. This will allow you to see a functional difference in the kit's behavior in addition to just the version number.

Note: *Just change the line* `app_psoc_button_count[0]++;` *to* `app_psoc_button_count[0]--;`

3. Build the application but do <u>NOT</u> program it to the kit.

4. Copy the *bin* file from the application to the directory containing the Windows bootloader application.

   The file will be in the application's root directory under *build/APP_CY8CKIT-062S2-43012/Debug/app.bin*.

5. Run the Windows bootloader and load the new firmware.

   Follow the instructions from 0. See the troubleshooting information in 9.3.1 if you run into issues.

Note: *If the device advertising times out, just reset the kit and wait for advertising to start before performing the OTA update. The device must be advertising for the Windows bootloader application to be able to connect to the device.*

Note: *Remember that bootloading will take about 1 minute and then the reboot/copy/validate will take about another minute.*

Note: *You will see lots of messages in the UART terminal during the bootloading process because we have the logging level for OTA set to* `CY_LOG_INFO`.

6. When bootloading completes, observe the new application version in the UART.

```
[INF] MCUBoot Bootloader Started
[INF] External Memory initialized w/ SFDP.
[INF] Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: none
[INF] boot_swap_type_multi: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] boot_swap_type_multi: Secondary image: magic=good, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: test
[INF] Erasing trailer; fa_id=1
[INF] Erasing trailer; fa_id=3
[INF] Erasing trailer; fa_id=2
[INF] User Application validated successfully
[INF] Starting User Application (wait)...
[INF] Start slot Address: 0x10018400
[INF] MCUBoot Bootloader finished
[INF] Deinitializing hardware...
**********Application Start*****************
**********Version: 1.1.0********************
********************************************
[F5] : [L1] : 0000 00:00:00.000 successfully Initialized QSPI
```

Note: *Once you see the application version, reset the kit so that other* `printf` *messages will appear in the UART.*

7. Use AIROC™ Connect to test the functionality to see that pressing the button on the kit now causes the count value to decrease instead of increase.

Trademarks

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

Published by

Infineon Technologies AG

81726 Munich, Germany