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 occur 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.

The exercise in this chapter uses an application from the MCUboot GitHub repo as a framework for the application.

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 application overview

The application descriptions and exercise in this chapter are based on the Bluetooth LE Battery Server with OTA code example. The code example for PSoC™ 6 uses version 2.x of the *ota-update* library while the code example for the CYW20829 uses version 3.x of the *ota-update* library. The differences between the two libraries and the code examples will be described, but the concept is the same for both.

The OTA process involves two separate projects – one is the bootloader project (MCUboot) while the other is the user application which also contains the OTA agent. Each of these projects is discussed separately below.

Note: *In the case of dual-core PSoC6 devices, the CM0+ runs the MCUboot project, while the CM4 runs the user application and OTA agent.*

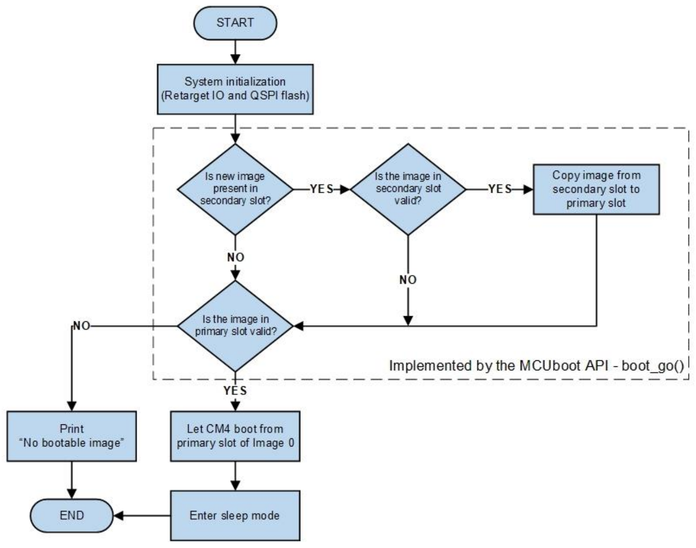
The OTA agent and MCUboot projects are responsible for the following:

- 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 user application.

The OTA update process can use either encrypted or unencrypted firmware images and the images may be signed. The firmware images may be stored in either internal or external flash memory.

9.1.1 MCUboot bootloader project

At powerup, the MCUbooth project runs first. If the OTA agent has downloaded new firmware to the secondary slot, it is the MCUboot bootloader that verifies its validity and then copies it to the primary location. If there is no new firmware in the secondary slot, the MCUboot project verifies that the primary firmware image is valid. Once a valid primary image is ready, MCUboot starts that firmware running and puts itself to sleep until another power cycle or OTA update.


START
System initialization (Retarget IO and QSPI flash)
Is new image present in secondary slot?
YES
Is the image in secondary slot valid?
YES
Copy image from secondary slot to primary slot
NO
NO
NO
Is the image in primary slot valid?
Implemented by the MCUboot API - boot_go()
YES
Print "No bootable image"
Let CM4 boot from primary slot of Image 0
END
Enter sleep mode

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

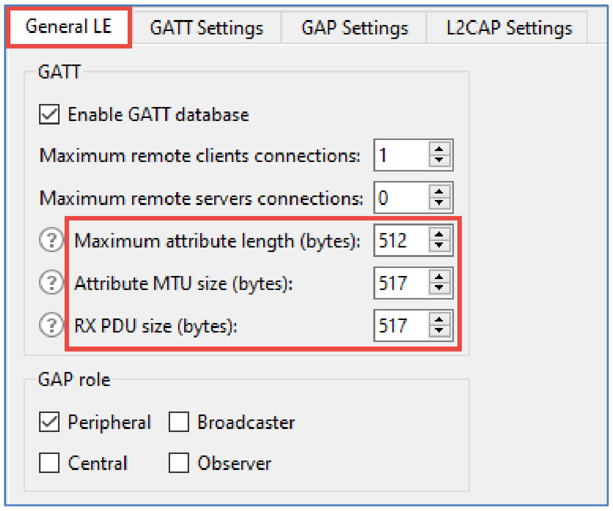
9.1.2 User application and OTA agent

The end user application contains the OTA agent which 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.

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, the **Attribute MTU size** to 517 and the **RX PDU size** to 517. It also requires that L2CAP is enabled and the **L2CAP MTU size** is set to 517.



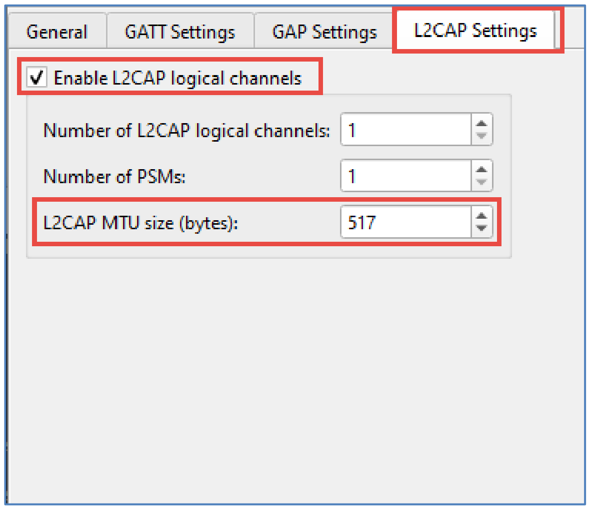
GATT

☑ Enable GATT database

Maximum remote clients connections:	1
Maximum remote servers connections:	0
Maximum attribute length (bytes):	512
Attribute MTU size (bytes):	517
RX PDU size (bytes):	517

GAP role

☑ Peripheral ☐ Broadcaster
☐ Central ☐ Observer



☑ Enable L2CAP logical channels

Number of L2CAP logical channels:	1
Number of PSMs:	1
L2CAP MTU size (bytes):	517

Bluetooth® OTA service

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 running it.

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

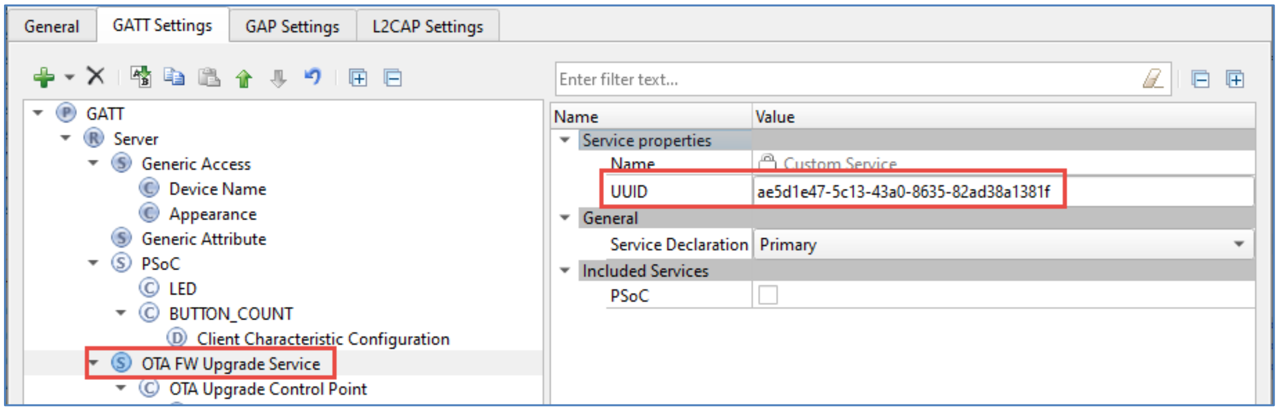
Note: *The UUIDs do not matter for the OTA service on the device, but the Windows application we will use to load new firmware uses hard-coded UUID values. So, you must match these in your Bluetooth® configuration to use the Windows application.*

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`

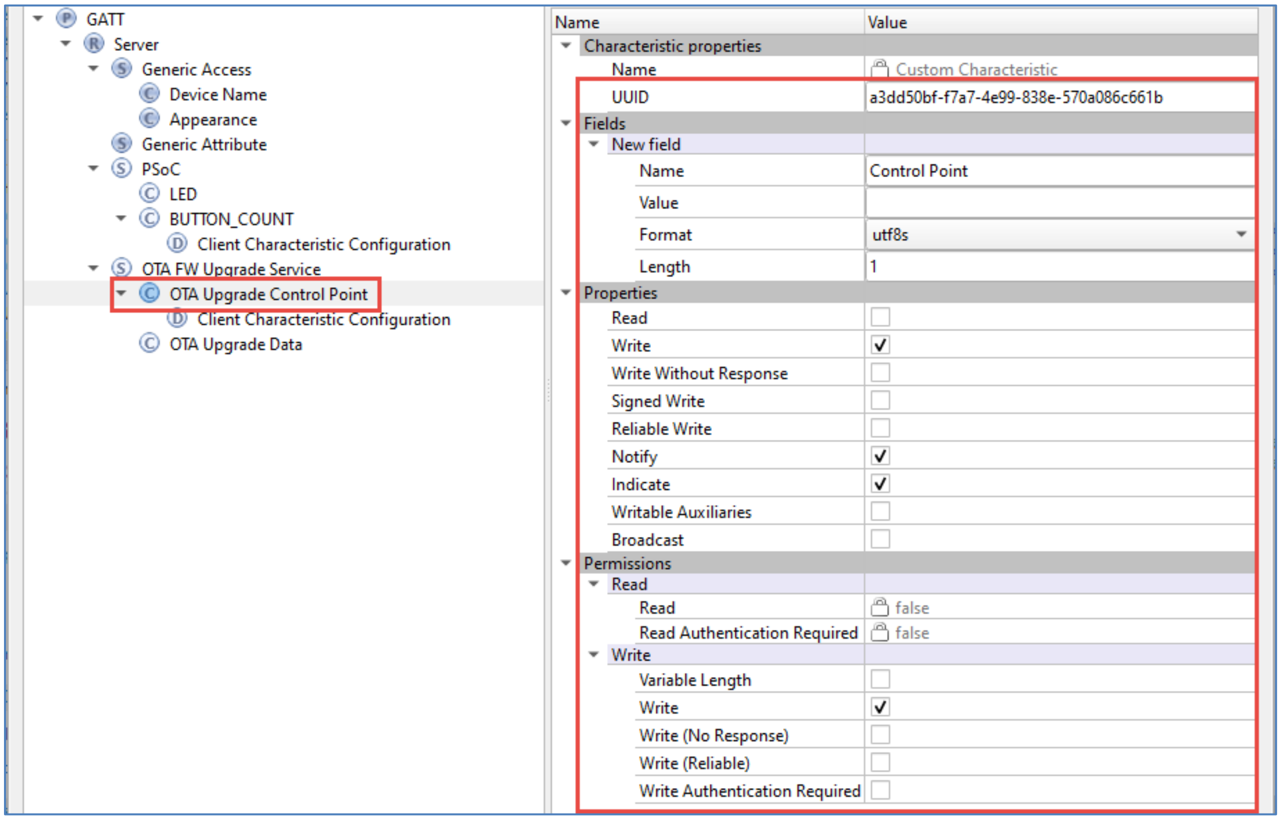
Name	UUID
OTA Upgrade Control Point	`a3dd50bf-f7a7-4e99-838e-570a086c661b`
OTA Upgrade Data	`a2e86c7a-d961-4091-b74f-2409e72efe26`

OTA FW Upgrade Service

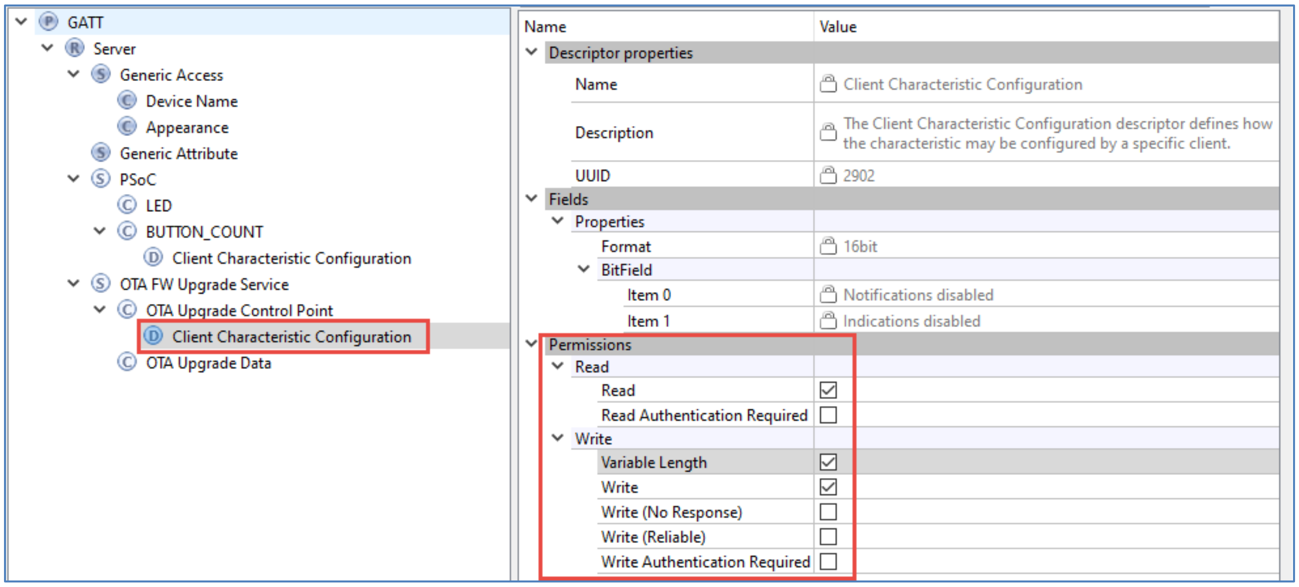


Name	Value
Service properties	
Name	Custom Service
UUID	ae5d1e47-5c13-43a0-8635-82ad38a1381f
General	
Service Declaration	Primary
Included Services	
PSoC	☐

OTA Upgrade Control Point



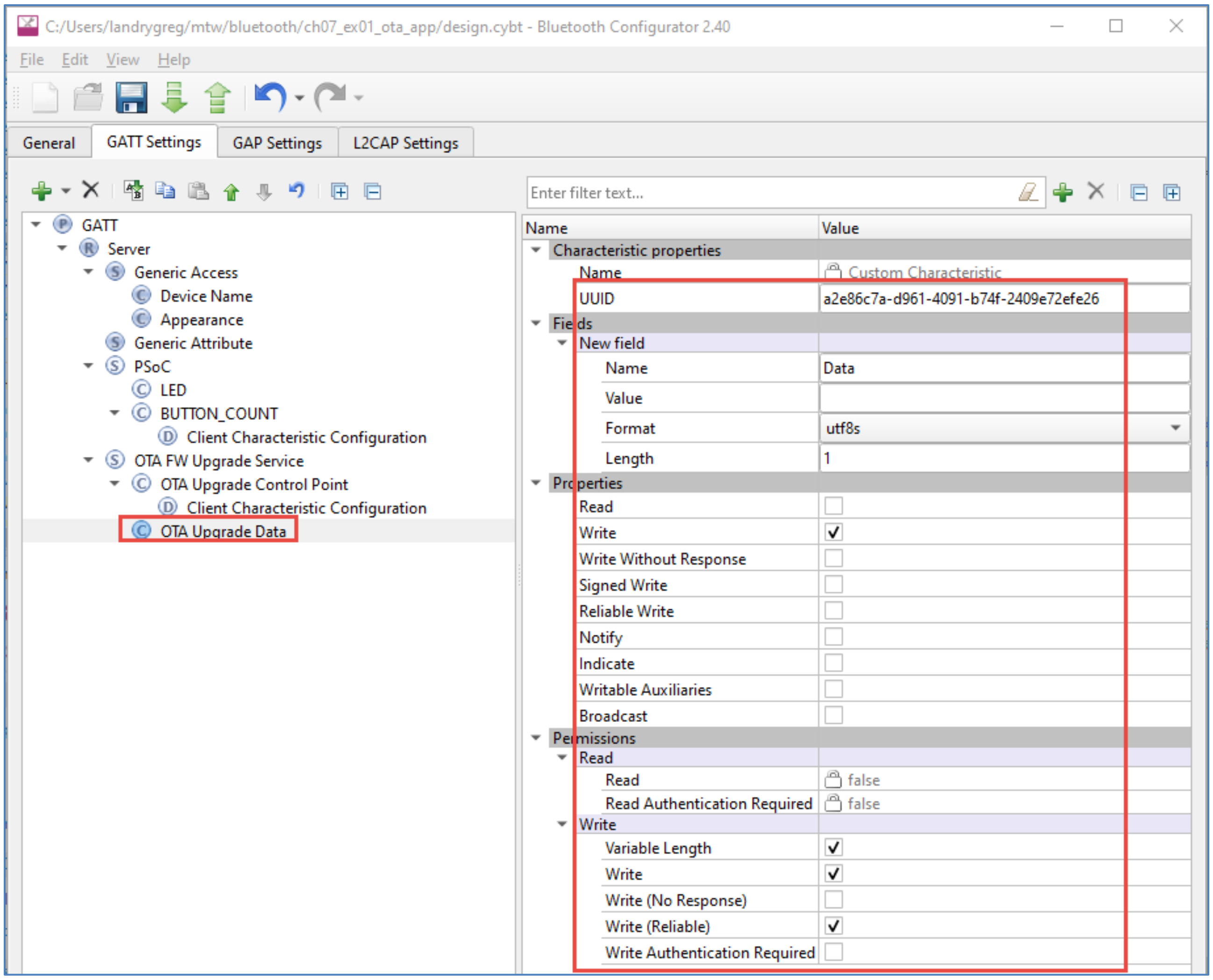
Name	Value
Characteristic properties	
Name	Custom Characteristic
UUID	a3dd50bf-f7a7-4e99-838e-570a086c661b
Fields	
New field	
Name	Control Point
Value	
Format	utf8s
Length	1
Properties	
Read	☐
Write	✓
Write Without Response	☐
Signed Write	☐
Reliable Write	☐
Notify	✓
Indicate	✓
Writable Auxiliaries	☐
Broadcast	☐
Permissions	
Read	
Read	false
Read Authentication Required	false
Write	
Variable Length	☐
Write	✓
Write (No Response)	☐
Write (Reliable)	☐
Write Authentication Required	☐



- 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	☐

OTA Upgrade Data



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

Over the air updates are accomplished using the library *ota-update*. It must be included as a dependency by the user application. The documentation for that library includes information on how to integrate the library into an application.

Note: The PSoC™ 6 + 43xxx OTA example uses ota-update version release-v2.x while the CYW20829 OTA example uses ota-update version release-v3.x.

The bootloader and the user application must have the same understanding of the memory layout. The information is contained inside flashmaps, which both applications must use. Example flashmaps are provided by the *ota-update* library in the directory *configs/flashmap*. The same flashmap file must be used when building both applications. The flashmaps that are available for various Infineon kits are shown below.

Target	Platform	Internal Flash size	Flashmap
CY8CKIT-062S2-43012 CY8CKIT-064B0S2-4343W CY8CPROTO-062-4343W CY8CEVAL-062S2-LAI-4373M2 CY8CEVAL-062S2-MUR-43439M2	PSOC_062_2M	2M	Default - psoc62_2m_ext_swap_single.json psoc62_2m_ext_overwrite_single.json psoc62_2m_int_overwrite_single.json psoc62_2m_int_swap_single.json
CY8CPROTO-062-4343W	PSOC_062_2M	2M	Default - psoc62_2m_int_swap_single.json
CY8CKIT-062-BLE	PSOC_062_1M	1M	Default - psoc62_1m_cm0_int_swap_single.json
CY8CPROTO-063-BLE CYBLE-416045-EVAL	PSOC_063_1M	1M	Default - psoc63_1m_cm0_int_swap_single.json
CY8CPROTO-062S3-4343W	PSOC_062_512K	512K	Default - psoc62_512k_xip_swap_single.json psoc62_512k_ext_overwrite_single.json psoc62_512k_ext_swap_single.json
CYW920829M2EVK-02	CYW20829	0K	Default - cyw20829_xip_swap_single.json cyw20829_xip_overwrite_single.json

9.1.2.3 Config file

There is a configuration file provided in the *ota-update* library that should be copied into the user application:

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

9.1.2.4 Makefile

The application's *Makefile* must be modified to configure OTA. The exact changes are different between the PSoC™ 6 + 43xxx and CYW20829 solutions and between different major versions of the library. The unique changes are listed separately below. See the *ota-update* library documentation for details on the required settings.

In the exercise, *Makefiles* with the required edits are provided for the the PSoC™ 6 + 43xxx and CYW20829 solutions.

Common

- A variable called `OTA_SUPPORT` is set to either 0 or 1. If it is set to 0, all the *Makefile* changes and the code changes in the .c/.h files required to support OTA are excluded. This allows the exercise to be build/programmed to work without OTA if desired. The Makefile uses the value of `OTA_SUPPORT` to determine what to exclude while the .c/.h files check to see if `ENABLE_OTA` is defined.
- Specify the application version. There are 3 levels of versioning: major, minor, and build. These can be used to verify the version of the firmware that is running on the device. The version is printed during initialization of the application.
- Set the `CY_IGNORE` variable to exclude libraries that are included as dependencies of the *ota-update* library that are not needed for Bluetooth® OTA updates.
- Exclude the ota-update library if OTA is disabled (i.e. if `OTA_SUPPORT` is 0).
- Define variables required for OTA such as `ENABLE_OTA`, `OTA_SUPPORT`, and `OTA_BT_SUPPORT`.
- Set variables for MCUBoot options. These must match the settings used in the MCUBoot project. This includes `OTA_FLASH_MAP` and `OTA_PLATFORM`.

PSoC™ 6 + 43xxx

Edits for OTA are in two sections – one just after the POSTBUILD settings and one at the end of the file.

- Add required items to the `COMPONENTS` variable such as, `OTA_BLUETOOTH`, and `OTA_PSOC_062`.
- Specify the core that the user application runs on (CM4)
- Define custom linker flags and specify a custom linker script based on the selected options.
- Provide code that creates the bootloadable image file and encrypts/signs the image.

CYW20829

Edits for OTA are in one section just after the POSTBUILD settings.

- Add a requirement for Python.
- Specify the path to the key file for signing the image.
- Set the option to not erase the flash when programming the application. This is necessary to prevent the MCUBoot project from being erased.
- Set the option to use the *hex* file during programming. This is necessary so that the file is placed in the correct location in flash. The default programming recipe for the CYW20829 uses the *bin* file and places it at the start of the flash. That would cause it to overwrite the MCUBoot project.
- Include Makefiles from the *ota-update* library.

9.1.2.5 Firmware

Updates are needed to the firmware to handle OTA setup and events are described below. As with the *Makefile*, some changes are different between the PSoC™ 6 + 43xxx and CYW20829 solutions and between different major versions of the library but the top-level changes are the same. The exercise templates already have the changes.

There are four sets of changes which are covered in more detail below. They are:

1. Add additional includes.
2. Initialize the OTA library in main before starting the stack. This also includes disabling the watchdog timer and enabling the serial flash QSPI interface.
3. Update the GATT event callback to handle confirmations that are received from the connected device after a notification is sent by the OTA service. The notification is sent when an OTA update has completed. The system is either rebooted or OTA is turned off, depending on the settings chosen.
4. Update the GATT write handler. Additional cases are added to handle write events to each of the OTA characteristics.

Includes

Additional header files must be included to have access to the *ota-update* library API and for application specific OTA functions that are defined. The OTA functionality uses the function `cy_rtos_delay_milliseconds` to create delays, so the *abstraction-rtos* library header is included. Finally, files to access the serial flash storage API are included. The exact list is different between the PSoC™ 6 + 43xxx and CYW20829 solutions, as shown here:

PSoC™ 6 + 43xxx

```
#ifdef ENABLE_OTA
       /* OTA related header files */
       #include "cy_ota_api.h"
       #include "ota.h"

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

       #include "cyabs_rtos.h" /* required for cy_rtos_delay_milliseconds */
#endif
```

CYW20829

```
#ifdef ENABLE_OTA
       /* OTA related header files */
       #include "cy_ota_api.h"
       #include "cy_ota_platform.h"
       #include "ota.h"

       #include "cybsp_smif_init.h"
       #include "ota_serial_flash.h"

       #include "cyabs_rtos.h" /* required for cy_rtos_delay_milliseconds */

#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.
2. The serial flash QSPI interface must be initialized. This is always required for the CYW20829 but is only needed for the PSoC™ 6 + CYW43xxx if external flash is used to store the secondary image during OTA.
3. The `ota_config` structure holding the OTA configuration must be initialized.
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.

The code used for each step above can be found in the Template projects. Look for code in the `#ifdef ENABLE_OTA` sections.

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 */
    {
     #ifdef ENABLE_OTA
       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_MIDDLEWARE, 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 */
       }
     #endif
     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 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.

For the PSoC™ 6 + CYW43xxx solution, the `app_bt_write_handler` calls a helper function called `app_bt_ota_write_handler` for all OTA writes. The helper function is located in the file *ota.c*. For the CYW20829 solution, the `app_bt_write_handler` function implements the OTA functionality flat instead of calling a helper function. In either case the OTA functionality is the same.

You should review the `app_bt_write_handler` function in *main.c* and the `app_bt_ota_write_handler` function in *ota.c* (for PSoC™ 6 + CYW43xxx) 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.

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_WARNING, NULL, NULL);
```

The level can be changed at any time during execution and can be set separately for OTA. For example, you can turn logging on for OTA messages with a level of `NOTICE` or higher:

```
cy_ota_set_log_level(CY_LOG_NOTICE);
```

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 uses `CYLF_MIDDLEWARE` for OTA messages.*

9.2 Over the air update application creation and programming

9.2.1 Create the user application

First, create a Bluetooth® application with the functionality that you require. Add OTA functionality to the application by following the description in section 9.1.2. As a reminder, the steps are:

1. Update Bluetooth® configurator settings for Maximum attribute length, Attribute MTU size, RX PDU size, and L2CAP MTU size.
2. Add the Bluetooth® OTA service.
3. Add the *ota-update* library and copy the config file to the application.
4. Update the *Makefile*.
5. Update the firmware (includes, initialization, GATT event handler, GATT write handler).
6. Optional: update logging settings.

The user application is created first so that the flash maps from the *ota-update* library will be available on disk when the MCUboot program is created. The user application will be programmed after creating and programming the MCUboot booloader project.

9.2.2 Create and program the MCUboot bootloader project

9.2.2.1 Getting the MCUboot project repo

The MCUboot project is provided in a GitHub repo. The first steps are to clone the repo, checkout a specific version, and update any submodules that are used. This can be done from a command terminal (e.g. modus-toolbox on Windows). The commands are:

```
git clone https://github.com/mcu-tools/mcuboot.git
cd mcuboot
git checkout v1.8.4-cypress
git submodule update --init --recursive
```

Note: *Version v1.8.4-cypress of the MCUboot library is used in the exercises. Newer versions may work but have not been tested.*

9.2.2.2 Installing required software

The next step is to get any software that is required to use the project. The commands are:

```
cd scripts
pip install -r requirements.txt
cd ..
```

The version of cysecuretools must be version 4.2.0 or above. Use the following command to check the version:

```
cysecuretools version
```

If the version is older than 4.2.0, use the following command to update it:

```
pip install --upgrade --force-reinstall cysecuretools
```

9.2.2.3 Copy the appropriate flashmap file

For the MCUboot project, the appropriate flashmap file will be copied from the *ota-update* library into the mcuboot/boot/cypress directory. The following commands illustrate howto copy the flashmap file. See section 9.1.2.2 or refer to the *ota-update* library documentation for available flashmaps.

```
cd boot/cypress
cp <workspace>/mtb_shared/ota-update/<version>/configs/flashmap/<file>.json ./
```

9.2.2.4 Build the MCUboot project

For the build to work it must know the toolchain path. This can either be done by editing the file *toolchains.mk* file in the *mcuboot/boot/cypress* directory or by overriding the value of `TOOLCHAIN_PATH` on the command line. For example, the line in *toolchains.mk* for GCC_ARM for ModusToolbox™ 3.1 would look like this:

```
TOOLCHAIN_PATH ?= c:/Users/$(USERNAME)/ModusToolbox/tools_3.1/gcc
```

Now that the MCUboot project is configured, it is time to build it. You must run this command from the *mcuboot/boot/cypress* directory. The command for the CY8CKIT-062-43012 using internal flash is:

```
make clean app APP_NAME=MCUBootApp PLATFORM=PSOC_062_2M
FLASH_MAP=./psoc62_2m_ext_swap_single.json
```

If you are using the CYW920829M2EVK-02 kit, the command has some additional arguments to specify that the application is stored in external flash and to select the UART to be used.

```
make clean app APP_NAME=MCUBootApp PLATFORM=CYW20829 USE_CUSTOM_DEBUG_UART=1
USE_EXTERNAL_FLASH=1 USE_XIP=1 FLASH_MAP=./cyw20829_xip_swap_single.json
```

Note: *The make command should be entered as a single command. It is shown on multiple lines due to space constraints.*

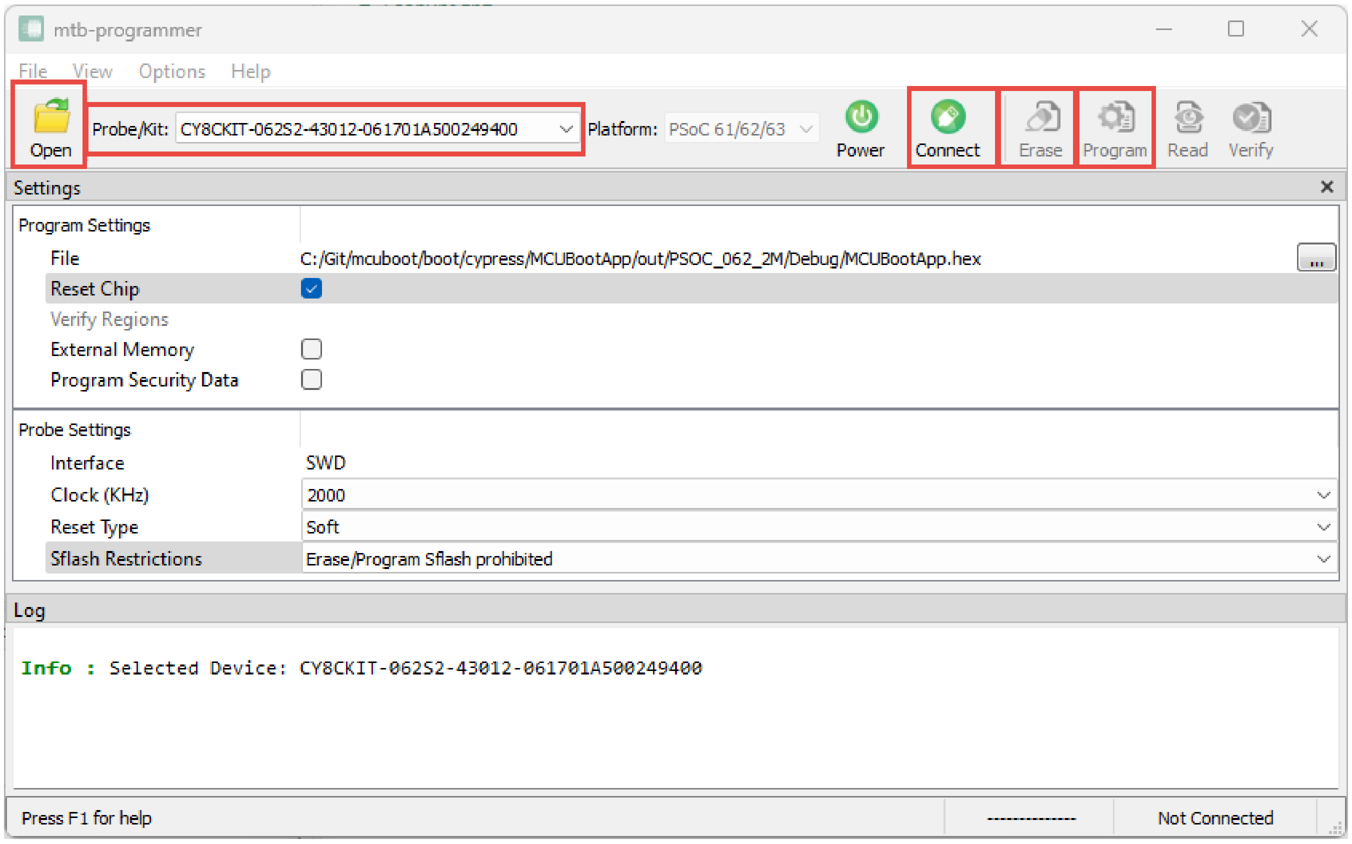
9.2.2.5 Program the MCUboot project

Once the project has been built, the hex file can be programmed into the device using ModusToolbox™ Programmer (mtb-programmer) or by executing OpenOCD from the command line. Both options are shown below.

Using ModusToolbox™ Programmer

The steps to use ModusToolbox™ Programmer to program the device are:

1. Run mtb-programmer.
2. Connect the kit to the computer.
3. Click Open and navigate to the hex file (*<mcuboot>/boot/cypress/MCUBootApp/out/<PLATFORM>/Debug/MCUBootApp.hex*).
4. Click the drop-down next to Probe/Kit and select your kit from the list.
5. Click Connect.
6. Click Erase.
7. Click Program.
8. Click Disconnect.



Probe/Kit: CY8CKIT-062S2-43012-061701A500249400
Platform: PSoC 61/62/63

Program Settings	
File	C:/Git/mcuboot/boot/cypress/MCUBootApp/out/PSOC_062_2M/Debug/MCUBootApp.hex
Reset Chip	☑
Verify Regions	
External Memory	☐
Program Security Data	☐
Probe Settings	
Interface	SWD
Clock (KHz)	2000
Reset Type	Soft
Sflash Restrictions	Erase/Program Sflash prohibited

```
Info : Selected Device: CY8CKIT-062S2-43012-061701A500249400
```

Using OpenOCD from the command line

If you want to use OpenOCD directly from the command line, there are three commands:

1. Set the OpenOCD path

```
export OPENOCD=C:/Users/${USERNAME}/ModusToolbox/tools_3.1/openocd
```

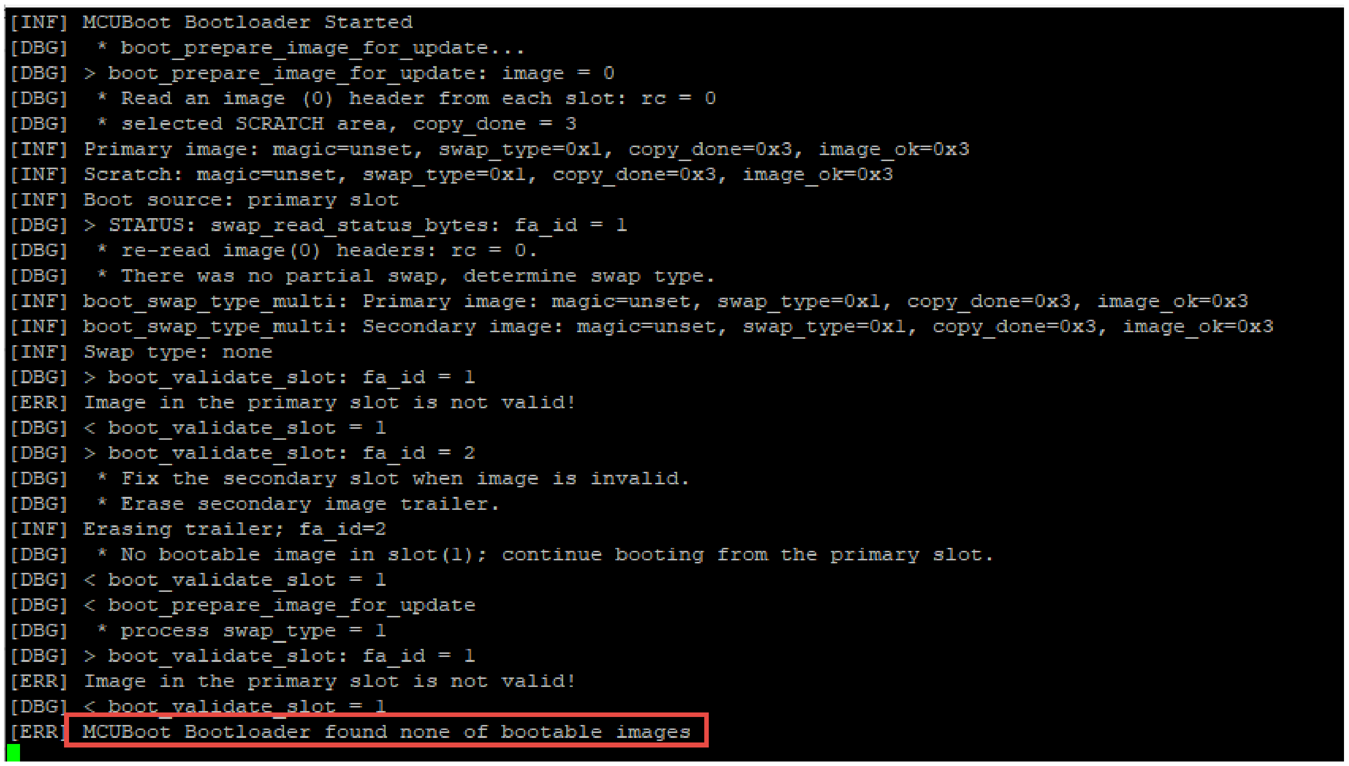
2. Erase the Device

```
${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
"$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
SMIF_BANKS { 0 {addr 0x60000000 size 0x800000 psize 0x100 esize 0x40000} }" -f
$OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash erase_address
0x60000000 0x100000; shutdown"
```

3. Program MCUBoot

```
${OPENOCD}/bin/openocd -s "$OPENOCD/scripts" -f
"$OPENOCD/scripts/interface/kitprog3.cfg" -c "set ENABLE_ACQUIRE 0" -c "set
SMIF_BANKS { 0 {addr 0x60000000 size 0x100000 psize 0x100 esize 0x1000} }" -f
$OPENOCD/scripts/target/cyw20829.cfg -c "init; reset init; flash write_image
"MCUBootApp/out/CYW20829/Debug/MCUBootApp.hex" 0x00000000; init; reset init;
reset run; shutdown"
```

After programming completes, the UART output should look like the following. Observe the messages indicate that there is no valid bootloadable image. This is correct since we have not yet programmed the user application into the device.



```
[INF] MCUBoot Bootloader Started
[DBG]  * boot_prepare_image_for_update...
[DBG] > boot_prepare_image_for_update: image = 0
[DBG]  * Read an image (0) header from each slot: rc = 0
[DBG]  * selected SCRATCH area, copy_done = 3
[INF] Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: primary slot
[DBG] > STATUS: swap_read_status_bytes: fa_id = 1
[DBG]  * re-read image(0) headers: rc = 0.
[DBG]  * There was no partial swap, determine swap type.
[INF] boot_swap_type_multi: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] boot_swap_type_multi: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: none
[DBG] > boot_validate_slot: fa_id = 1
[ERR] Image in the primary slot is not valid!
[DBG] < boot_validate_slot = 1
[DBG] > boot_validate_slot: fa_id = 2
[DBG]  * Fix the secondary slot when image is invalid.
[DBG]  * Erase secondary image trailer.
[INF] Erasing trailer; fa_id=2
[DBG]  * No bootable image in slot(1); continue booting from the primary slot.
[DBG] < boot_validate_slot = 1
[DBG] < boot_prepare_image_for_update
[DBG]  * process swap_type = 1
[DBG] > boot_validate_slot: fa_id = 1
[ERR] Image in the primary slot is not valid!
[DBG] < boot_validate_slot = 1
[ERR] MCUBoot Bootloader found none of bootable images
```

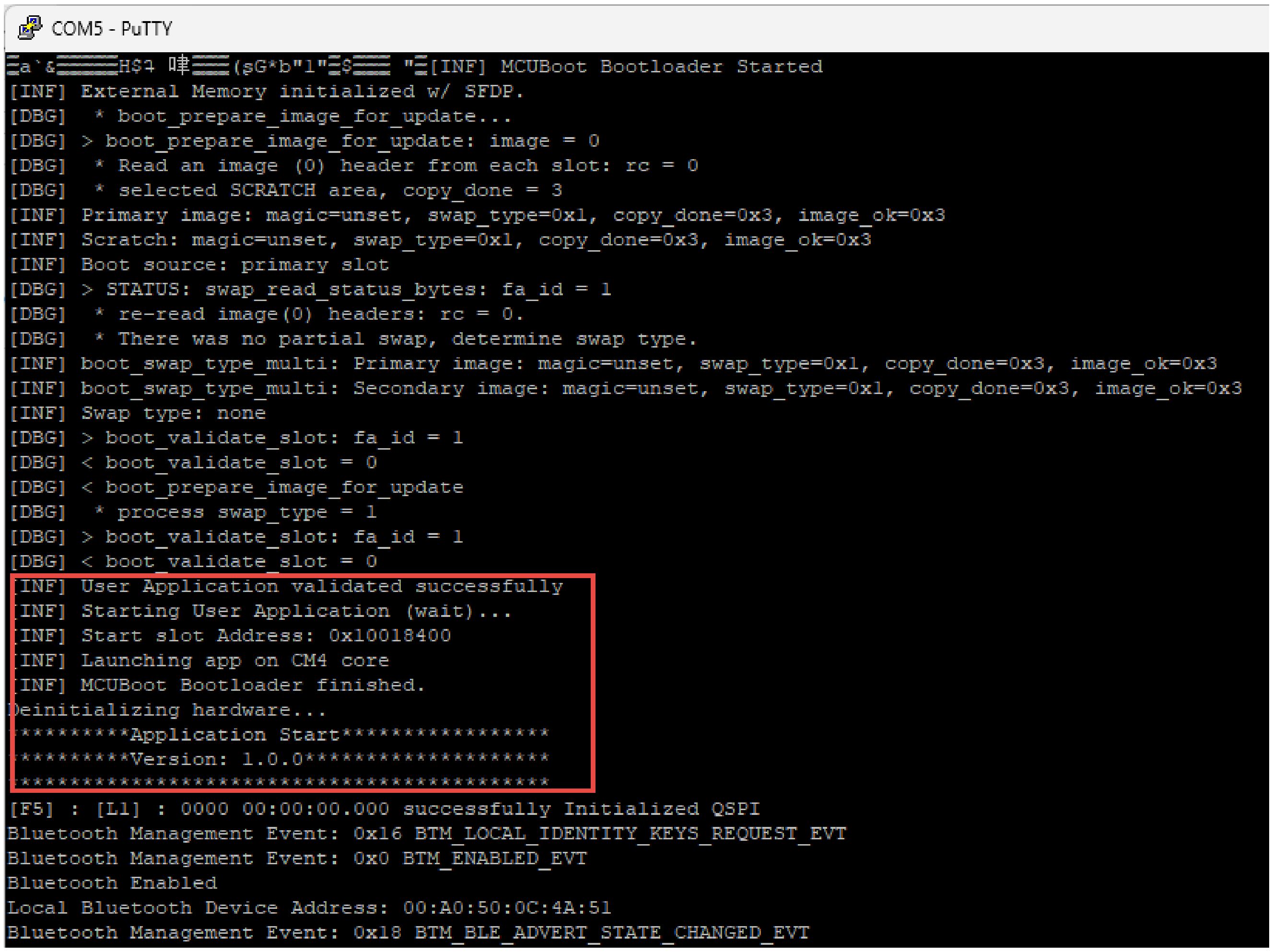
9.2.3 Programming the user application

Once the MCUboot project has been programmed to the device, it is time to program the user application. If you are using a CYW20829 device, this should be done using `make program` from the command line. Do NOT program the application from Eclipse or VS Code for the CYW20829 since they are setup to erase the flash before programming the device.

If you get a build error message that says `cysecuretools` can't be found, you will need add the appropriate location to the user's environment variable for the `Path`. For example:

C:\Users\<username>\ModusToolbox\tools_3.1\python\Scripts

When the user application has been programmed, the device will reset and the MCUBoot project will run. It will find a valid image in the primary slot and will launch it as shown below.



```
COM5 - PuTTY
☰a`&☰☰☰H$♩ 啡☰☰(şG*b"l"☰$☰☰ "☰[INF] MCUBoot Bootloader Started
[INF] External Memory initialized w/ SFDP.
[DBG]  * boot_prepare_image_for_update...
[DBG] > boot_prepare_image_for_update: image = 0
[DBG]  * Read an image (0) header from each slot: rc = 0
[DBG]  * selected SCRATCH area, copy_done = 3
[INF] Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Boot source: primary slot
[DBG] > STATUS: swap_read_status_bytes: fa_id = 1
[DBG]  * re-read image(0) headers: rc = 0.
[DBG]  * There was no partial swap, determine swap type.
[INF] boot_swap_type_multi: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] boot_swap_type_multi: Secondary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: none
[DBG] > boot_validate_slot: fa_id = 1
[DBG] < boot_validate_slot = 0
[DBG] < boot_prepare_image_for_update
[DBG]  * process swap_type = 1
[DBG] > boot_validate_slot: fa_id = 1
[DBG] < boot_validate_slot = 0
[INF] User Application validated successfully
[INF] Starting User Application (wait)...
[INF] Start slot Address: 0x10018400
[INF] Launching app on CM4 core
[INF] MCUBoot Bootloader finished.
Deinitializing hardware...
*********Application Start*****************
*********Version: 1.0.0********************
*******************************************
[F5] : [L1] : 0000 00:00:00.000 successfully Initialized QSPI
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:0C:4A:51
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
```

9.3 Secure OTA update

The MCUboot project used in the exercise contains public key encryption to sign the image. The private key is used to sign the build output of the user application project as a post-build step. The public key (which is included during the build of the MCUboot project) is used to validate the image when it is loaded during the OTA process. This ensures that only an image from a trusted source can be installed onto the device using OTA.

The keys used by the MCUboot library are in *mcuboot/boot/cypress/keys*. The file *cypress-test-ec-p256.pem* is the private key and the file *cypress-test-ec-p356.pub* is the public key. Those file names are specified in the *Makefile* for the MCUBoot project. The user application template has its own copy of the keys stored in the *keys* directory. Each project has a copy of the keys despite only using one of the keys, as described above.

9.3.1 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 project in the scripts directory.*

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

```
python imgtool.py keygen -k cypress-test-ec-p256.pem -t ecdsa-p256
python imgtool.py getpub -k cypress-test-ec-p256.pem >> cypress-test-ec-p256.pub
```

If you generate new keys, be sure to use the same key pair in both the MCUBoot project and the user application. Otherwise, OTA will fail. The commands above will generate the keys in the *scripts* directory. You must copy your new keys to the *mcuboot/boot/cypress/keys* directory and the user application's *keys* directory for them to be used.

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).

If you don't use the name *cypress-test-ec-p256* for the key files, you must update the name in the MCUBoot project and the application project Makefiles. The variables are:

Project	Varaible Name	Default Value
MCUBoot	`SIGN_KEY_FILE`	`cypress-test-ec-p256`
PSoC™ 6 + 43xxx user application	`MCUBOOT_KEY_FILE`	`cypress-test-ec-p256.pem`
CYW20829 user application	`CY_SIGN_KEY_PATH`	`./keys/cypress-test-ec-p256.pem`

9.3.2 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 secure services such as secure boot and secure 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.

9.4 Bluetooth® OTA update process

A Windows application is provided that can be used to perform the Bluetooth® OTA update process. However, this application is provided only as an example. For a real product, the developer typically provides their own application either as a stand-alone app or as part of an existing app. Many times, 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

Note: *In newer versions of ModusToolbox™, The OTA update application is available in from the Bluetooth® technology pack.*

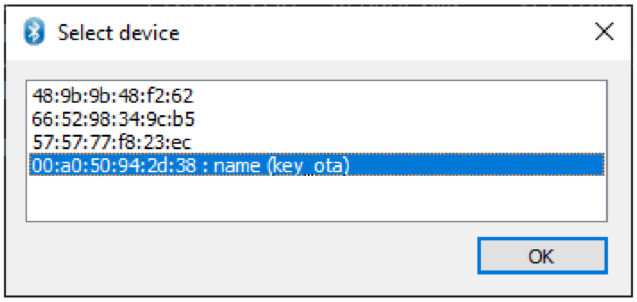
To use the Windows peer application, you must first copy the **.bin* file from the build Debug directory of the application into the same directory as the Windows peer application. Then run the application with the **.bin* file provided as an argument. For example, from modus-shell:

For PSoC™ 6 applications: `./WsOtaUpgrade.exe app.bin`

For CYW20829 applications: `./WsOtaUpgrade.exe app.signed.bin`

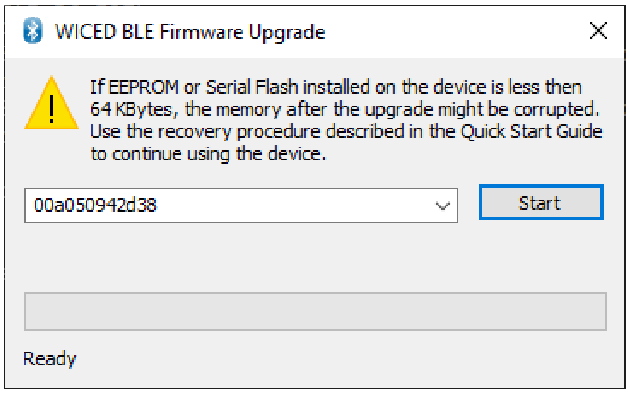
Note: *You can also just drag the .bin file onto WSOtaUpgrade.exe from a Windows file explorer window.*

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



```
48:9b:9b:48:f2:62
66:52:98:34:9c:b5
57:57:77:f8:23:ec
00:a0:50:94:2d:38 : name (key_ota)
```

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

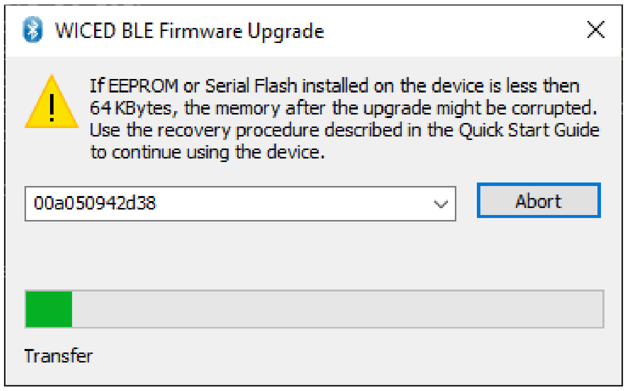

WICED BLE Firmware Upgrade

If EEPROM or Serial Flash installed on the device is less then 64 KBytes, the memory after the upgrade might be corrupted. Use the recovery procedure described in the Quick Start Guide to continue using the device.

00a050942d38

Ready

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. It may also take some time (~20 seconds) for the update to start, so only click the **Start** button once.

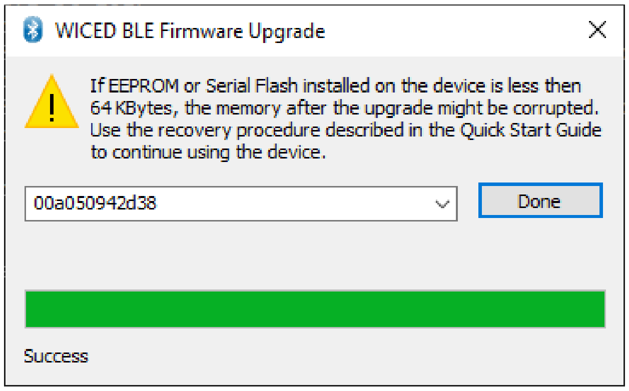

WICED BLE Firmware Upgrade

If EEPROM or Serial Flash installed on the device is less then 64 KBytes, the memory after the upgrade might be corrupted. Use the recovery procedure described in the Quick Start Guide to continue using the device.

00a050942d38

Transfer

When it finishes, the window shows "Success" at the bottom if the update worked. Click **Done** to close the window.

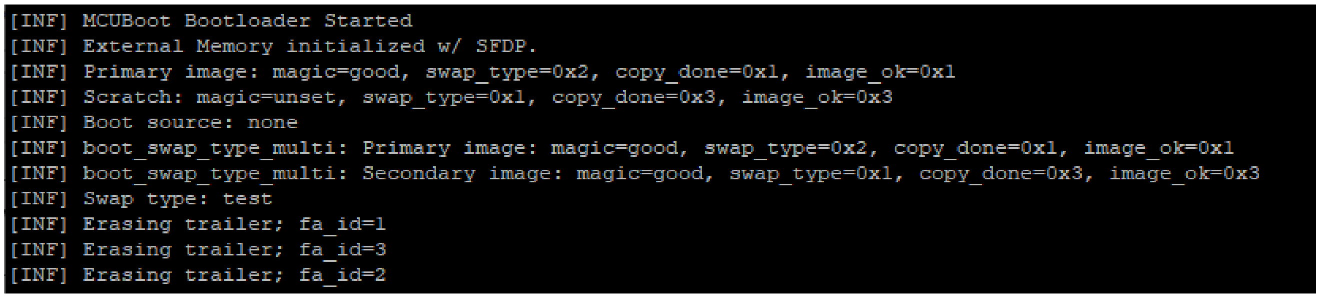

WICED BLE Firmware Upgrade

If EEPROM or Serial Flash installed on the device is less then 64 KBytes, the memory after the upgrade might be corrupted. Use the recovery procedure described in the Quick Start Guide to continue using the device.

00a050942d38

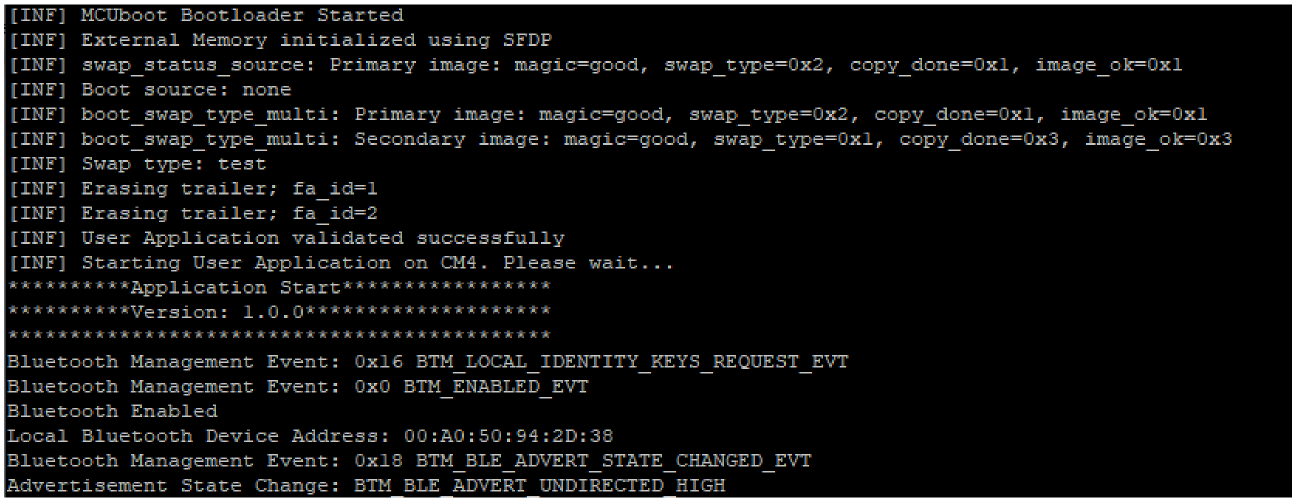
Success

When 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
```

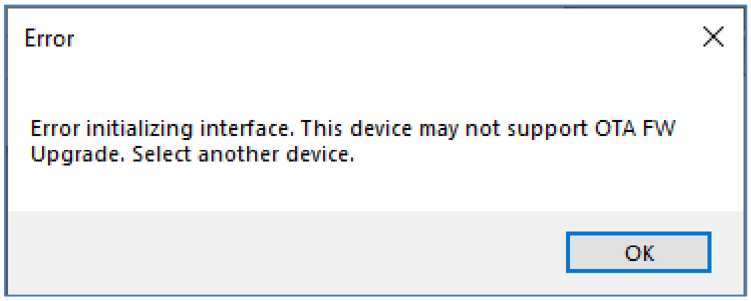
Once the new image has been validated and copied, the new application will start as normal:



```
[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
```

9.4.1 Troubleshooting

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



Error initializing interface. This device may not support OTA FW Upgrade. Select another device.

This message can be caused by several different issues. Try the following methods to 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.

9.5 Exercises

Exercise 1: Bluetooth® LE application with OTA update capability

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

1. Create a Bluetooth® project using a template that has been modified to support OTA firmware updates.
2. Create, update and program the MCUboot bootloader project onto the device.
3. Program the Bluetooth® application project to the device and test the Bluetooth® functionality.
4. Modify the Bluetooth® project's functionality and load it onto the device 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 BSP you are using.

On the application template page, use the **Browse** button to start from the template in *Templates/ch09_ex01_ota_psoc or Templates/ch09_ex01_ota_20829* depending on the kit you are using. Keep the application name the same.

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

☐ 3. On the **General LE** 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 *cy_ota_config.h* file from the *ota-update* library's *configs* directory to the application's *configs* directory. The default settings in this file are OK so we won't need to change them.

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

☐ 8. The *Makefile* already has the changes needed for OTA. Review the file to understand what 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 for the code are already done. Review the code to understand the changes that were described earlier in this chapter.

Clone and configure the MCUboot projectand program it to the kit

☐ 1. Follow the steps in section 9.2.2 to get the MCUboot project, configure it, and program it to the kit. As a reminder, you must:

   a. Clone the *mcuboot* repo from GitHub and update submodules.
   b. Install any required software (this only needs to be done once).
   c. Copy the correct flashmap from the *ota-update* library.
   d. Update the path to the tools and build the project.
   e. Program the project to the kit using ModusToolbox™ Programmer or OpenOCD from the command line.

Program the Bluetooth® application project and test its functionality

☐ 1. Now that we have the bootloader programmed, we need to program the Bluetooth® application project. Program the *ch09_ex01_ota_psoc* or *ch09_ex01_ota_20829* application from the command line (e.g. `make program` command in modus-shell).

Note: *If you are using a CYW20829 device and you program the application using the launches from Eclipse or VS Code, it will erase the MCUboot project from flash before programming the Bluetooth® application, so it will not work. You will need to re-program the MCUboot project first and then program the Bluetooth® application from the ModusToolbox™ command line.*

☐ 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 user application, open the *Makefile* and change OTA_*APP_VERSION_MINOR* to 1.

☐ 2. Open the 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 NOT 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_<bsp name>/Debug.* The file name is *app.bin* for PSoC™ 6 and *app.signed.bin* for the CYW20289.

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

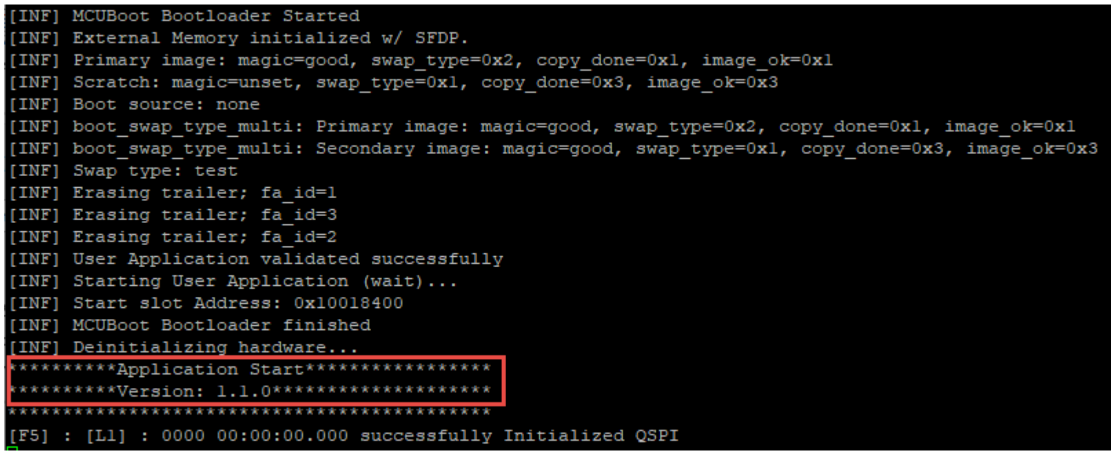
Follow the instructions in section 9.4. See the troubleshooting information in 9.4.1 if you run into issues.

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
```

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

Published by

Infineon Technologies AG

81726 Munich, Germany