

Chapter 7: 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

7.1	Over the air update firmware architecture	2
7.1.1	CM0+ project.....	2
7.1.2	CM4 project.....	5
7.2	Secure OTA update	13
7.2.1	Generate a public/private key pair	13
7.2.2	Enable secure OTA update in the CM0+ MCUboot application	13
7.2.3	Enable image signing in the CM4 project	14
7.3	Bluetooth® OTA update Windows peer application	14
7.3.1	Troubleshooting.....	17
7.4	Exercises	18
	Exercise 1: Bluetooth® LE application with OTA update capability	18
	Exercise 2: Secure OTA update	21

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR) ; make build
<i>Italics</i>	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 .

7.1 Over the air update firmware architecture

The OTA process involves both the CM0+ and the CM4 so you will need firmware for both MCUs. This can either be done as a single dual-core application or it can be done as two individual applications. 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.

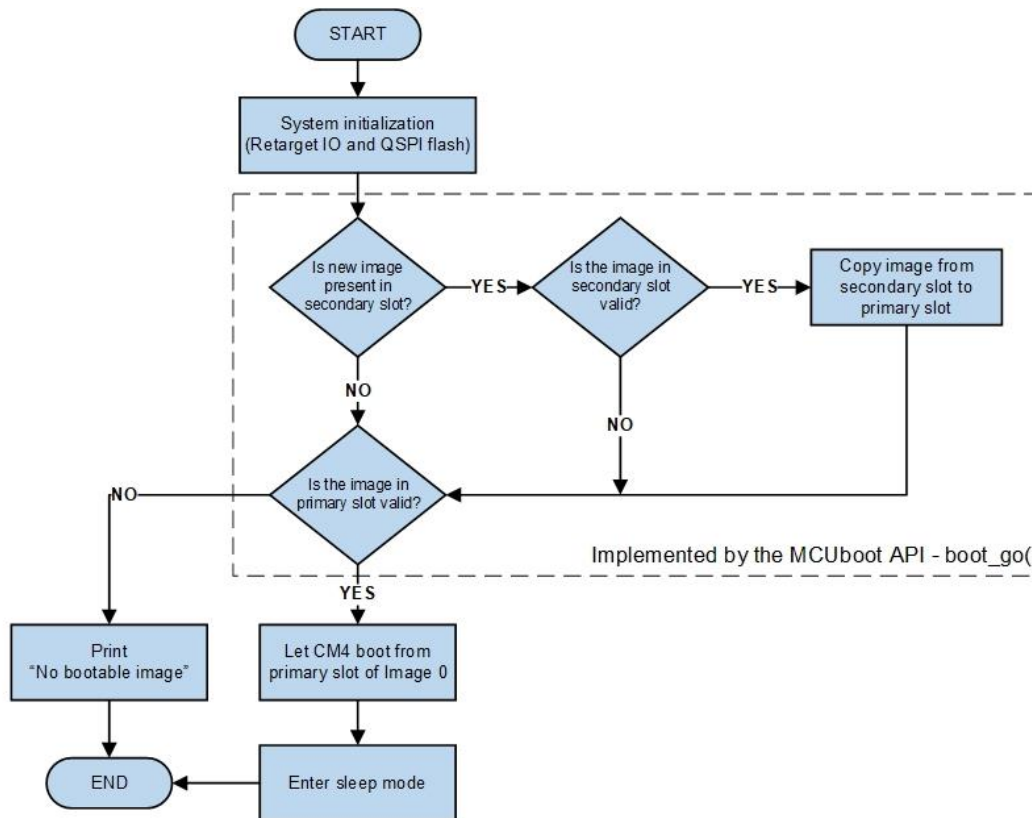
7.1.1 CM0+ project

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: The MCUboot library does not appear in the library manager. Rather, it is included via an mtb file in the deps directory of the project that runs on the CM0+.



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

7.1.1.1 CM0+ MCUboot example

The easiest way to get the CM0+ application is to copy it 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 a few 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. If you are using an IDE, you may not see the README.md file appear because it is at a directory level above the individual CM0+ and CM4 projects. You can either open it with a Markdown viewer locally or you can view it on GitHub at:

<https://github.com/Infineon/mtb-example-psoc6-mcuboot-basic>

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

- Modify the memory layout to match the CM4 Bluetooth® project

This is done by changing the values on 3 lines in the *shared_config.mk* file:

```
ifeq ($(USE_EXT_FLASH), 1)
MCUBOOT_SLOT_SIZE=0x1C0000
else
MCUBOOT_SLOT_SIZE=0xEE000
endif.
.
.
MCUBOOT_SCRATCH_SIZE=0x4000
```

Note: The values must match the values used in the CM4 project for *CY_BOOT_PRIMARY_1_SIZE* and *CY_BOOT_SCRATCH_SIZE*. In our examples, those values are defined in the file *ota.mk*.

- Copy the *config* directory from the *mcuboot* library into the CM0+ project's root directory.
The directory can be found in *mtb_shared/mcuboot/<version>/boot/cypress/MCUBootApp*. The entire contents should be copied to the CM0+ project.
- Edit the *app.mk* file in the CM0+ project's root directory to change the path to the *config* directory to use the local configuration files.

Old: \$(MCUBOOTAPP_PATH)/config\

New: ./config\

- Disable image signature checking (this will be enabled when we do secure OTA updates)
This is done by commenting out 2 lines in the CM0+ project's *config/mcuboot_config/mcuboot_config.h* file.

```
//#define MCUBOOT_SIGN_EC256
//#define NUM_ECC_BYTES (256 / 8) // P-256 curve size in bytes
```

- Change setting to use external flash to store the secondary image and enable the MCUBoot swap feature.

This is done by editing two lines in the CM0+ project's Makefile:

```
SWAP_UPGRADE ?= 1
USE_EXT_FLASH ?= 1
```

- Install tools for creating a bootloadable image

Open a command line terminal (modus-shell for Windows), and navigate to the *mtb_shared/mcuboot/<version>/scripts* directory. Run the command:

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

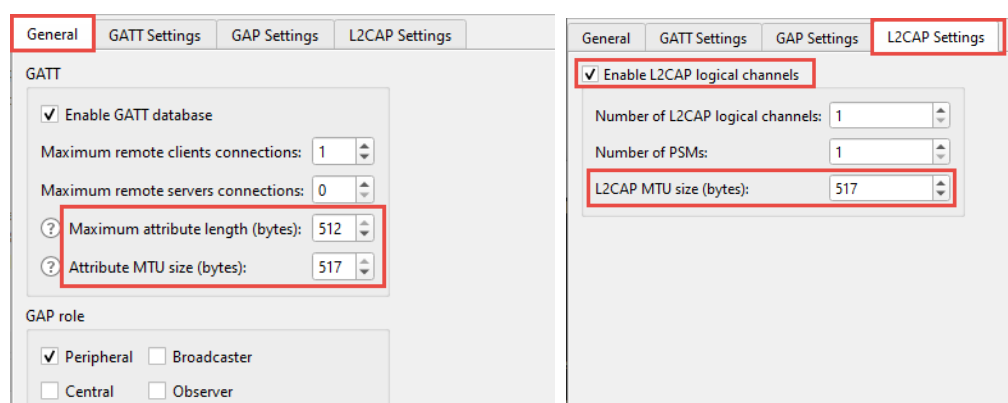
7.1.2 CM4 project

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.

7.1.2.1 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.



The image shows two side-by-side screenshots of the Bluetooth Configurator application. The left screenshot is the 'GATT' settings tab, where 'Maximum attribute length (bytes)' is set to 512 and 'Attribute MTU size (bytes)' is set to 517. The right screenshot is the 'L2CAP Settings' tab, where 'Enable L2CAP logical channels' is checked and 'L2CAP MTU size (bytes)' is set to 517. Red boxes highlight these specific settings in both screenshots.

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.

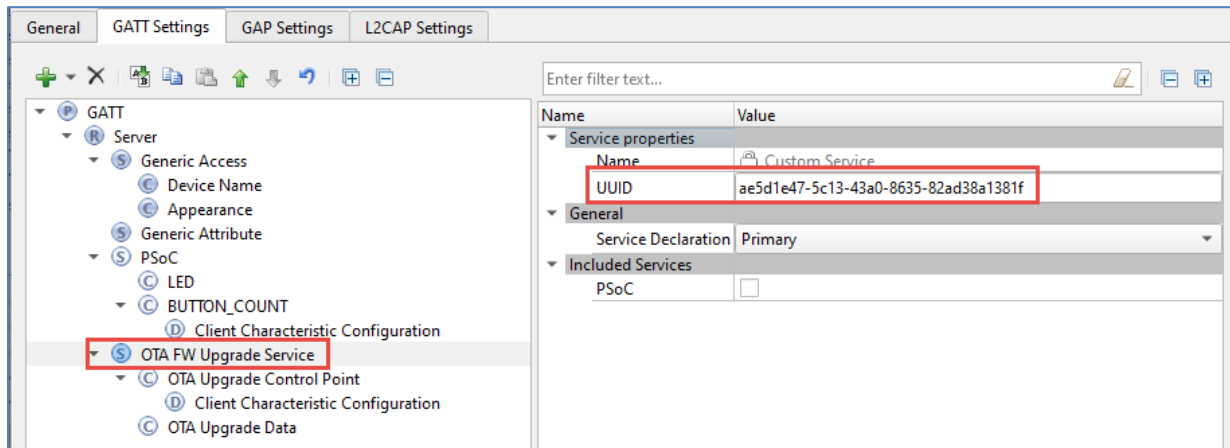
Note: The ModusToolbox™ any cloud OTA library relies on the service and characteristic names so you must match them exactly with what is shown below.

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

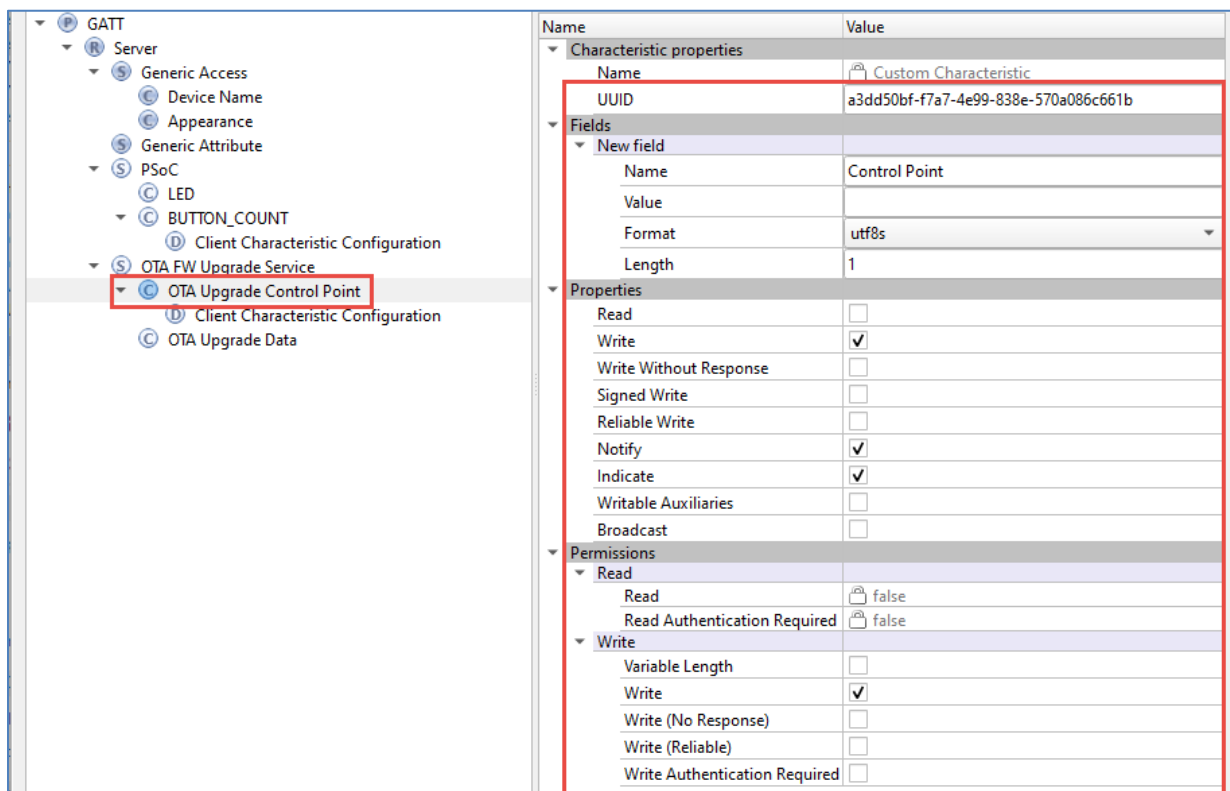
OTA FW Upgrade Service



The screenshot shows the GATT Settings window with the 'OTA FW Upgrade Service' selected in the left-hand tree. The right-hand pane displays the service properties:

Name	Value
Service properties	
Name	Custom Service
UUID	ae5d1e47-5c13-43a0-8635-82ad38a1381f
General	
Service Declaration	Primary
Included Services	
PSoC	<input type="checkbox"/>

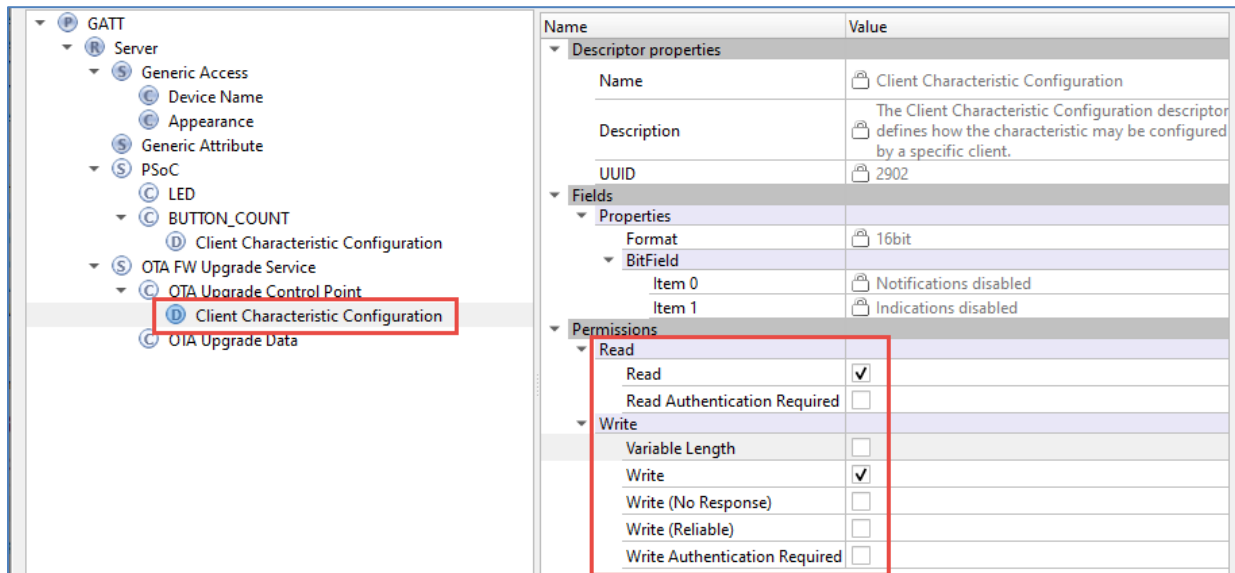
OTA Upgrade Control Point Characteristic



The screenshot shows the GATT Settings window with the 'OTA Upgrade Control Point' selected in the left-hand tree. The right-hand pane displays the characteristic properties:

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	<input type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write Without Response	<input type="checkbox"/>
Signed Write	<input type="checkbox"/>
Reliable Write	<input type="checkbox"/>
Notify	<input checked="" type="checkbox"/>
Indicate	<input checked="" type="checkbox"/>
Writable Auxiliaries	<input type="checkbox"/>
Broadcast	<input type="checkbox"/>
Permissions	
Read	
Read	false
Read Authentication Required	false
Write	
Variable Length	<input type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write (No Response)	<input type="checkbox"/>
Write (Reliable)	<input type="checkbox"/>
Write Authentication Required	<input type="checkbox"/>

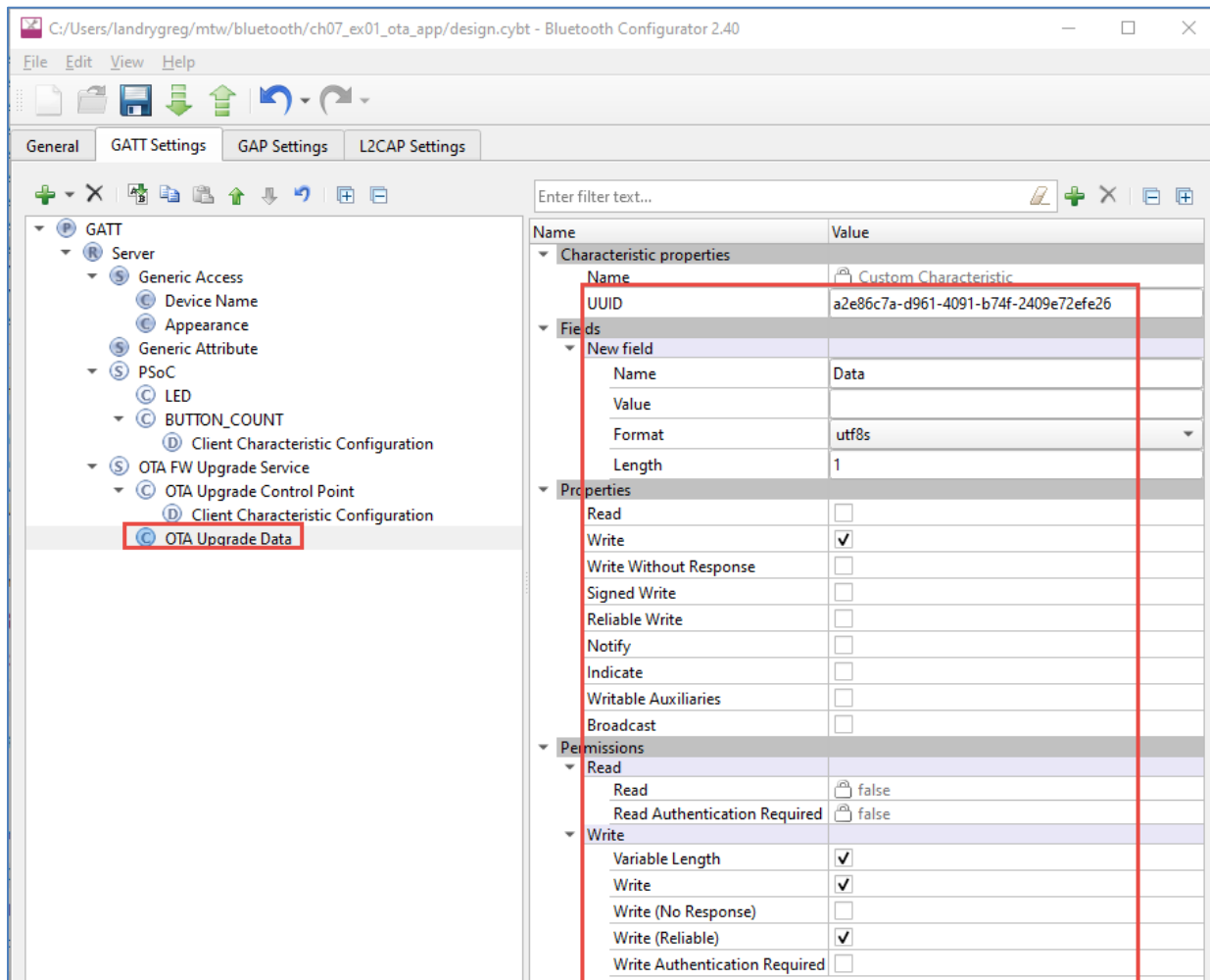
OTA Upgrade Control Point CCCD



The screenshot shows the Bluetooth Configurator interface. On the left, the tree view shows the 'Client Characteristic Configuration' selected under the 'OTA Upgrade Control Point'. On the right, the properties table is displayed:

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	<input checked="" type="checkbox"/>
Read Authentication Required	<input type="checkbox"/>
Write	
Variable Length	<input type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write (No Response)	<input type="checkbox"/>
Write (Reliable)	<input type="checkbox"/>
Write Authentication Required	<input type="checkbox"/>

OTA Upgrade Data Characteristic

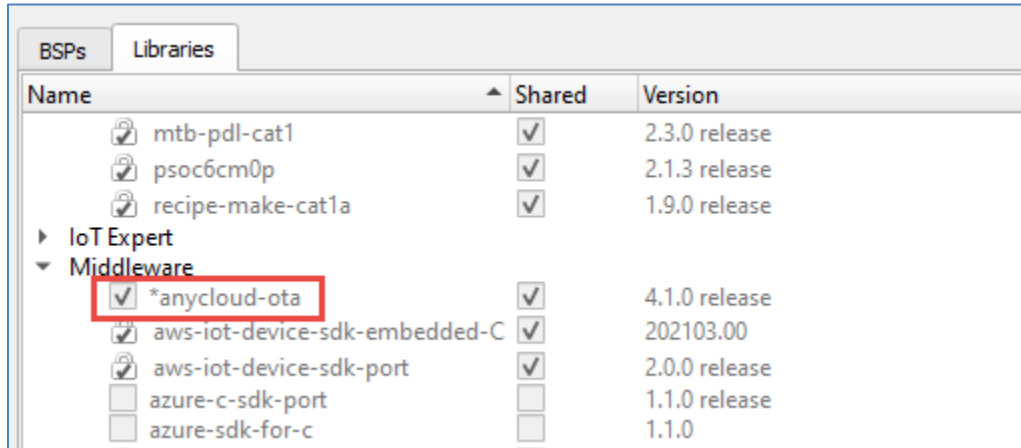


The screenshot shows the Bluetooth Configurator interface. On the left, the tree view shows the 'OTA Upgrade Data' selected under the 'Client Characteristic Configuration'. On the right, the properties table is displayed:

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	<input type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write Without Response	<input type="checkbox"/>
Signed Write	<input type="checkbox"/>
Reliable Write	<input type="checkbox"/>
Notify	<input type="checkbox"/>
Indicate	<input type="checkbox"/>
Writable Auxiliaries	<input type="checkbox"/>
Broadcast	<input type="checkbox"/>
Permissions	
Read	
Read	false
Read Authentication Required	false
Write	
Variable Length	<input checked="" type="checkbox"/>
Write	<input checked="" type="checkbox"/>
Write (No Response)	<input type="checkbox"/>
Write (Reliable)	<input checked="" type="checkbox"/>
Write Authentication Required	<input type="checkbox"/>

7.1.2.2 OTA Library

On PSoc 6 + 43012 devices, over the air updates are accomplished using the library *anyccloud-ota*. The documentation for that library includes information on how to integrate the library into an application and includes lots of very useful code examples.



7.1.2.3 Makefile

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

1. Specify the application version (optional, but useful in verifying the correct image is running).
2. Specify the CM4 as the processor for the project.
3. Define variables `OTA_SUPPORT` and `OTA_USE_EXTERNAL_FLASH` and add `OTA_BLUETOOTH` to the `COMPONENTS` variable.
4. Exclude the libraries that are dependencies of the *anyccloud-ota* but are not needed for Bluetooth®.
5. Set variables for MCUBoot. These must match the values used in the MCUBoot application.
6. Define custom linker flags and specify a custom linker script.
7. Provide code that creates the boot loadable image file and encrypts/signs the image if requested.

As you will see in the exercises, an `app.mk` file is provided in the Templates directory to accomplish the required modifications. That file can just be included in the CM4 project and can be included by the Makefile like this:

```
-include ./ota.mk
```

Note: *This include line must be done at the end of the advanced configuration section of the Makefile. Specifically, it must be done after the `COMPONENTS`, `DEFINES`, and `LINKER_SCRIPT` variables are set in the Makefile so that the custom script values are not overridden. The simplest thing to do is to place the include line just before the Paths section in the Makefile.*

7.1.2.4 Firmware

Updates 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

The OTA library functions require a header file to be specified:

```
/* OTA related header files */  
#include "cy_ota_api.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 */  
#ifndef CY_BOOT_USE_EXTERNAL_FLASH /* Set in ota.mk */  
#include "cy_smif_psoc6.h"  
#endif
```

Global Variables and macros

Several macros and global variables are used by the OTA API. You will see how these are used in the exercise.

```
/* Determine if reboot is initiated after OTA competes */  
#define REBOOT_AFTER_OTA (1)  
  
/* Variables for OTA */  
cy_ota_context_ptr ota_context; /* Context used by the OTA agent */  
cy_ota_network_params_t ota_network_params = {CY_OTA_CONNECTION_UNKNOWN};  
cy_ota_agent_params_t ota_agent_params = {0};  
uint16_t bt_ota_config_descriptor = 0; /*Remember if OTA is using notify or indicate*/
```

Initialization in main

Two things are done 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. 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();
```

Initialize in Bluetooth® enabled event

If the external flash is used to store the secondary image, the quad SPI block must be initialized during the Bluetooth® management event `BTM_ENABELD_EVT`:

```
/*Initialize QuadSPI if using external flash*/  
#ifndef CY_BOOT_USE_EXTERNAL_FLASH  
psoc6_qspi_init();  
#endif
```

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 the define for `REBOOT_AFTER_OTA`.

```

case GATT_HANDLE_VALUE_CONF: /* Value confirmation */
{
    cy_ota_agent_state_t ota_lib_state;
    cy_ota_get_state(ota_context, &ota_lib_state);
    if (ota_lib_state == CY_OTA_STATE_OTA_COMPLETE)
    {
        if(REBOOT_AFTER_OTA == 1)
        {
            printf( "OTA done, reboot in 2 seconds\n" );
            vTaskDelay(2000);
            NVIC_SystemReset();
        }
        else
        {
            printf( "OTA done, auto-reboot disabled\n" );
            cy_ota_agent_stop(&ota_context);
        }
    }
    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 section 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 code is shown in a single block on the following page.

```

/* OTA GATT write handling */
switch ( p_write_req->handle )
{
    case HDLD_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_CONTROL_POINT_CLIENT_CHAR_CONFIG:
        bt_ota_config_descriptor = p_write_req->p_val[0]; /* Save Configuration descriptor */
        return WICED_BT_GATT_SUCCESS;

    case HDLC_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_CONTROL_POINT_VALUE:
        switch (p_write_req->p_val[0])
        {
            case CY_OTA_UPGRADE_COMMAND_PREPARE_DOWNLOAD:
                /* Initialize OTA */
                memset(&ota_network_params, 0, sizeof(ota_network_params));
                memset(&ota_agent_params, 0, sizeof(ota_agent_params));

                /* Common Network Parameters */
                ota_network_params.initial_connection = CY_OTA_CONNECTION_BLE;

                /* OTA Agent parameters - used for ALL transport types*/
                ota_agent_params.validate_after_reboot = 1; /* Validate after reboot */

                result = cy_ota_agent_start(&ota_network_params, &ota_agent_params, &ota_context);
                if (CY_RSLT_SUCCESS != result)
                {
                    printf("cy_ota_agent_start() Failed - result: 0x%x\r\n", result);
                }
                else
                {
                    printf("OTA Agent Started \r\n");
                }
            }
        }
}

```

```

    }

    result = cy_ota_ble_download_prepare(ota_context, connection_id, bt_ota_config_descriptor);
    if (CY_RSLT_SUCCESS != result)
    {
        printf("Download preparation Failed - result: 0x%lx\r\n", result);
        return WICED_BT_GATT_ERROR;
    }
    return WICED_BT_GATT_SUCCESS;

case CY_OTA_UPGRADE_COMMAND_DOWNLOAD:
/* let OTA lib know what is going on */
result = cy_ota_ble_download(ota_context, p_data, connection_id, bt_ota_config_descriptor);
if (CY_RSLT_SUCCESS != result)
{
    printf("Download Failed - result: 0x%lx\r\n", result);
    return WICED_BT_GATT_ERROR;
}
return WICED_BT_GATT_SUCCESS;

case CY_OTA_UPGRADE_COMMAND_VERIFY:
result = cy_ota_ble_download_verify(ota_context, p_data, connection_id);
if (CY_RSLT_SUCCESS != result)
{
    printf("Verification and Indication failed: 0x%d\r\n", status);
    return WICED_BT_GATT_ERROR;
}
return status;

case CY_OTA_UPGRADE_COMMAND_ABORT:
result = cy_ota_ble_download_abort(ota_context);
return WICED_BT_GATT_SUCCESS;
}

case HDLC_OTA_FW_UPGRADE_SERVICE_OTA_UPGRADE_DATA_VALUE:
result = cy_ota_ble_download_write(ota_context, p_data);
return (result == CY_RSLT_SUCCESS) ? WICED_BT_GATT_SUCCESS : WICED_BT_GATT_ERROR;
} /* End of OTA write handling */

/* Normal GATT write handling */
. . .

```

7.1.2.5 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 *anyccloud-ota* 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:

```
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:

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

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

7.2 Secure OTA update

The *anyccloud-ota* library allows for secure OTA updates using a signed image and a different custom service. However, MCUboot already supports a secure OTA update process including image encryption and signing, so it is simplest to use that functionality.

The differences between what we have done so far and the secure OTA update process are:

1. The image is encrypted and is then digitally signed using a private key.
2. The MCUboot firmware loaded onto the CM0+ contains the matching public key.
3. MCUboot validates the image in the primary slot whenever the device rebooted.
4. When a new image is provided, MCUboot decrypts and validates it using the public key before copying it to the primary slot. If the image is not properly signed, it will be rejected.

Everything else works exactly the same.

There are three sets of changes required to enable the secure OTA update process:

7.2.1 Generate a public/private key pair

The MCUboot library has a public/private key pair that is used for many of the code examples such as the MCUboot basic bootloader app example. While that key pair can be used for an example, 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).*

7.2.2 Enable secure OTA update in the CM0+ MCUboot application

This is done by uncommenting 2 lines in *config/mcuboot_config/mcuboot_config.h* to enable decryption. You should also verify that the line to perform validation of the image is not commented out.

```
#define MCUBOOT_SIGN_EC256
#define NUM_ECC_BYTES (256 / 8) // P-256 curve size in bytes
.
.

#define MCUBOOT_VALIDATE_PRIMARY_SLOT
```

The public key is imbedded in the MCUboot firmware. The default key location is the *keys* directory of the CM0+ project. If you generate your own key pair, you can either replace the existing keys with your own in the same directory, or else you must update the include path in the *app.mk* file:

```
INCLUDES+=\  
./keys\
```

The name of the key file is specified by a make variable so if you change the name of the key file, you must update the variable *SIGN_KEY_FILE* in the *shared_config.mk* file. For example, if your key is called *my_key.pub* the *shared_config.mk* file would have:

```
SIGN_KEY_FILE=my_key
```

Note: For the public key, you should not include the file extension of *.pub*.

7.2.3 Enable image signing in the CM4 project

This is done by enabling image signing during post-build steps in the *ota.mk* file and specifying the path to the key file:

```
IMGTOOL_COMMAND_ARG=sign  
CY_SIGNING_KEY_ARG="-k $(MCUBOOT_KEY_FILE) "
```

The default private key is taken from an example key pair in the *anyccloud-ota* library. If you generate your own key pair, you need to change the variables to specify the location and name of your private key. The values are set in *app.mk*. For example, if your private key is in the application in a directory called *keys* and the file is called *my_key.pem*, you would use:

```
MCUBOOT_KEY_DIR=./keys  
MCUBOOT_KEY_FILE=$(MCUBOOT_KEY_DIR)/my_key.pem
```

Note: For the private key name, you must include the full file name including the file extension (*.pem* in this case)

Note: The examples shown here demonstrate the image signing and validation feature 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](#).

7.3 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 directories (for 64-bit and 32-bit versions of Windows, respectively):

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

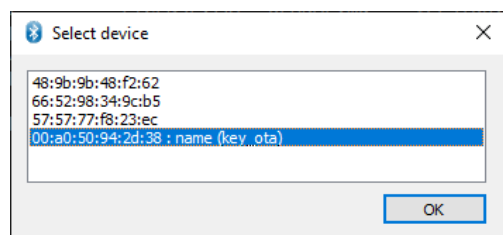
btsdk-peer-apps-ota/Windows/WSotaUpgrade/Release/x64/WSotaUpgrade.exe

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:

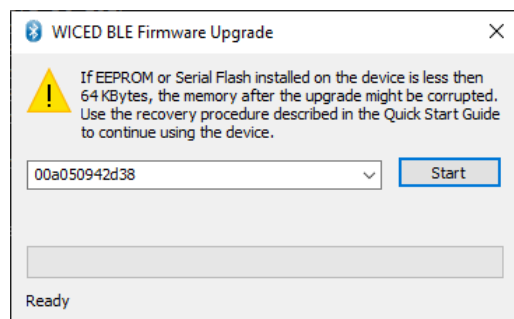
```
./WSotaUpgrade.exe app.bin
```

Note: You can also just drag the .bin file onto WSotaUpgrade.exe from a Windows file explorer window to run the application with the bin file as its input.

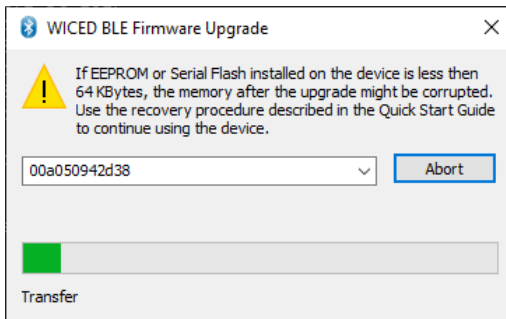
You will get a window that looks like the following. Select the device you want to update and click **OK**.



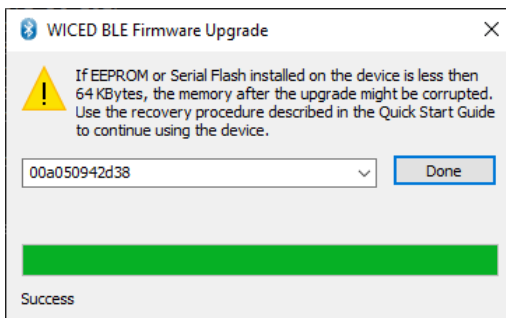
On the next window, verify that the device type is correct and click **Start**.



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:

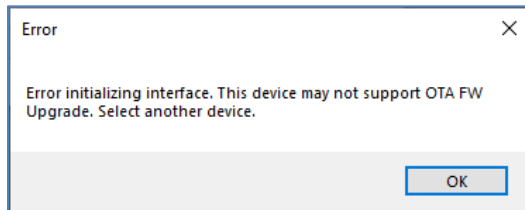
```
[INF] MCUboot Bootloader Started
[INF] External Memory initialized using SFDP
[INF] swap_status_source: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] Boot source: none
[INF] boot_swap_type_multi: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
[INF] 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
```

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
```


7.3.1 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 they 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.

7.4 Exercises

Exercise 1: Bluetooth® LE application with OTA update capability

In this exercise, you will first create, update and program a Bluetooth® project for the CM4 MCU using a template that has been modified from a prior exercise to support OTA firmware update.

Next you will create, update and program the MCUboot project for the CM0+ MCU.

Finally, you will 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, update project settings, and program the CM4



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

Use the import functionality in project creator to start from the template in *Templates/ch07_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 the **Maximum attribute length** to 512 and the **Attribute MTU 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 (7.1.2.1) 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. Edit the Makefile to include the ota.mk file.

```
-include ./ota.mk
```

Note: This line must go at the end of the Advanced Configuration section of the file.



8. The ota.mk file already has all of the changes needed for OTA. Review the file to understand what was changed.



9. Use the Library Manager to add the *anycLOUD-ota* library to the application.



10. Copy the file *cy_ota_config.h* file from the *anycLOUD-ota* library to the application. This file contains OTA settings for Wi-Fi so we won't need to change it.

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



11. All changes required to the code are already done in the template. Review the code to understand the changes that were described earlier in this chapter.

☐ 12. Build and program the application to the kit.

Note: *After programming completes, there will be errors in the console output and the application will not run. This is expected because the CM0+ application containing MCUboot has not yet been programmed:*

```
Error: psoc6.cpu.cm4 -- clearing lockup after double fault
psoc6.cpu.cm4 halted due to debug-request, current mode: Handler HardFault
xPSR: 0x21000003 pc: 0x8008f3ee msp: 0x080ff7e0
Error executing event reset-deassert-post on target psoc6.cpu.cm4
```

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

☐ 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 *ch07_ex01_ota_bootloader*.

Note: *This is a dual-core application so you will see two projects in the project explorer inside the Eclipse IDE for ModusToolbox™. We will only use the CM0+ bootloader project.*

```
ch07_ex01_ota_bootloader.bootloader_cm0p
ch07_ex01_ota_bootloader.blinky_cm4
```

Note: *If you look at the directory hierarchy on disk, it will look like this:*

```
ch07_ex01_ota_bootloader/
  bootloader_cm0p/
  blinky.cm4/
  ...
  README.md
```

☐ 2. Go to the CM0+ project and make the updates described in [7.1.1.1](#). As a reminder, you must:

- Edit the *shared_config.mk* file to update the slot and scratch sizes to match the sizes used in the CM4 project's *ota.mk* file.
- Copy the config directory from the mcuboot library to the CM0+ project's root directory.
- Edit the *app.mk* file to change the path to the *config* directory.
- Edit the *mcuboot_config.h* file to disable image signature checking.
- Edit the *Makefile* file to enable external flash and the MCUboot swap upgrade feature
- Ensure that the tools required for creating the bootloadable image are installed.

☐ 3. Optional: Since we are not using the blink.cm4 project, you can delete it if desired.

- ☐ 4. Build and program the CM0+ project to the kit.

Once the CM0+ and CM4 applications have both been programmed, the application should start up as normal. You will see the MCUboot project run first and when it finds a valid primary image, it launches it.

```
[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=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
[INF] Swap type: none
[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
```

Test Bluetooth® functionality

- ☐ 1. With the application running and advertising, use CySmart 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 two new characteristics from the OTA update service. Don't write to them since they are intended for the OTA update process. You will get a chance to try that next.

- ☐ 2. Disconnect when you are done so that the OTA update process will be able to connect to the device.
- ☐ 3. Remove the device from the paired Bluetooth device list on the phone or from the Device List on the Windows version of CySmart.

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 (ch07_ex01_ota_app), open the file ota.mk and change APP_VERSION_MINOR to 1.
- ☐ 2. Open the CM4 application's main.c file. Change all occurrences of CYBSP_USER_LED to CYBSP_LED_RGB_BLUE. This will change the LED control from the orange LED to the blue LED in the RGB LED so that you will see a functional difference in the kit's behavior in addition to just the version number.

Note: Make sure you don't change any occurrences of CYBSP_USER_LED2 since that is used for showing the advertising status.

- ☐ 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/CY8CKIT-062S2-43012/Debug/app.bin*.
- ☐ 5. Run the Windows bootloader and load the new firmware.

Follow the instructions from 7.3. See the troubleshooting information in 7.3.1 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 to reboot/copy/validate the image will take about another minute.

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

```
[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: 1.1.0*****
*****
Bluetooth Management Event: 0x16 BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT
Bluetooth Management Event: 0x0 BTM_ENABLED_EVT
Bluetooth Enabled
Local Bluetooth Device Address: 00:A0:50:25:06:90
Bluetooth Management Event: 0x18 BTM_BLE_ADVERT_STATE_CHANGED_EVT
Advertisement State Change: BTM_BLE_ADVERT_UNDIRECTED_HIGH
```

- ☐ 7. Connect to the device using CySmart and test the functionality to see that the blue LED turns on/off now instead of the orange LED.

Exercise 2: Secure OTA update

In this exercise, we will take the CM0+ and CM4 projects from the previous exercise and enable secure OTA.

- ☐ 1. Open a command terminal (modus-shell for Windows) and create a public/private key pair using *imgtool*. See section 7.2.1 for details. We will copy these keys into the projects in a minute.
- ☐ 2. Create two new applications from the CM0+ and CM4 projects from the previous exercise.
Name the applications *ch07_ex02_ota_app_secure* and *ch07_ex02_ota_bootloader_secure*.

If you did not complete the previous exercise, the solution can be found in the *Projects* directory.

Note: When you create the CM0+ application if you choose the lower level CM0+ project as the starting point it will be created without the extra level of hierarchy from the original dual-core MCUBoot application. If you choose the lower level, you will have to update the `CY_GETLIBS_SHARED_PATH` variable in the Makefile to remove the extra level of hierarchy. This is how the solution projects are done.

Note: You can create both applications simultaneously in the project creator tool. Just use the Import button twice, check both boxes and enter the new name next to each application, then click Create.

- ☐ 3. Go to the CM0+ project of the bootloader application.
- ☐ 4. Update the `config/mcuboot_config/mcuboot_config.h` file to perform decryption and image checking as described in section 7.2.2.
- ☐ 5. Go to the `keys` directory and delete the existing key files. Copy the public key (`.pub`) from step 1 to the `keys` directory.
- ☐ 6. Edit the `SIGN_KEY_FILE` variable in `shared_config.mk` to match the name of the key you created.

Note: The name does not include the extension `pub`. For example: `SIGN_KEY_FILE=my_key`

- ☐ 7. Go to the CM4 application.
- ☐ 8. Create a directory called `keys` in the application's root directory. Copy the private key file (`.pem`) that you created to the `keys` directory.
- ☐ 9. Edit the `ota.mk` file to sign the image as described in section 7.2.3.
- ☐ 10. Edit the `ota.mk` file to specify the key directory as `./keys` and the key name to match your key's file name. For example:

```
MCUBOOT_KEY_DIR=./keys
MCUBOOT_KEY_FILE=$(MCUBOOT_KEY_DIR)/my_key.pem
```
- ☐ 11. Build and program both projects to the kit.
- ☐ 12. Modify the CM4 project so that the version is 1.2.0 (in `ota.mk`) and change `CYBSP_LED_RGB_BLUE` to `CYBSP_LED_RGB_GREEN` (two places in `main.c`). Build the project but do NOT program it.
- ☐ 13. Use the OTA update procedure to update the firmware on the kit.
- ☐ 14. Optional – Attempt to use OTA to load unsigned firmware from the previous exercise onto the device and observe that the new firmware is rejected by MCUBoot after loading completes because the signature validation fails. You will see the message "Image in the secondary slot is not valid!" in the UART and the existing application from the primary slot will run.

Trademarks

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

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

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

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

WARNINGS

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

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