AIROC™ Firmware Upgrade Library

ModusToolbox™

About this document

Scope and purpose

The firmware upgrade feature provided in ModusToolbox® allows an external device to install a newer firmware version on devices equipped with AIROC™ Bluetooth® chips. This document describes the functionality of the AIROC™ Firmware Upgrade Library used in various ModusToolbox™ sample applications. The remainder of the document uses CYW20819 for example, but the feature and library usage are the same for all Infineon Bluetooth® devices supporting the firmware upgrade feature.

Intended audience

This document is intended for application developers using a ModusToolbox™ Bluetooth® Software Development Kit (SDK) to create and test designs based on AIROC™ Bluetooth® devices.

Acronyms and abbreviations

In most cases, acronyms and abbreviations are defined on first use.

Note: *For a comprehensive list of acronyms and other terms used in the documents, go to the **Glossary**.*

Reference

[1] **Bluetooth® Core Specification 5.2**

[2] **AIROC™ Secure Over-the-Air Firmware Upgrade**

[3] **MeshClient and ClientControlMesh App user guide**

Table of contents

About this document **1**
Table of contents **2**
1 Introduction **3**
1.1 IoT resources and technical support 3
2 Design and architecture **4**
2.1 Dual partitions 4
2.2 Copy from storage to first partition 5
2.3 GATT database 7
2.4 OTA firmware upgrade procedure 8
2.5 HCI firmware update procedure 10
3 Library reference **14**
3.1 Firmware Upgrade Library initialization 14
3.2 Firmware upgrade initialize nonvolatile storage locations 15
3.3 Firmware upgrade store data to nonvolatile storage 15
3.4 Firmware upgrade retrieve data from nonvolatile storage 16
3.5 Firmware upgrade finish 16
3.6 OTA firmware upgrade initialization 17
3.7 OTA firmware upgrade connection status 17
3.8 OTA firmware upgrade read handler 18
3.9 OTA firmware upgrade write handler 18
3.10 OTA firmware upgrade indication confirmation 18
Revision history **19**

1 Introduction

The AIROC™ Firmware Upgrade Library is split into two parts. The top-level firmware upgrade module of the library provides a state machine with commands interfaces, status callbacks, and data handling. This module can be driven by applications that respond to HCI or Bluetooth® physical interfaces. For over-the-air (OTA) firmware upgrades, the module provides a simple implementation of the GATT procedures to interact with the device performing the upgrade. The Hardware Abstraction Library (HAL) firmware upgrade module of the library provides support for storing data in the nonvolatile memory and switching the device to use the new firmware when the upgrade is completed. Embedded applications may use OTA module functions (which in turn use HAL module functions), or the application may choose to use HAL module functions directly. It is assumed that the reader is familiar with the Bluetooth® Core Specification **[1]**.

The library supports secure and non-secure versions of the upgrade. In the non-secure version, a simple CRC32 verification is performed to validate that all bytes that have been sent from the device performing the upgrade are correctly saved in the serial flash of the device. The secure version of the upgrade validates that the image is correctly signed and has correct production information in the header. See the *AIROC™ Secure Over-the-Air Firmware Upgrade* application note **[2]** for the details of image generation and verification. In addition, see *MeshClient and ClientControlMesh App user guide* **[3]** for details regarding over-the-air firmware upgrades for mesh applications.

1.1 IoT resources and technical support

The wealth of data available **here** will help you to select the right IoT device for your design, and quickly and effectively integrate the device into your design. You can access a wide range of information, including technical documentation, schematic diagrams, product bill of materials, PCB layout information, and software updates. You can acquire technical documentation and software from the **Support Community website**.

2 Design and architecture

2.1 Dual partitions

The dual-partition method is used for devices without execute-in-place (XIP) code sections. To ensure a failsafe upgrade, the external or on-chip flash (OCF) memory of Infineon AIROC™ chips is organized with two firmware partitions: DS1 and DS2. During the startup operation, the boot code of the chip checks the first firmware partition (DS1), and if a valid image is found, assumes that the first partition is active and starts executing the code in the first partition.

If the first partition does not contain a valid image, the boot code checks the second partition (DS2) and starts the execution of the code in the second partition if a valid image is found there. If neither partition is valid, the boot code enters the download mode and waits for the code to be downloaded over HCI UART. Addresses of the partitions are programmed in a file with a *.btp* extension located in the platform directory of the SDK. For example, the *.btp* file for the CYW20719 device can be found in the ModusToolbox™ IDE under the *mtb-shared\wiced_btsdk* project folder in the Project Explorer pane, which is created and used by all AIROC™ applications:

mtb-shared\wiced_btsdk\dev-kit\baselib\20719B2\<version>\platforms\20719_OCF.btp

The firmware upgrade process stores the received data in the inactive partition. When the download procedure is completed and the received image is verified and activated, the currently active partition is invalidated, and then the chip is rebooted. After the chip reboots, the previously inactive partition becomes active. If, for some reason, the download or the verification step is interrupted, the valid partition remains valid and chip is not rebooted. This guarantees the failsafe procedure.

Table 1 shows the recommended memory section configuration values for an application supporting the firmware upgrade feature to be executed on a device with an external 4-Mbit serial flash.

Table 1 Recommended memory section offsets and lengths for external flash

Section name	Offset	Length	Description
Static Section (SS)	0x0000	0x2000	Static section used internally by the chip firmware.
Volatile Section (VS1)	0x2000	0x1000	First volatile section used for the application and the stack to store data in the external or on-chip flash memory. One serial flash sector.
Volatile Section (VS2)	0x3000	0x1000	Used internally by the firmware when VS1 needs to be defragmented.
Data Section (DS1)	0x4000	0x3E000	First partition.
Data Section (DS2)	0x42000	0x3E000	Second partition.

Table 2 shows the recommended layout for on-chip flash. These settings are configured on a per-platform basis by the **.btp* file.

Table 2 Recommended memory section offsets and lengths for on-chip flash

Section name	Offset	Length	Description
Static Section (SS)	0x500000	0x400	Static section used internally by the chip firmware.
Volatile Section (VS1)	0x500400	0x1000	First volatile section used for the application and the stack to store data in the external or on-chip flash memory. One serial flash sector.

Section name	Offset	Length	Description
Volatile Section (VS2)	N/A	N/A	
Data Section (DS1)	0x501400	0x1F600	First partition.
Data Section (DS2)	0x520A00	0x1F600	Second partition.

2.2 Copy from storage to first partition

A third upgrade layout option exists that uses an on-chip or external (off-chip) flash memory area to temporarily store the upgrade image. This option is used for devices with XIP code sections. XIP code is built to execute from a fixed location within the first flash partition. To upgrade, a new firmware image is downloaded to the designated storage location. After the download is validated, it is copied over the active partition. This procedure is performed in a failsafe manner by using a small second flash partition to perform the copy operation as described below. The flash layout for devices with XIP is shown in **Table 3**.

Table 3 Example memory section offsets and lengths for on-chip flash with external flash upgrade storage

Section name	Offset	Length	Description
Static Section (SS)	0x500000	0x400	Static section used internally by the chip firmware.
Volatile Section (VS1)	0x500400	0x1000	First volatile section used for the application and the stack to store data in the external or on-chip flash memory. One serial flash sector.
Volatile Section (VS2)	N/A	N/A	
Data Section (DS1)	0x501400	0x3DC00*	First partition. *Limited to 0x1EE00 if storage is on-chip
Data Section (DS2)	0x53F000	0x1000	Second partition, app that copies from storage to first partition.
Storage			On-chip or external

During firmware upgrade, the device performing the procedure (Downloader) pushes chunks of the new image to the device being upgraded. The embedded application receives the image and stores it in the external or on-chip flash. When all data has been transferred, the Downloader sends a command to verify the image passing a 32-bit CRC checksum. The embedded app reads the image from the flash and verifies the image. For the non-secure download, the library calculates the checksum and verifies that it matches received CRC. For the secure download case, the library performs Elliptic Curve Digital Signature Algorithm (ECDSA) verification and verifies that the Product Information stored in the new image is consistent with the Product Information of the firmware currently being executed on the device. If verification succeeds, the embedded application invalidates the active partition and reboots the chip.

When rebooted, the chip will find a valid image in the second partition. This is a small application that performs the copy of the validated image from the storage location to the active partition. Once the copy is completed, the active partition is validated and the device is rebooted. After rebooting, the device will use the new firmware that has been copied to the first partition.

The upgrade image storage location can be designated “on_chip” or “external_sflash” in the kit-specific makefile found in the kit’s *BSP* folder. For example, *mtb-shared/wiced_btsdk/dev-kit/bsp/TARGET_<target>/<version>/<target>.mk*. The makefile variable is `CY_CORE_OTA_FW_UPGRADE_STORE`. If the storage location is “on_chip”, it will reside in the upper half of the DS1 partition. This limits the size of both the active partition and the storage location to 0x1EE00 (126,464 bytes) for the example layout in **Table 3**.

If the storage location is designated as “external_sflash”, the full extent of DS1 can be used as the active partition. Larger firmware images may require external storage for upgrades. By default, when external storage is selected, the image in the external flash is encrypted. The key for this encryption is regenerated as a random number during each firmware upgrade and is stored in non-volatile memory in the VS1 partition (shown in **Table 1**). VS1 is on-chip, by default. The key is recalled from VS by the small application kept in DS2. This application will decrypt the image using the key while copying it to the active partition. The optional encryption is enabled by `CY_APP_OTA_DEFINES+=-DOTA_ENCRYPT_SFLASH_DATA` in the kit-specific makefile.

Note that this method of upgrade completes all firmware upgrade transactions after the download image is validated and confirmation is sent to the Downloader. The device is out of communication for several seconds while rebooting, copying the download image to the active partition, and then rebooting again with the upgraded firmware.

Figure 1 shows a block diagram of the Firmware Upgrade Library modules.

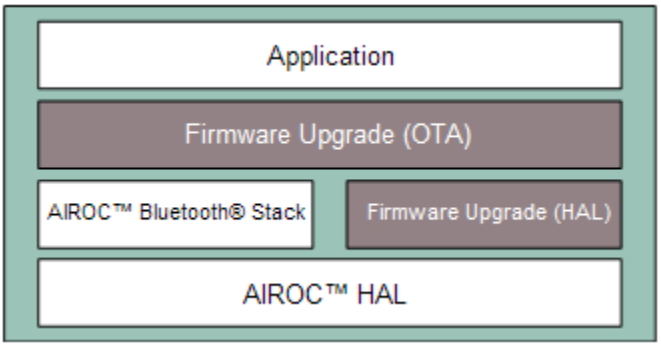

Figure 1 Firmware upgrade modules

While different upgrade methods (for example, a different OTA procedure, SPI, or UART download) may require different firmware upgrade module implementations, the HAL firmware upgrade implementation will likely be the same, and will not require changes to that module of the library. The sample OTA firmware upgrade module is provided in the *mtb-shared\wiced_btsdk* project, created and used by applications created in ModusToolbox™, in the Project Explorer pane under:

mtb-shared\wiced_btsdk\dev-kit\libraries\btsdk-ota\<version>\COMPONENT_fw_upgrade_lib

The implementation of the HAL firmware upgrade module is provided in:

mtb-shared\wiced_btsdk\dev-kit\libraries\btsdk-ota\<version>\COMPONENT_fw_upgrade_lib\<device>\fw_upgrade.c

The "ota_firmware_upgrade" sample application that exercises the library is available using the “New Application” wizard in the ModusToolbox™ Quick Panel. Run the wizard, select your board, and under the "Bluetooth®" category, choose the "Firmware Upgrade via OTA" sample application.

The sample application demonstrates the use of the required `OTA_FW_UPGRADE` make variable, as well as the optional secure configuration, and where supported, options to configure the storage location of the update image (on-chip flash vs. external flash).

Similarly, the “hci_firmware_upgrade” sample application (under the Quick Panel "Manufacturing" category) demonstrates firmware upgrades via the HCI UART.

2.3 GATT database

The GATT services and characteristics listed below along with the correct UUIDs can be added to an app by using the Bluetooth® Configurator. Depending on the secure or non-secure method that the application wants to use, the GATT database of the device capable of receiving an OTA firmware upgrade will contain either an OTA Secure Upgrade or an OTA Upgrade service declaration using one of the UUIDs listed in **Table 4**.

Table 4 OTA upgrade service

Service name	UUID
OTA Upgrade Service	`{ae5d1e47-5c13-43a0-8635-82ad38a1381f}`
OTA Secure Upgrade Service	`{C7261110-F425-447A-A1BD-9D7246768BD8}`

The service will contain Control Point and Data characteristics. The Control Point characteristic shall also contain a standard Client Characteristic Configuration descriptor with mandatory properties defined in **Table 5**.

Table 5 OTA firmware upgrade service characteristics

Characteristic name	UUID	Mandatory properties
OTA Upgrade Control Point	`{a3dd50bf-f7a7-4e99-838e-570a086c661b}`	Write, Indicate, Notify
OTA Upgrade Control Point Client Characteristic Configuration Descriptor	`0x2902`	Read, Write
OTA Upgrade Data	`{a2e86c7a-d961-4091-b74f-2409e72efe26}`	Write

If the application requires a secure link between the Downloader and the embedded application, the Characteristics shall be defined in the GATT database to include `LEGATTDB_PERM_AUTH_WRITABLE`.

2.4 OTA firmware upgrade procedure

A message sequence chart showing an OTA firmware upgrade procedure is shown in **Figure 2**.

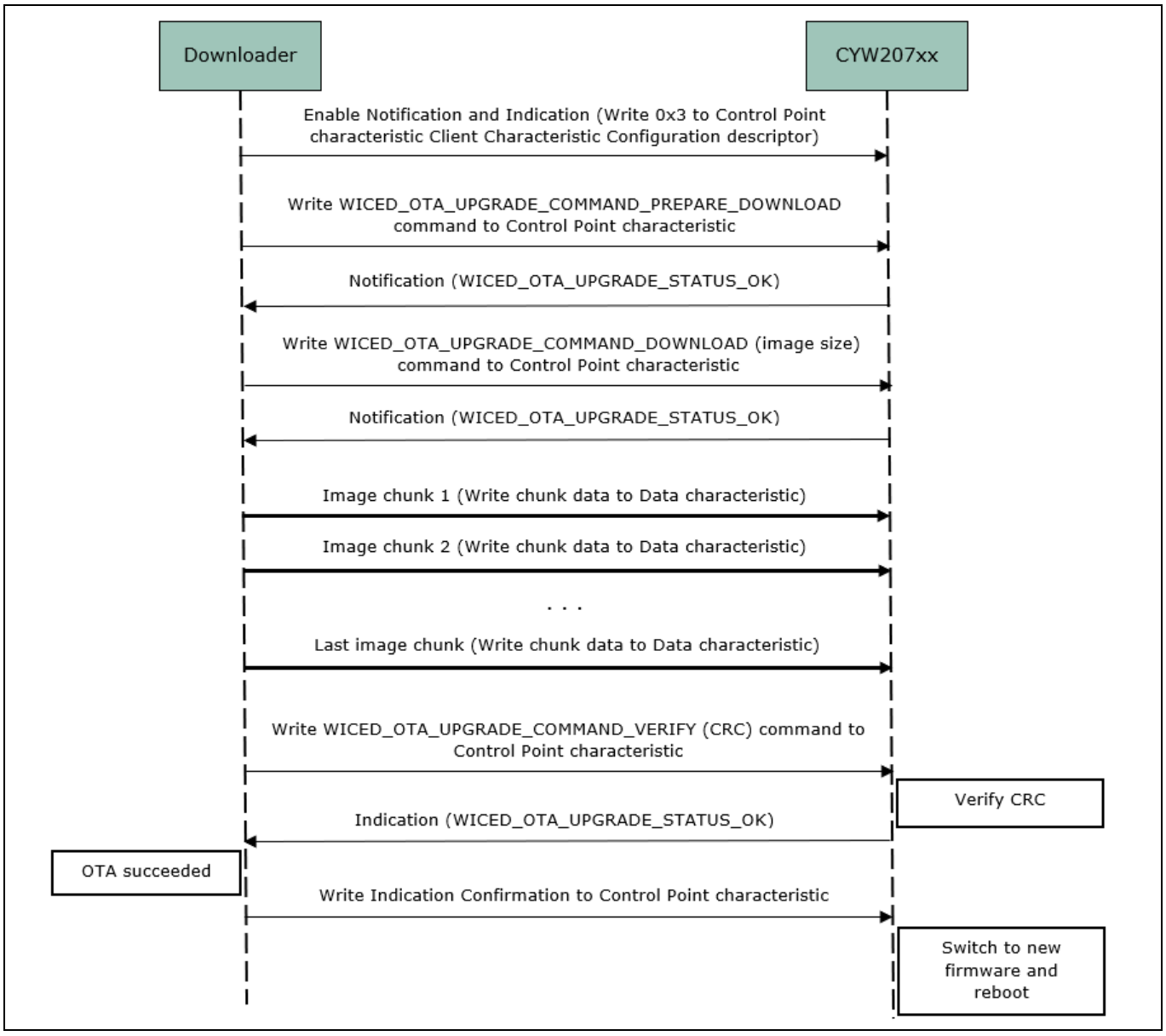

Figure 2 OTA firmware upgrade message sequence chart

*Note: Thin lines in **Figure 2** correspond to the messages sent using the Control Point characteristic. Thick lines indicate messages sent using the Data characteristic.*

Before performing the upgrade procedure, the Downloader should enable notifications and indications for the Control Point characteristic by writing the corresponding value to the Client Characteristic Configuration descriptor. If the Downloader is using a Bluetooth® stack that does not allow the configuration of simultaneous notifications and indications, at least one of them must be configured.

All multi-octet values, for example the size of the image and the CRC32, are sent using little-endian format.

To start the upgrade, the Downloader sends the `WICED_OTA_UPGRADE_COMMAND_PREPARE_DOWNLOAD` command (see **Table 6** and **Table 7** for details of the commands and events). This indicates that a new upgrade

process is being started. The data received after that command will be stored from the zero-offset position of the inactive logical memory partition.

After the Downloader receives the `WICED_OTA_UPGRADE_STATUS_OK` message, it must send the `WICED_OTA_UPGRADE_COMMAND_DOWNLOAD` command, passing four bytes that specify the memory image size to be downloaded. If `WICED_OTA_UPGRADE_STATUS_OK` is received in reply, the Downloader starts sending chunks of data.

When the library receives the `WICED_OTA_UPGRADE_COMMAND_DOWNLOAD` command from the Downloader, it verifies the configuration of active and inactive partitions. If the configuration is not valid, the library sends `WICED_OTA_UPGRADE_STATUS_INVALID_IMAGE`.

After the final image chunk is sent, the Downloader sends the `WICED_OTA_UPGRADE_COMMAND_VERIFY` command that passes the image checksum calculated on the host. The library verifies the stored image and sends the `WICED_OTA_UPGRADE_STATUS_OK` or `WICED_OTA_UPGRADE_STATUS_VERIFICATION_FAILED` message to the Downloader. If verification was successful, the firmware automatically reboots the chip. If the verification was not successful, the firmware sends a `WICED_OTA_UPGRADE_STATUS_VERIFICATION_FAILED` status to the Downloader. If the download process is interrupted or if the verification fails, the embedded application continues its execution. To restart the process, the Downloader will need to start from the beginning by sending `WICED_OTA_UPGRADE_COMMAND_PREPARE_DOWNLOAD`.

All commands and data packets are sent from the Downloader to the embedded application using the GATT Write request procedure. All messages to the Downloader except for the final verification `WICED_OTA_UPGRADE_STATUS_OK` message are sent using the GATT Notification procedure. The Verification OK message is sent using the GATT Indication procedure. The library reboots the chip as soon as it receives the Indication Confirmation from the Downloader. If the Downloader had enabled notifications, and did not allow indications, the verification `WICED_OTA_UPGRADE_STATUS_OK` message is sent using the GATT Notify procedure. In that case, the library waits for one second after sending the notification, marks the newly updated partition as valid, invalidates the current partition, and then reboots the chip.

The library accepts data chunks of up to 512 octets in length. For better performance, it is recommended that the Downloader negotiates the largest possible maximum transmission unit (MTU) and sends data chunks of (MTU minus 3) octets.

Table 6 OTA firmware upgrade commands

Command Name	Value	Parameters
`WICED_OTA_UPGRADE_COMMAND_PREPARE_DOWNLOAD`	1	
`WICED_OTA_UPGRADE_COMMAND_DOWNLOAD`	2	4-byte image size
`WICED_OTA_UPGRADE_COMMAND_VERIFY`	3	4-byte CRC32
`WICED_OTA_UPGRADE_COMMAND_ABORT`	7	

Table 7 OTA firmware upgrade events

Event Name	Value	Parameters
`WICED_OTA_UPGRADE_STATUS_OK`	0	
`WICED_OTA_UPGRADE_STATUS_UNSUPPORTED_COMMAND`	1	
`WICED_OTA_UPGRADE_STATUS_ILLEGAL_STATE`	2	
`WICED_OTA_UPGRADE_STATUS_VERIFICATION_FAILED`	3	
`WICED_OTA_UPGRADE_STATUS_INVALID_IMAGE`	4	

2.5 HCI firmware update procedure

The firmware can be updated via the HCI interface. This method can be useful when the update is performed by an MCU directly connected to the HCI UART interface of the Bluetooth® device. In this method, the embedded application running on the Bluetooth® device can use the firmware update library by responding to HCI commands with library procedure calls, and responding to library status callbacks with HCI events.

A message sequence chart showing an HCI firmware upgrade procedure is shown in **Figure 3**. The chart starts with the `wiced_hci_firmware_upgrade_init()` call to initialize the library. This is called from the hci firmware update application initialization. This call provides a status update callback procedure, labeled `status_update` in the chart. In the *hci_firmware_update* code example, this is the `hci_dfu_status_callback()` function.

Next, the chart shows HCI commands `HCI_CONTROL_MISC_COMMAND_GET_VERSION` and `HCI_CONTROL_DFU_COMMAND_READ_CONFIG` that are handled by the hci firmware upgrade application directly. The application responds to these commands with HCI events containing the requested information. The `get_version` event includes the BTSDK build version, the Bluetooth® device identification, and the HCI firmware update application identifier. The config command response event provides the sector size used to store data to flash. This is also the size expected for all data transfers except the last.

The rest of the chart shows HCI commands that result in library calls to `hci_firmware_upgrade_handle_command()` and `hci_firmware_upgrade_handle_data()`. The HCI command `HCI_CONTROL_DFU_COMMAND_WRITE_COMMAND` has a command value in the first byte of the payload. These commands are shown in the chart as prepare, download, and verify. The responses to these commands are generated in the status callback handler.

For the command `WICED_HCI_UPGRADE_COMMAND_PREPARE_DOWNLOAD`, the `HCI_CONTROL_DFU_EVENT_STARTED` event is returned. After that the download size is sent using the command `WICED_HCI_UPGRADE_COMMAND_DOWNLOAD`. The first four bytes of the command payload should contain the update image byte count. No response is returned for this command.

The command `WICED_HCI_UPGRADE_COMMAND_VERIFY` takes time to complete, so two events are provided: `HCI_CONTROL_DFU_EVENT_VERIFICATION` when verification starts and `HCI_CONTROL_DFU_EVENT_VERIFIED` when verification is successful. If verification fails, the library status callback passes the status `HCI_FW_UPGRADE_STATUS_ABORTED`. This is used to generate the HCI event `HCI_CONTROL_DFU_EVENT_ABORTED`.

Data is transferred using the HCI command `HCI_CONTROL_DFU_COMMAND_WRITE_DATA`. This command causes the application to call the library function `hci_firmware_upgrade_handle_data()`. If the return value from this call indicates success, then the HCI event `HCI_CONTROL_DFU_EVENT_DATA` is sent. If the data command results in failure, then no HCI event is generated. The HCI host should wait long enough for the worst-case flash erase, write, and processing time before aborting, if there is not response from the data write command.

At any point, the firmware upgrade process can be aborted with the HCI command `HCI_CONTROL_DFU_COMMAND_WRITE_COMMAND` with parameter `WICED_HCI_UPGRADE_COMMAND_ABORT`. This command stops the process and returns the upgrade process to the initial state. Subsequent upgrade activity should start again from the beginning.

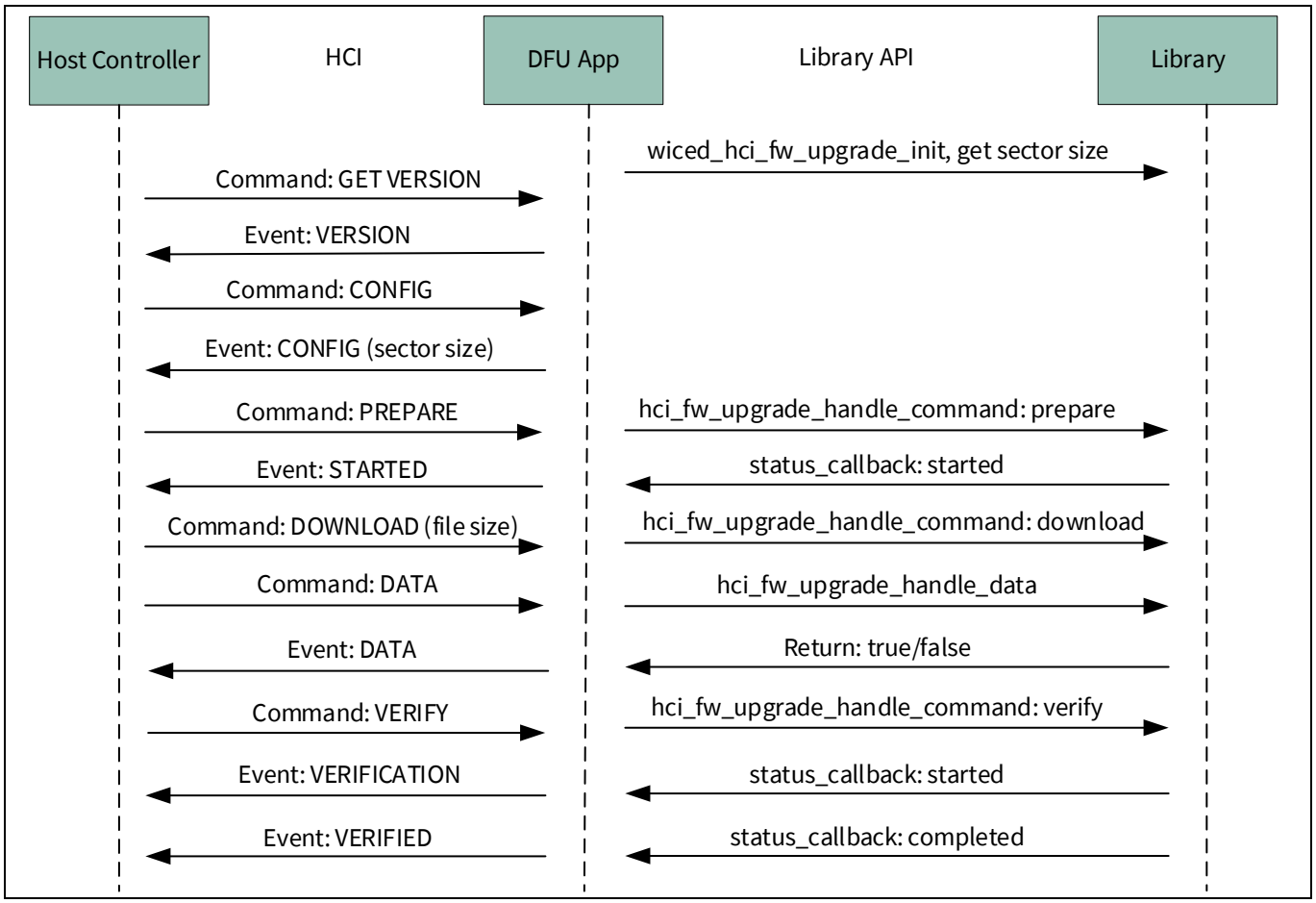

Figure 3 HCI firmware upgrade message sequence chart

Another message sequence chart shown in **Figure 4** illustrates how procedural failures are handled. The first scenario occurs during data transfer. If no acknowledgment event is returned within a time out period after the `HCI_CONTROL_DFU_COMMAND_WRITE_DATA` HCI command, then the transfer is aborted.
The second scenario is a verification failure. In that case the `HCI_CONTROL_DFU_COMMAND_WRITE_COMMAND` is provided with the `WICED_HCI_UPGRADE_COMMAND_VERIFY` payload. The response event `HCI_CONTROL_DFU_EVENT_VERIFICATION` indicates verification is in progress. A subsequent `HCI_FW_UPGRADE_STATUS_ABORTED` event signals the failure. Any further attempt to update firmware must restart from the beginning.

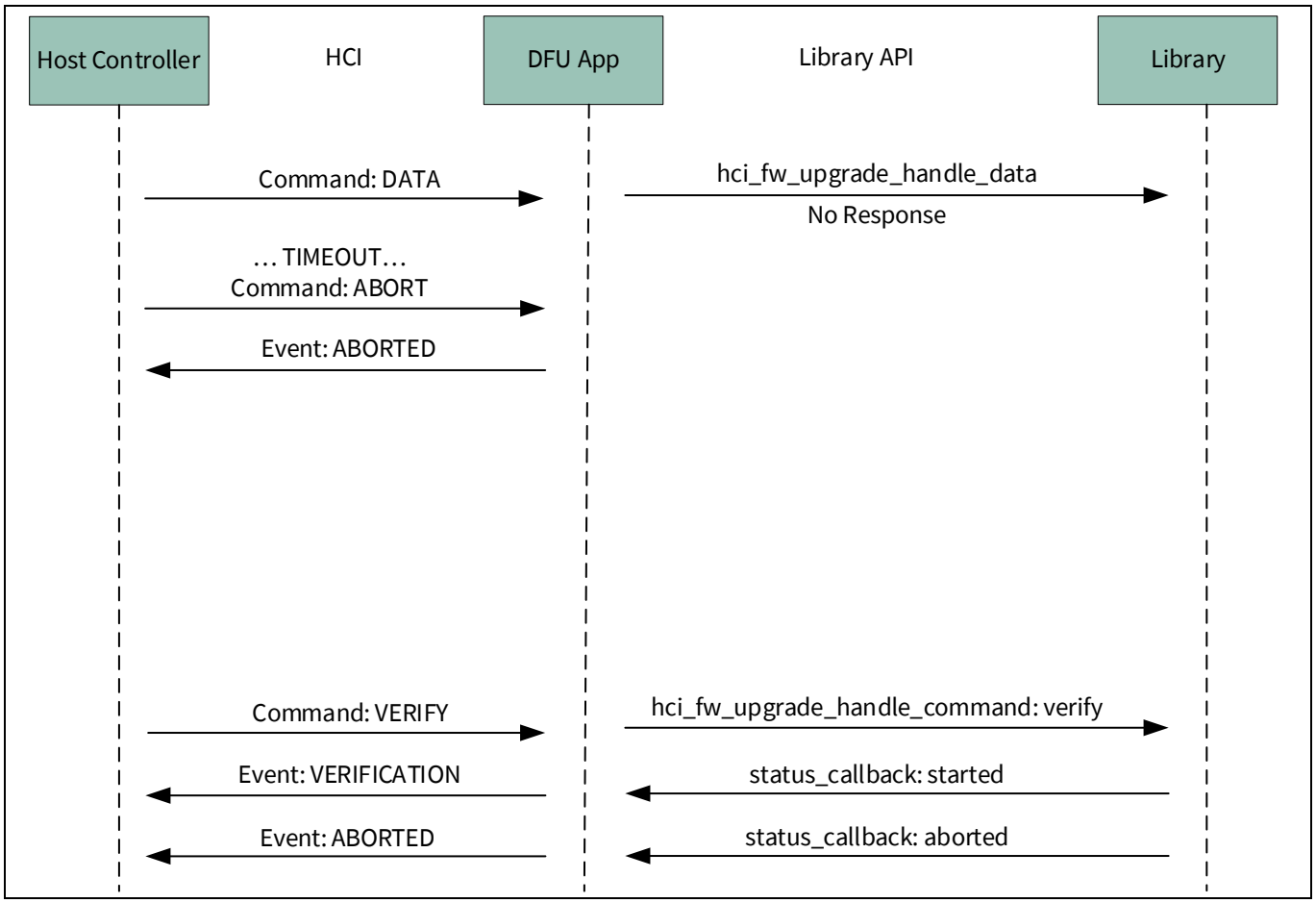

Figure 4 HCI firmware upgrade failures

The list of HCI command and events as described above are listed in **Table 8** and **Table 9**.

Table 8 HCI firmware upgrade commands (wiced_hci_control.h)

Command name	Value	Command type	Value	Parameters
HCI_CONTROL_DFU_COMMAND_READ_CONFIG	0			
HCI_CONTROL_DFU_COMMAND_WRITE_COMMAND	1	WICED_HCI_UPGRADE_COMMAND_PREPARE_DOWNLOAD	1	
		WICED_HCI_UPGRADE_COMMAND_DOWNLOAD	2	File size
		WICED_HCI_UPGRADE_COMMAND_VERIFY	3	4-byte CRC32
		WICED_HCI_UPGRADE_COMMAND_ABORT	7	
HCI_CONTROL_DFU_COMMAND_WRITE_DATA	2			

Table 9 HCI firmware upgrade events (wiced_hci_control.h)

Event name	Value	Parameters
HCI_CONTROL_DFU_EVENT_CONFIG	1	Sector size
HCI_CONTROL_DFU_EVENT_STARTED	2	

Event name	Value	Parameters
HCI_CONTROL_DFU_EVENT_DATA	3	
HCI_CONTROL_DFU_EVENT_VERIFICATION	4	
HCI_CONTROL_DFU_EVENT_VERIFIED	5	
HCI_CONTROL_DFU_EVENT_ABORTED	6	

3 Library reference

This section describes the functions exposed by the HAL firmware upgrade module followed by the OTA firmware upgrade module. You can use the OTA or HCI sample protocols described in this document and call `ota_firmware_upgrade_...` or `hci_firmware_upgrade_...` functions. You can also develop a completely different method to deliver the firmware image to the embedded application and call `wiced_firmware_upgrade_...` functions directly.

Table 10 Function call hierarchy for example code

Operation	Example application call	Library/module internal call	Note
Initialization	`wiced_ota_fw_upgrade_init`	`wiced_firmware_upgrade_init`	init
		`wiced_firmware_upgrade_init_nv_locations`	At start of update
GATT Connect / Disconnect	`wiced_ota_fw_upgrade_connection_status_event`		Upon GATT connect and disconnect
Read	`wiced_ota_fw_upgrade_read_handler`		
Write	`wiced_ota_fw_upgrade_write_handler`	`wiced_firmware_upgrade_store_to_nv`	When updating
Indication Confirmation	`wiced_ota_fw_upgrade_indication_cfm_handler`	`wiced_firmware_upgrade_retrieve_from_nv`	When verifying after transfer completed
Completion		`wiced_firmware_upgrade_finish`	After image is verified

3.1 Firmware Upgrade Library initialization

This function is typically called by the OTA firmware upgrade module (see OTA firmware upgrade) or the application during initialization to configure serial flash sections' locations and lengths. The module will use flash layout defaults that are defined by the **.btp* file and any makefile overrides for defined values such as `SS_LOCATION`, `VS_LOCATION`, `VS_LENGTH`, and `DS_LOCATION`. An application can supply the values directly by calling the initialization function directly.

Prototype

```
wiced_bool_t wiced_firmware_upgrade_init(wiced_fw_upgrade_nv_loc_len_t
*p_sflash_nv_loc_len, uint32_t sflash_size);
```

Parameters

- `p_sflash_nv_loc_len`: Locations and lengths of different sections present in the serial flash. These must match the values configured in the platform *.btp* file during the build process. It is not possible to change these values during the firmware upgrade procedure. The values are passed to the firmware upgrade module in the `wiced_fw_upgrade_nv_loc_len_t` structure as follows:

```
typedef struct
{
    uint32_t ss_loc;    // static section location
```

```
    uint32_t ds1_loc;   // ds1 location
    uint32_t ds1_len;   // ds1 length
    uint32_t ds2_loc;   // ds2 location
    uint32_t ds2_len;   // ds2 length
    uint32_t vs1_loc;   // vendor specific location 1
    uint32_t vs1_len;   // vendor specific location 1 length
    uint32_t vs2_loc;   // vendor specific location 2
    uint32_t vs2_len;   // vendor specific location 2 length
} wiced_fw_upgrade_nv_loc_len_t;
```

- `p_sflash_size`: Serial flash size present on the tag board.

Returns

`WICED_TRUE` if locations and length were validated successfully. If the initialization function returns `WICED_FALSE`, future attempts to start another firmware upgrade would fail. In this state, the only way to program a new version is to program the serial flash directly or over the HCI UART.

3.2 Firmware upgrade initialize nonvolatile storage locations

The OTA firmware upgrade module or the application must call this during the start of the firmware download process to set up memory locations. If a download was started but not successfully completed, this function must be called again. The module will call this function when the module's state changes from "idle" to "ready for download" based on commands received from the control point. This description is provided for completeness in case customization of the download process is desired.

Prototype

```
wiced_bool_t wiced_firmware_upgrade_init_nv_locations(void);
```

Parameters

None.

Returns

`WICED_TRUE` if success; `WICED_FALSE` otherwise.

3.3 Firmware upgrade store data to nonvolatile storage

This function can be called by the OTA firmware upgrade module or by the application to store a chunk of data to the physical nonvolatile storage medium. The inactive partition will be written to. The application does not need to know which type of memory is used or which partition is being upgraded. Typically, the OTA procedure will call this function when it receives the next data packet from the Downloader. This description is provided for completeness. The example applications provided rely on the upgrade module to call this function as needed.

Prototype

```
uint32_t wiced_firmware_upgrade_store_to_nv(uint32_t offset, uint8_t *data, uint32_t
len);
```

Parameters

- `offset`: Memory offset where the data will be stored.
- `data`: Pointer to the chunk of data to be stored.

- `len`: Size of the memory chunk to be stored.

Returns

Number of bytes stored to the storage if successful; 0 otherwise.

3.4 Firmware upgrade retrieve data from nonvolatile storage

This function can be called by the OTA firmware upgrade module or by the application to retrieve a chunk of data from the physical nonvolatile storage medium. The data is read from the inactive DS partition. The application does not need to know which type of memory is used or which partition is being upgraded. Typically, the OTA procedure will call this function during the verification to validate that the full and correct image has been stored. This description is provided for completeness. The example applications provided rely on the upgrade module to call this function as needed.

Prototype

```
uint32_t wiced_firmware_upgrade_retrieve_from_nv(uint32_t offset, uint8_t *data,
uint32_t len);
```

Parameters

- `offset`: Memory offset from which the data will be retrieved.
- `data`: Pointer to where the library will deposit the retrieved data.
- `len`: Size of the memory chunk to be retrieved.

Returns

Number of bytes retrieved from the storage if successful; 0 otherwise.

3.5 Firmware upgrade finish

After the download is completed and verified, this function may be called to switch the active partition with the one that has been receiving the new image. This function invalidates the previously active partition and initiates the reboot.

Prototype

```
void wiced_firmware_upgrade_finish(void);
```

Parameters

None.

Returns

None.

3.6 OTA firmware upgrade initialization

The application that wants to utilize the OTA firmware upgrade module functionality must call this function during startup. It can optionally register a callback to be issued at the end of the upgrade procedure just before the chip is rebooted. The application that wants to use the ECDSA firmware verification method must pass a pointer for valid public key. If the application uses simple CRC32 verification, the pointer to the public key must be set to NULL.

Prototype

```
wiced_bool_t wiced_ota_fw_upgrade_init(void *p_public_key,
wiced_ota_firmware_upgrade_status_callback_t *p_status_callback,
wiced_ota_firmware_upgrade_send_data_callback_t *p_send_data_callback);
```

Parameters

- `p_public_key`: If the application requires ECDSA verification, it must pass the pointer to the public key stored in the image. Otherwise, the application must pass NULL pointer.
- `p_status_callback`: Optional callback to be executed when the Firmware Upgrade state changes. NULL if not used. The callback is defined as:

  ```
  typedef void (wiced_ota_firmware_event_callback_t)(uint16_t event, void
  *p_data);
  ```

- `p_send_data_callback`: Optional callback to be executed to before sending data over the air. NULL if not used. The callback is defined as:

  ```
  typedef wiced_bt_gatt_status_t
  (wiced_ota_firmware_upgrade_send_data_callback_t) (wiced_bool_t
  is_notification, uint16_t conn_id, uint16_t attr_handle, uint16_t val_len,
  uint8_t *p_val);
  ```

Returns

None.

3.7 OTA firmware upgrade connection status

The application utilizing the OTA firmware upgrade module must call the function when a peer device establishes a Bluetooth® Low Energy (LE) connection or the connection goes down.

Prototype

```
void wiced_ota_fw_upgrade_connection_status_event(wiced_bt_gatt_connection_status_t
*p_status);
```

Parameters

- `p_status`: Pointer to a GATT Connection Status structure as received by the application from the stack.

Returns

None.

3.8 OTA firmware upgrade read handler

The application utilizing the OTA firmware upgrade module must call this function to pass GATT Read requests to the library for the attributes that belong to the OTA Upgrade Service. The function returns the data and the error code that must be passed back to the stack.

Prototype

```
wiced_bt_gatt_status_t wiced_ota_fw_upgrade_read_handler(uint16_t conn_id,
wiced_bt_gatt_read_t *p_read_data);
```

Parameters

- `conn_id`: GATT connection ID.
- `p_read_data`: Pointer to the GATT Read structure that the application receives from the stack.

Returns

Status of the GATT read operation.

3.9 OTA firmware upgrade write handler

The application that uses the OTA firmware upgrade module must call this function to pass GATT Write requests to the library for the attributes that belong to the OTA Upgrade Service. This function must not be called if the application is using the ECDSA verification method.

Prototype

```
wiced_bt_gatt_status_t wiced_ota_fw_upgrade_write_handler(uint16_t conn_id,
wiced_bt_gatt_write_t *p_write_data);
```

Parameters

- `conn_id`: GATT connection ID.
- `p_write_data`: Pointer to the GATT Write structure that the application receives from the stack.

Returns

Status of the GATT Write operation.

3.10 OTA firmware upgrade indication confirmation

The application utilizing the OTA firmware upgrade module must call this function to pass GATT Indication Confirm requests to the library for the attributes that belong to the OTA Upgrade Service.

Prototype

```
wiced_bt_gatt_status_t wiced_ota_fw_upgrade_indication_cfm_handler(uint16_t conn_id,
uint16_t handle);
```

Parameters

- `conn_id`: GATT connection ID.
- `handle`: Attribute handle for which the indication confirm message has been received.

Returns

Status of the GATT indication confirm operation.

Revision history

Document version	Date of release	Description of changes
**	2017-08-23	Initial release
*A	2018-10-15	Updated for ModusToolbox
*B	2019-02-18	Updated for CYW20819
*C	2019-04-24	Removed Associated Part Family Updated for BTSDK release
*D	2019-10-15	Sunset Review Updated for ModusToolbox 2.0
*E	2020-01-30	Updated for “Copy from Storage to First Partition” method.
*F	2020-11-23	Added HCI firmware Updated descriptions Moved to Infineon Template
*G	2021-02-25	Updated Figure 2
*H	2021-10-28	Updated terminology per Bluetooth® SIG and Marketing

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

Edition 2021-10-28
Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?
Go to www.cypress.com/support

Document reference
002-19289 Rev. *H

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

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.