lib_dfu: Device Firmware Upgrade (DFU)

Publication Date: 2026/5/1
Document Number: XM-015482-UG v2.1.0

XMOS

IN THIS DOCUMENT

1 Introduction . . . 3
2 Usage . . . 3
3 Operation . . . 4
3.1 DFU Overview . . . 4
3.2 DFU Configuration . . . 4
3.3 DFU Resources . . . 5
3.3.1 Design Constraints . . . 5
3.4 DFU Lifecycle . . . 6
3.5 DFU and Device Security . . . 6
3.6 DFU Operations . . . 7
3.6.1 Summary . . . 7
3.6.2 XCORE Boot Process . . . 7
3.6.3 Description . . . 7
3.6.4 Determining the DFU mode . . . 11
3.6.5 Extensions for non-USB transports . . . 11
4 Host applications . . . 12
4.1 Building the host applications . . . 12
4.1.1 Host dependencies . . . 12
4.2 Host `dfu_i2c` application . . . 12
4.3 Host `suffix_generator` application . . . 13
4.4 Host `libsuffix_verifier` library . . . 13
4.5 Host `xmosdfu` application . . . 13
5 Example application . . . 14
5.1 Building the example . . . 14
5.2 Building the example host app . . . 14
5.3 Example Hardware Setup . . . 14
5.4 Determining the XTAG adapter ID . . . 16
5.5 Running the example . . . 16
5.6 Running the host example . . . 17
5.6.1 Output when running the example . . . 17
6 References . . . 18
7 Functional API . . . 19
7.1 Data Structures/Types . . . 19
7.2 API Functions . . . 22
7.2.1 <dfu.h> . . . 22
7.2.2 <dfu_reboot.h> . . . 23
7.2.3 <dfu_usb_server.h> . . . 23
7.2.4 <dfu_usb_requests.h> . . . 23
7.2.5 <dfu_control_server.h> . . . 25
7.3 DFU Configuration Options . . . 25

1 Introduction

The Device Firmware Upgrade (DFU) library provides functionality to facilitate firmware upgrades over various transport mechanisms. It includes support for handling DFU packets, managing firmware images, and ensuring the integrity of the upgrade process.

2 Usage

`lib_dfu` is intended to be used with the XCommon CMake , the *XMOS* application build and dependency management system.

To use this library in an application include `lib_dfu` in the application's `APP_DEPENDENT_MODULES` list in *CMakeLists.txt*, for example:

```
set(APP_DEPENDENT_MODULES "lib_dfu")
```

Note

Dependent modules should be pinned to release versions where possible, otherwise the latest commit on the *develop* branch will be used. For further details on managing modules, pinning to a release version and other options, please see the page xcommon-cmake Dependency Management.

For flash memory support in an application, this library requires linking to the Tool's Quad Flash library `quadflash` in the application's `APP_COMPILER_FLAGS` list in *CMakeLists.txt*, for example:

```
set(APP_COMPILER_FLAGS  ... -lquadflash ...)
```

All `lib_dfu` functions can be accessed via the `dfu.h` header file, for example:

```
#include "dfu.h"
```

The library uses an optional header, `dfu_conf.h`, when present in the application allows the user to reconfigure the library. For example the following configuration allows the user to define the device's firmware version. Please see the *DFU Configuration Options* section for details on all available configuration options.

```
#define DFU_BCD_DEVICE 0x0101
```

3 Operation

3.1 DFU Overview

In lib_dfu, the DFU implementation is transport agnostic, and can be used with any physical transport by implementing the necessary transport layer to receive DFU commands and send them to the DFU task.

This library is intended for a host device to update a target device, so the DFU implementation is designed to be used in the target device, and a separate host application will need to be used to send the DFU commands from the host to the target device.

For DFU over USB, the DFU implementation in lib_dfu is compliant with version 1.1 of Universal Serial Bus Device Class Specification for Device Firmware Upgrade.

This section describes the DFU implementation in lib_dfu. For information about using a DFU loader to send DFU commands to a USB device, refer to app-note AN02019: Using Device Firmware Upgrade (DFU) in USB Audio.

For detailed information on creating a USB device with DFU capabilities, please see lib_xua: Device Firmware Upgrade (DFU) over USB

For DFU over non-USB transports, the transport layer will need to be selected by the user, the example application in this library demonstrates how to implement DFU over I2C using lib_device_control.

This library uses the flash user library to manage the flash memory where the firmware images are stored, please see XMOS flash user library.

As seen in the flash library documentation the flash memory is divided into two sections, the factory image section and the upgrade image section. Under normal circumstances the factory image will remain present for the life of the device and is used as a fallback in case the upgrade image is invalid. The upgrade image is present after the new firmware is written during the DFU download process.

3.2 DFU Configuration

Table 1 lists a few important DFU related configuration options.

For full details of all configuration options please see the *DFU Configuration Options* section.

Table 1: DFU defines

Define	Description	Default
DFU_ENABLE	Enable DFU functionality for device applications, disable for host applications.	1 (Enabled)
DFU_USB_EN	Enable DFU over USB support. Disable for non-USB transports.	0 (Disabled)
DFU_CONTROL_SERVER	Enable (non-USB) DFU over lib_device_control support. Disable for USB transports.	0 (Disabled)
DFU_BCD_DEVICE	Device release number in binary-coded decimal (BCD) format.	0x0100 (Version 1.0)

3.3 DFU Resources

The DFU implementation in `lib_dfu` uses a number of threads to perform the DFU process.

For USB transports, there is the USB `lib_xud` thread, which handles the USB communication and passes it to the `endpoint0` thread, which are outlined in Fig. 1. The `endpoint0` thread receives the DFU commands from the host and sends them to the DFU task. Due to the use of the `i_dfu` interface, the DFU task can be `distributable` and thus called directly from the `endpoint0` thread, so no additional threads are typically needed for the DFU task. However, if XUD is not run on tile[0], DFU will require its own thread on tile[0].

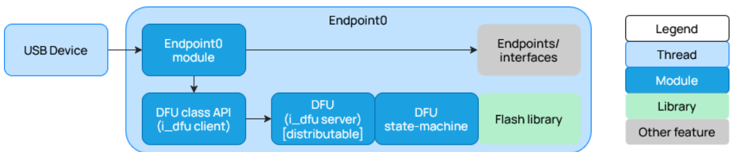


Fig. 1: DFU thread diagram for USB transports

For non-USB transports, there is a thread for the physical transport layer, a thread for the lib_device_control `control` client, and one thread for the `control` server with the DFU task, which are outlined in Fig. 2. Three threads in total.

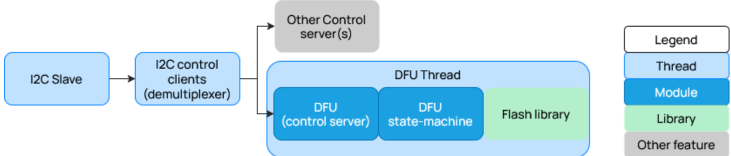


Fig. 2: DFU thread diagram for non-USB transports

3.3.1 Design Constraints

The main constraint to be aware of when designing DFU functionality into an application is that the DFU task must be running on `tile[0]`, as the flash memory is only accessible from tile 0. For details on working with flash memory on XCORE devices, please see Design and manufacture systems with flash memory.

The physical transport can be on either tile as there is an interface to bridge the communications.

The secondary constraint is the time taken to perform the DFU process, particularly the flash erase and write operations, which can take a a few minutes to complete with very large images (>512 kB).

The timing of the DFU process can be managed by setting the appropriate response values for the DFU status requests managed through the configuration defines that start `POLL_TIMEOUT_DNLOAD_`, see *DFU Configuration Options*, to allow the host to know how long to wait before sending the next command. Many of these values can be taken from the flash memory data-sheet.

3.4 DFU Lifecycle

The lifecycle of a product with DFU is an important consideration when designing the DFU functionality, as it can impact the user experience and the security of the device.

The main stages of the product are:

1. **Manufacturing**: During this stage, the device is produced and the factory firmware image is flashed onto the device. The DFU functionality should be tested during this stage to ensure that it is working correctly.
2. **Deployment**: During this stage, the device is shipped to the end-users. The DFU functionality should be available to allow for firmware updates and bug fixes.
3. **Maintenance**: During this stage, the device is in use and may require firmware updates to add new features or fix bugs. The DFU functionality should be reliable and secure to ensure that firmware updates can be performed without issues.
4. **End of Life**: During this stage, the device is no longer supported and may not receive firmware updates. The DFU functionality may still be present, but it may not be secure or reliable. This can also include decommissioning of the device, where the DFU functionality can be used to erase the firmware and render the device inoperable.

Carefully considering the DFU lifecycle, and designing the DFU functionality accordingly, can help ensure that the device is secure and provides a good user experience throughout its lifecycle.

3.5 DFU and Device Security

Device security is an important consideration when implementing DFU functionality, as it involves allowing firmware updates which, if not properly secured, could be exploited by malicious actors.

> **⚠ Warning**
>
> No security is implemented in the DFU implementation in `lib_dfu`. It is the responsibility of the user to implement necessary security measures in their application when using DFU functionality.

For information on securing XCORE devices, please see Safeguard IP and device authenticity.

Here are some security considerations to keep in mind when implementing DFU:

1. **Authentication**: Implement authentication mechanisms to ensure that only authorized users can perform firmware updates.
2. **Encryption**: Consider encrypting the firmware images to protect them from being tampered with or reverse engineered.
3. **Access Control**: Implement access control mechanisms to restrict who can initiate the DFU process.
4. **Secure Boot**: Implement a secure boot process that verifies the integrity of the firmware at startup.

5. **Rollback Protection**: Implement mechanisms to prevent rollback attacks, where an attacker tries to install an older, vulnerable version of the firmware.

6. **Logging and Monitoring**: Implement logging of DFU operations and monitor for any suspicious activity.

By carefully considering these security aspects and implementing appropriate measures, can help ensure that the DFU functionality in the user's device is secure and resistant to potential attacks.

3.6 DFU Operations

3.6.1 Summary

The DFU operations process is as follows:

1. The device starts in runtime mode, running the factory image or the upgrade image if it is valid.
2. The host sends a detach command to the device to switch it to DFU mode.
3. The device reboots into DFU mode and enumerates with the DFU descriptors.
4. The host sends DFU download or upload commands to write or read the firmware image to or from the device.
5. The host sends a detach command to the device to switch it back to runtime mode.

> **Note**
>
> For non-USB transports, the device simply switches to DFU mode without rebooting, and the host can determine the mode switch by querying the device for its descriptors.

> **Note**
>
> To return to runtime mode, the host typically sends a detach command to the device. A bus-reset is supported by the device but Windows hosts do not support sending a bus reset over USB when using the WINUSB driver.

3.6.2 XCORE Boot Process

The DFU depends on the XCORE boot process, and the role of the flash loader to run the upgrade image when valid, for more details please see Design and manufacture systems with flash memory.

3.6.3 Description

Before starting the DFU upload or download process, the host sends a `DFU_DETACH` command to detach the device from runtime to DFU mode, see Fig. 3. In response to the `DFU_DETACH` command, the device reboots itself into DFU mode and enumerates using the DFU mode descriptors. For non-USB transports such as I2C, the device simply transfers into DFU mode without rebooting, after a pause the host can determine the mode switch by querying the device for its descriptors. Once the device is in DFU mode, the DFU interface can accept commands defined by the DFU 1.1 class specification.

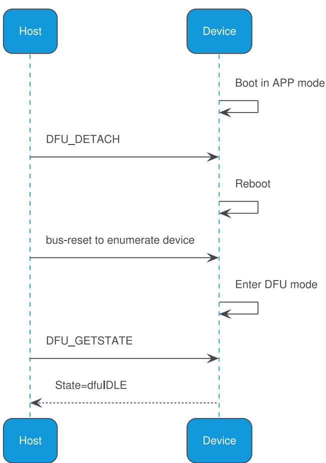


Fig. 3: Message sequence chart for the DFU entry operation

After detaching the device, the host proceeds with the DFU download/upload commands to write/read the firmware upgrade image to/from the device.

During the DFU download process, on receiving the first `DFU_DNLOAD` command, the device starts to erase `FLASH_MAX_UPGRADE_SIZE` bytes of the upgrade section of the flash, see Fig. 4. This is done by repeatedly calling the flash erase function until the entire upgrade section is erased, and can take several seconds. To avoid the `DFU_DNLOAD` request timing out, the flash erase is instead done in the `DFU_GETSTATUS` handling code. So, the device ends up returning the status as `dfuDNBUSY` several times while the flash erase is in progress.

Fig. 5 describes the DFU download process following the erase operation, where the device receives data blocks and writes pages to flash (flash page is typically 256 bytes).

Once the DFU download or upload process is complete, the host sends a bus-reset to the device to switch it back to runtime mode.

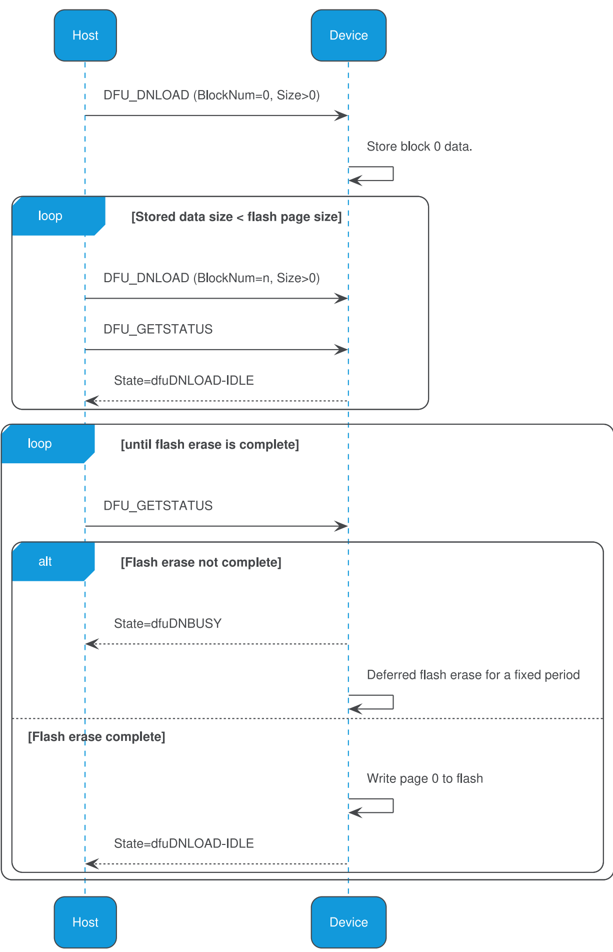


Fig. 4: Message sequence chart for the DFU erase operation

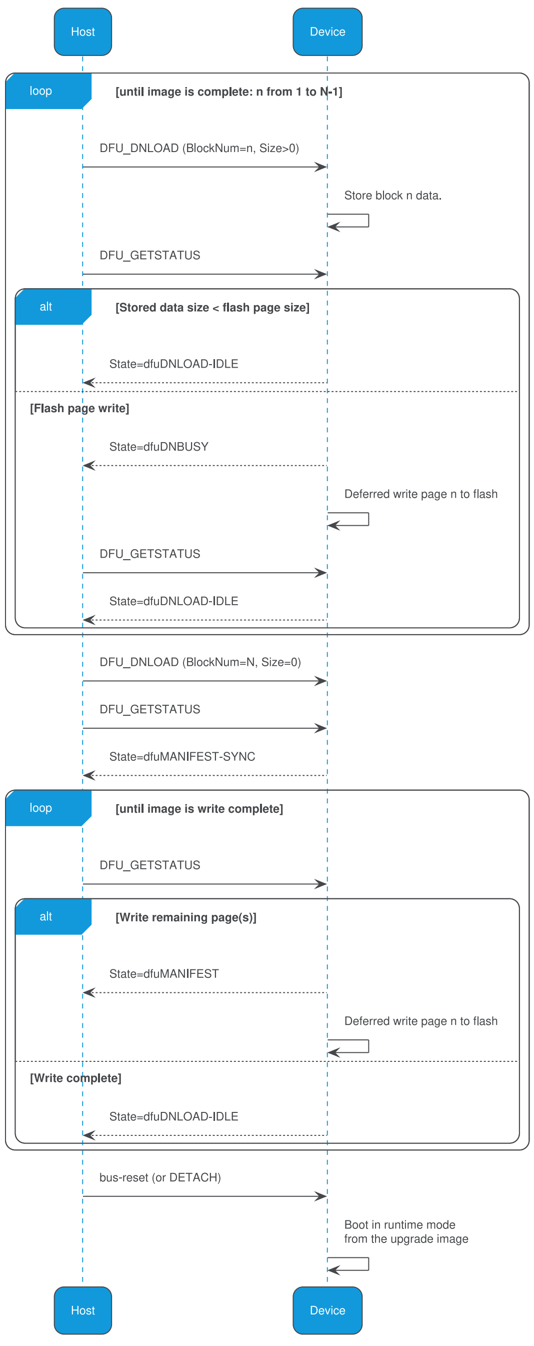


Fig. 5: Message sequence chart for the DFU download operation

> **Note**
>
> Once a valid upgrade image is loaded in flash, on subsequent reboots, the device will boot from the upgrade image. If the upgrade image is invalid, the factory image will be loaded. To revert back to the factory image, send the command `XMOS_DFU_REVERTFACTORY` to erase the upgrade image.

3.6.4 Determining the DFU mode

For USB transport, the device will enumerate with different USB descriptors, depending on whether it is in runtime mode or DFU mode.

For non-USB transports there is an extended command to allow the host to query the device, `XMOS_DFU_GET_DESCRIPTOR`.

3.6.5 Extensions for non-USB transports

To emulate the behaviour of a USB host there are some additional commands defined in the `dfu_cmd_request` enum for non-USB transports, these are:

- **XMOS_DFU_BUS_RESET**: For emulating bus/device reset
- **XMOS_DFU_GET_DESCRIPTOR**: For emulating getting a descriptor

Most of the slow operations are handled during the hold-off time allowed for after the `DFU_GETSTATUS` request by the *poll_timeout* value response. But there are a few other operations for non-USB transports to be aware of, these are described below.

Timings for `XMOS_DFU_BUS_RESET`, when transitioning to DFU mode, the host should hold off for at least 100ms to allow the device to prepare the flash memory for the DFU process. When transitioning to APP mode, the host should hold-off the next request for at least 500ms after sending the bus reset command to allow the device to reboot and be ready for the next request.

Timings for `XMOS_DFU_REVERTFACTORY`, the host should hold-off the next request for at least 250ms after sending the command to allow the device to erase the upgrade image and be ready for the next request.

4 Host applications

4.1 Building the host applications

This section assumes that the host compiler is installed and in the path, for details per host OS please see *Host dependencies*.

For Linux and Mac hosts, the host app can be built from a command terminal with the commands as shown:

```
cd lib_dfu/host
cmake -G "Unix Makefiles" -B build
xmake -j -C build
```

For Windows hosts the process is the same except the Ninja generator is recommended to be used with CMake and the executable will have a `.exe` extension. The commands as shown:

```
cd lib_dfu\host
cmake -G "Ninja" -B build
cmake --build build
```

The built host application executable will be found in the `bin` subdirectory under each of the host directories.

4.1.1 Host dependencies

For Windows hosts, the supported compiler is *MSVC*. The Ninja build system is recommended to be used with CMake, but it is not required. The `libusb` library is provided in the library for applications such as `xmosdfu`.

For Linux hosts, including Raspberry Pi, the supported compiler is *GCC*. The `libusb-1.0-0-dev` package must be installed for USB transport.

For OSX hosts, the supported compiler is *Clang*. The `libusb` library is provided in the library for applications such as `xmosdfu`.

4.2 Host `dfu_i2c` application

This application provides a fully featured DFU host implementation over I2C using `lib_device_control` to send control commands.

Note

The application is only supported on a Linux host with I2C, such as a Raspberry Pi.

The application supports DFU with the following command-line arguments:

- `--help`: Show help message and exit
- `detach_and_bus_reset`: Send the detach command followed by a bus reset to switch the device from runtime to DFU mode
- `reboot`: Send a bus reset to switch the device from DFU to runtime mode
- `write_upgrade <boot.dfu>`: Write the specified firmware image to the device using DFU download commands, the file must include the DFU suffix.

- upload <file.bin>: Read the firmware image from the device using DFU upload commands and save it to the specified file, the file will be binary and not include the DFU suffix.
- `revert_factory`: Revert back to the factory image by erasing the first flash sector of the upgrade image.

The I2C address can be configured using --i2c-address <address> argument, which defaults to 0x2C.

4.3 Host `suffix_generator` application

This is host app to generate the suffix of the firmware image in flash, this is used in the DFU download process to write the suffix to flash after writing the firmware image.

Note

The USB DFU specification requires a valid suffix to be present in the firmware image.

Before generating a suffixed binary file, the xe file will need to be processed using `xflash`.

```
After building the example...
cd examples/i2c/device
xflash --factory-version 15.3 --upgrade 1 bin/update/i2c_update.xe -o bin/update/i2c_update.bin
```

After extracting the binary file, a suffixed binary file can be generated using the `suffix_generator` application with the following command-line arguments:

```
cd lib_dfu/host/suffix_generator
./bin/dfu_suffix_generator <VID> <PID> <bcdDevice> <input_binary> <output_file>
```

So for example the following would generate a suffixed binary file with the VID, PID and bcdDevice values for the example application:

```
./bin/dfu_suffix_generator 0x20b1 0x1234 0x0101 i2c_update.bin i2c_update_suffixed.bin
```

It is recommended to update the `bcdDevice` value for each new firmware version to allow the host to easily identify different versions of the firmware.

4.4 Host `libsuffix_verifier` library

This is host library can be used to verify the suffix of the firmware image in flash is correct, *Host dfu_i2c application* uses this to verify firmware images before upgrading the device.

4.5 Host `xmosdfu` application

`xmosdfu` is a USB DFU host application that can be used to perform DFU operations such as firmware upgrades on XMOS devices over USB. It uses the `libusb` library to communicate with the device over USB and provides a command-line interface.

For detailed instructions on using `xmosdfu`, please see AN02019: Using Device Firmware Upgrade (DFU) in USB Audio.

5 Example application

5.1 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `lib_dfu` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd lib_dfu/examples/i2c/device
cmake -G "Unix Makefiles" -B build
```

If any dependencies are missing they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `examples/i2c/device/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

5.2 Building the example host app

This is very similar to building the device example, please follow the build instructions in the *Building the example* section to build the host application, except the host example is in the `examples/i2c/host_xcore` directory.

5.3 Example Hardware Setup

The example is *XCORE* to *XCORE* over I2C, so two *XCORE* boards are needed. The example was developed and tested using the `XK-EVK-XU316` and the `XK-VOICE-L71`.

Connect three jumper wires between the `XK-EVK-XU316` and the `XK-VOICE-L71` to allow the I2C communication between the host and device.

For the `XK-EVK-XU316`, connect the jumper wires as shown in the Fig. 6 and Fig. 7, connecting the I2C *SCL*, *SDA* and *GND* pins to the corresponding pins on the `XK-VOICE-L71`.

The `XK-VOICE-L71` uses the Raspberry Pi GPIO pins for I2C communication. For more information, refer to the Raspberry Pi GPIO Documentation.

To run the example, connect a USB cable to power the `XK-VOICE-L71` board as shown in Fig. 8, and plug the XTAG to the board and connect the XTAG USB cable to the devel-

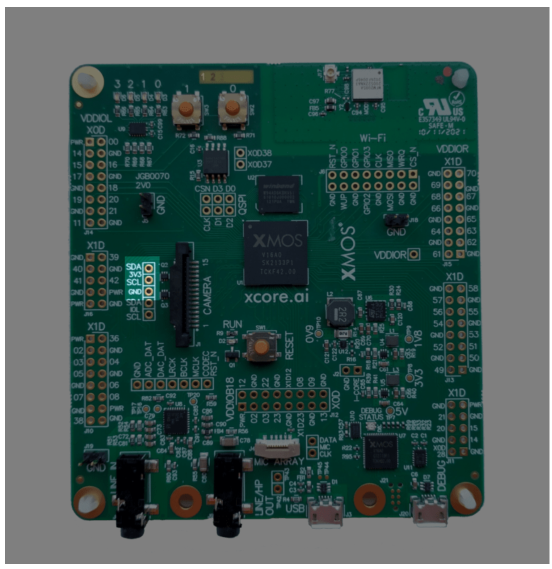


Fig. 6: XK-EVK-XU316 I2C Hardware setup

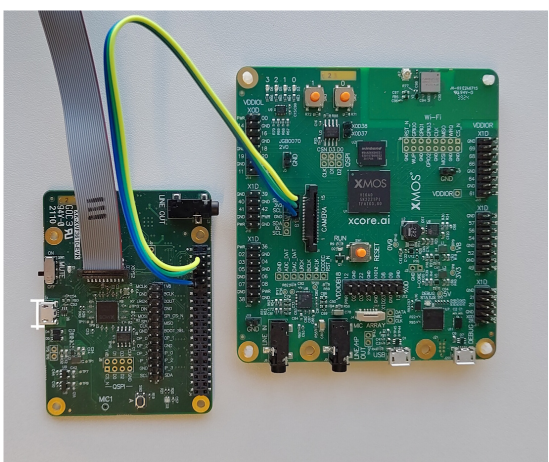


Fig. 7: Connecting I2C between the XK-EVK-XU316 and XK-VOICE-L71

opment machine. And also connect two USB cables to the `XK-EVK-XU316` to power it and allow programming and xscope communication.

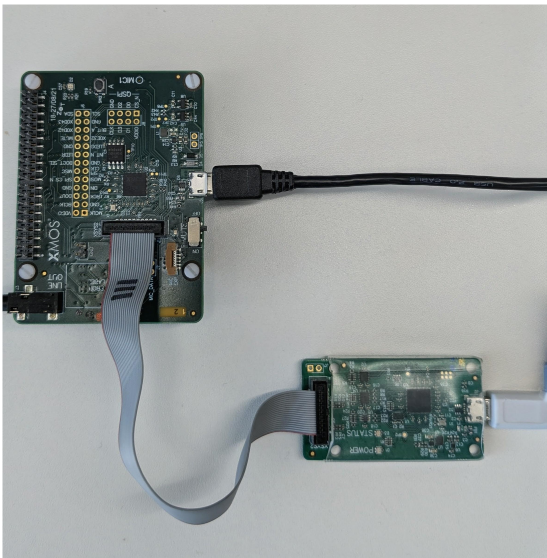
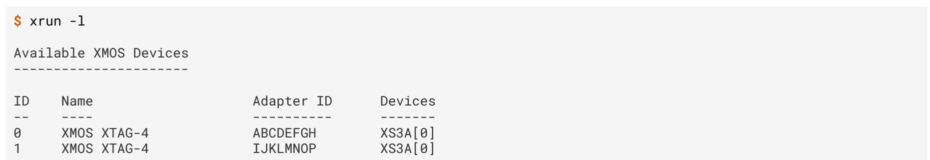


Fig. 8: XK-VOICE-L71 USB Hardware setup

5.4 Determining the XTAG adapter ID

When running two XTAGs connected to the development machine, one for the host and one for the device, the XTAG adapter ID will need to be provided to *xrun* to specify which XTAG to use for the host and which one to use for the device. The XTAG adapter ID's can be found using the *xrun* command from an XTC command prompt. An example output is shown below:



```
$ xrun -l

Available XMOS Devices
----------------------

ID    Name                Adapter ID     Devices
--    ----                ----------     -------
0     XMOS XTAG-4         ABCDEFGH       XS3A[0]
1     XMOS XTAG-4         IJKLMNOP       XS3A[0]
```

5.5 Running the example

The device example needs to be running, otherwise the host will fail to connect to the device.

Note

The example is not a fully featured DFU implementation, it is designed to show the DFU detach process from runtime to DFU mode, so the device example will start in runtime mode, and only switch to DFU mode after receiving the detach command from the host.

From an XTC command prompt, the following command should be run from the `examples/i2c/device` directory:

```
xrun --xscope --adapter-id <device_xtag_adapter_id> ./bin/i2c.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash --adapter-id <device_xtag_adapter_id> ./bin/i2c.xe
```

5.6 Running the host example

From an XTC command prompt, the following command should be run from the `examples/i2c/host_xcore` directory:

```
xrun --xscope --adapter-id <host_xtag_adapter_id> ./bin/host_i2c_xcore.xe
```

5.6.1 Output when running the example

The output from the host shows the DFU detach process, followed by a simple control data exchange between the host and device to show that the communication is working as expected.

```
xrun --xscope --adapter-id <host_xtag_adapter_id> ./bin/host_i2c_xcore.xe
i2c ready
Starting Control DFU example
DFU state: 0
DFU status: 0, timeout: 0 ms, next state: 0
Sent detach command
DFU status: 0, timeout: 0 ms, next state: 1
Sent bus reset command
DFU status: 0, timeout: 0 ms, next state: 2
control write command 9 failed with 5
Sent bus reset command
DFU status: 0, timeout: 0 ms, next state: 0
Starting Control data example
Written and read back command with payload: 0x00
Written and read back command with payload: 0x01
Written and read back command with payload: 0x02
Written and read back command with payload: 0x03
done
```

6 References

- USB Firmware Upgrade (DFU) 1.1
- XMOS flash user library
- AN02019: Using Device Firmware Upgrade (DFU) in USB Audio
- Safeguard IP and device authenticity
- Design and manufacture systems with flash memory

7 Functional API

See *Usage* section and *Example application* for details on usage of the following.

7.1 Data Structures/Types

enum **dfu_state**

DFU state machine states, from USB DFU spec v1.1

Values:

enumerator **STATE_APP_IDLE**

enumerator **STATE_APP_DETACH**

enumerator **STATE_DFU_IDLE**

enumerator **STATE_DFU_DOWNLOAD_SYNC**

enumerator **STATE_DFU_DOWNLOAD_BUSY**

enumerator **STATE_DFU_DOWNLOAD_IDLE**

enumerator **STATE_DFU_MANIFEST_SYNC**

enumerator **STATE_DFU_MANIFEST**

enumerator **STATE_DFU_MANIFEST_WAIT_RESET**

enumerator **STATE_DFU_UPLOAD_IDLE**

enumerator **STATE_DFU_ERROR**

enum **dfu_status**

DFU device status code, from USB DFU spec v1.1

Values:

enumerator **DFU_OK**

enumerator **DFU_errTARGET**

enumerator **DFU_errFILE**

enumerator **DFU_errWRITE**

enumerator **DFU_errERASE**

enumerator **DFU_errCHECK_ERASED**

enumerator **DFU_errPROG**

enumerator **DFU_errVERIFY**

enumerator **DFU_errADDRESS**

enumerator **DFU_errNOTDONE**

enumerator **DFU_errFIRMWARE**

enumerator **DFU_errVENDOR**

enumerator **DFU_errUSBR**

enumerator **DFU_errPOR**

enumerator **DFU_errUNKNOWN**

enumerator **DFU_errSTALLED_PKT**

enum **dfu_cmd_request**

DFU request types

Largely follows the USB DFU class specification requests, with some additional custom requests for non-USB transports and deferred actions. Deferred action requests are not actual requests sent by the host, but are used internally to indicate slow operations to carry out after responding to the initial request, such as flash programming and rebooting, which cannot be completed within the time constraints of a single request/response transaction.

Values:

enumerator **DFU_DETACH**

enumerator **DFU_DNLOAD**

enumerator **DFU_UPLOAD**

enumerator **DFU_GETSTATUS**

enumerator **DFU_CLRSTATUS**

enumerator **DFU_GETSTATE**

enumerator **DFU_ABORT**

enumerator **XMOS_DFU_BUS_RESET**

For emulating bus/device reset on transports other than USB.

enumerator **XMOS_DFU_GET_DESCRIPTOR**

For emulating getting a descriptor on transports other than USB.

enumerator **DFU_DEFERRED_ACTION_REBOOT**

enumerator **DFU_DEFERRED_ACTION_REBOOT_TO_DFU**

enumerator **DFU_DEFERRED_ACTION_REVERT_FACTORY**

enumerator **DFU_DEFERRED_ACTION_FLASH_CONNECT**

enumerator **DFU_DEFERRED_ACTION_FLASH_WRITE**

enumerator **DFU_DEFERRED_ACTION_FLASH_MANIFEST**

enumerator **XMOS_DFU_GETPROFILE**

enumerator **XMOS_DFU_REVERTFACTORY**

Additional command that erases only the first flash sector of the upgrade image, to revert to the factory image

enum **dfu_api_status**

API function return values

Values:

enumerator **DFU_API_SUCCESS**

enumerator **DFU_API_ERROR**

enumerator **DFU_API_DATA_LENGTH_ERROR**

enumerator **DFU_API_BAD_PARAM**

struct **dfu_cmd_response**

DFU request response structure return type

Public Members

enum *dfu_api_status* **status**

The status of the DFU request

int32_t **return_data_len**

The length of the data returned by the DFU request, only used for read operations.

enum *dfu_cmd_request* **deferred_request**

The deferred DFU request, if any, used to action slow operations.

7.2 API Functions

7.2.1 <dfu.h>

group **API**

Functions

struct *dfu_cmd_response* **dfu_request_with_arguments**(
enum *dfu_cmd_request* request, NULLABLE_ARRAY_OF(uint8_t, block), int32_t block_size_bytes, NULLABLE_REFERENCE_PARAM(int32_t, block_num),
)

DFU request handling with arguments

Parameters

- **request** – the DFU request to handle
- **block** – pointer to the data payload of the request for read-/write operations, usage depends on command, for download it's the data block to write, for upload it's the buffer to fill with the data block read from flash, for other commands it's unused and can be null
- **block_size_bytes** – the size of the data block to read-/write for upload/download commands, for other commands it's unused and can be 0
- **block_num** – for download command, the block number to write, for other commands it's unused and can be null

Return values

- **DFU_API_SUCCESS** – if command was handled successfully, the value is the upload block-number for upload command, 0 otherwise.
- **DFU_API_ERROR** – if there was an error handling the command
- **DFU_API_BAD_PARAM** – if the command or parameters were invalid

Returns

struct *dfu_cmd_response* containing status and any return value, usage depends on command, for upload the return_data_len is the size of the block to upload, for other commands it's unused and can be 0, the deferred_request field is used to indicate any slow operations to carry out after responding to the request, such as flash programming and rebooting, which cannot be completed within the time constraints of a single request/response transaction.

struct *dfu_cmd_response* **dfu_request**(enum *dfu_cmd_request* request)

Send request to DFU with no data

Parameters

- **request** – the DFU request to send

Return values

- **DFU_API_SUCCESS** – for status, if command was handled successfully
- **DFU_API_ERROR** – for status, if there was an error handling the command

- **DFU_API_BAD_PARAM** – for status, if the command or parameters were invalid

Returns
struct *dfu_cmd_response* identical to dfu_request_with_arguments return value.

7.2.2 <dfu_reboot.h>

void **dfu_reboot**(int32_t delay_ms)

Reboot the device

This function reboots the device by writing to the PLL config register. This will call dfu_user_pre_reboot() before rebooting, which allows the user to perform any necessary cleanup or preparation before the device reboots.

Parameters

- **delay_ms** – The delay in milliseconds before the reboot occurs.

void **dfu_user_pre_reboot**(void)

Function to notify user's application that a reboot is about to occur.

This is called from dfu_reboot() before the device is rebooted, and can be used to perform any necessary cleanup or preparation before the reboot occurs.

The default weak implementation does nothing, but the user can override this function with their own implementation if desired.

7.2.3 <dfu_usb_server.h>

void **dfu_usb_server**(SERVER_INTERFACE(i_dfu, i))

DFU USB server function

This links the USB device to the DFU library, allowing the device to receive DFU commands over USB. Call this function from the application's `Endpoint0` task.

Parameters

- **i** – The usb interface to use for connection to the transport.

7.2.4 <dfu_usb_requests.h>

group **USB API**

Functions

int **dfu_is_mode_active**(void)

Check if DFU mode is active

This is used in a variety of places to coordinate the application. Such as; resolving the interface number to respond to, determining which USB descriptors to return, whether to allow certain commands, etc.

Returns
1 if DFU mode is active, 0 if DFU mode is not active.

void **dfu_set_mode_active**(void)

Set DFU mode as active. Typically not used in the application.

void **dfu_set_mode_inactive**(void)

Set DFU mode as inactive. Typically not used in the application.

int **dfu_check_init_state**(void)

Check the initial state of DFU mode

Used during boot to determine whether to enter DFU mode or not.

When entering DFU mode during boot, this function returns 1, and the application can then disable or halt other tasks not needed during DFU.

Return values

- `0` – if DFU mode should not be active (normal boot)
- `1` – if DFU mode should be active (boot to DFU)

void **dfu_force_dfu_mode_active**(CLIENT_INTERFACE(i_dfu, i))

Force DFU state machine into dfuIDLE state.

Used during boot to force the device into DFU mode. Typically called after *dfu_check_init_state()* returns true, to ensure the device enters DFU mode.

> **Note**
>
> Call only after the DFU thread is running, as otherwise it might deadlock the system.

int **dfu_process_reset_state**(CLIENT_INTERFACE(i_dfu, i))

Handle USB reset events

Call from endpoint0 task when `(result == XUD_RES_UPDATE)` and `XUD_BusState_t busState = XUD_GetBusState(...)` returns `(busState == XUD_BUS_RESET)`.

> **Note**
>
> This function will call the DFU interface, so should only be called after the DFU thread is running. If called before the DFU thread is running, or before the interface is available, it may cause a deadlock.

Returns

1 if the device should be in DFU mode, 0 if it should be in application mode.

int **dfu_usb_vendor_requests**(
XUD_ep ep0_out, XUD_ep ep0_in, REFERENCE_PARAM(USB_SetupPacket_t, sp), CLIENT_INTERFACE(i_dfu, dfuInterface), unsigned int xua_dfu_interface_num,
)

Handle XMOS specific DFU requests

Call from endpoint0 task.

Returns

XUD_RES_OKAY if request was handled, XUD_RES_ERR if request was not recognised/handled.

int **dfu_usb_class_int_requests**(
XUD_ep ep0_out, XUD_ep ep0_in, REFERENCE_PARAM(USB_SetupPacket_t, sp), CLIENT_INTERFACE(i_dfu, dfuInterface),
)

Handle standard DFU requests

Call from endpoint0 task.

Returns

XUD_RES_OKAY if request was handled, XUD_RES_ERR if request was not recognised/handled.

void **dfu_usb_set_configured_state**(void)

Handle standard SET_CONFIGURATION request

Informs DFU layer of USB device configured state.

void **dfu_usb_clear_configured_state**(void)

Handle clearing of the USB configured state

Informs the DFU layer that the USB device is no longer configured, for example after SET_CONFIGURATION(0) or equivalent de-configuration.

7.2.5 <dfu_control_server.h>

group **Control API**

Functions

void **dfu_control_server**(SERVER_INTERFACE(control, dfu_control_interface))

DFU control server function

This links the control library to the DFU library, allowing the device to receive DFU commands over any transport supported by the control library (USB, I2C, SPI, XSCOPE). Call this function from `main()`.

The server registers the resource ID for DFU `RESOURCE_ID_DFU`, then listens for read and write commands to that resource.

Parameters

- **dfu_control_interface** – The control interface to use for connection to the transport.

7.3 DFU Configuration Options

group **Configuration Options**

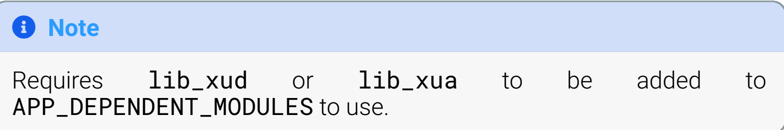
Defines

DFU_ENABLE

Main control for the function of the DFU library. When enabled, the application must provide “-lquadflash” to link the flash library. When disabled will include empty stubs for the API functions. These stubs are weak and can be overridden locally.

DFU_USB_EN

Main control for the USB functionality of the DFU library. When enabled, the DFU library will include USB support. When disabled, USB support will be excluded. Other transports can be used.


Note

Requires `lib_xud` or `lib_xua` to be added to `APP_DEPENDENT_MODULES` to use.

DFU_BCD_DEVICE

USB device release number in binary-coded decimal (BCD) format
Used in DFU for identifying the device version. The default value is 0x0100, which corresponds to version 1.00.
Reported to the host via the USB device descriptor when DFU_USB_EN is enabled. Or, when DFU_USB_EN is disabled, this can be accessed by the host using request XMOS_DFU_GET_DESCRIPTOR.

DFU_USER_FLASH_DEVICE

Optional, user defined flash device specification for DFU to use.
If not defined (default), the libquadflash will attempt to use JEDEC `JESD216` Serial Flash Discoverable Parameters (SFDP) to identify the connected flash device and its capabilities. This requires the flash to be SFDP compliant, and may not work with all flash devices.
If DFU_USER_FLASH_DEVICE is defined, this should be defined as a fl_QuadDeviceSpec or fl_DeviceSpec structure, depending on whether DFU_QUAD_SPI_FLASH is set.

DFU_QUAD_SPI_FLASH

Whether DFU uses Quad SPI Flash, or older SPI flash (when 0)

DFU_TRANSFER_SIZE_BYTES

Number of bytes transferred in each DFU packet on the given physical transport

DFU_FLASH_PAGE_SIZE_BYTES

Number of bytes in a flash page for the target device

NUM_TRANSFER_BLOCKS_PER_FLASH_PAGE

Number of DFU packets per flash page

DFU_CONFIG_USB_INBAND_FUNCTIONS

Whether to enable USB in-band functions for DFU Inband functions occur during the host request transaction. Out-of-band requests are deferred until after the host request has been completed, which require the DFU task to be allocated to a thread.

FLASH_MAX_UPGRADE_SIZE

Defines flash area to erase on first DFU download request received
Flash library will round it up to the nearest sector, e.g. 4KB

CLKBLK_DFU_FLASHLIB

Clock block for use by DFU flash operations

POLL_TIMEOUT_DNLOAD_ENTRY_MSEC

Poll timeout for DFU download entry in milliseconds, time taken to find the upgrade image, if it exists, and erase the first sector.
The first sector erase can often take significantly longer than subsequent erases, as it may require additional setup such as checking whether an upgrade image exists and its size.
Values shorter than the actual time taken may cause the host to timeout or cause clock-stretching if using I2C.

POLL_TIMEOUT_DNLOAD_ERASE_MSEC

Poll timeout for DFU download erase in milliseconds, time given to erase a number of sectors to prepare for the upgrade image.
The intent is that the device should attempt to erase a number of sectors within this time.
This value is that reported to the host to hold-off further requests until a block of erasing is done. DFU_FLASH_ERASE_CYCLE_MSEC is the time duration the device will use which must be shorter than this value to ensure the host does not timeout.
Values shorter than the actual time taken by the flash chip to erase one sector may cause the host to timeout or cause clock-stretching if using I2C.

DFU_FLASH_ERASE_CYCLE_MSEC

The cycle time for the flash erase operation in milliseconds. Allows repeated erase operations to be performed within this time frame.
The margin between this value and POLL_TIMEOUT_DNLOAD_ERASE_MSEC should be sufficient to allow the device flash preparation operations at various stages of the process.

POLL_TIMEOUT_DNLOAD_FIRST_WRITE_MSEC

Poll timeout for DFU download first page write in milliseconds, time taken to prepare for the write and write the first page of the upgrade image.
The first page can often take significantly longer to write than subsequent pages, as it may require additional setup such as preparing the flash for writing.
Values shorter than the actual time taken by the flash chip to write one page may cause the host to timeout or cause clock-stretching if using I2C.

POLL_TIMEOUT_DNLOAD_WRITE_MSEC

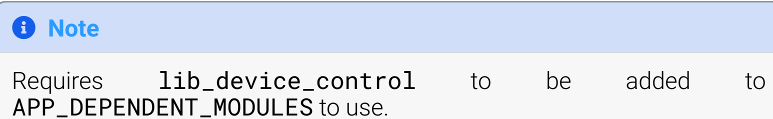
Poll timeout for DFU download write in milliseconds, time taken to write subsequent pages of the upgrade image.
The value should be taken from the Flash memory datasheet, for write operations.
Values shorter than the actual time taken may cause the host to timeout or cause clock-stretching if using I2C.

POLL_TIMEOUT_DNLOAD_MANIFEST_MSEC

Poll timeout for DFU manifest in milliseconds, time taken to write the last page and finalise the write process.
The value should be taken from the Flash memory datasheet, for write operations.
Values shorter than the actual time taken may cause the host to timeout or cause clock-stretching if using I2C.

DFU_CONTROL_SERVER

Main control for the DFU lib_control_device server functionality. When enabled, the DFU library will include the control server for non-USB transports. When disabled, the control server will be excluded.



Note

Requires `lib_device_control` to be added to `APP_DEPENDENT_MODULES` to use.

Copyright © 2026, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.