XMOS XVF3800 - Programming Guide

Release: 3.2.1
Publication Date: 2024/10/29

XMOS

Table of Contents

1 Overview

The XMOS VocalFusion ® XVF3800 is a high-performance voice processor that uses microphone array processing and a sophisticated audio processing pipeline to capture clear, high-quality speech from anywhere in a room. The XVF3800 uses the XMOS xcore.ai processor and supports a range of integrated and accessory voice communication applications.

Fig. 1.1 shows the XVF3800 in context. Only one of the alternate reference audio paths may be present in the product design.
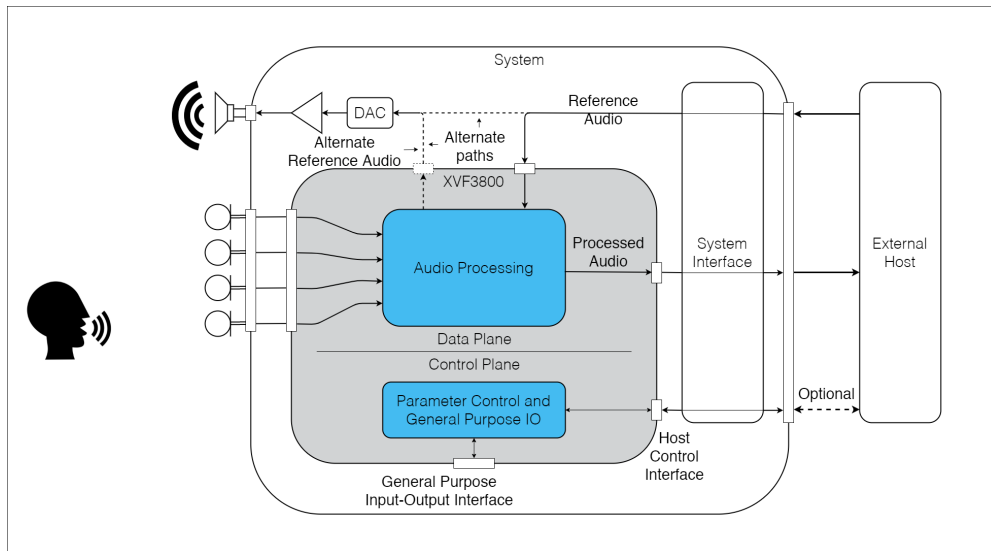
Fig. 1.1: Context Diagram

This document provides information on:

- The XVF3800's theory of operation,

- Advanced options for configuring and building the firmware,

- Methods for testing changes to the XVF3800 firmware, and

- Areas within the XVF3800 firmware intended for customisation.

2 Theory of Operation

2.1 System Architecture Overview

The XVF3800 system subdivides into two major sections: a Control Plane and a Data Plane.

The Control Plane includes all control interfaces, related logic, and housekeeping functions. Control Plane functions have low performance requirements, relaxed timing constraints, and complex logic. The XVF3800 design uses a Real Time Operating System (RTOS) to distribute Control Plane functionality across xCORE tile boundaries.

The Data Plane includes all functions that handle audio processing. These functions have hard real-time constraints and operate isochronously. They generally pass their audio data using buffers. The xCORE processor package imposes a constraint on Data Plane functions receiving data from or providing data to an external source due to the number and width of ports connected to physical pins within the package. The XVF3800 operates the Data Plane functions bare-metal, i.e. without the help of an RTOS.

Both the Control Plane and the Data Plane consist of several modules.

2.1.1 Control Plane Modules

The Control Plane includes the following modules:

- Device Control (DC)
- Device Firmware Update controller (DFU)
- General Purpose Input and Output (GPIO)
- Human Interface Device (HID)
- Input Output Configuration (IO Config)
- Inter-Integrated Circuit Master (I^2C Master)
- Inter-Integrated Circuit Slave (I^2C Slave)
- Quad Serial Peripheral Interface (QSPI)
- Serial Peripheral Interface Slave (SPI Slave)
- Servicers (SER)

2.1.2 Data Plane Modules

The Data Plane includes the following modules:

- Acoustic Echo Cancellation (AEC)
- Audio Manager (AM)
- Beamforming and Post-processing (BAP)
- Customer DSP (DSP)
- Inter-IC Sound (I^2S)
- Microphone Array (MIC)
- Software Phase-Locked Loop (SW PLL)

- Universal Serial Bus (USB)

2.2 Control Plane Module Responsibilities

2.2.1 Device Control

The Device Control module handles the transfer of control messages between a host and the device. It connects to the host control interface, e.g. I^{2}C Slave, SPI Slave or USB, on one end and the command servicers on the other end. It routes a command and its response between the host control interface and the intended servicer for that command.

2.2.2 Device Firmware Update Controller

The Device Firmware Update (DFU) controller processes DFU messages received from a host control interface, e.g. I^{2}C Slave, SPI Slave or USB, and interacts with the QSPI Flash module to read/write to the external flash.

2.2.3 General Purpose Input Output

The General Purpose Input Output (GPIO) module reads General Purpose Input (GPI) pins and writes to General Purpose Output (GPO) pins. These pins allow device interaction with buttons, sliders or knobs for input and Light Emitting Diodes (LEDs) for output. The GPIO module includes the logic to drive GPO pins using Pulse-Width Modulation (PWM).

2.2.4 Human Interface Device

The Human Interface Device (HID) module allows the XVF3800 to operate as a human interface device according to the USB Human Interface Devices specification. Compliance with the USB HID specification allows host devices to interact with physical controls and indicators connected to the XVF3800 through the GPIO module such as buttons.

2.2.5 Input Output Configuration

The Input Output Configuration module configures GPIO devices and an attached Digital to Analogue Converter (DAC). The number of GPIO devices can be extended using the I^{2}C-to-GPIO expander. An example of the IO expander is described in *the Expanding available IO for extended HID support* section.

2.2.6 Inter-Integrated Circuit Master

The Inter-Integrated Circuit Master (I^2C Master) module provides an XVF3800-clocked I^2C data transport for DAC configuration.

2.2.7 Inter-Integrated Circuit Slave

The Inter-Integrated Circuit Slave (I^2C Slave) module provides an externally-clocked I^2C data transport for receiving and responding to control commands including the Direction of Arrival command. The XVF3800 cannot include both this module and the Serial Peripheral Interface Slave module in the same build configuration.

2.2.8 Quad Serial Peripheral Interface

The Quad Serial Peripheral Interface (QSPI) module provides a QSPI data transport for input and output to an attached QSPI Flash memory device. The XVF3800 uses this data transport when booting up from QSPI Flash and during the DFU process.

2.2.9 Serial Peripheral Interface Slave

The Serial Peripheral Interface (SPI) Slave module provides an externally-clocked SPI data transport for receiving and responding to control commands including the Direction of Arrival command. The XVF3800 cannot include both this module and the Inter-Integrated Circuit Slave module in the same build configuration.

2.2.10 Servicers

A set of Servicer (SER) modules handle requests from the Device Control module to get or set controllable parameters. It also provides a response back to the Device Control module. Each Servicer handles a request either on its own or through an underlying resource. When using an underlying resource, each Servicer manages the associated control packet queue and ensures thread safety when modifying shared memory or altering a Data Plane module's controllable parameter.

2.3 Data Plane Module Responsibilities

2.3.1 Acoustic Echo Cancellation

The Acoustic Echo Cancellation (AEC) module removes from the microphone signal the acoustic echos of the reference signal projected into the room by the loudspeaker.

2.3.2 Audio Manager

The Audio Manager (AM) performs a number of functions. It collects individual samples from the microphone array and the reference signal source, and it assembles them into a block for further audio processing. It prepares the reference and microphone signals for acoustic processing by, for instance, changing the reference signal sample rate, amplifying either signal as required, converting them between integer and floating point format, and/or adding any necessary delay to synchronise them. It also includes an audio packing facility that allows the XVF3800 to send a selection of six 16 kHz signals which it time-division multiplexes into two 48 kHz I^2S or USB channels.

2.3.3 Beamforming and Post-processing

After the completion of acoustic echo cancellation, the Beamforming and Post-processing (BAP) module further enhances the audio signal through the use of a multi-beam beamformer, de-reverberation, generalised side-lobe cancellation, dynamic echo and noise suppression, automatic gain control, and application of a limiter.

2.3.4 Customer DSP

The Customer DSP module includes two separate digital signal processing functions set aside to allow customisation of signals as desired for a particular product. The first function allows the customer to alter the reference signal before use by the Acoustic Echo Cancellation module and transmission over I^2S. This function operates at the audio interface sample rate. The second function allows the customer to add processing after the signal emerges from the Beamforming and Post-processing module. This function operates at the internal audio processing sample rate. In it, the customer has access to all four beam signals produced by the BAP module and to the residual signals produced by the AEC module.

2.3.5   Inter-IC Sound

The Inter-IC Sound (I^2S) module provides an audio interface to an integrated processor which supplies the reference signal, consumes the processed audio signal, or both. It also includes an audio unpacking facility that allows the XVF3800 to receive the 16 kHz reference signal and four 16 kHz substitute microphone signals as two 48 kHz time-division multiplexed I^2S channels.

2.3.6   Microphone Array

The Microphone Array (MIC) operates four PDM microphones in either a linear or a square/rectangular configuration. It converts the sample rate of the microphone output to match the audio processing sample rate.

2.3.7   Software Phase-Locked Loop

The Software Phase-Locked Loop (SW PLL) module enables the XVF3800 to synchronize the clock signal used by the microphones with the reference audio signal received via I^2S or USB.

2.3.8   Universal Serial Bus

The Universal Serial Bus (USB) module provides a USB Audio Class 2 (UAC2) interface to a USB host. The host supplies the reference signal, consumes the processed audio signal, or both. This module includes an audio unpacking facility that allows the XVF3800 to receive a 16 kHz reference signal and four 16 kHz substitute microphone signals as two 48 kHz time-division multiplexed USB channels. It also provides a control interface, a DFU interface, and a HID interface used by the Control Plane.

2.4   Product Configurations

The XVF3800 supports two primary use cases:

- Integrated device
- USB accessory

The integrated device use case embeds the XVF3800 within a system that includes a separate, primary microcontroller. The primary microcontroller provides the reference signal to the XVF3800, receives the processed microphone signal from the XVF3800, and initiates any control commands sent to the XVF3800. It also provides all system functionality outside of the audio processing performed by the XVF3800.

The USB accessory use case embeds the XVF3800 within a system that connects to a USB host. The USB host provides the reference signal, receives the processed microphone signal, initiates any control commands, and provides all functionality outside of the XVF3800.

Interface variations for each use case appear in the table below:

Table 2.1: Use Case Interface Variations

Interface Attribute	Integrated Device	USB Accessory
Control Protocol	I2C slave or SPI slave	USB
Data Bit Depth	32	16, 24, or 32
Data Protocol	I^2S slave	USB and I^2S master
Master Clock	Derived or Input	Output

All use cases support either a linear or a square/rectangular geometry of four microphones. Likewise, all use cases support either 16 kHz or 48 kHz operation of the data interface.

A system diagram for each use case appears in Fig. 2.1 and Fig. 2.2.
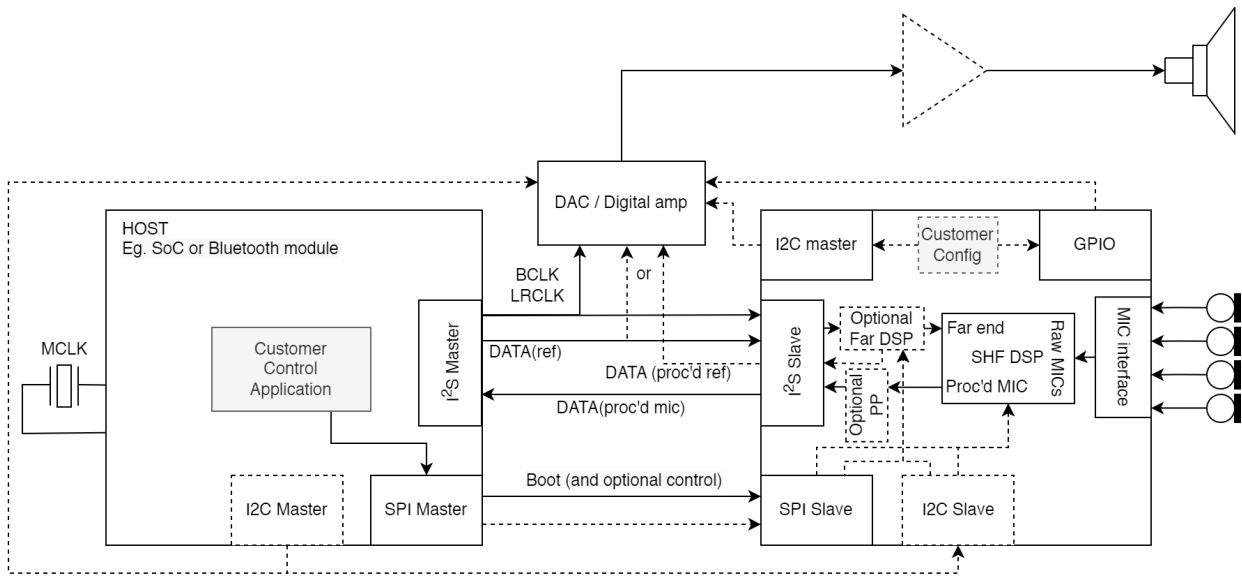
Fig. 2.1: XVF3800 Integrated Device System Diagram
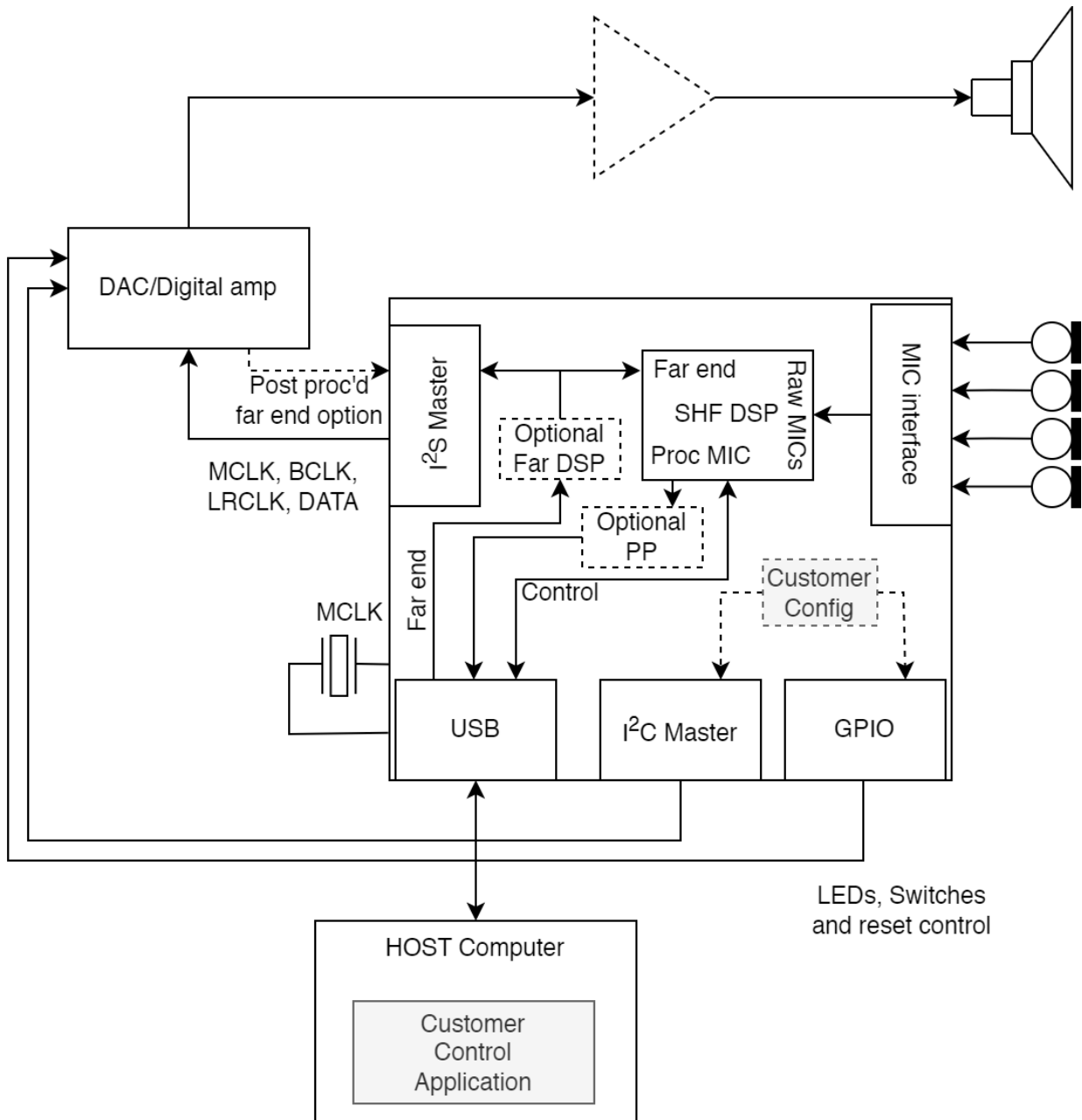
Fig. 2.2: XVF3800 USB Accessory System Diagram

2.5 Module Placement and Interconnection

The diagrams in this section show the location of the XVF3800 modules on the two tiles of the xcore.ai and the interconnections between them.

Note: These diagrams do not depict logical cores or channel interconnections.

One diagram is included for the USB Accessory (Fig. 2.5) use case. The Integrated Device use case supports a control data transport over either SPI or I^{2}C, so two diagrams (Fig. 2.3 and Fig. 2.4) appear for it.

2.5.1 Integrated Device with SPI Control
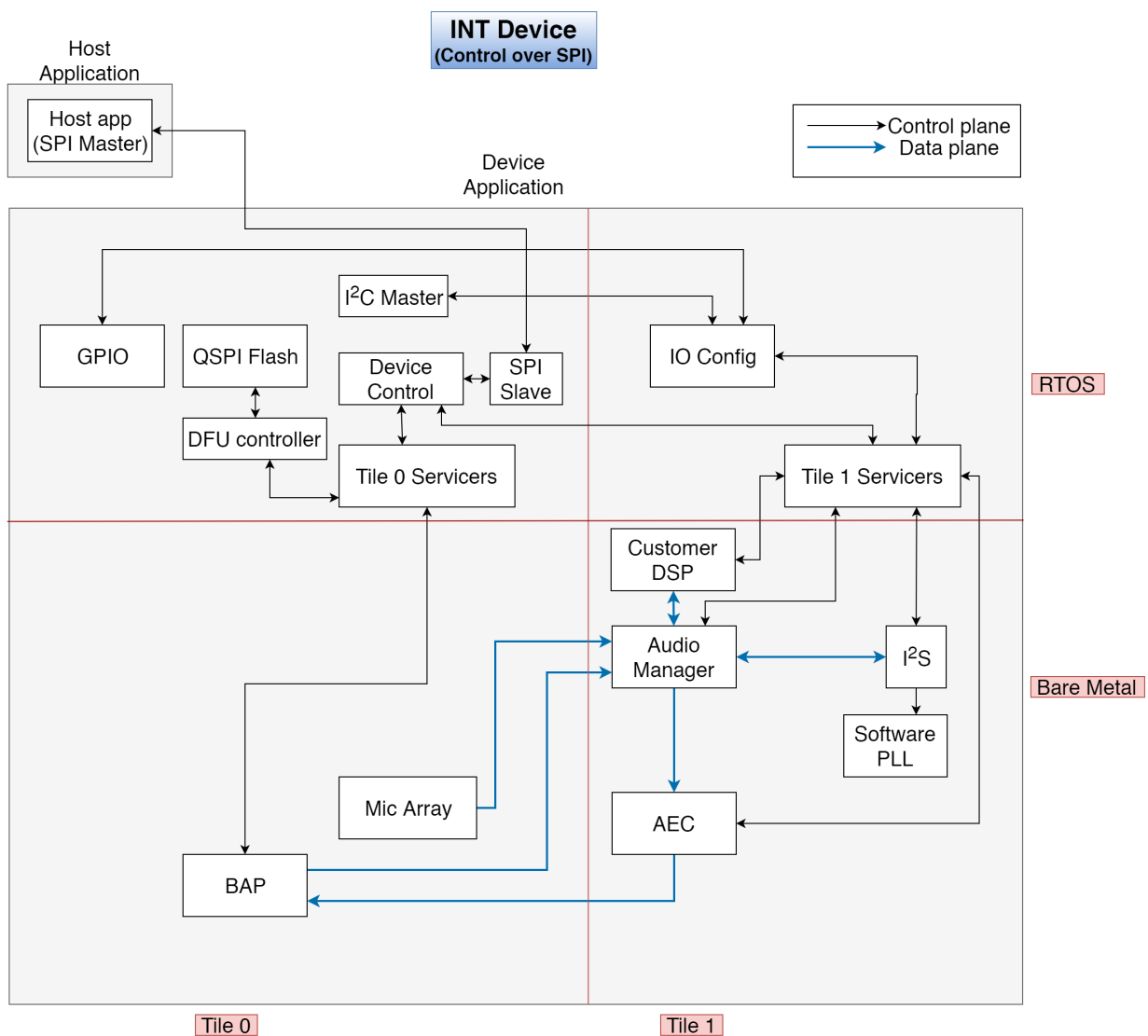
Fig. 2.3: XVF3800 Integrated Device (SPI control) Location Diagram

2.5.2 Integrated Device with I^{2}C Control
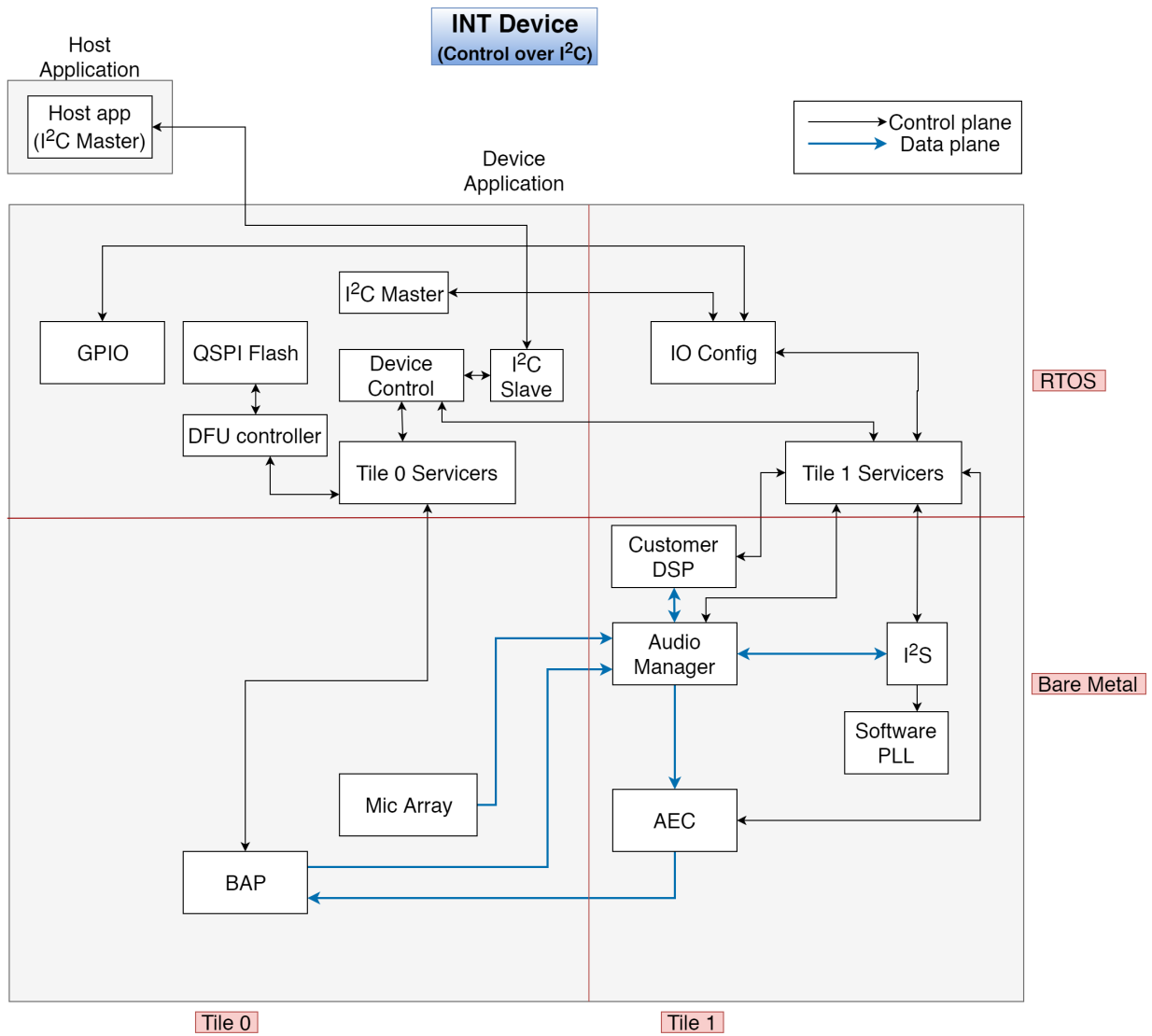
Fig. 2.4: XVF3800 Integrated Device (I^{2}C control) Location Diagram

2.5.3 USB Accessory
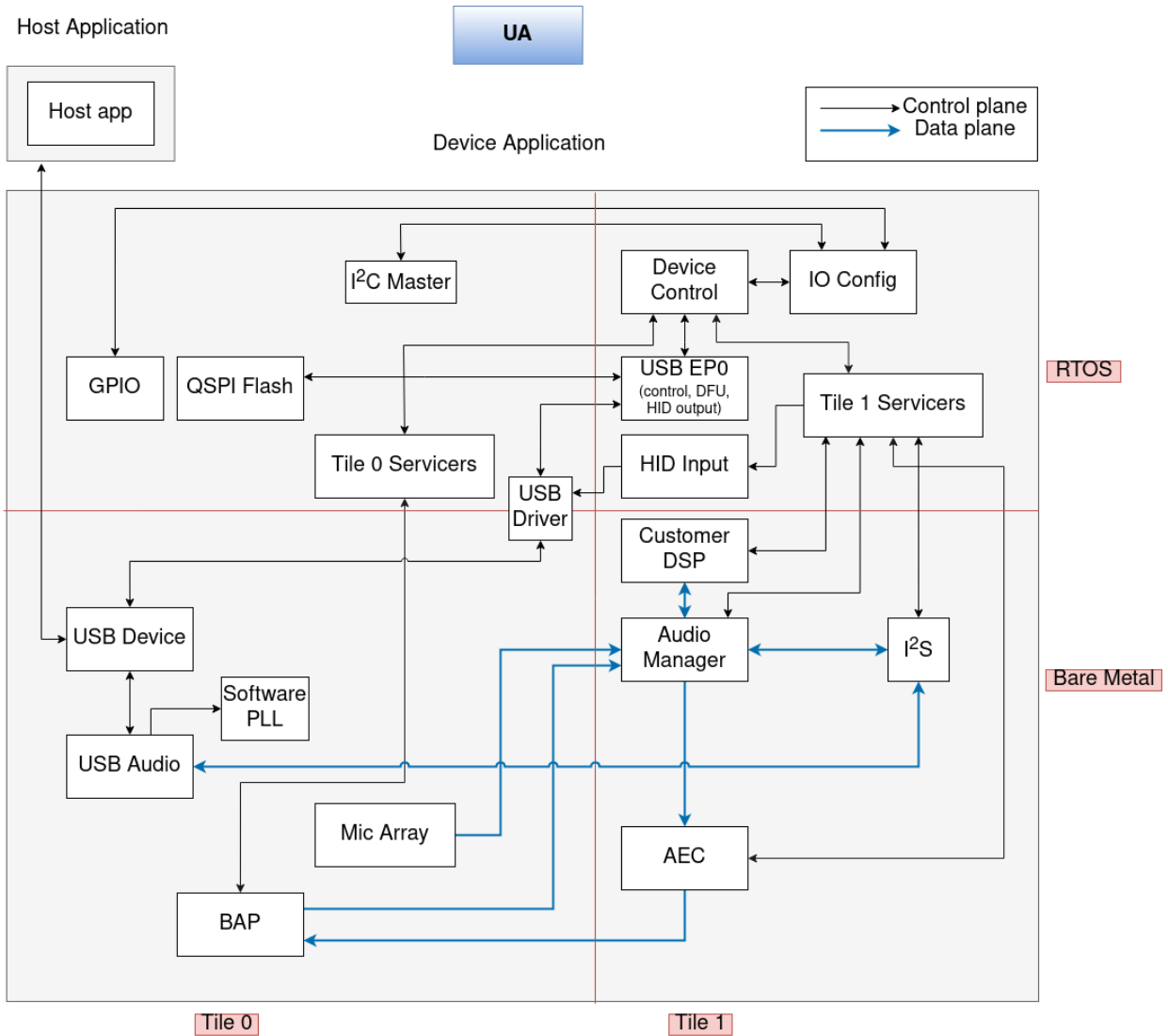
Fig. 2.5: XVF3800 USB Accessory Location Diagram

2.6 Control Plane Detailed Design

2.6.1 Control Plane Structure and Operation

Fig. 2.6 shows the modules involved in processing control commands. In order to concentrate on their processing, it does not include Control Plane modules, such as the DFU controller, HID, I^2C Master or QSPI, that are not directly involved with control command processing.
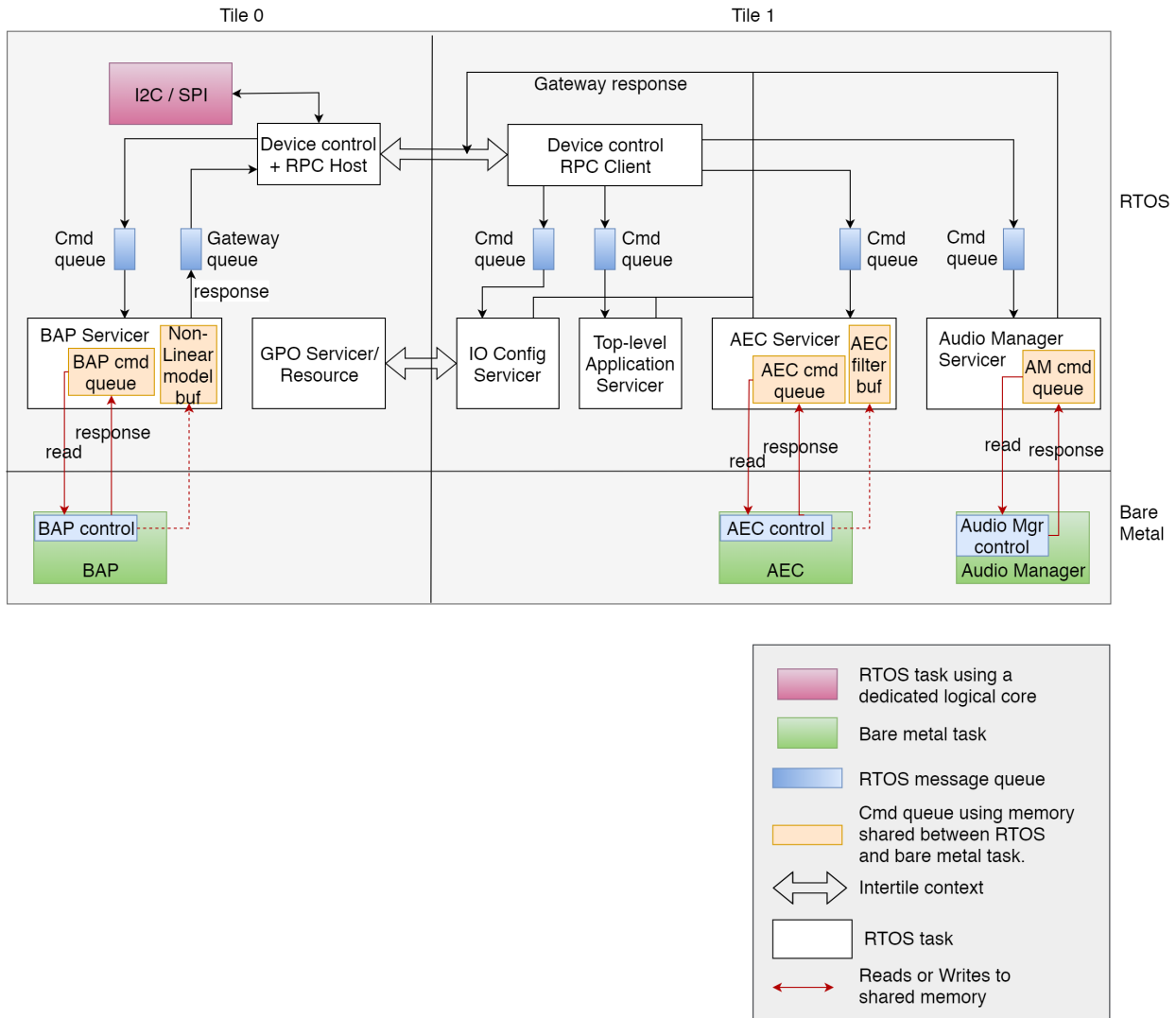
Fig. 2.6: XVF3800 Control Plane Components Diagram

shows the interaction between the Device Control module and a Servicer. In this diagram, boxes with the same colour reside in the same RTOS task.
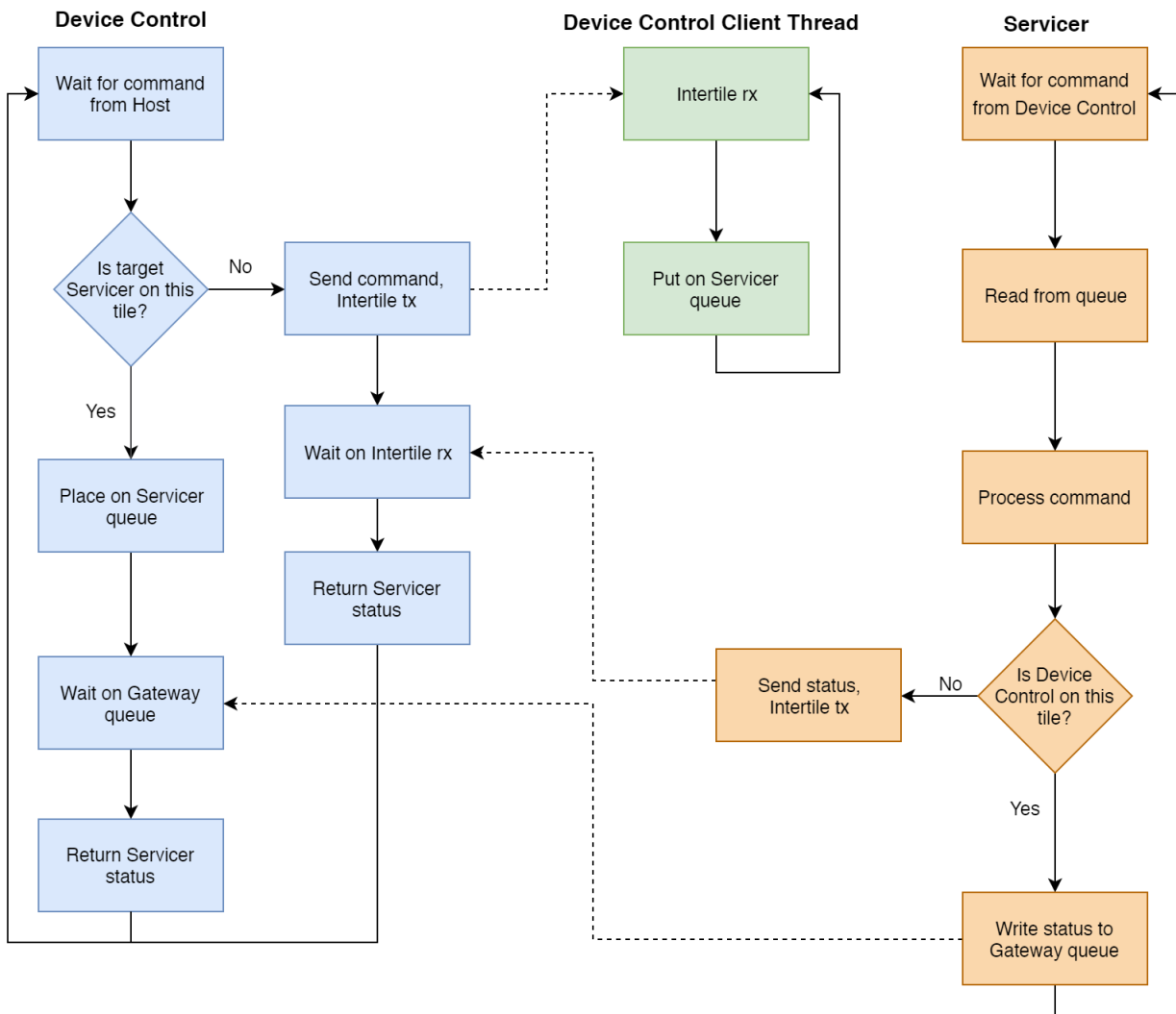
Fig. 2.7: XVF3800 Device Control − Servicer Flow Chart

This diagram shows a critical aspect of Control Plane operation. The Device Control module, having placed a command on a Servicer's command queue, waits on either the Gateway queue or on the Inter-tile context for a response. As a result, it ensures processing of a single control command at a time. Limiting Control Plane operation to a single command in-flight reduces the complexity of the control protocol and eliminates several potential error cases.

Note: Since the Control Plane design requires the host application to poll read commands, limiting operation to a single command in-flight does not limit operation to a single read transaction at a time. For example, a host application may issue a read command to a particular Servicer, receive a status value indicating that it should poll the device for the completion of that read operation, issue a second read command to the same or a different Servicer, receive a status value indicating that it should poll the device for the completion of the second read operation, and then issue additional read commands for either operation in any order until they complete.

2.6.2   Control Protocol

The XVF3800 uses a packet protocol to receive control commands and send each corresponding response. Because packet transmission occurs over a very short-haul transport, e.g. I^{2}C or SPI, or as the payload within a USB packet, the protocol does not include fields for error detection or correction such as start-of-frame and end-of-frame symbols, a cyclical redundancy check or an error correcting code. Fig. 2.8 depicts the structure of each packet.
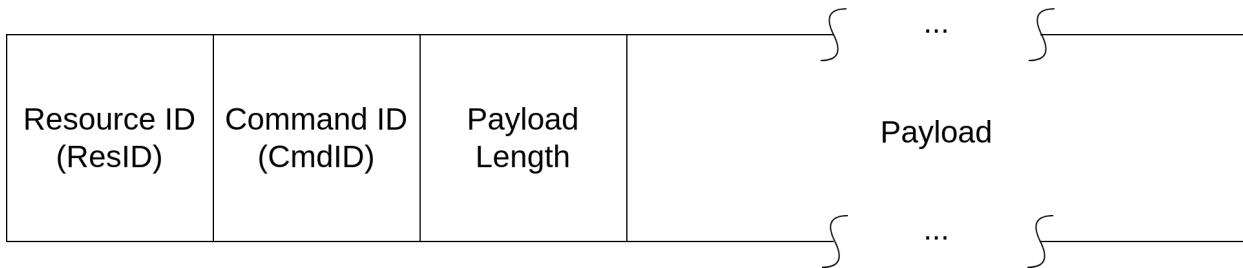
Fig. 2.8: XVF3800 Control Plane Packet Diagram

Packets containing a response from the XVF3800 to the host application place a status value in the first byte of the payload.

2.7 Data Plane Detailed Design

Fig. 2.9 shows the activities within each Data Plane logical core.
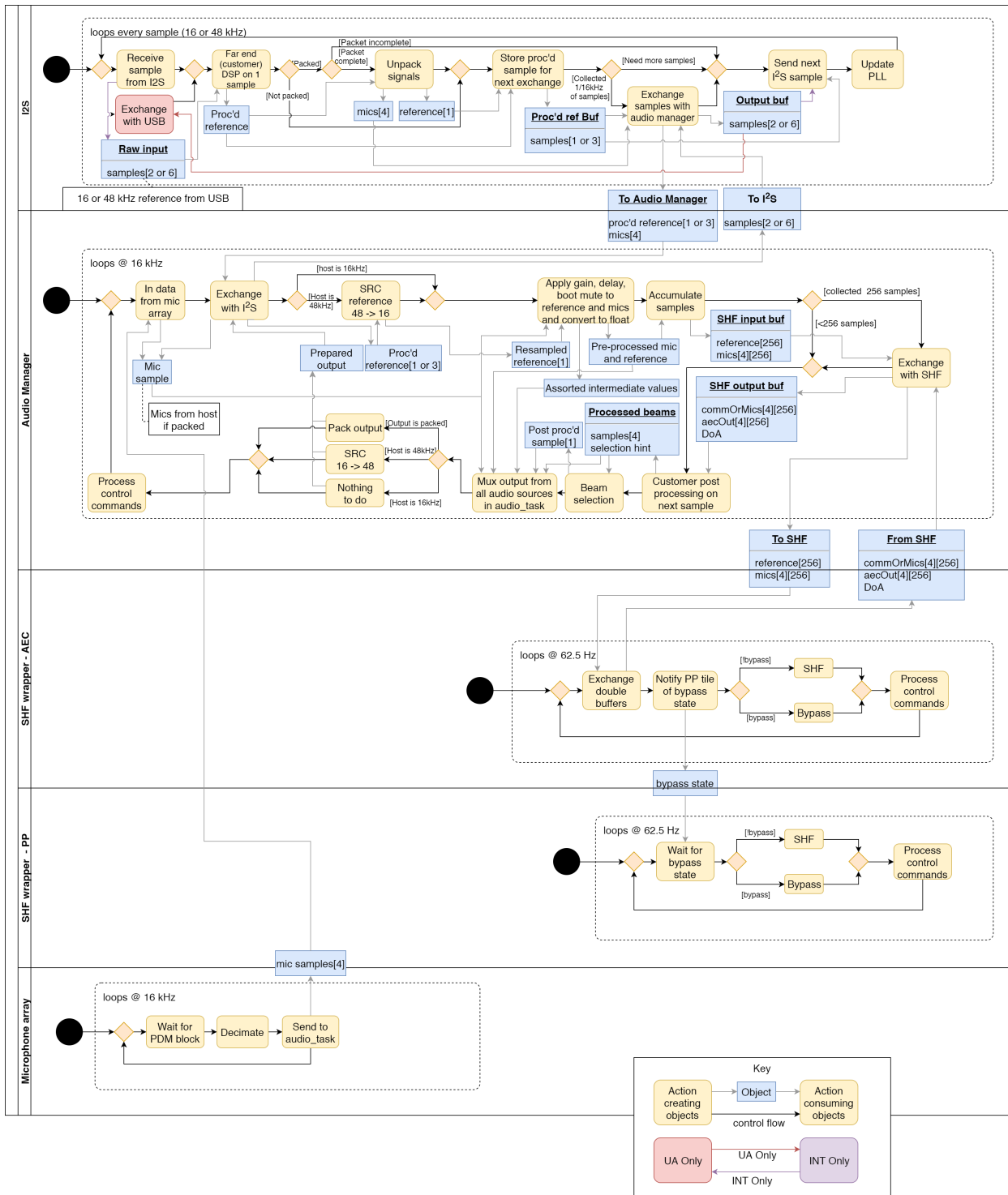
Fig. 2.9: XVF3800 Data Plane Activity Diagram

The portion of the Customer DSP module that allows processing of the reference signal prior to use by the Acoustic Echo Cancellation module appears in the I²S logical core. The other portion of the Customer DSP module, which allows further processing of the audio produced by the Beamforming and Post-processing module, appears in the Audio Manager logical core.

The Software Phase Locked Loop module appears in the I^{2}S logical core. The other Data Plane modules each appear in the logical core of the same name.

2.8   Device Firmware update (DFU) Design

The Device Firmware Update (DFU) allows updating the firmware of the device from a host computer, and it can be performed over I^{2}C or USB. This interface closely follows the principles set out in version 1.1 of the Universal Serial Bus Device Class Specification for Device Firmware Upgrade, including implementing the state machine and command structure described there.

The DFU process is internally managed by the DFU controller module within the firmware. This module is tasked with overseeing the DFU state machine and executing DFU operations. The list of states and transactions are represented in the diagram in Fig. 2.10.
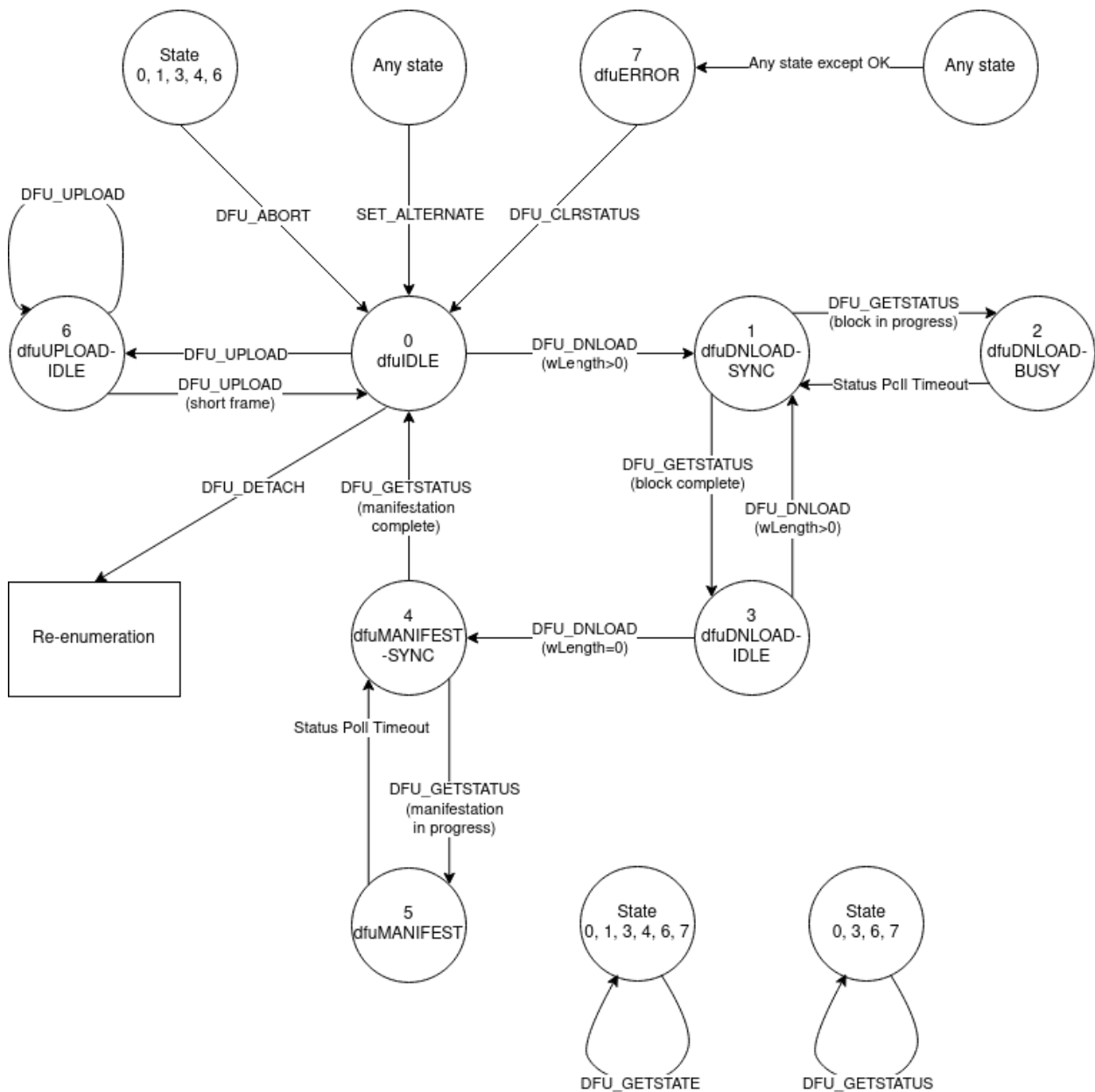
Fig. 2.10: State diagram of the DFU operations

The main differences with the state diagram in version 1.1 of Universal Serial Bus Device Class Specification for Device Firmware Upgrade are:

- the `appIDLE` and `appDETACH` states are not implemented, and the device is started in the `dfuIDLE` state
- the device goes into the `dfuIDLE` state when a `SET_ALTERNATE` message is received
- the device is rebooted when a `DFU_DETACH` command is received.

The DFU allows the following operations:

- download of an upgrade image to the device
- upload of factory and upgrade images from the device
- reboot of the device.

The rest of this section describes the message sequence charts of the supported operations.

A message sequence chart of the download operation is below:
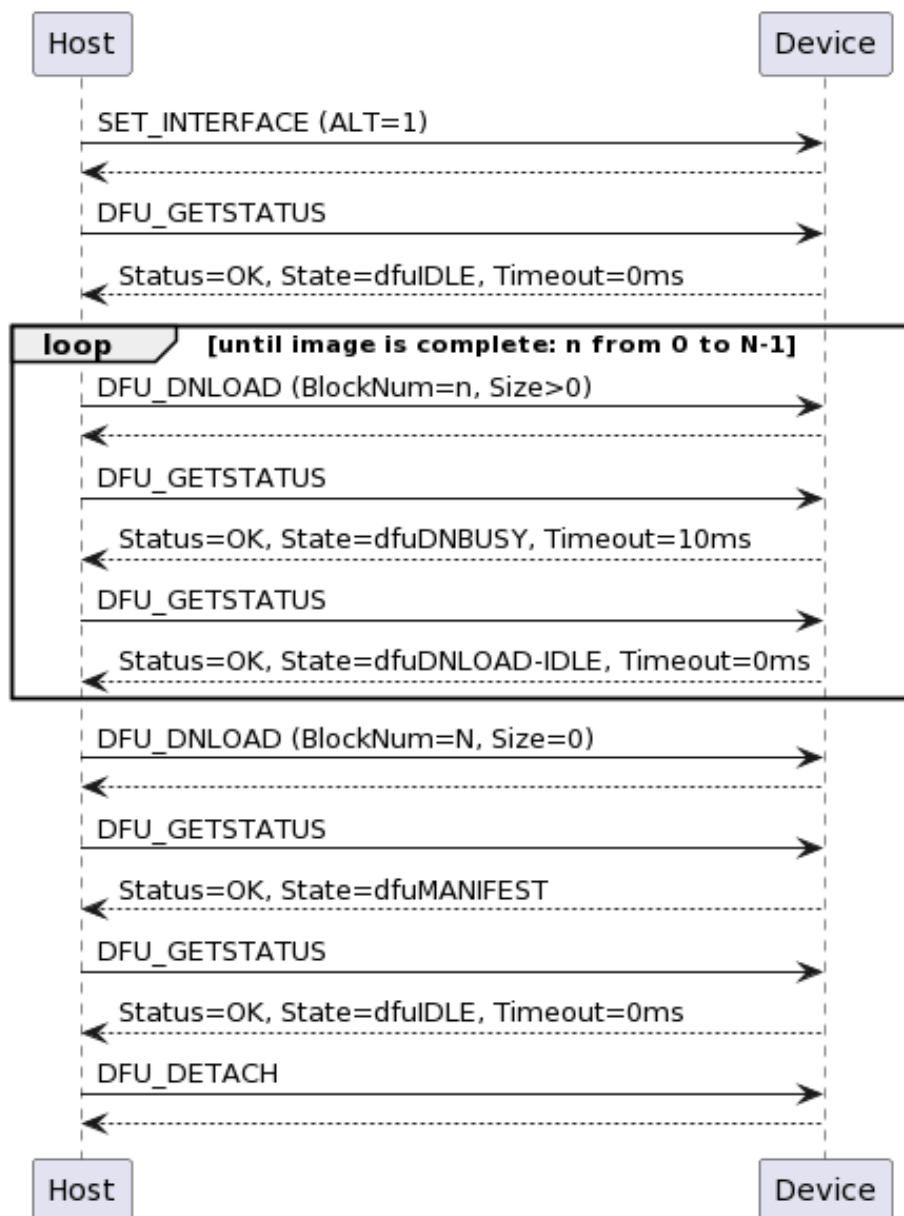
Fig. 2.11: Message sequence chart of the download operation

Note: The end of the image transfer is indicated by a `DFU_DNLOAD` message of size 0.

Note: The `DFU_DETACH` message is used to trigger the reboot.

Note: For the I^{2}C implementation, specification of the block number in download is not supported; all downloads must start with block number 0 and must be run to completion. The device will track this progress internally.

A message sequence chart of the reboot operation is below:
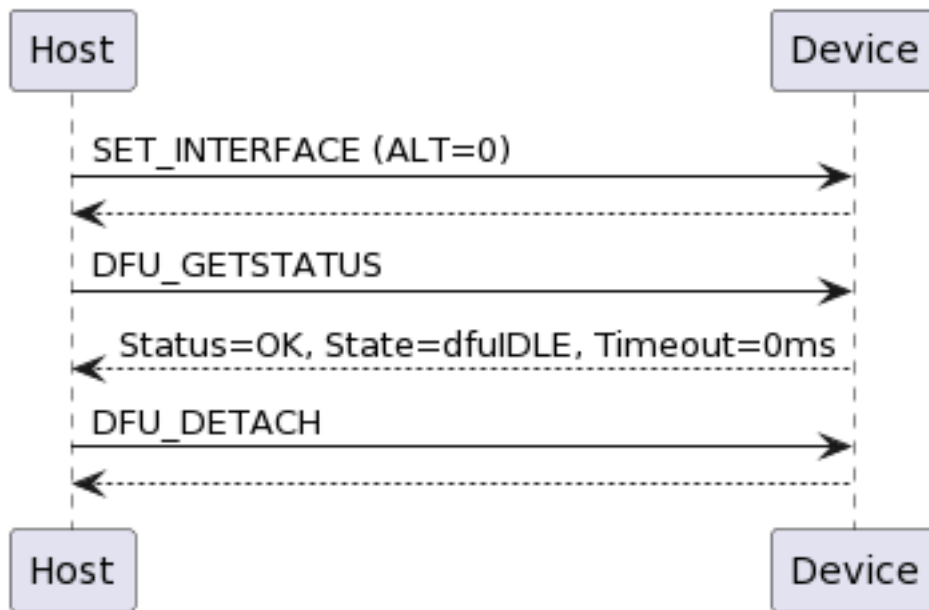
Fig. 2.12: Message sequence chart of the reboot operation

Note: The `DFU_DETACH` message is used to trigger the reboot.

A message sequence chart thisof the upload operation is below:
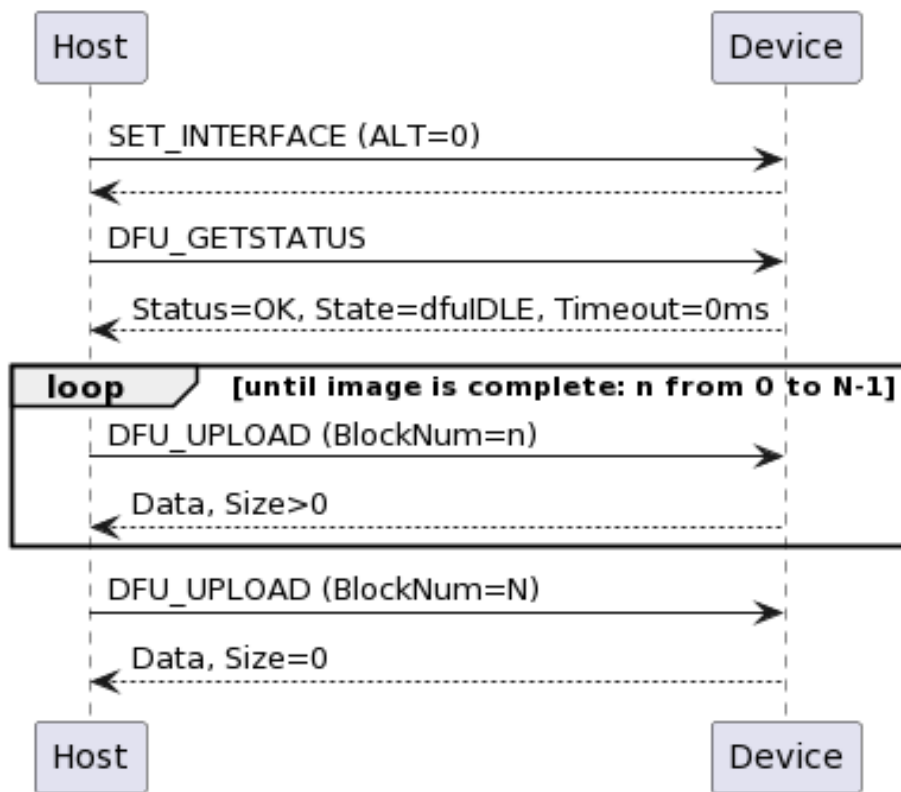
Fig. 2.13: Message sequence chart of the upload operation

Note: The end of the image transfer is indicated by a `DFU_UPLOAD` message of size less than the transport medium maximum; this is 256 bytes in UA and 128 bytes in INT.

2.8.1 DFU over USB implementation

The UA variant of the device make use of a USB connection for handling DFU operations. This interface is a relatively standard, specification-compliant implementation. The implementation is encapsulated within the tinyUSB library, which provides a USB stack for the XVF3800.

2.8.2 DFU over I^{2}C implementation

The INT variant of the device presents a DFU interface that may be controlled over I^{2}C.

The INT DFU state machine is driven by use of control commands, as described in *Control Plane Detailed Design*. The DFU state machine has its own servicer, which then interacts with a separate RTOS task in order to asynchronously perform flash read/write operations.

Mirroring the USB DFU specification, the INT implementation supports a set of 9 control commands intended to drive the state machine, along with an additional 2 utility commands:

Table 2.2: DFU commands

Name	ID	Length	Payload Structure	Purpose
DFU_DETACH	0	1	Payload unused	Write-only command. Restarts the device. Payload is required for protocol, but is discarded within the device. This command has a defined purpose in the USB DFU specification, but in a deviation to that specification it is used with I^2C simply to reboot the device. Future versions of the XMOS DFU-by-device-control protocol (but not future versions of this product) may choose to alter the function of this command to more closely align with the USB DFU specification.
DFU_DNLOAD	1	130	2 bytes length marker, followed by 128 bytes of data buffer	Write-only command. The first two bytes indicate how many bytes of data are being transmitted in this packet. These bytes are little-endian, so byte 0 represents the low byte and byte 1 represents the high byte of an unsigned 16b integer. The remaining 128 bytes are a data buffer for transfer to the device. All control command packets are a fixed length, and therefore all 128 bytes must be included in the command, even if unused. For example, a payload with length of 100 should have the first 100 bytes of data set, but must send an additional 28 bytes of arbitrary data.
DFU_UPLOAD	2	130	2 bytes length marker, followed by 128 bytes of data buffer	Read-only command. The first two bytes indicate how many bytes of data are being transmitted in this packet. These bytes are little-endian, so byte 0 represents the low byte and byte 1 represents the high byte of an unsigned 16b integer. The remaining 128 bytes are a data buffer of data received from the device. All control command packets are a fixed length, and therefore this buffer will be padded to length 128 by the device before transmission. The device will, as per the USB DFU specification, mark the end of the upload process by sending a "short frame" - a packet with a length marker less than 128 bytes.

Table 2.2 – continued from previous page

Name	ID	Length	Payload Structure	Purpose
DFU_GETSTATUS	3	5	1 byte representing device status, 3 bytes representing the requested timeout, 1 byte representing the next device state.	Read-only command. The first byte returns the device status code, as described in the USB DFU specification in the table in section 6.1.2. The next 3 bytes represent the amount of time the host should wait, in ms, before issuing any other commands. This timeout is used in the DNLOAD process to allow the device time to write to flash. This value is little-endian, so bytes 1, 2, and 3 represent the low, middle, and high bytes respectively of an unsigned 24b integer. The final byte returns the number of the state that the device will move into immediately following the return of this request, as described in the USB DFU specification in the table in section 6.1.2.
DFU_CLRSTATUS	4	1	Payload unused	Write-only command. Moves the device out of state 10, dfuERROR. Payload is required for protocol, but is discarded within the device.
DFU_GETSTATE	5	1	1 byte representing current device state.	Read-only command. The first (and only) byte represents the number of the state that the device is currently in, as described in the USB DFU specification in the table in section 6.1.2.
DFU_ABORT	6	1	Payload unused	Write-only command. Aborts an ongoing upload or download process. Payload is required for protocol, but is discarded within the device.
DFU_SETALTERNATE	64	1	1 byte representing either factory (0) or upgrade (1) DFU target images	Write-only command. Sets which of the factory or upgrade images should be targeted by any subsequent upload or download commands. Use of this command entirely resets the DFU state machine to initial conditions: the device will move to dfuIDLE, clear all error conditions, wipe all internal DFU data buffers, and reset all other DFU state apart from the DFU_TRANSFERBLOCK value. This command is included to emulate the SET_ALTERNATE request available in USB.

Table 2.2 – continued from previous page

Name	ID	Length	Payload Structure	Purpose
DFU_TRANSFERBLOCK	65	2	2 bytes, representing the target transfer block for an upload process.	Read/write command. Sets/gets a 2 byte value specifying the transfer block number to use for a subsequent upload operation. A complete image may be conceptually divided into 128-byte blocks. These blocks may then be numbered from 0 upwards. Setting this value sets which block will be returned by a subsequent DFU_UPLOAD request. This value is initialised to 0, and autoincrements after each successful DFU_UPLOAD request has been serviced. Therefore, to read a whole image from the start, there is no need to issue this command - this command need only be used to select a specific section to read. Because this value is automatically incremented after a DFU_UPLOAD command is successfully serviced, reading it will give the value of the next block to be read (and this will be one greater than the previous block read, if it has not been altered in the interim). This value is reset to 0 at the successful completion of a DFU_UPLOAD process. It is not reset after a DFU_ABORT, nor after a DFU_SETALTERNATE call. This command is included to emulate the ability in a USB request to send values in the header of the request - the device control protocol used here does not allow sending any data with a read request such as DFU_UPLOAD.
DFU_GETVERSION	88	3	3 bytes, representing major.minor.patch version of device	Read-only command. Bytes 0, 1, and 2 represent the major, minor, and patch versions respectively of the device. This is a utility command intended to provide an easy mechanism by which to verify that a firmware download has been successful.
DFU_REBOOT	89	1	Payload unused	Write-only command. Restarts the device. Payload is required for protocol, but is discarded within the device. This is a utility command intended to provide a clear and unambiguous interface for restarting the device. Use of this command should be preferred over DFU_DETACH for this purpose.

These commands are then used to drive the state machine described in the *Device Firmware update (DFU) Design*.

When writing a custom compliant host application, the use of XMOS' fwk_rtos library is advised; the **device_control** library provided there gives a host API that can communicate effectively with the XVF3800, as demonstrated in the *xvf_host* application. However, a description of the I^{2}C bus activity during the execution

of the above DFU commands is provided below, in the instance that usage of the **device_control** library is inconvenient or impossible.

The XVF3800's I^{2}C address is set by default as 0x2C. This may be confirmed by examination of the `I2C_ADDRESS` field in the `transport_config.yaml` file, found in the release package at `sources/app_xvf3800/autogeneration/yaml_files/settings_and_defaults`. The XVF3800's I^{2}C address may also be altered by editing this file. The DFU resource has an internal "resource ID" of 0xF0. This maps to the register that read/write operations on the DFU resource should target - therefore, the register to write to will always be 0xF0.

To issue a write command (e.g. DFU_SETALTERNATE):

- First, set up a write to the device address. For a default device configuration, a write operation will always start by a write token to 0x2C (START, 7 bits of address [0x2C], R/W bit [0 to specify write]), wait for ACK, followed by specifying the register to write [Resource ID 0xF0] (and again wait for ACK).

- Then, write the command ID (in this example, 64 [0x40]) from the above table.

- Then, write the total transfer size, *including the register byte*. In this example, that will be 4 bytes (register byte, command ID, length byte, and 1 byte of payload), so write 0x04.

- Finally, send the payload - e.g. 1 to set the alternate setting to "upgrade".

- The full sequence for this write command will therefore be START, 7 bits of address [0x2C], 0 (to specify write), hold for ACK, 0xF0, hold for ACK, 0x40, hold for ACK, 0x04, hold for ACK, 0x01, hold for ACK, STOP.

- To complete the transaction, the device must then be queried; set up a read to 0x2C (START, 7 bits of address [0x2C], R/W bit [1 to specify read], wait for ACK). The device will clock-stretch until it is ready, at which point it will release the clock and transmit one byte of status information. This will be a value from the enum `control_ret_t` from `device_control_shared.h`, found in `sources\modules\fwk_xvf\modules\rtos\modules\sw_services\device_control\api`.

To issue a read command (e.g. DFU_GETSTATUS):

- Set up a write to the device; as above, this will mean sending START, 7 bits of device address [0x2C], 0 (to specify write), hold for ACK. Send the DFU resource ID [0xF0], hold for ACK.

- Then, write the command ID (in this example, 3), bitwise ANDed with 0x80 (to specify this as a read command) - in this example therefore 0x83 should be sent, and hold for ACK.

- Then, write the total length of the expected reply. In this example, the command has a payload of 5 bytes. The device will also prepend the payload with a status byte. Therefore, the expected reply length will be 6 bytes [0x06]. Hold for ACK.

- Then, issue a repeated START. Follow this with a read from the device: the repeated START, 7 bits of device address [0x2C], 1 (to specify read), hold for ACK. The device will clock-stretch until it is ready. It will then send a status byte (from the enum `control_ret_t` as described above), followed by a payload of requested data - in this example, the device will send 5 bytes. ACK each received byte. After the last expected byte, issue a STOP.

It is heavily advised that those wishing to write a custom host application to drive the DFU process for the XVF3800 over I^{2}C familiarise themselves with version 1.1 of the Universal Serial Bus Device Class Specification for Device Firmware Upgrade.

2.9 HID Interface design

The UA variant of the device presents a USB HID interface. The HID interface adds support for USB standard input and output reports for a telephony device.

2.9.1 HID descriptors

As shown in the trace captured during device enumeration (Fig. 2.14), the HID interface presents itself as interface 5 in the configuration descriptor. The device enumerates with Endpoint 2 as the HID Input endpoint which is responsible for sending HID input reports to the host.

Interface Descriptor		Radix: auto
bLength	9	
bDescriptorType	INTERFACE (0x04)	
bInterfaceNumber	5	
bAlternateSetting	0	
bNumEndpoints	1	
bInterfaceClass	Human Interface Device (0x03)	
bInterfaceSubClass	None (0x00)	
bInterfaceProtocol	None (0x00)	
iInterface	XMOS HID (8)	

HID Descriptor		Radix: auto
bLength	9	
bDescriptorType	HID (33)	
bcdHID	1.11 (0x0111)	
bCountryCode	0x00	
bNumDescriptors	1	
bDescriptorType	REPORT (34)	
wDescriptorLength	188	

Endpoint Descriptor		Radix: auto
bLength	7	
bDescriptorType	ENDPOINT (0x05)	
bEndpointAddress	2 IN (0b10000010)	
bmAttributes.TransferType	Interrupt (0b11)	
wMaxPacketSize.PacketSize	64	
wMaxPacketSize.Transactions	One transaction per microframe if HS (0b00)	
bInterval	7	

Fig. 2.14: HID descriptor during enumeration

The device also supports using the UC Qualification (UCQ) descriptor for HID to inform the host of its capabilities. The UCQ string descriptor returned by the device is "UCQ01001000001000" indicating a speakerphone device with AEC capability.

Input reports are sent in response to button presses on the device or in response to specific HID output reports. HID output reports are sent by the host on Endpoint 0. Feature reports are sent and received by the host on Endpoint 0.

Note: While the HID device descriptor currently describes one feature report, the current HID implementation doesn't support parsing feature reports received from the host or responding with a non-zero feature report to the host.

The HID descriptor structure is transcluded below from `sources/modules/fwk_xvf/modules/xvf/src/usb/config/usb_descriptors.c`:

```
uint8_t const desc_hid_report[] =
{
  TUD_HID_REPORT_DESC_MISC_BUTTONS    (HID_REPORT_ID(REPORT_ID_MISC_BUTTONS              )),
  TUD_HID_REPORT_DESC_VOLUME_BUTTONS (HID_REPORT_ID(REPORT_ID_VOLUME_BUTTONS            )),
  TUD_HID_REPORT_DESC_TEAMS_ASP      ( HID_REPORT_ID(REPORT_ID_TEAMS_ASP                )),
  TUD_HID_REPORT_DESC_TEAMS_BUTTON   ( HID_REPORT_ID(REPORT_ID_TEAMS_BUTTON             )),
};
```

The macros defining the individual reports that are part of the `desc_hid_report` structure are transcluded below from `sources/modules/fwk_xvf/modules/xvf/src/usb/config/hid_telephony_device.h`:

```
#define TUD_HID_REPORT_DESC_MISC_BUTTONS(...) \
  HID_USAGE_PAGE ( HID_USAGE_PAGE_TELEPHONY                    ) ,\
  HID_USAGE      ( HID_USAGE_TELEPHONY_PHONE                   ) ,\
  HID_COLLECTION ( HID_COLLECTION_APPLICATION                  ) ,\
    /* Report ID if any */\
    __VA_ARGS__ \
    HID_USAGE      ( HID_USAGE_TELEPHONY_HOOKSWITCH          ) ,\
    HID_LOGICAL_MIN ( 0                                      ) ,\
    HID_LOGICAL_MAX ( 1                                      ) ,\
    HID_REPORT_COUNT( 1                                      ) ,\
    HID_REPORT_SIZE ( 1                                      ) ,\
    HID_INPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE      ( HID_USAGE_TELEPHONY_PHONE_MUTE         ) ,\
    HID_INPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE      ( HID_USAGE_TELEPHONY_FLASH              ) ,\
    HID_INPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE      ( HID_USAGE_TELEPHONY_REDIAL             ) ,\
    HID_INPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE_PAGE ( HID_USAGE_PAGE_BUTTON                  ) ,\
    /* This button has been chosen to support the HOLD procedure in Teams */ \
    HID_USAGE      ( HID_USAGE_BUTTON_7                     ) ,\
    HID_INPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    /* 3 bit padding */ \
    HID_REPORT_COUNT( 1                                      ) ,\
    HID_REPORT_SIZE ( 3                                      ) ,\
    HID_INPUT      ( HID_CONSTANT                           ) ,\
    HID_USAGE_PAGE ( HID_USAGE_PAGE_TELEPHONY               ) ,\
    HID_USAGE      ( HID_USAGE_TELEPHONY_KEYPAD             ) ,\
    HID_LOGICAL_MIN ( 1                                      ) ,\
    HID_LOGICAL_MAX ( 12                                     ) ,\
    HID_REPORT_COUNT( 1                                      ) ,\
    HID_REPORT_SIZE ( 4                                      ) ,\
    HID_USAGE_MIN  ( 0xB0                                   ) ,\
    HID_USAGE_MAX  ( 0xBB                                   ) ,\
    HID_INPUT      ( HID_DATA | HID_ARRAY | HID_ABSOLUTE    ) ,\
    HID_LOGICAL_MIN ( 0                                      ) ,\
    HID_LOGICAL_MAX ( 1                                      ) ,\
    /* 4 bit padding */ \
    HID_REPORT_COUNT( 1                                      ) ,\
    HID_REPORT_SIZE ( 4                                      ) ,\
    HID_INPUT      ( HID_CONSTANT                           ) ,\
    HID_USAGE_PAGE ( HID_USAGE_PAGE_LED                     ) ,\
    HID_USAGE      ( HID_USAGE_LED_OFF_HOOK                 ) ,\
    HID_LOGICAL_MIN ( 0                                      ) ,\
    HID_LOGICAL_MAX ( 1                                      ) ,\
    HID_REPORT_COUNT( 1                                      ) ,\
    HID_REPORT_SIZE ( 1                                      ) ,\
    HID_OUTPUT     ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE      ( HID_USAGE_LED_MUTE                     ) ,\
    HID_OUTPUT     ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
    HID_USAGE      ( HID_USAGE_LED_RING                     ) ,\
```

```
     HID_OUTPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
     HID_USAGE       ( HID_USAGE_LED_HOLD                      ) ,\
     HID_OUTPUT      ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
     /* 4 bit padding */ \
     HID_REPORT_COUNT( 1                                       ) ,\
     HID_REPORT_SIZE ( 4                                       ) ,\
     HID_OUTPUT      ( HID_CONSTANT                            ) ,\
   HID_COLLECTION_END  \


#define TUD_HID_REPORT_DESC_VOLUME_BUTTONS(...) \
   HID_USAGE_PAGE ( HID_USAGE_PAGE_CONSUMER                 ), \
   HID_USAGE      ( HID_USAGE_CONSUMER_CONTROL              ) ,\
   HID_COLLECTION ( HID_COLLECTION_APPLICATION              ) ,\
     /* Report ID if any */                                  \
     __VA_ARGS__                                             \
     HID_LOGICAL_MIN ( 0                                    ) ,\
     HID_LOGICAL_MAX ( 1                                    ) ,\
     HID_REPORT_COUNT( 1                                    ) ,\
     HID_REPORT_SIZE ( 1                                    ) ,\
     HID_USAGE       ( HID_USAGE_CONSUMER_VOLUME_INCREMENT   )  ,\
     HID_INPUT       ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
     HID_USAGE       ( HID_USAGE_CONSUMER_VOLUME_DECREMENT    ) ,\
     HID_INPUT       ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
     /* 6 bit padding */                                     \
     HID_REPORT_COUNT( 1                                    ) ,\
     HID_REPORT_SIZE ( 6                                    ) ,\
     HID_INPUT       ( HID_CONSTANT                         ) ,\
   HID_COLLECTION_END  \


#define TUD_HID_REPORT_DESC_TEAMS_BUTTON(...) \
   HID_USAGE_PAGE_N ( 0xFF99, 2                             ) ,\
   HID_USAGE      ( 0x01                                    ) ,\
   HID_COLLECTION ( HID_COLLECTION_APPLICATION              ) ,\
     /* Report ID if any */                                  \
     __VA_ARGS__                                             \
     HID_LOGICAL_MIN ( 0                                    ) ,\
     HID_LOGICAL_MAX ( 1                                    ) ,\
     HID_REPORT_COUNT( 1                                    ) ,\
     HID_REPORT_SIZE ( 1                                    ) ,\
     HID_USAGE       ( 0x04                                 ) ,\
     HID_INPUT       ( HID_DATA | HID_VARIABLE | HID_RELATIVE ) ,\
     /* 7 bit padding */                                     \
     HID_REPORT_COUNT( 1                                    ) ,\
     HID_REPORT_SIZE ( 7                                    ) ,\
     HID_INPUT       ( HID_CONSTANT                         ) ,\
   HID_COLLECTION_END                                         \


#define TUD_HID_REPORT_DESC_TEAMS_ASP(...) \
   HID_USAGE_PAGE_N ( 0xFF99, 2                             ) ,\
   HID_USAGE       ( 0x03                                   ) ,\
   HID_COLLECTION   ( HID_COLLECTION_APPLICATION            ) ,\
     /* Report ID if any */                                  \
     __VA_ARGS__                                             \
     HID_LOGICAL_MIN   ( 0                                  ) ,\
     HID_LOGICAL_MAX_N ( 255, 2                             ) ,\
     HID_USAGE_MIN     ( 0x00                               ) ,\
     HID_USAGE_MAX     ( 0xff                               ) ,\
     HID_REPORT_COUNT_N( 63, 2                              ) ,\
     HID_REPORT_SIZE   ( 8                                  ) ,\
     HID_FEATURE       ( HID_DATA | HID_VARIABLE | HID_ABSOLUTE ) ,\
   HID_COLLECTION_END  \
```
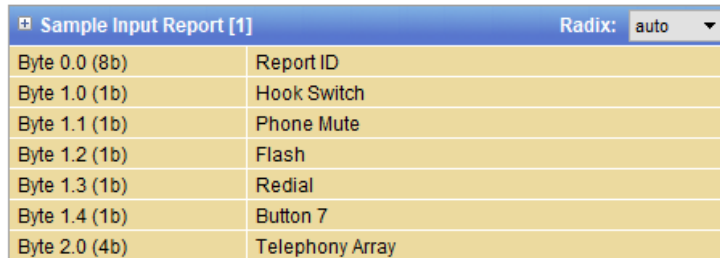
The HID report descriptor translates to three input reports, one output report and one feature report.

Input report ID 1 contains the Hook Switch, Mute, Flash, Redial and the Dialpad buttons.

Note: The current HID implementation doesn't support Dialpad buttons.

⊞ Sample Input Report [1]		Radix: auto ▼
Byte 0.0 (8b)	Report ID	
Byte 1.0 (1b)	Hook Switch	
Byte 1.1 (1b)	Phone Mute	
Byte 1.2 (1b)	Flash	
Byte 1.3 (1b)	Redial	
Byte 1.4 (1b)	Button 7	
Byte 2.0 (4b)	Telephony Array	

Fig. 2.15: Input report ID 1
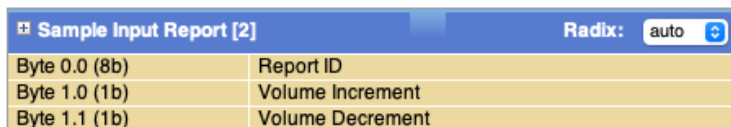
Output report ID 1 contains the Off-hook, Mute, Ring and Hold LEDs.

⊞ Sample Output Report [1]		Radix: auto ⬍
Byte 0.0 (8b)	Report ID	
Byte 1.0 (1b)	Off-Hook	
Byte 1.1 (1b)	Mute	
Byte 1.2 (1b)	Ring	
Byte 1.3 (1b)	Hold	

Fig. 2.16: Output report ID 1

Input report ID 2 contains the Volume Increment and Volume Decrement buttons.

⊞ Sample Input Report [2]		Radix: auto ⬍
Byte 0.0 (8b)	Report ID	
Byte 1.0 (1b)	Volume Increment	
Byte 1.1 (1b)	Volume Decrement	

Fig. 2.17: Input report ID 2

Input report ID 155 contains a custom button meant to be used as the Teams button.

Note: The Teams button implementation requires the device to respond to feature reports and is currently unimplemented.

⊞ Sample Input Report [155]		Radix: auto ⬍
Byte 0.0 (8b)	Report ID	
Byte 1.0 (1b)	Undefined	

Fig. 2.18: Input report ID 155

2.9.2 HID System Design

The HID interface is implemented in the RTOS. Fig. 2.19 shows the tasks involved in the HID implementation.
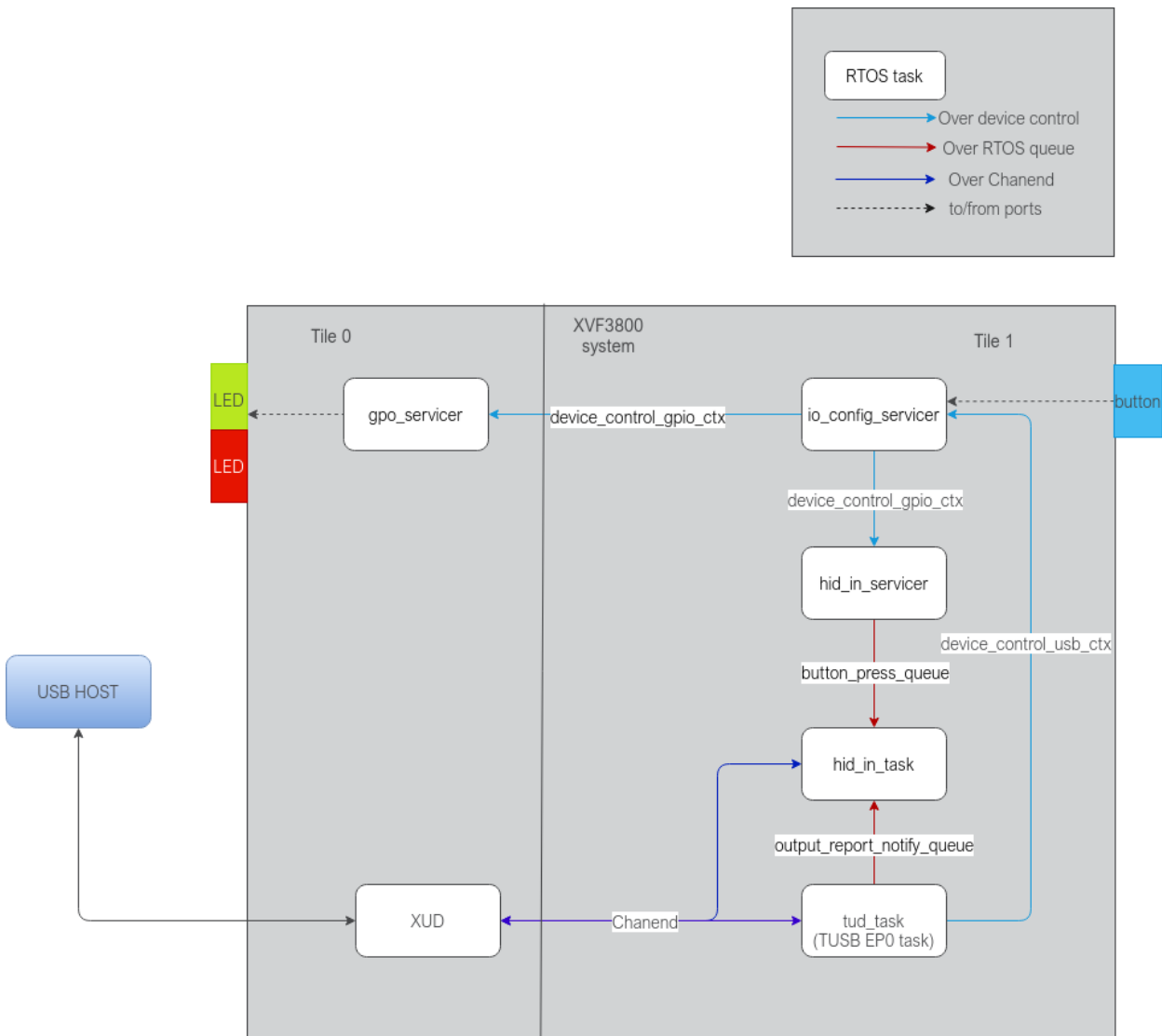
Fig. 2.19: HID task diagram

The `io_config_servicer` detects button presses and notifies a button press event to the `hid_in_servicer` through a control command sent on the `device_control_gpio_ctx`. It also forwards the GPO commands for LED control to the `gpo_servicer`.

The `gpo_servicer` receives GPO LED commands over the `device_control_gpio_ctx` and programmes the LEDs.

The `hid_in_servicer` notifies the `hid_in_task` of the button press event notifications that it receives from the `io_config_servicer`.

The `tud_task` implements USB Endpoint 0 and calls the callback function `tud_hid_set_report_cb` on receiving a HID output report from the host on Endpoint 0. If an LED state needs to change in response to the output report, `tud_hid_set_report_cb` initiates it by sending GPO LED control commands to the `io_config_servicer`. `tud_hid_set_report_cb` also notifies the the `hid_in_task` when specific output reports are received.

The `hid_in_task` implements the HID Input endpoint. It is a timer based task that wakes up periodically and

sends a HID input report to the host over the HID Input endpoint. The logic for deciding which report to send is as follows:

- If the previous iteration attempted to send a report but could not, re-attempt sending the report,
- else, if a One Shot Control (OSC) button press sequence is in progress, send the report to complete the button press sequence,
- else, check if there is an output report notification from `tud_hid_set_report_cb` that requires sending an input report in response,
- else, check if there is a button press notification from `io_config_servicer` and send an input report corresponding to the pressed button.

To summarise, Fig. 2.20 shows the path through which a GPI button press translates into a HID input report, and Fig. 2.21 shows the path through which a HID output report that the device receives translates to a GPO LED program. Additionally, Fig. 2.21 also shows the path through which a HID output report translates to a HID input report sent from the device to USB.
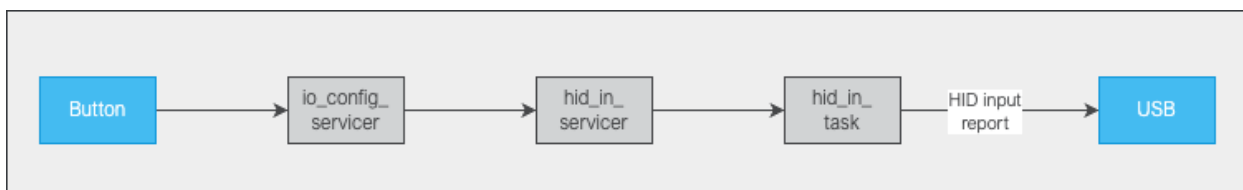
Fig. 2.20: HID button path
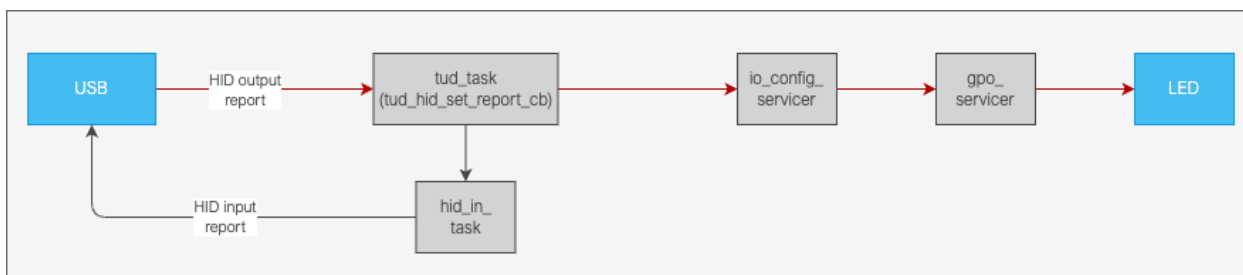
Fig. 2.21: HID LED path

2.9.3   HID Initialisation

During device initialisation, the `hid_init` function is called to do all HID related initialisations. The HID button and LED configurations are captured in the `hid_button_config_t` and `hid_led_config_t` structures. These structures contain the mapping from a given HID button to the GPI button on the device and from a given HID LED to the GPO LED on the device.

The `init_hid_button_config` function initialises the HID buttons. There is one button on the XK-VOICE-SQ66 development kit (`XS1_PORT_4A` pin 3) and it's currently mapped to the HID Mute button as can be seen in the code transcluded from the `init_hid_button_config` in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c`:

```
    hid_button_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = BUTTON_MUTE_OFFSET,
↪.size = 1, .gpi_source = GPI_SOURCE_EVK, .gpi_pin_index = EVK_BUTTON_INDEX, .button_type = BUTTON_
↪TYPE_OSC, .button_press_precondition=HOOKSWITCH_BUTTON};
    hid_button_config[MUTE_BUTTON] = config;
```

Note the `.gpi_source = GPI_SOURCE_EVK` and `.gpi_pin_index = EVK_BUTTON_INDEX` in the code above that map the button on the XK-VOICE-SQ66 development kit to the HID Mute button.

Note: The HID implementation supports two types of buttons, One Shot Control (OSC) and Re-Trigger Control (RTC). The Volume Increment and Volume Decrement buttons are set as RTC while all the other buttons are configured to be of OSC type. For the OSC buttons, only after the button is released, the device sends an input report with the button set to 1 followed by another input report with the button set to 0. For the RTC buttons, in a button press event, an input report with button set to 1 is sent and in a button release event, an input report with button set to 0 is sent by the device.

The `init_hid_led_config` function initialises the HID LEDs. The Red LED (`XS1_PORT_8C`, pin 6) on the XK-VOICE-SQ66 development kit is mapped to the MUTE_LED and the Green LED (`XS1_PORT_8C`, pin 7) on the XK-VOICE-SQ66 development kit is mapped to the OFFHOOK_LED as shown in the code transcluded from `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c` below:

```
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_OFFHOOK_OFFSET, .
→gpo_source = GPO_SOURCE_EVK, .gpo_pin_index = EVK_LED_GREEN, .notify_hid_task = true, .trigger_hid_
→input_index = HOOKSWITCH_BUTTON, .led_mode=LED_MODE_STEADY};
    hid_led_config[OFFHOOK_LED] = config;
```

Note the `.gpo_source = GPO_SOURCE_EVK` and `.gpo_pin_index = EVK_LED_GREEN` in the code above that map the green LED on the XK-VOICE-SQ66 development kit to the HID OffHook LED.

```
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_MUTE_OFFSET, .gpo_
→source = GPO_SOURCE_EVK, .gpo_pin_index = EVK_LED_RED, .notify_hid_task = false, .trigger_hid_
→input_index = NO_HID_IN_TRIGGER, .led_mode=LED_MODE_STEADY};
    hid_led_config[MUTE_LED] = config;
```

Note the `.gpo_source = GPO_SOURCE_EVK` and `.gpo_pin_index = EVK_LED_RED` in the code above that map the red LED on the XK-VOICE-SQ66 development kit to the HID Mute LED.

The HID buttons/LEDs to GPIO button/LEDs mapping can be changed by modifying the `init_hid_button_config` and `init_hid_led_config` functions. *Modifying the HID to GPIO mapping* describes this in detail.

2.9.4   HID Operation

With the HID Buttons and LEDs initialised as described in *HID Initialisation*, the following use cases are implemented.

2.9.4.1   Mute/Unmute device

When the device is used in an ongoing Teams call, the user pressing the Mute button on the device ends up with the microphone mute status toggled on the Teams client. Fig. 2.22 depicts this use case. The Mute button is a One Shot Control button type which means that a 0 -> 1 in the button state in the HID report triggers an event and a 1 -> 0 transition in the button state has to happen before the next event. As a result, the Mute button pressed on the device has the device sending two hid reports, with the Mute button set to 1 and 0 respectively.

Fig. 2.22: Mic mute/unmute

2.9.4.2 Inform call start

At the start of the Teams call, the Teams client sends a HID output report with Off-Hook LED set to 1. The device responds with an input report with HookSwitch set to 1. Note that this is an example of the device sending an input report in response to a given HID output report.
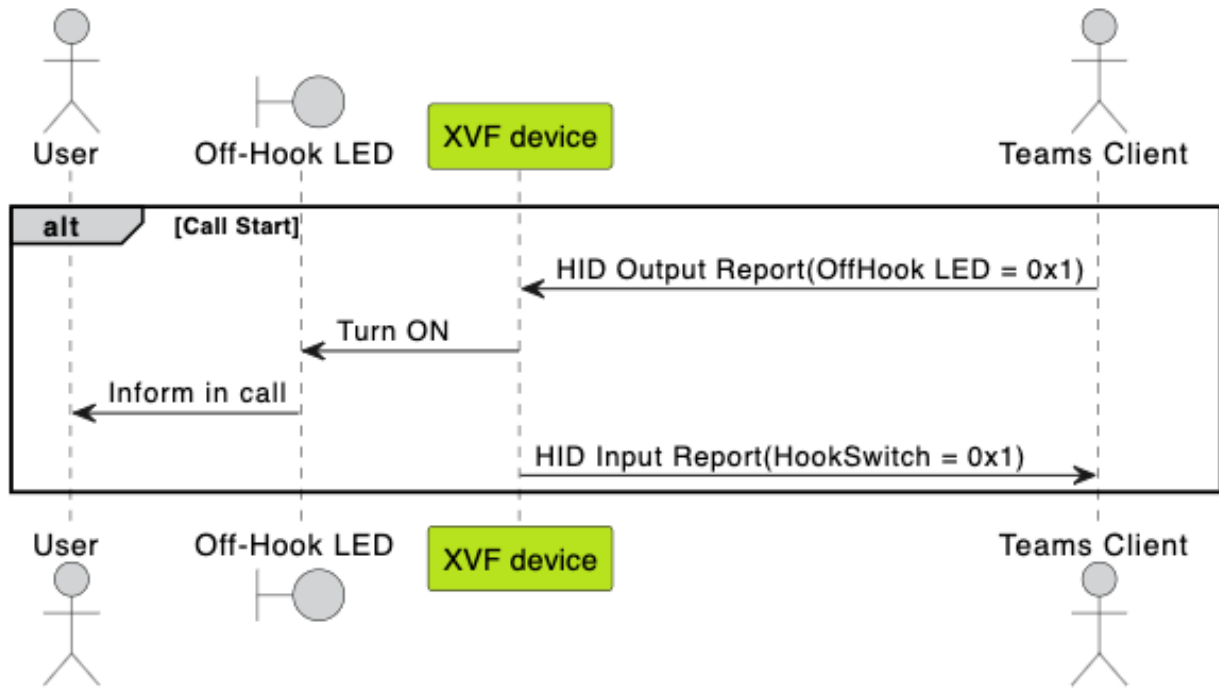

Fig. 2.23: Call Start

2.9.4.3 Inform call end

Similar to *Inform call start*, to inform call end, the host sends an output report with Off-Hook set to 0 which the device follows up with an input report with HookSwitch set to 0.
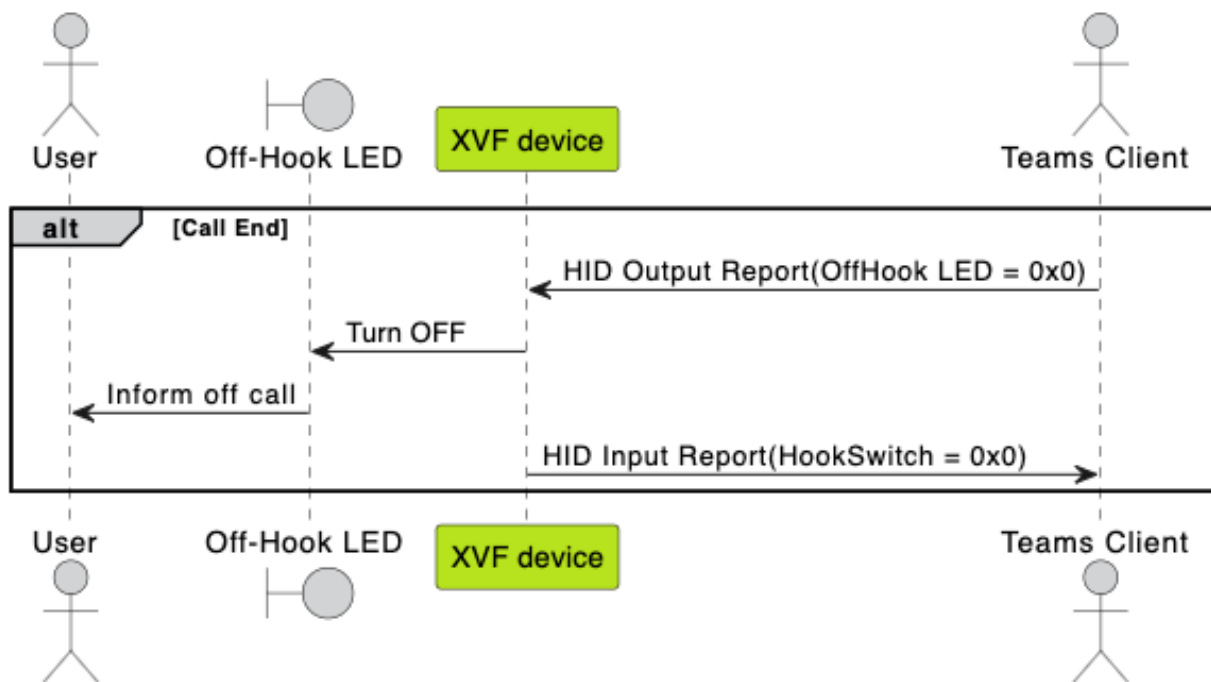
Fig. 2.24: Call End

2.10 Expanding available IO for extended HID support

This section describes extending the HID support by expanding the number of GPIOs available on the device for mapping to HID events. This is done by attaching an IO expander to the XK-VOICE-SQ66 development kit device. IO Expander ICs provide programmable GPIO and are controlled via an interface. The XVF3800 supports a build configuration (`application_xvf3800_ua-io48-lin-io-exp`) that adds support for an IO expander connected to the XK-VOICE-SQ66 development kit. The IO expander used is the PCAL6416A, which is a GPIO expander providing remote IO support and is controlled over the I^2C bus.

The PCAL6416A I^2C expander (referred to as just I^2C expander in the rest of the document) has the I^2C slave address of 0x21. Pins 0, 1, 2 and 3 on Port 0 are configured as input pins and are used for extra buttons. The incoming logic levels of these pins is read from the Input port register (00h) of the I^2C expander.

The default GPI pin to HID button mapping for the `application_xvf3800_ua-io48-lin-io-exp` build is summarised in *GPI to HID button mapping*:

Table 2.3: GPI to HID button mapping

GPI button	HID button
I^2C expander, register 00h, pin 0	Mute button
I^2C expander, register 00h, pin 1	Volume Increment button
I^2C expander, register 00h, pin 2	Flash button
I^2C expander, register 00h, pin 3	Volume Decrement button
XK-VOICE-SQ66 development kit button,	HookSwitch button

This mapping can be seen in the code wihin the `#if (IO_EXPANDER_ENABLED)` define in the `init_hid_button_config` function in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c`.

Pin 4 on Port 0 is configured as an output pin and is used for one extra LED. The output logic level of this pin is written to the Output port register register 02h of the I^2C expander. In addition to the PCAL6416A

I²C expander, the I²C expander board that is connected to the XK-VOICE-SQ66 development kit in the `application_xvf3800_ua-io48-lin-io-exp` build configuration has another LED, IS31FL3193, that is programmable via the I²C interface on address 0x68.

The default GPO pin to HID LED mapping for the `application_xvf3800_ua-io48-lin-io-exp` build is summarised in *GPO to HID LED mapping*:

<div align="center">

Table 2.4: GPO to HID LED mapping

</div>

GPO LED	HID LED
PCAL6416A I²C expander, register 02h, pin 4	Mute LED
IS31FL3193 LED	Ring LED
IS31FL3193 LED	Hold LED
XK-VOICE-SQ66 development kit Green LED,	Off-Hook LED

This mapping can be seen in the code within the `#if  (IO_EXPANDER_ENABLED)` define in the `init_hid_led_config` function in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c`.

Note: The Ring and Hold LED are mapped to the same GPO LED but with different flash modes (look at the `led_mode` field initialisation in `init_hid_led_config`). This allows distinguishing between the two events. The Ring event causes a fast flash while the Hold event causes a slow flash of the LED. The `led_mode` field initialisation in the `init_hid_led_config` configures the flash mode.

2.10.1 System Design

The system design in *HID System Design* is extended to add the I²C expander tasks. Fig. 2.25 describes the extended design:
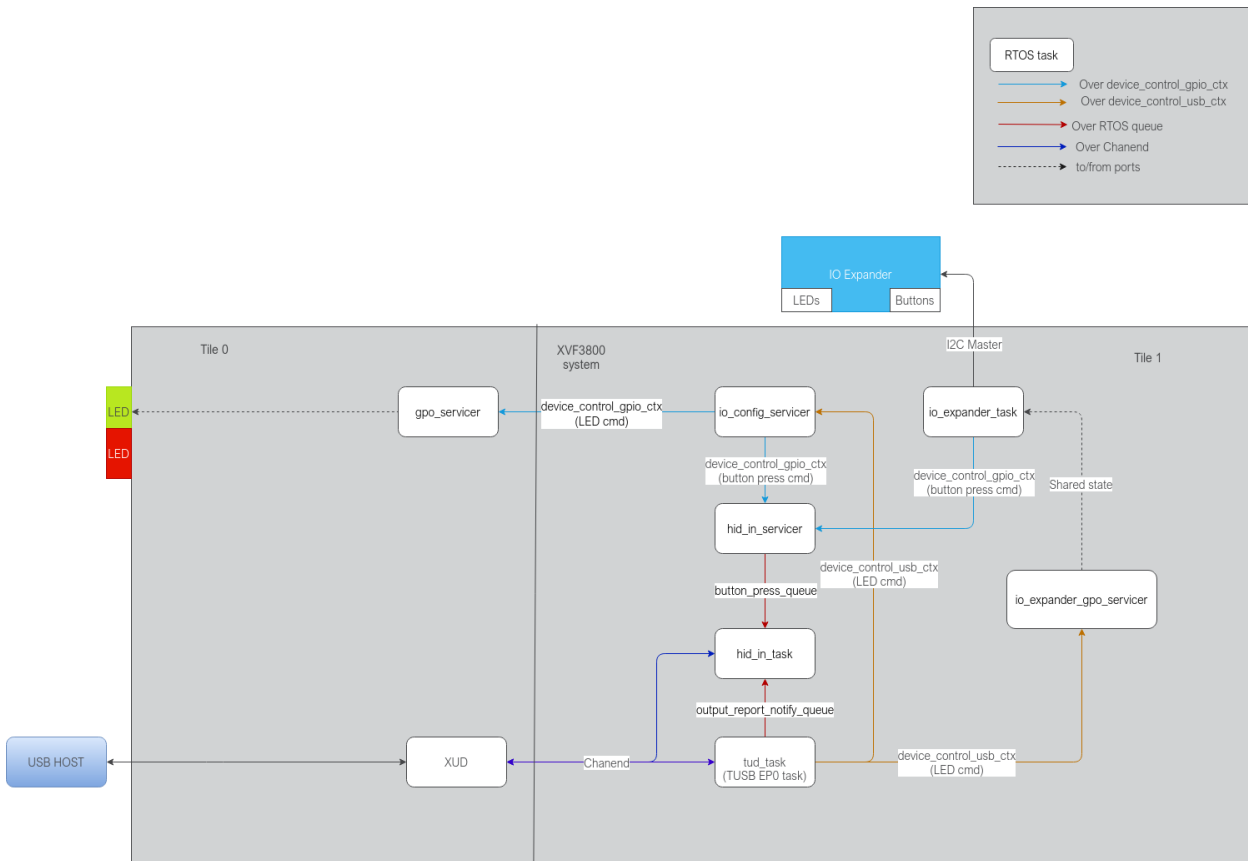
Fig. 2.25: HID with IO expander task diagram

The `io_expander_task` reads button states and programs LEDs on the I^{2}C expander over the I^{2}C master interface. It is a timer driven task that wakes up periodically, reads the button statuses and notifies button press events to the `hid_in_servicer` through control commands sent over the `device_control_gpio_ctx`. It also configures the I^{2}C expander LEDs if notified to do so by the `io_expander_gpo_servicer` task.

The `io_expander_gpo_servicer` handles GPO commands sent from `tud_hid_set_report_cb` to program LEDs in response to HID output reports. It updates the LED state in a structure shared in memory with the `io_expander_task` which then configures the LED registers over the I^{2}C interface.

The `hid_in_task` gets notified by both `io_expander_task` and `io_config_servicer` tasks with different commands sent over the same device control context about button press events.

The `tud_hid_set_report_cb` sends GPO control commands to `io_config_servicer` and `io_expander_gpo_servicer` depending on where the LED that needs to be configured resides.

Fig. 2.26 shows the updated path through which a GPI button press translates into a HID input report.
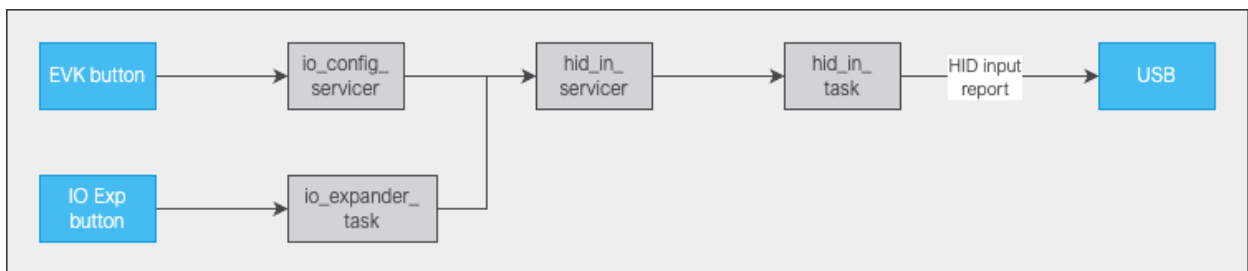
Fig. 2.26: HID button path with IO expander present

Fig. 2.27 shows the updated path through which a HID output report translates into a GPO LED state change.
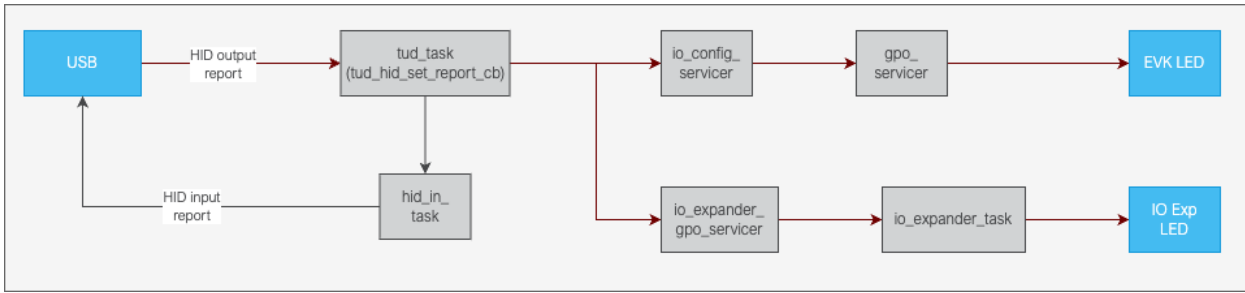
Fig. 2.27: HID LED path with IO expander present

Note: The system design can be modified to support an I^2C expander different from the PCAL6416A. Section *Adding a different I2C Expander* describes the steps required to do so.

2.10.2 HID Operation with expanded GPIO set

With the I^2C expander providing extra buttons and LEDs available to map to HID events, some more use cases are implemented as described below.

2.10.2.1 Handle Incoming Call

Fig. 2.28 shows the device handling an incoming Teams call. The two scenarios shown are the user choosing to accept or reject the call by pressing the respective buttons on the device.
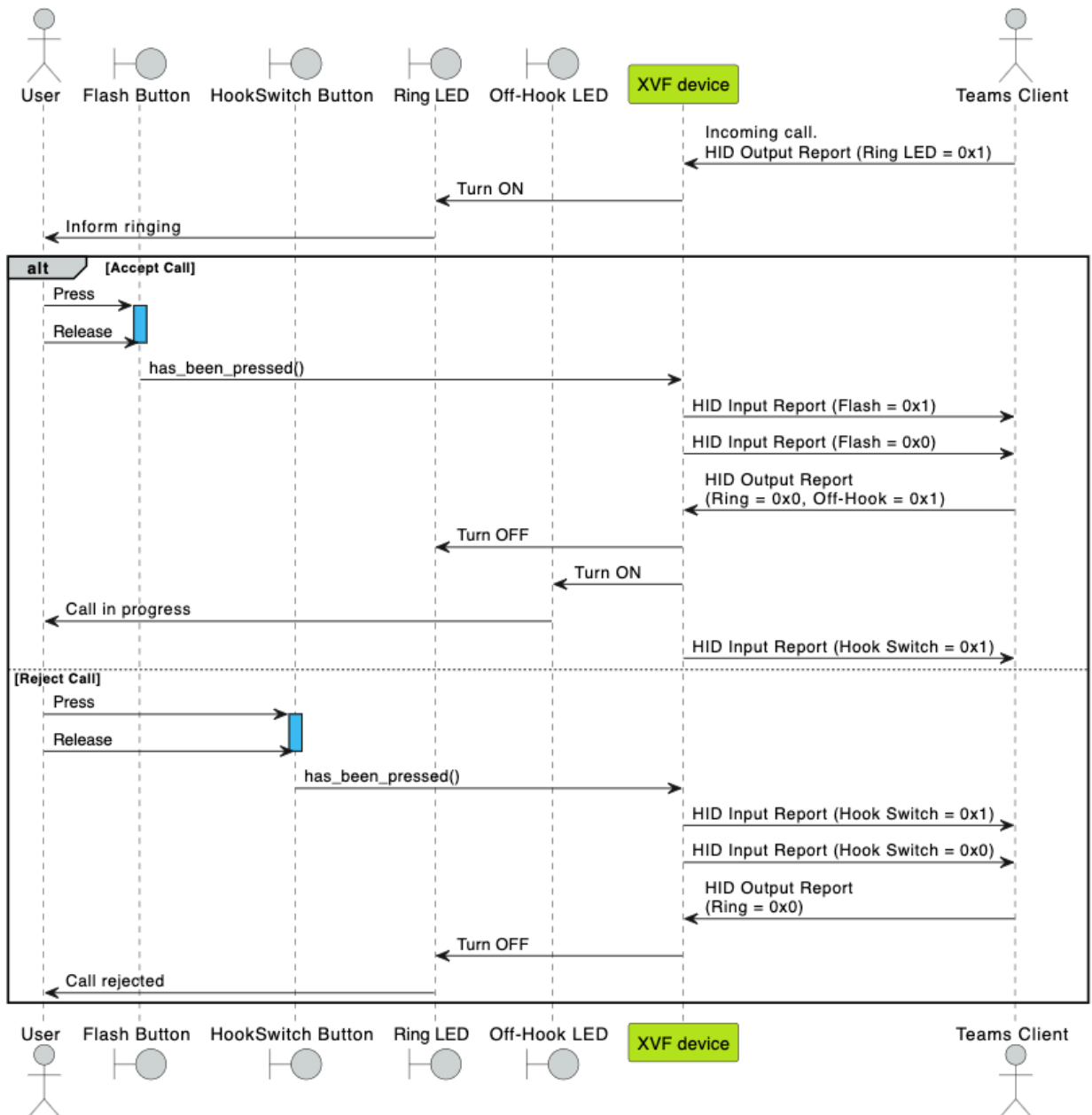
Fig. 2.28: Handle Incoming Call

2.10.2.2 End Call

Fig. 2.29 shows the user ending an ongoing Teams call by pressing a button on the device. This use case assumes an ongoing call as its precondition.
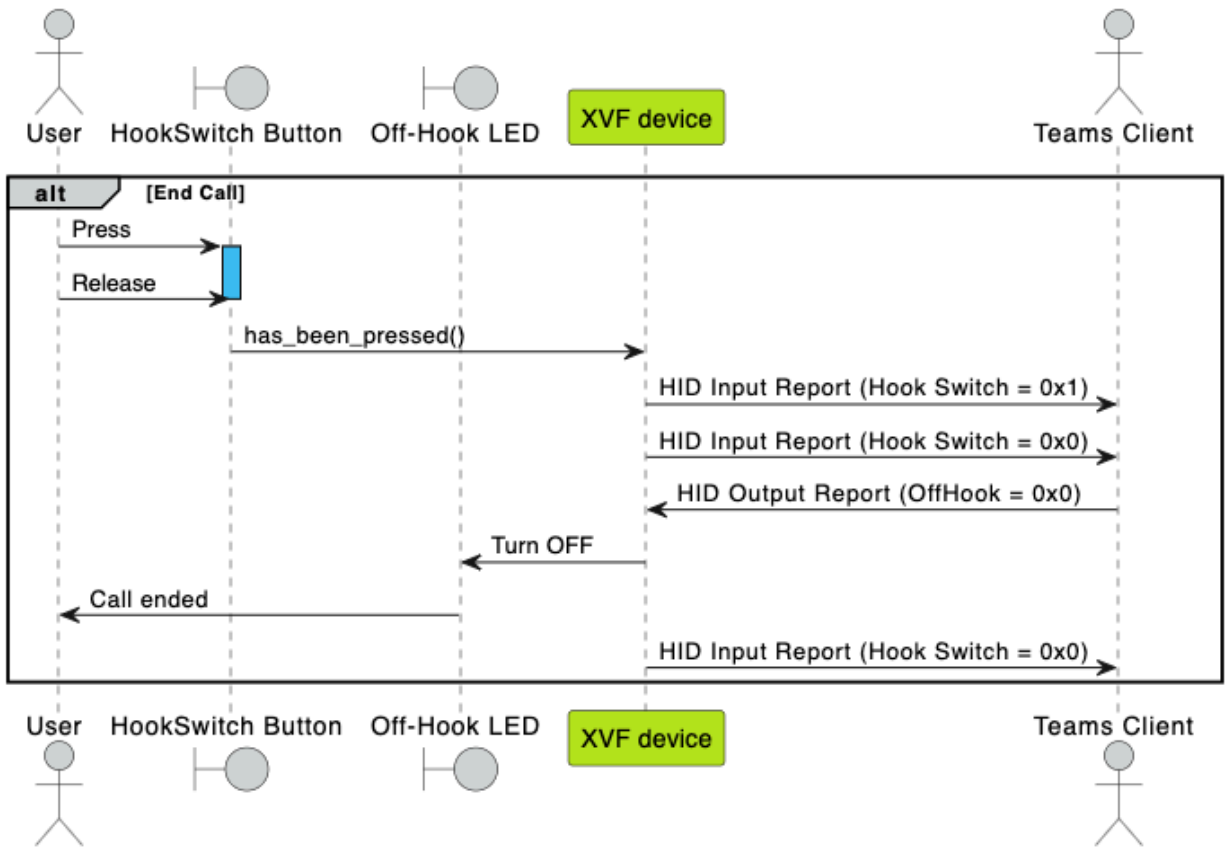
Fig. 2.29: End Call

2.10.2.3   Hold/Unhold Call

Fig. 2.30 shows the use case for placing an ongoing Teams call on hold, followed by bringing an on-hold call off hold. This use case assumes as ongoing call as its precondition.
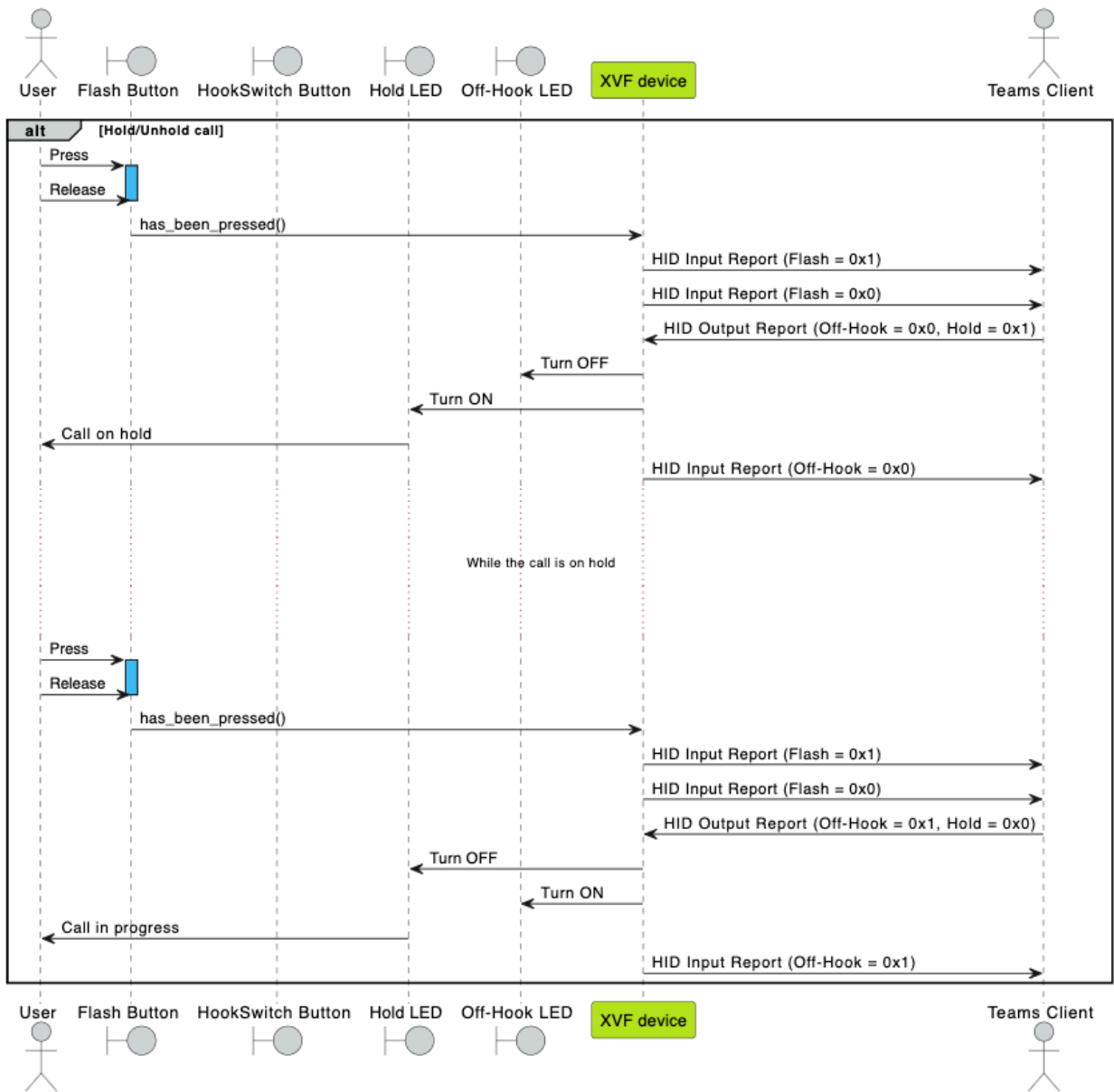
Fig. 2.30: Hold/Unhold Call

2.10.2.4 Volume Increment/Decrement

Fig. 2.31 shows the user adjusting the device volume by pressing the Volume up or down buttons on the device.
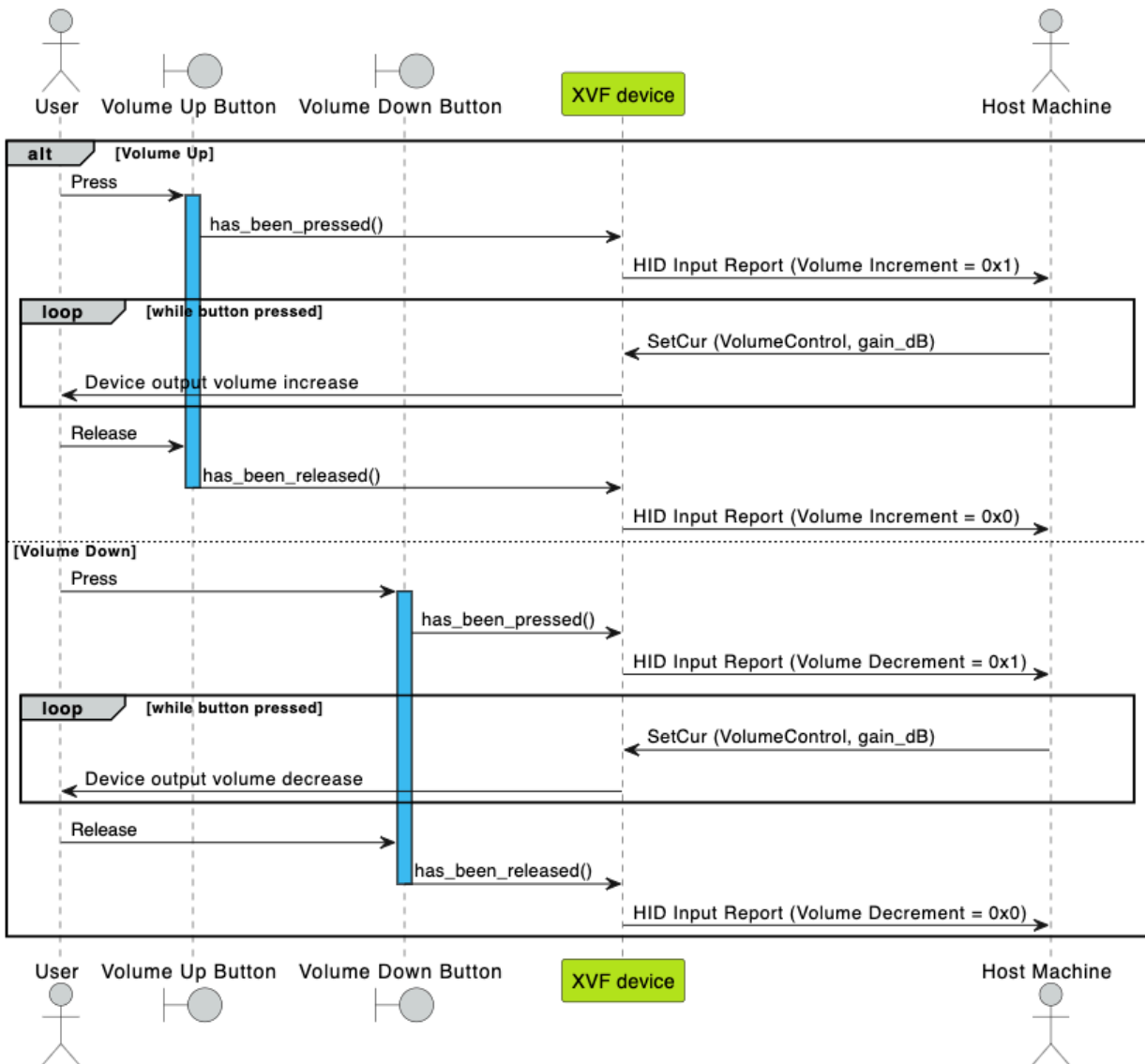
Fig. 2.31: Adjust Volume

Note: Unlike the other use cases, the volume buttons work outside of a Teams call as well. Pressing the volume button on the device has the host send SetCur volume control commands on Endpoint 0 for controlling the USB output volume. The volumes buttons are Re-Trigger Control (RTC) type which means the host continues to send the SetCur commands as long as the button is pressed.

---

3 Working With The Build System

This section will provide details on how the software is constructed. The basic steps and build requirements can be found in the README.md file which is distributed with the source.

3.1 Configuring CMake and Building the Firmware

The XVF3800 firmware follows a standard approach to building software using CMake. CMake is a cross platform build tool that supports most targets through configurable toolchains. For more details on CMake and to learn more about what the build scripts do, see the documentation at the CMake website. Each release is built with CMake 3.24.1, but any version greater than 3.21 should work.

The build process for XVF3800 works by creating CMake targets with `add_executable()` and `add_library()` that specify source files and compile flags. Each target is then linked together with `target_link_libraries()` forming the final binary. If a library is `STATIC` then it is compiled into an archive (`.a` file) using only the compilation flags that are added to that library and the ones it inherits from the libraries it is linked to. If a library is `INTERFACE` then it is not compiled into an archive; instead, all the source files and compilation flags are added to the library or executable it is linked against. The key difference is that when applying compilation flags to an executable, they will be applied to linked `INTERFACE` libraries, but not the linked `STATIC` libraries. The XVF3800 build process uses both `INTERFACE` and `STATIC` libraries, but mostly uses `INTERFACE`. This means that adding flags to the executable will usually have the desired effect.

While the above describes a standard CMake build, the toolchain used in the XVF3800 presents one key difference to the standard approach, which stems from the multi-tile design of the xcore. To enable each tile to have a separate build configuration, the toolchain constructs the XVF3800 application by merging two applications together. Multiple ".xe" files are produced for each target, one for each tile and a final one that is their combination. The CMake function `merge_binaries()`, which is defined in `xmos_macros.cmake`, creates the combined application. Only the merged binary is important; the others can be ignored.

The CMake build process follows two stages. The first is configuration; in this stage the script in the top level `CMakeLists.txt` is run, which in turn runs many other `CMakeLists.txt` and CMake scripts. To run the configuration a number of parameters need to be passed into CMake. In order to ensure the correct flags are used, a `CMakePresets.json` has been included which provides the necessary configuration presets:

- `rel_app_xvf3800` for Linux and macOS
- `rel_app_xvf3800_windows` for Windows

Configuration will generate the Makefiles for building the executable in the directory specified by the preset. The XVF3800 build process is designed so that one CMake configuration defines all target executables so managing multiple configurations is not required.

The second stage is the build stage. In this stage, GNU Make (when using Linux, macOS, or Raspberry Pi OS) or Ninja (when using Windows) is used to compile the sources as specified in `CMakeLists.txt`. To build a specific target there is a choice of using one of the build presets specified in `CMakePresets.json` or to name the target explicitly. Presets are available for existing targets, but will not exist for any new targets added to the released package. Below shows the two ways to build the same target:

```
# Build with preset.
cmake --build --preset=intdev-lr48-lin-i2c

# Build with explicit target name.
cmake --build build/app_xvf3800 --target application_xvf3800_intdev-lr48-lin-i2c
```

3.1.1 Adding or Modifying Build Configurations

The source release of XVF3800 defines a number of executables with a different combination of features. Each of these are defined in a section titled "Build profiles" in the `CMakeLists.txt` that can be found at `sources/app_xvf3800` in the source release package. The script takes the following steps:

1. Define a variable named `BUILD_PROFILES` which is a list of the names of all the targets.

2. Iterate through the build profiles, defining a list of definitions, source files and libraries for each one. A set of patterns are checked for common build flags; this is documented as comments in `CMakeLists.txt`.

3. Iterate through them again defining the libraries and executables required for the target, including the extra flags defined in the previous step.

To add a new target, the name must be included in the BUILD_PROFILES list. There are two pathways available for configuring a new target. The first is to build the name out of the patterns described in `CMakeLists.txt` to configure the desired combination of data rate, microphone geometry, etc. Alternatively, choose a completely different name and then add the flags that are needed. Care must be taken to ensure a valid combination of flags and sources are used, start by copying from an existing target. An example of what that may look like is shown in this example which creates a target named "my-custom-target". This shows the sections of `CMakeLists.txt` which would need to be modified, "…" represents skipped parts of the file.

```
set(BUILD_PROFILES
    "intdev-lr16-lin-spi"
    "intdev-lr16-sqr-spi"
    # ...
    "my-custom-target"    # 1. Add target to the list.
)

# ...

foreach(PROFILE ${BUILD_PROFILES})

    # ... assorted checks for common patterns ...

    # 2. Add build flags for the target with the new name.
    #    Note that this must be *above* the check for EXTRA_BUILD_INCLUDED.
    if(PROFILE STREQUAL "my-custom-target")
        add_to_build_opts("INT_DEVICE=1")
        add_to_build_opts("appconfLRCLK_NOMINAL_HZ=48000")
        add_to_build_opts("appconfSPI_CTRL_ENABLED=0")
        add_to_build_opts("appconfI2C_CTRL_ENABLED=1")
        add_to_build_opts("appconfUSER_CONFIG_ENABLED=0")
        add_to_build_sources(src/default_params/product_defaults.c)
        add_to_build_libs(fwk_xvf::bsp_config::xk_voice_sq66_evk)

        # Set a boolean to indicate if this is an extra build.
        set(EXTRA_BUILD_INCLUDED ON)
    endif()
endforeach()
```

This will create a target named `application_xvf3800_my-custom-target` which can be compiled as explained in section *Configuring CMake and Building the Firmware*.

3.1.2 Adding New Files and Compilation Flags to the Build

For simple changes, such as adding individual files and new definitions, the example shown in *Adding or Modifying Build Configurations* can easily be extended to add as many files and defines as desired using `add_to_build_sources()` and `add_to_build_opts()`. If something more complex is desired, the best option will be to define a new `INTERFACE` library and include it in the build.

The following is a basic example of how to do this. This will not cover anything an experienced CMake user hasn't seen before. To create a library named my_custom_lib, create a directory named `my_custom_lib` under the `sources/app_xvf3800` directory with the contents shown:

```
sources/app\_\ |project_lc|
    CMakeLists.txt
    my_custom_lib
        CMakeLists.txt
        my_custom_source.c
        my_other_source.c
        my_custom_source.h
```

Somewhere near the top of `sources/app_xvf3800/CMakeLists.txt` add the following line so that CMake knows the new directory exists:

```
add_subdirectory(my_custom_lib)
```

Now add the following to `sources/app_xvf3800/my_custom_lib/CMakeLists.txt`:

```
# create the target.
add_library(my_custom_lib INTERFACE)

# add the sources.
target_sources(my_custom_lib INTERFACE my_custom_source.c my_other_source.c)

# add current directory to the search path so the header
# file can be used in other places.
target_include_directories(my_custom_lib INTERFACE ${CMAKE_CURRENT_LIST_DIR})

# Add arbitrary defines and flags and anything else CMake will allow.
target_compile_definitions(my_custom_lib INTERFACE MY_CUSTOM_FEATURE=1)

# Add flag to individual file.
set_source_files_properties(my_other_source.c PROPERTIES COMPILE_OPTIONS "-O0")
```

The final step is to link the library against the new target. This requires adding the following line to `sources/app_xvf3800/CMakeLists.txt` alongside the rest of the configuration for the target that is shown in *Adding or Modifying Build Configurations*.

```
add_to_build_libs(my_custom_lib)
```

3.2 Building the Host Control App

The host control application is distributed with the release as a binary alongside its associated shared libraries. The control app requires awareness of all the control parameters in the software. The control parameters are defined in YAML files that are located at `sources/app_xvf3800/autogeneration/yaml_files/control_commands` in the source release package. When one of these files is modified it is not necessary to rebuild the whole host control application; only the shared library containing the command map needs to be updated. This shared library is named `command_map.dll` for Windows, `libcommand_map.so` for Linux and Raspbian, and `libcommand_map.dylib` for macOS.

> **Warning:** The shf_aec_cmds.yaml and shf_pp_cmds.yaml files are auto-generated and should not be manually updated.

`(lib)command_map.(so/dll/dylib)` is compiled from source files that are generated based on the YAML files. Updating the command map library requires:

- the XVF3800 source release package, including all sub-modules
- CMake with minimum version 3.13
- Python with the required packages.

CMake can be installed from any location using:

```
sudo apt install -y cmake
```

The required Python packages can be installed by running the following commands from the directory containing the source release package:

```
pushd sources
pip3 install -r requirements.txt
popd
```

To build the command map, run the following commands from the directory containing the source release package:

```
pushd sources/modules/fwk_xvf/modules/host_cmd_map
cmake -B build
pushd build
make
popd
popd
```

The compiled `libcommand_map.so` should then be copied to the same directory as `xvf_host`, and it will be used automatically. The build process is defined by `sources/modules/fwk_xvf/modules/host_cmd_map/CMakeLists.txt`. This includes finding the YAML files, generating the source code and finally building the shared library. The default output folder is `sources/build/release/host/product/<OS>`, where `OS` is the name of the platform used to build the files.

It is possible to generate different `libcommand_map` files for different products. For example a user may need multiple files each with different USB product and vendor ID's. These values are set in the file `sources/app_xvf3800/autogeneration/yaml_files/settings_and_defaults/product/usb_param_values.yaml`. To have multiple `libcommand_map` files, do the following:

1. Create a new folder in `sources/app_xvf3800/autogeneration/yaml_files/settings_and_defaults/`, for example `new_product`

2. Copy the YAML files in `sources/app_xvf3800/autogeneration/yaml_files/settings_and_defaults/product` into the new folder

3. Update the values in the new YAML files

4. Overwrite the CMake variable `PRODUCT_DEFAULT_SPECIFIER` in `sources/modules/fwk_xvf/modules/host_cmd_map/CMakeLists.txt` for example by adding:

   ```
   set(PRODUCT_DEFAULT_SPECIFIER new_product)
   ```

5. Re-build the command map.

The new files will be generated and stored in `sources/build/release/host/new_product/<OS>`, where `OS` is the name of the platform used to build the files.

4 Testing the Software

The XVF3800 is supplied as a verified package built under a CI system with extensive regression tests to cover all key aspects of the functionality. Since it is supplied as source, any user modifications may potentially affect functionality, and/or timing, of the firmware. The multi-core and hard real-time XCORE architecture lends itself very well to running multiple tasks robustly; however, there are ultimately cycle and memory limitations which are present. The following section describes the various tests that can be performed following source code modification of the firmware to verify that it is still functional and works as expected.

4.1 Test Capabilities

Each of the following sections details the classes of test that are supported. These generally use the audio muliplexer (mux) capability of the XVF3800, which allows output sources to choose between a number of internal (and external input) signals.

The mux and routing capability is extremely flexible and powerful. A single output mux (represented in Fig. 4.1) has the ability to individually choose a specific signal. Each output channel (left or right) on an output stream has its own mux.
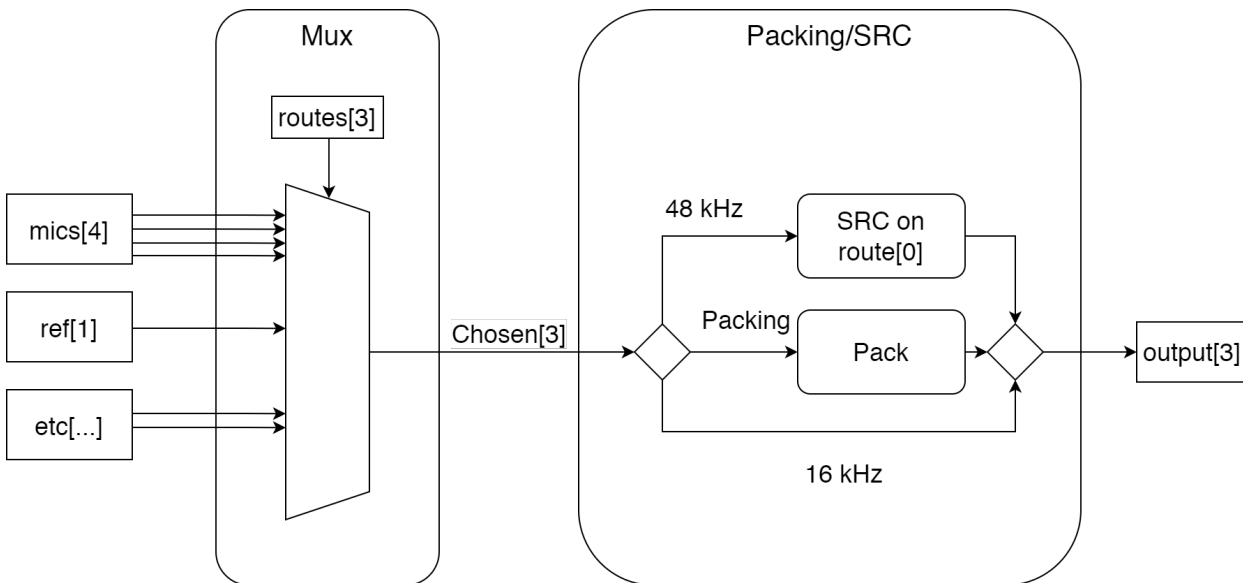
Fig. 4.1: A single output channel's Mux and Packing/SRC block

The complete signal path of the XVF3800 is shown in Fig. 4.2. There are many useful signals which can be routed out of the device and the following sections provide some practical examples of using this capability for testing specific parts of the system.
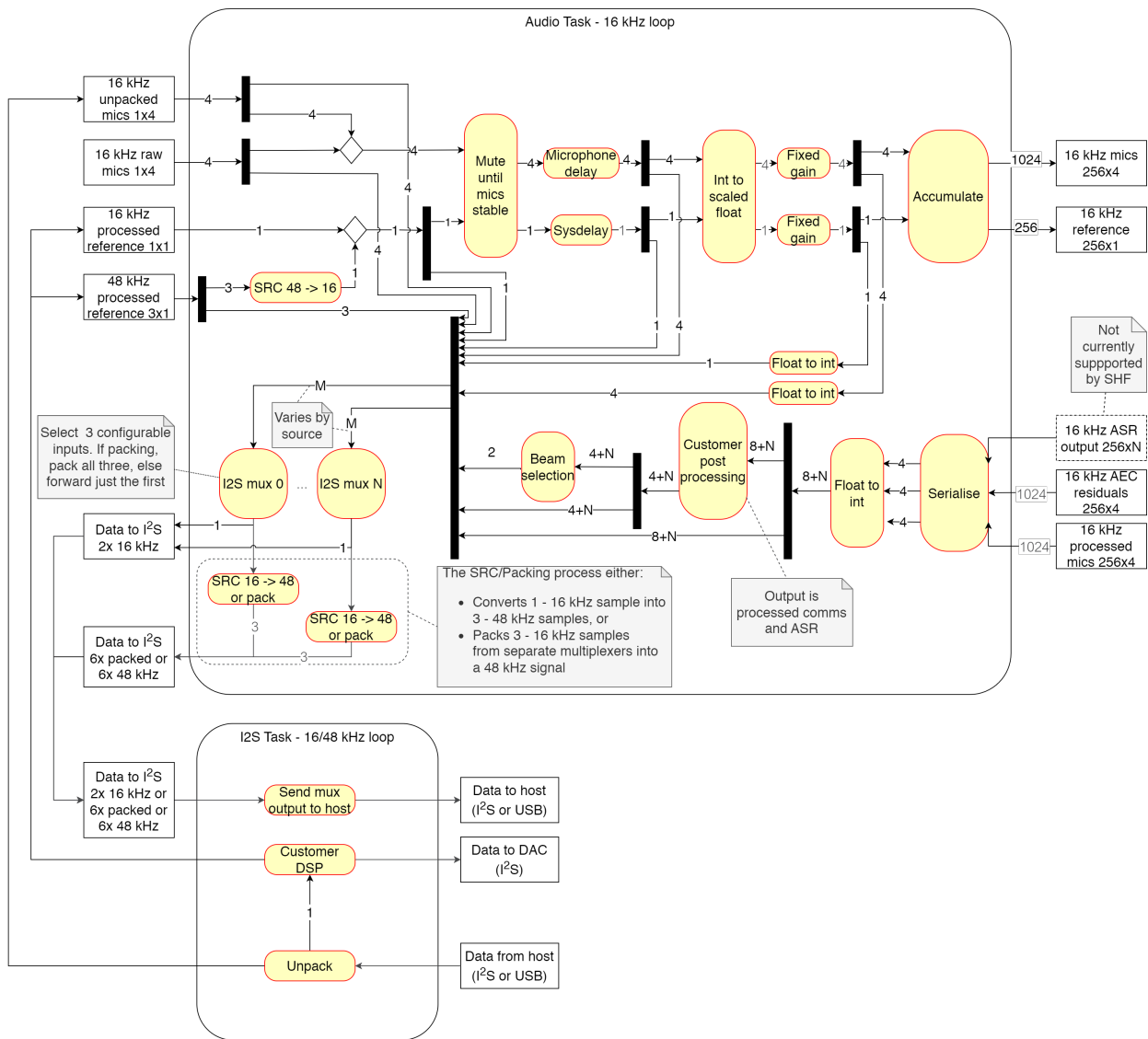
Fig. 4.2: Complete audio path through the XVF3800

The Setting Up the Hardware section of the XVF3800 User Guide provides additional details on the numerous signals available via the mux and how to translate the desired audio source into the enumerated value for passing to the host app.

4.1.1 Loopbacks

When integrating the XVF3800 into the overall system it can be helpful to test signal path from and back to the host.

Note: All examples below assume a host rate of 48 kHz, which is 3 the voice-DSP rate.

Note: In the rest of this document when using the `xvf_host` app in the code examples, the command is written as `(sudo) xvf_host(.exe)`. The `.exe` extension is only required on Windows. The `sudo` command is only required on Linux, macOS and Raspberry Pi OS if the user does not have the necessary permissions to access the device. On these platforms it may be necessary to use `./` before the command if the directory

containing the `xvf_host` app is not in the `PATH`.

This 'round trip' test is useful for validating the host->device and device->host paths at the same time. Normally, an audio stream provides the far-end reference, the microphones provide the near-end and an audio output stream provides the processed microphone output.

In this example, the mux will be modified so that the far-end reference is looped back to the audio output stream. This can be useful for evaluating propagation delays and/or volume scaling within the complete system. The first command ensures that the native signal is used (instead of being down-sampled and then up-sampled) and the second command routes the raw far-end DSP signal to the output mux.

Note: Any far-end DSP used will be included in this signal path and so it can also be used to check far-end DSP operation. See *Modifying the Software* for an example of adding custom far-end DSP.

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 0 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_ALL 10 0 10 2 10 4 10 1 10 3 10 5
```

Another useful feature is to be able to capture the raw microphone signals and listen to them without passing them through the voice-DSP. This can be helpful when validating custom hardware to check that the microphones are properly connected and also evaluate the relative gain between them. This is useful for debugging when developing custom enclosures.

The first command ensures that up-sampling is used (if the host interface is different from the native microphone rate of 16 kHz) and the subsequent commands route the raw microphone signals to the output mux. No microphone gain will be applied. The microphone front-end is tuned to support the acoustic overload point of the microphones without clipping and hence sounds quiet for normal listening without gain.

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 1 1
# Set the left output to raw mic 0 and the right output to raw mic 1
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 1 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 1 1
# Set the left output to raw mic 2 and the right output to raw mic 3
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 1 2
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 1 3
```

The amplified microphone signal (again without voice-DSP) is also available at the mux. This is helpful for tuning the system for the voice-DSP which is optimised for a certain microphone level.

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 1 1
# Set the left output to amplified mic 0 and the right output to amplified mic 1
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 3 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 3 1
# Set the left output to amplified mic 2 and the right output to amplified mic 3
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 3 2
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 3 3
```

The AEC residuals are also available at the mux. These signals are the output directly from the AEC and can be helpful for tuning other echo suppression functions such as non-linear echo and echo suppression.

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 1 1
# Set the left output to aec residuals of mic 0 and the right output to aec residuals of
# mic 1
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 7 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 7 1
# Set the left output to aec residuals of mic 2 and the right output to aec residuals of
# mic 3
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 7 2
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 7 3
```

Finally, it may useful to test the background noise of a system. The mux can also provide zero samples. The below commands show how to do this.

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 1 1
# Set the left and right outputs to silence
(sudo) xvf_host(.exe) AUDIO_MGR_OP_L 0 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_R 0 0
```

For instruction on capturing more than two signals at a time, please see the *Signal Capture* section.

4.1.2 Signal Capture

It can be very useful to capture the entire voice-DSP pipeline input including the unprocessed microphones and the far-end reference. This allows a test vector for a particular acoustic environment to be captured which can then be inspected or even processed offline, such as being run through a simulated or hardware voice-DSP processing system. Processing the same vector offline allows repeatable testing while tuning parameters for example, or even providing a test vector when requesting technical support. See the *Signal Injection* and *Signal Injection and Capture Simultaneously* sections for how to provide a test vector and re-use it in the system.

Note: All signal injection/capture features require the host audio rate to be running at 3 the voice-DSP rate. The reason for this is that the voice-DSP requires multiple channels (microphones and reference signals) and the output is normally 2 channels. The multiple channels of the voice-DSP signals are packed into a stereo signal at one third the rate. Scripts are provided for packing/unpacking the signals.

The signal injection/capture features support different bit depths depending on the firmware configuration and the host platform. Over I^{2}S the XVF3800-INT device requires a bit depth of 32 bits. Over USB the XVF3800-UA device supports 16, 24 and 32 bit depths, but there are some limitations due to the host OS platforms, as described in the table below.

Note: The USB bit depth does not affect the precision of internal computations, and will not affect the algorithm performance.

Table 4.1: USB bit depths supported by host platform

Host platform	Supported bit depths
Linux	16, 24 and 32
macOS	16 and 24
Windows	16 and 24

32 bit audio devices are not supported by Windows and macOS operating systems. However, 24 bit audio provides 144 dB of dynamic range, which is more than enough for professional audio applications. 16 bit audio provides 96 dB of dynamic range, which should be adequate for most applications.

Note: Whilst 32 bit packed recordings can only be made on Linux, the resulting wav files can be unpacked

---

and processed on any operating system.

Windows may experience issues when changing the bit depth between 16 and 24 bits. If an issue occurs, the audio device can be reinstalled with the following steps:

1. Connect the XK-VOICE-SQ66 development kit to the host computer using a USB cable, configured at the desired sample rate.

2. Open *Device Manager* and find the "XVF3800 Voice Processor" under *Sound, video and game controllers*.

3. Right-click on the device and select *Uninstall device*. Click *Uninstall* on the warning box that appears.

4. Right-click anywhere in the list of devices and select *Scan for hardware changes*.

5. The XVF3800 should now reinstall and enumerate at the correct bit depth.

The Python scripts in this section make use of the script `xvf_tools.py` which is located in the `sources` folder of the XVF3800 source release package. This program allows the user to call a script from the `sources` folder, by mapping the desired script name to the correct location in the source release package. It is used as follows:

```
python3 xvf_tools.py <script_name without .py extension> [command arguments]
```

and the help menu of the script to run can be printed with:

```
python3 xvf_tools.py <script_name without .py extension> --command-help
```

In the remainder of this section the script to be used will be referred to as a command.

For making packed recordings on USB devices, `xvf_tools.py` provides the command `packed_recorder`. This command requires the installation of the Python packages listed in `requirements_build.txt`. If using a Linux platform, the package `sounddevice` requires the installation of the `libportaudio2` library.

To make a recording while playing back a 48 kHz reference signal, use the command:

```
python3 xvf_tools.py packed_recorder <host_app> --playback_file <my_48kHz_stereo_reference_signal.
↪wav> --packed_output
```

where `<host_app>` is the path to the `xvf_host` file, this can be an absolute path or a path relative to the current working directory.

To make a 10 second recording without playback, specify a recording length instead of a playback file:

```
python3 xvf_tools.py packed_recorder <host_app> --recording_length 10 --packed_output
```

By default the output is saved to `packed_rec.wav` and `unpacked_rec.wav`. The paths can be changed using the `--packed_output_file` and `--unpacked_output_file` command line arguments. The following 6 channels are output:

- Channel 1 is the left far-end reference, with reference gain and system delay applied
- Channel 2 is the processed output (autoselect beam)
- Channel 3 is MIC 0, with mic gain and system delay applied
- Channel 4 is MIC 1, with mic gain and system delay applied
- Channel 5 is MIC 2, with mic gain and system delay applied
- Channel 6 is MIC 3, with mic gain and system delay applied

`packed_rec.wav` contains the 6 channels interleaved in a stereo 48 kHz stream, with packing markers in the LSB. `unpacked_rec.wav` splits this into a 16 kHz 6 channel signal.

An image of an example 6 channel unpacked captured wav can be seen in Fig. 4.3. The far-end reference can be seen playing and each of the microphones (0..3) have been tapped in sequence to show a large noise source being captured.

Fig. 4.3: Unpacked output from the XVF3800

To change the channels that are packed, use the `--op_all` flag. The available categories and sources are as detailed in the Output Selection section of the XVF3800 User Guide. For additional usage instructions, please run `python3 xvf_tools.py packed_recorder --command-help`.

If not using the provided python script or using an I2S device, the following commands can be used to capture the mono far-end signal (with delay and gain if enabled), the four amplified (and optionally delayed) raw microphone signals and the processed output (autoselect beam in this case):

```
# Enable packed output
(sudo) xvf_host(.exe) AUDIO_MGR_OP_PACKED 1 1
# Set the 48 kHz stereo to output all 5 channels of input to the voice-DSP
(sudo) xvf_host(.exe) AUDIO_MGR_OP_ALL 12 0  3 0  3 2  6 3  3 1  3 3
```

Next, the output signal may be recorded and unpacked to a channel wav file. The example below uses the Linux `arecord` utility to capture the signal of a XVF3800-INT device connected to a Raspberry Pi over I2S. It may be beneficial to invoke a background `aplay` instance to provide far-end audio before this is run:

```
# Play the desired far-end reference signal in the background
aplay <my_48kHz_stereo_far_end_reference_signal.wav> &
# Run a stereo audio capture at 48 kHz with 32b bit depth for 60 sec
arecord -r 48000 -f S32_LE -c 2 -d 60 <capture_48k_2ch.wav>
# Unpack the 48 kHz stereo packed file into a 6ch 16 kHz unpacked wav using 32b sample depth
python3 xvf_tools.py packing unpack <capture_48k_2ch.wav> <unpacked_16k_6ch.wav> -b 32
```

The file *<capture_48k_2ch.wav>* has been recorded at 48 kHz stereo and at the full bit width of I^{2}S of 32b which includes the packing markers in the LSB. The output from the unpack operation is a 16 kHz, 6 channel signal with the channel designation as in the previous example.

Since the packer needs chunks of three samples, it is likely that the first and last frame are not complete; this may cause the following warning which means partial frames at the start and finish have been discarded. Discarding partial frames is important to ensure the remaining unpacked samples are correctly time aligned:

```
Warning: Bad indices: [[113999      0]]
```

The `packing` command may produce the following error message with no output file generated:

```
Error: Over 50 markers incorrectly spaced so giving up.
```

It means that:

- The audio mux in the XVF3800 has not been configured to properly produce a packed output,
- The sample resolution is incorrect,
- The capture process has been corrupted (perhaps by volume scaling), or
- The `--skip-leading-zeros` option was not included on the command line (USB build configurations only).

Re-check the mux configuration commands, the host system controls and the options on the `packing.py` command.

Note: Signal packing uses least-significant bit markers to encode the channel packing sequence. These are then stripped so do not contribute to noise. For 32b audio, the least-significant bit is over 190 dB down from full scale and the loss of precision is insignificant. However, it is critical to ensure that any volume controls are disabled (volume = 100%) to prevent the packed audio frame being corrupted.

4.1.3   Signal Injection

The XVF3800 supports a mode where the input to DSP pipeline can be fed directly from a 5-channel test vector which may either be pre-generated or even pre-captured by recording directly from an XVF3800 device - see *Signal Capture*. This can be helpful when re-creating a previously seen scenario or when tuning the system via the control interface in the presence of a fixed and repeatable test vector.

Note: The XVF3800 contains a lot of state such as pre-learned AEC coefficients. When re-running a particular test vector it is important to ensure the device is reset, either by individually resetting the various blocks or alternatively by resetting the entire firmware using `(sudo) xvf_host(.exe) TEST_CORE_BURN 0`, which will force a reboot of the firmware from the host application.

The vector injection mode works by packing 6-channel 16 kHz input data (four microphones, a mono reference and an unused channel) into a 48 kHz stereo input signal. The device then unpacks the 48 kHz wav file into a 16 kHz multi-channel input and feeds it directly into the front end of the voice-DSP pipeline.

Note: It is essential to use a 48 kHz host audio rate for this process to work since the higher rate is needed to support channel packing.

The required format of the 6 channel test vector should be as follows:

- Channel 1 is the far-end reference signal
- Channel 2 is ignored

- Channel 3 is the amplified raw MIC 0

- Channel 4 is the amplified raw MIC 1

- Channel 5 is the amplified raw MIC 2

- Channel 6 is the amplified raw MIC 3

Suitable test vectors may be obtained directly from the XVF3800 using the *Signal Capture* procedure.

If using `packed_recorder`, the `packed_rec.wav` can be used as a packed input. This can then be replayed using:

```
python3 xvf_tools.py packed_recorder <host_app> --playback_file my_packed_vector.wav --packed_input
```

The output audio location can be set using the `--unpacked_output_file` command line argument.

Otherwise, to turn the six channel 16 kHz test vector into a 48 kHz packed file call:

```
# Pack the 6ch 16 kHz unpacked wav into a 48 kHz stereo file using 32b sample depth
python3 xvf_tools.py packing pack <my_6ch_vector.wav> <my_packed_vector.wav> -b 32
```

Then, with the firmware freshly booted and running, configure the input to accept a packed signal. It is important to consider gain and system delay when playing a packed input signal. The commands below indicate how to remove those completely. When packed input is enabled, the fixed 6 dB attenuation is skipped on the assumption that the packed reference signal was captured from the device and has already been attenuated. If this is not the case then set AUDIO_MGR_REF_GAIN to 0.5 to emulate this behaviour.

```
# Configure I2S or USB to unpack a 48 kHz packed input
(sudo) xvf_host(.exe) I2S_INPUT_PACKED 1
# The next steps assume the input signal was captured after gain and system delay was applied.
# These commands will prevent additional gain and system delay being applied to the test signal.
(sudo) xvf_host(.exe) AUDIO_MGR_MIC_GAIN 1.0
(sudo) xvf_host(.exe) AUDIO_MGR_REF_GAIN 1.0
(sudo) xvf_host(.exe) AUDIO_MGR_SYS_DELAY 0
```

With the firmware ready to run a packed input, now is a good time to perform any configuration via the control utility such as tweaking tuning parameters.

Finally, play the input vector. In this case a background `arecord` session has also been run to capture the output from the XVF3800 simultaneously.

```
# Run a stereo audio capture at 48 kHz with 32b bit depth in the background
arecord -r 48000 -f S32_LE -c 2 <test_output.wav> &
# Play the pre-packed test vector signal and terminate the recording when done
aplay <my_packed_vector.wav> && killall arecord
```

Listen to and inspect the output file, which contains the processed output from the input test vector.

4.1.4 Signal Injection and Capture Simultaneously

Enabling multi-channel input and output at the same time allows a full hardware-in-the-loop (HIL) system with a high degree of repeatability and visibility, as represented in Fig. 4.4. Not only can the same test vector be repeatably run through the system but multiple outputs may be observed simultaneously from different parts of the system including:

- The raw inputs to the voice-DSP to ensure correct transport and injection/capture.

- The delayed microphone/far-end inputs to check that the input to the AEC is causal (the far-end reference must arrive before the acoustically coupled echo).

- The amplified microphones to ensure that the microphone amplifier gain has been tuned correctly.

- Multiple beam outputs to help determine which of the beams performs best in the desired application.

- The AEC residual signals to determine how much to tune the post-processing stages to trade off echo cancellation versus double-talk performance.

- Processed far-end DSP to ensure that the far-end DSP is performing as expected.
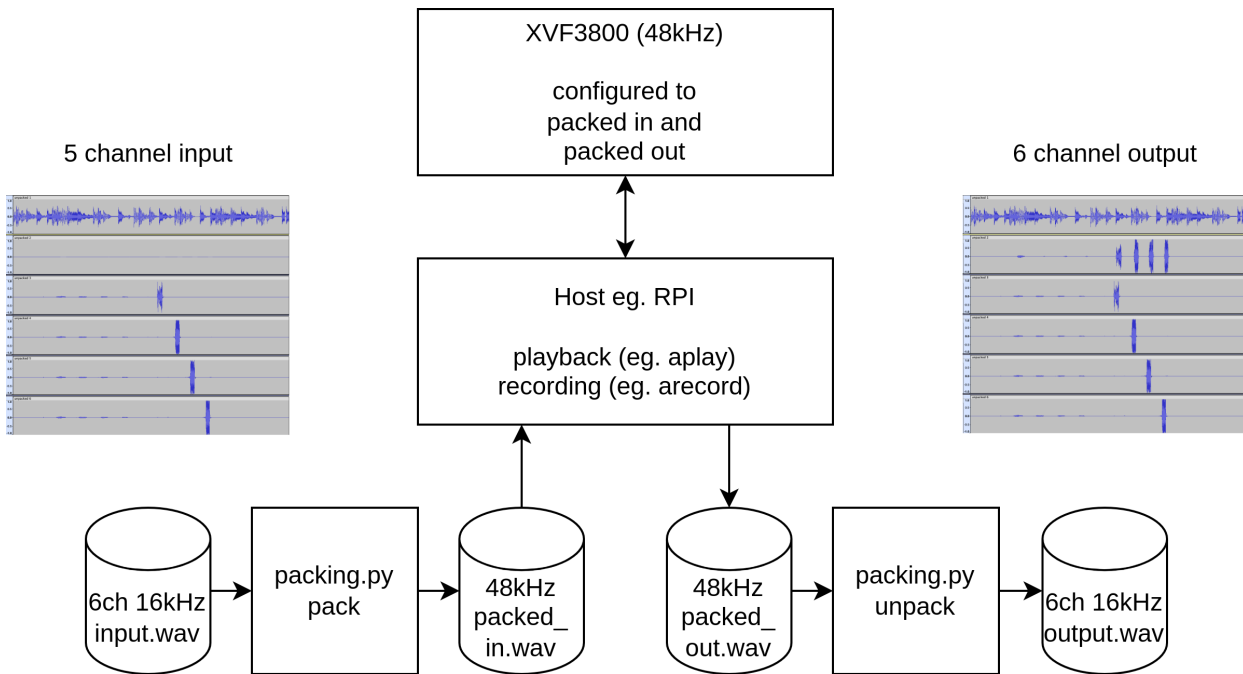
- ...and many more.

Fig. 4.4: Representation of a fully HIL workflow with the XVF3800

If using `packed_recorder`, simultaneous packed input and output can be used by calling:

```
python3 xvf_tools.py packed_recorder <host_app> --playback_file <my_packed_vector.wav> --packed_
↪input --packed_output
```

The `--op_all` flag may be used to customise the packed channels.

Otherwise, an example of simultaneously injecting 5 channels of near and far-end signals whilst capturing the same 5 channels (plus the auto-select processed output) is shown below. For further details on the individual steps shown below, please consult the *Signal Capture* and *Signal Injection* sections above and be aware that resetting the firmware before each run is essential for consistency. See *here for instructions on how to reset the firmware from the host app*.

```
# Enable packed output
(sudo) xvf_host(.exe) AUDIO_MGR_OP_PACKED 1 1
# Set the 48 kHz stereo to output all 5 channels of input to the voice-DSP
(sudo) xvf_host(.exe) AUDIO_MGR_OP_ALL 5 0 3 0 3 2 6 3 3 1 3 3
# Configure I2S or USB to unpack a 48 kHz packed input
(sudo) xvf_host(.exe) I2S_INPUT_PACKED 1
# The next steps assume the input signal was captured after gain and system delay was applied.
# These commands will prevent additional gain and system delay being applied to the test signal.
(sudo) xvf_host(.exe) AUDIO_MGR_MIC_GAIN 1.0
(sudo) xvf_host(.exe) AUDIO_MGR_REF_GAIN 1.0
(sudo) xvf_host(.exe) AUDIO_MGR_SYS_DELAY 0
```

```
# Pack the 6ch 16 kHz unpacked wav into a 48 kHz stereo file using 32b sample depth
python3 xvf_tools.py packing pack <my_6ch_vector.wav> <my_packed_vector.wav> -b 32

# Play the test vector in the background
```

```
aplay <my_packed_vector.wav> &
# Run a stereo audio capture at 48 kHz with 32b bit depth for 60 sec and stop the
# test_vector playback when done
arecord -r 48000 -f S32_LE -c 2 -d 60 <packed_capture.wav> && killall aplay

# Unpack the 48 kHz stereo packed file into a 6ch 16 kHz unpacked wav using 32b
# sample depth
python3 xvf_tools.py packing unpack <packed_capture.wav> <unpacked_capture.wav> -b 32
```

4.2 Measuring Resources

In any embedded system, resources are limited and the XVF3800 is no exception. The main resources of concern are memory and processing cycles when adding additional source code. The methods for ensuring the application remains within the available resources are described in the following subsections.

4.2.1 Measuring Available Cycles

Adding extra control code for initialising hardware, in general, has no effect on the real-time portion of the firmware thanks to the hardware scheduler and multiple logical cores of the XCORE. However, limited processing cycles are available to add user-DSP. Exceeding these limits will cause audio glitching or in some cases cause firmware instability.

Tools are provided, via the command interface, to allow quantifying of the number of cycles available and exercise the worst case timing case within the firmware. Exercising the worst case timing in the firmware and ensuring a non-zero number of processing cycles are still available is the best way to gain confidence that the code will always meet timing when deployed in the field under any operating conditions.

A number of control commands are available to assist with verifying that timing has not been violated. These commands report idle time, which is the amount of free time available within the audio loops. As this number approaches zero there is an increasing risk of violating timing. A basic command help string is available by executing (sudo) xvf_host(.exe) --list-commands. A more detailed description of the function of these commands can be seen below.

Note: The *TEST_CORE_BURN* command is hidden from --list-commands since it is used only in test situations.

I^{2}S idle times relate to far-end user DSP integration and audio manager idle times relate to output DSP integration. However, it is recommended to monitor both idle times since there is some interaction between these two tasks.

The commands in Table 4.2 provide information and control mechanisms relevant to achieving the necessary timing constraints.

Table 4.2: Control commands relevant to measuring timing

Function	Comment
I2S_CURRENT_IDLE_TIME	This command provides the current idle time of the last I^{2}S loop executed in system timer ticks of 10 ns. This reports the real-time value and should only be used as an indication of how much the processing varies. Note that the amount of idle time heavily depends on the frequency of the I^{2}S interface. For example, a 16 kHz I^{2}S interface will offer significantly more idle time than the 48 kHz setting due to having fewer samples to process.
I2S_MIN_IDLE_TIME	This command provides the minimum idle time of all I^{2}S loops executed in system timer ticks of 10 ns. This reports the worst case value since boot or since the idle time metric was last reset. It is the value that should be used to close timing when adding DSP.
I2S_RESET_MIN_IDLE_TIME	Use this control to reset the minimum idle time.
AUDIO_MGR_CURRENT _IDLE_TIME	This command provides the current idle time of the last audio manager loop executed in system timer ticks of 10 ns. This reports the real-time value and should only be used as an indication of how much the processing varies.
AUDIO_MGR_MIN _IDLE_TIME	This command provides the minimum idle time of all audio manager loops executed in system timer ticks of 10 ns. This reports the worst case value since the idle time metric was last reset. It is the value that should be used to close timing when adding DSP.
AUDIO_MGR_RESET_MIN _IDLE_TIME	Use this control to reset the minimum idle time.
SHF_BYPASS	The bypass command causes the far-field voice pipeline to be bypassed and the raw (but amplified) microphone signals to be passed to the output. In addition, it adds poll loops to burn processor cycles up to the maximum available for the voice pipeline section of the design. This means it exercises a tougher timing case compared with running the normal voice-DSP operation and so should be used when ascertaining the worst-case.
TEST_CORE_BURN	This hidden command places the firmware in 'burn' mode. Note this resets the firmware and consequently subsequent commands may be ignored while the firmware initialises. Because it performs a reset, all previously written parameters will be lost and all internal state will be set to the defaults that would be expected following power-on-reset. For this reason, it is recommended that this command be executed at the beginning of the timing-closure session.

Note: The TEST_CORE_BURN command ensures all idle cycles in other parts of the XVF3800 are used up by creating polling loops. This significantly increases core power consumption of the XVF3800 when set to 1 (enable core burn). Expect increased power consumption of up to double the normal operating value when this test mode is used.

A typical sequence of commands to ascertain worst-case timing would be:

```
# Switch on burn mode and reboot the device. Wait at least 2 seconds before the next
# command to allow reboot time.
(sudo) xvf_host(.exe) TEST_CORE_BURN 1
sleep(2)
# Bypass the voice-DSP and exercise worst case timing of the voice pipeline
(sudo) xvf_host(.exe) SHF_BYPASS 1

# Allow the application to run for a while. Try restarting I2S a few times.
for i in {0..10}; do arecord -r 48000 -f S32_LE -c 2 -d 1 /dev/null; sleep 0.5; done
```

```
# Read the I2S minimum idle time (important if doing far-end DSP)
(sudo) xvf_host(.exe) I2S_MIN_IDLE_TIME
# Read the audio manager minimum idle time (important if doing post voice-pipeline DSP)
(sudo) xvf_host(.exe) AUDIO_MGR_MIN_IDLE_TIME


# Enable the voice-DSP and reset the minimum idle times
(sudo) xvf_host(.exe) SHF_BYPASS 0
(sudo) xvf_host(.exe) I2S_RESET_MIN_IDLE_TIME 1
(sudo) xvf_host(.exe) AUDIO_MGR_RESET_MIN_IDLE_TIME 1

# Allow the application to run for a while. Try restarting I2S a few times.
for i in {0..10}; do arecord -r 48000 -f S32_LE -c 2 -d 1 /dev/null; sleep 0.5; done


# Read the I2S minimum idle time (important if doing far-end DSP)
(sudo) xvf_host(.exe) I2S_MIN_IDLE_TIME
# Read the audio manager minimum idle time (important if doing post voice-pipeline DSP)
(sudo) xvf_host(.exe) AUDIO_MGR_MIN_IDLE_TIME
```

Always use the smallest minimum idle times noted in the previous steps as a guide to how many cycles remain to ensure that the worst case has been covered. Always start and stop I^{2}S a few times (10 times is enough), if possible, as this will further exercise the timing paths. If at any point the number of available cycles becomes close to zero, then the code exceeds the time slot available, and it will be necessary to either optimise the DSP code to meet timing or reduce the amount of processing required.

4.2.2 Measuring Available Memory

When *building the software*, the makefile settings are configured to produce a memory report which is printed at the end of the build. This report is generated by the compiler to account for all statically allocated memory used by the firmware. The firmware runs on two XCORE tiles, each with its own on-chip memory and consequently its own report. The majority of user configuration when *modifying the software*, such as adding hardware configuration code and user-DSP, will take place on **tile[1]** and so this is generally the number that will normally decrease with added functionality.

> **Warning:** The compiler memory report includes the memory set aside for the RTOS heap and stacks. The compiler cannot, however, track dynamic memory allocations from these set aside areas. Changes to an RTOS task that increases its heap or stack usage, including creating new tasks, adding parameters or local variables to a function called from an RTOS task, or operations on RTOS primitives such semaphores, queues, etc., may result in running out of memory. The specific effect of running out of memory will vary. Possible effects include hardware exceptions or the firmware hanging.
>
> The `appconfTOTAL_HEAP_SIZE` symbols in app_conf.h determine the amount of memory set aside on each tile for the RTOS heap and stacks.

> **Note:** The compiler builds the entire application twice due to the need to build the FreeRTOS kernel twice, once for each tile. This means that the compiler generates two memory constraints reports. In all cases, use the **highest** number number of the two reports.

The report shown below (with irrelevant lines deleted for clarity) shows a typical report for the `application_xvf3800_intdev-lr48-lin-i2c` firmware. Note that the precise memory usage varies considerably depending on the actual build, firmware version and the code that may have been added:

```
Constraint check for tile[0]:
Memory available:       524288,   used:        6980 .  OKAY
(Stack: 356, Code: 4064, Data: 2560)
Constraints checks PASSED.
Constraint check for tile[0]:
Memory available:       524288,   used:      491112 .  OKAY
(Stack: 10068, Code: 358940, Data: 122104)
Constraints checks PASSED WITH CAVEATS.
Constraint check for tile[1]:
Memory available:       524288,   used:      487908 .  OKAY
(Stack: 8340, Code: 397580, Data: 81988)
Constraints checks PASSED WITH CAVEATS.
Constraint check for tile[1]:
Memory available:       524288,   used:        6324 .  OKAY
(Stack: 356, Code: 3520, Data: 2448)
Constraints checks PASSED
```

Ignoring the lower numbers reported, in this case Tile[0] is using 491112 of 524288 bytes available and Tile[1] is using 487908 of 524288 bytes available. Therefore the free memory available is:

- Tile[0]: 524288 - 491112 = 33176 Bytes
- Tile[1]: 524288 - 487908 = 36380 Bytes

If usage exceeds the available memory the link stage of compilation will fail and the compiler will issue a clear error. In this case, reduce the memory usage.

5 Modifying the Software

5.1 Adding a Control Command

The XVF3800 software allows for easily extensible control. Each time the firmware is built, the command definition YAML files are parsed and the firmware hooks and enums are updated automatically. See *Working With The Build System* for how to build the XVF3800 firmware.

The command definition files can be found in `sources/app_xvf3800/autogeneration/yaml_files/Control_commands`. There is a file for each control servicer within the firmware. The control servicers are:

- `aec_cmds.yaml` - This is where high level voice-DSP parameters are accessed as well as AEC information. The voice-DSP is not modifiable other than the published API. It is not expected that this file will need to be modified.

- `application_cmds.yaml` - This is where build information is accessed and some test features. The application servicer does not directly connect to any peripherals, however commands requiring internal storage or calling a user API may be added here.

- `audio_cmds.yaml` - This is where high-level aspects of the audio framework including SRC, packing, I^2S and user-DSP are accessed. If extending the DSP capabilities of the design it is likely that commands may be added here, for example to control user-DSP. See *Adding Custom Digital Signal Processing*. Note that two tasks are controlled by this servicer (Audio Manager and I^2S) with the I^2S task being accessed via a shared-memory structure.

- `dfu_cmds.yaml` - This is where DFU messages are handled and processed. It is not expected that this file will need to be modified. This servicer is only used in the INT device.

- `hid_task_cmds.yaml` - This is where internal messages to send HID IN events are handled. It is not expected that this file will need to be modified. This servicer is only used in the UA device.

- `io_config_cmds.yaml` - This is where GPIO parameters are accessed. Commands are already provided for manipulating many aspects of these pins, although any custom requirements involving GPIO access may be added here.

- `io_expander_cmds.yaml` - This is where internal messages to control the IO expander GPO's are handled. It is not expected that this file will need to be modified. This servicer is only used in the builds with the IO expander enabled.

- `pll_cmds.yaml` - This is where PLL information is accessed. Generation of the MCLK signal uses the PLL.

- `pp_cmds.yaml` - This is where high level voice-DSP parameters are accessed as well as post processing information. The voice-DSP is not modifiable other than the published API. It is not expected that this file will need to be modified.

- `shf_aec_cmds.yaml` - This is where low-level voice-DSP parameters are accessed. The voice-DSP is not modifiable other than the published API. This file is auto-generated and should not be modified.

- `shf_pp_cmds.yaml` - This is where low-level voice-DSP parameters are accessed. The voice-DSP is not modifiable other than the published API. This file is auto-generated and should not be modified.

- `usb_buffer_cmds.yaml` - This is where USB information and parameters are accessed. It only applies to UA configurations.

5.1.1 Adding a new control command

This process is illustrated by adding a simple read/write parameter via `application_cmds.yaml`. As an example of how to extend this to controlling IO, see the *FAR_END_DSP_ENABLE* parameter contained in `audio_cmds.yaml`.

First, add a command to the YAML file. The valid types that can be used for command parameters are as follows:

```
TYPE_INT32
TYPE_UINT32
TYPE_INT16
TYPE_UINT16
TYPE_INT8
TYPE_UINT8
TYPE_CHAR
TYPE_FLOAT
TYPE_RADIANS
```

Any number of these parameters may be defined in a control command, up to the total maximum command size of 64 bytes. Commands attributable to the *pp_cmds* servicer are exceptions; these are limited to 20 bytes.

The following access permissions may be assigned to parameters:

```
CMD_READ_ONLY
CMD_WRITE_ONLY
CMD_READ_WRITE
```

For write commands, a range must be provided for each value. If no value range is specified for such commands, the firmware code will fail to compile. The ranges must be listed in the *value_ranges* array and must follow one of the two formats:

1. list of intervals - each interval is listed using the syntax *[A .. B]*

    - the syntax is the same for both integers and float values

    - multiple intervals can be specified, for example *[0 .. 5, 10 .. 15]*

    - all the intervals must be closed, meaning that they include all the limit points

    - if only one value is valid, the range can be specified as *[E .. E]*

2. any value is valid - this is declared using the word *any* and the range depends on the maximum and minimum values of the specific type. For example, *TYPE_UINT8* can have values from 0 to 255.

An example of a command with two arguments, where the first requires a list of intervals and the second accepts any value, is shown below:

```
value_ranges:
    - value0: [0 .. 5, 10 .. 15]
    - value1: any
```

Note: The host control application performs range checking before sending the control command to the device and it returns an error if any argument value is out of range.

An example of adding a command to `application_cmds.yaml` is shown below. The position in the list at which the command is added is not important so long as it is in the appropriate section:

```
- cmd: MY_INTERNAL_REGISTER
  number_of_values: 1
  type: CMD_READ_WRITE
```

```
help: A simple example of setting / getting a variable in the firmware
value_type: TYPE_UINT32
```

Next, in the appropriate servicer C file, add the handlers for the command. In this case we are adding the following code to `sources/modules/fwk_xvf/modules/xvf/src/control_plane/application_servicer.c`:

```
// Global variable to get or set
uint32_t my_var = 0;
```

In the function *control_ret_t application_servicer_read_cmd()* add the following case. Note the pre-pending of the resource ID to the command name:

```
case APPLICATION_SERVICER_RESID_MY_INTERNAL_REGISTER:
    memcpy(payload, &my_var, sizeof(my_var));
    break;
```

In the function *control_ret_t application_servicer_write_cmd()* add the following case:

```
case APPLICATION_SERVICER_RESID_MY_INTERNAL_REGISTER:
    memcpy(&my_var, payload, sizeof(my_var));
    break;
```

Next, build the firmware and host app; see *Working With The Build System* for instructions on this. Test the new command:

```
(sudo) xvf_host(.exe) MY_INTERNAL_REGISTER
0
(sudo) xvf_host(.exe) MY_INTERNAL_REGISTER 1066
(sudo) xvf_host(.exe) MY_INTERNAL_REGISTER
1066
```

5.2   Adding Custom Digital Signal Processing

The XVF3800 supports the addition of custom DSP at three locations in the signal path. These points are:

- Far-end reference signal DSP within the DAC
- Far-end reference signal DSP within the XVF3800 firmware
- Post-processing of the voice signal within the XVF3800 firmware after it exits the voice pipeline.

5.2.1   I^{2}S Usage

Understanding the two options for far-end reference signal DSP requires knowledge of how the XVF3800 uses the I^{2}S signals. The purpose of each signal depends on the device configuration as shown in Table 5.1.

Table 5.1: I^{2}S usage

Signal	Integrated Device (INT)	USB Accessory (UA)
I2S_DATA0	Input from host	Output to DAC
I2S_DATA1	Output to host	Optional input from DAC
I2S_DATA2	Optional output to DAC	Not used

5.2.2 Custom DSP Within the DAC

Some Digital to Analogue Converters include optional DSP with the ability to transmit the altered signal before conversion via I^{2}S and after conversion to an amplifier or loudspeaker. For designs using such a DAC and receiving the reference audio via USB, the XVF3800 UA configuration includes an option to accept a processed far-end reference signal over an I^{2}S interface.

To enable this option, use the host control application to set the *I2S_DAC_DSP_ENABLE* control to 1 or modify the default value of the *I2S_DAC_DSP_ENABLE* control to 1. Information about using the host control application appears in the Using the Host Application section of the XVF3800 User Guide. Information about modifying the default value of a control parameter appears in the Changing Default Parameter Values section of the XVF3800 User Guide.

5.2.3 Custom DSP Within the XVF3800 Firmware

5.2.3.1 Customer Far-End Reference DSP

Custom DSP between the far-end (reference) input and the start of the far-field voice pipeline allows additional processing of the far-end signal, for instance correcting speaker/amplifier imperfections to ensure optimum AEC performance. The voice pipeline, and optionally the DAC via I^{2}S, receives a copy of the processed far-end signal.

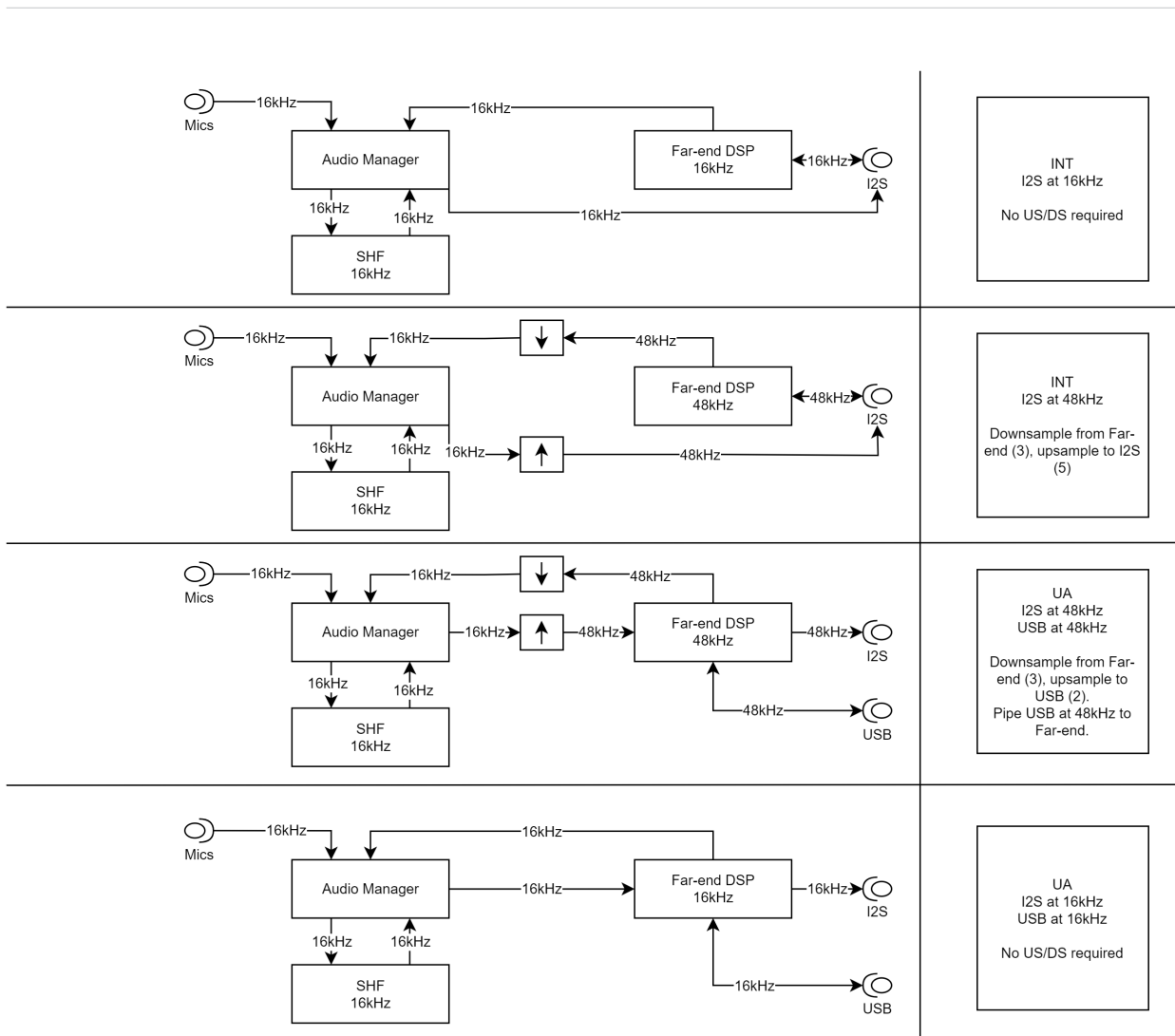Fig. 5.1 shows the audio paths for the far-end DSP as well as where up/down-sampling may occur.

Fig. 5.1: Audio paths for far-end DSP and where up/down-sampling may occur in the XVF3800.

5.2.3.2   Custom Voice Post-Processing DSP

While the standard voice processing offers a wide range of typically needed functions such as AGC, high-pass filtering and automatic beam selection, some users may wish to augment these functions. Custom DSP post-processing occurs immediately after the processed microphone signal leaves the voice pipeline and before the XVF3800 sends it to the host.

5.2.3.3   Common Aspects to Custom DSP Within the XVF3800 Firmware

Both DSP hooks provide an API in which processing of a single sample occurs for each function call. This approach reduces latency (block based algorithms introduce a minimum latency of the block size) and simplifies integration into the main firmware framework. The lib_xcore_math library provides various DSP functions including FIR filters and Biquad IIR filters. An example of the latter is given further on.

Note: Integration of user DSP consumes processing cycles from the processor. These are limited according to the build and host sample rates used. Please see *Meeting timing* for details.

The API for the firmware-based custom DSP functions is transcluded below from `sources/modules/fwk_xvf/modules/xvf/src/user_interfaces/user_dsp.h`:

```c
// Copyright 2022-2023 XMOS LIMITED.
// This Software is subject to the terms of the XCORE VocalFusion Licence.

#ifndef __USER_DSP_H_
#define __USER_DSP_H_

#include "aec_cmds.h"
#include "shf_wrapper.h"
#include <stdint.h>

/// There is a timing limit on the time spent in these functions. Please use the
/// minimum idle time control commands in conjunction with the TEST_CORE_BURN command to
/// characterise the amount of cycles available.
/// The far_end_dsp function is called from I2S and so check min_idle time for that task
/// The far_end_dsp function is called from Audio so check min_idle time for that task


/// @brief callback to pre-process one sample of the far end before outputting to DAC/SHF DSP input
/// Note that this callback runs at the I2S sample rate.
/// @param far_end_sample     input and output (sample is processed in place)
/// @param far_end_dsp_enable Set to 1 to enable, 0 to disable. This is handled by the user.
void far_end_dsp(int32_t far_end_samples[BECLEAR_NUMBER_OF_FAR], bool far_end_dsp_enable);

/// Struct passed to post_shf_dsp which contains all the information that is available
/// for additional post-processing.
typedef struct {
    /// Pointer to array containing the BECLEAR_NUMBER_OF_OUTPUTS processed microphone channels.
    int32_t* post_shf_processed_mic_samples;

    /// BECLEAR_NUMBER_OF_MICS channels containing the microphones after AEC before post-processing.
    int32_t* aec_residuals;

    /// Pointer to array of BECLEAR_NUMBER_OF_OUTPUTS azimuths, each element is the azimuth for the
    /// post_shf_processed_mic_samples of the same index. It can be NULL if azimuths have not been
    /// calculated.
    float* azimuths;

    /// The spenergy (speech energy) for each beam in post_shf_processed_mic_samples. If the spenergy
    /// is non-zero then it contains energy that is likely speech. The value will be higher for louder
    /// or closer voices, noise and distortion will cause the speech energy to decrease. This points
    /// to an array of size BECLEAR_NUMBER_OF_OUTPUTS where each value corresponds to the beam of the
    /// same index. It can be NULL if spenergy has not been calculated.
    float* spenergy;

    /// Output of xmos algorithm to determine the direction of voice, NAN if no voice detected
    float direction_of_voice;
} user_dsp_post_shf_input_t;

/// @brief callback to post-process one sample of audio after the SHF voice DSP stage
/// Note that this callback runs at the SHF sample rate.
/// @param out Array to fill with the output of this function.
/// @param input See user_dsp_post_shf_input_t comments for details.
void post_shf_dsp(int32_t out[BECLEAR_NUMBER_OF_OUTPUTS],
                  user_dsp_post_shf_input_t* input);

#define USER_DSP_NUM_OUTPUT_CHANNELS 2

/// @brief called immediately after post_shf_dsp and will be used to determine the
/// channels that MUX_USER_CHOSEN_CHANNELS will consist of. It also sets the azimuth
/// of each chosen channel so that it can be requested via control command.
///
/// @param[in,out] out_idx 2 chosen channels from `out` which were written by
```

```
///    the function `post_shf_dsp`. Before being passed to beam_selection,
///    out_idx will be populated by the suggested indices.
/// @param[out] out_azimuths azimuths of the two channels that have been
///    selected. Most likely a copy of the correct input from above.
void beam_selection(uint8_t out_idx[USER_DSP_NUM_OUTPUT_CHANNELS],
                    float out_azimuths[USER_DSP_NUM_OUTPUT_CHANNELS]);

#endif
```

5.2.4   Meeting Timing

When adding custom DSP, it is important to check timing. **It is not sufficient just to check that audio is playing cleanly** because the available cycles within the XVF3800 varies significantly depending on operation and will increase under certain conditions. A comprehensive method for checking *worst case timing* is included and can be found in the *Testing the Software section of the Programming Guide*.

5.2.5   Adding Control to Custom DSP

In some cases it may be desirable to add a new control command to allow the host application to enable/disable or adjust the custom DSP. The steps in *Adding a Control Command* show how to add controls to the firmware and the below examples include adding a control for each case.

For far-end reference DSP within the XVF3800 firmware, a built-in control named *AUDIO_MGR_FAR_END_DSP_ENABLE* has already been provided. It is possible to add further controls if needed.

5.2.6   Far-End Reference DSP

Both far-end reference DSP locations offer the opportunity to add processing between the host audio signal and the DAC output. If adding non-linear processing (eg. bass-enhancement, dynamic-range compression), placing that processing in one of these locations will avoid degradation in AEC performance. If it is located somewhere else in the system, this DSP may significantly degrade AEC performance due to the far-end reference given to the voice pipeline differing from the signal played through the loudspeaker. The remainder of this section will discuss far-end reference DSP placed within the XVF3800 firmware.

The rate of the DSP is the host interface rate. For example, if I^{2}S runs at 48 kHz, then the far-end DSP also runs at 48 kHz. The samples will not be sent to the voice pipeline until after the far-end DSP has occurred and will be down-sampled if required.

Because the far-end DSP block will add delay to the voice pipeline reference signal, it is essential that any delay added is not so large that the direct path (loudspeaker to microphone) of the far-end signal arriving at the microphones gets to the voice pipeline before the processed far-end signal does. This non-causal relationship will cause a rapid degradation in AEC performance. For more information on this topic, including how to measure this effect, see the Tuning the Application section of the XVF3800 User Guide.

Depending on the external audio path and the type of processing applied (linear vs non-linear) it may be necessary to add an additional audio line. For example, an I^{2}S connected system may need one audio line for the reference input, one audio line for the processed microphone output and an additional line for the processed far-end output to the DAC. Where the far-end audio source is USB, this additional audio line is not necessary since the processed far-end output will always be sent to the DAC pin.

The XVF3800 allows provision of a third I^{2}S line for processed far-end output using the following steps:

- Increase `appconfNUM_I2S_PINS_OUT` in `app_conf.h` from 1 to 2. This will enable PORT_I2S_DATA2 as

an I^{2}S output pin. The default is to set the second pin to output the post-processed far-end input with sample rate conversion disabled.

- Set `USE_FAR_END_DSP` in `far_end_dsp.c` to 1.
- Make sure the DAC input is connected to the processed far-end DSP signal.

Note: Far-end reference DSP is disabled by default in the standard build to avoid using extra memory and processing cycles.

In the USB build, the pin formerly used as the I^{2}S input to the device is used by default as an I^{2}S output to send the far-end signal to the DAC. Therefore, for the USB build, it is usually sufficient to keep `appconfNUM_I2S_PINS_OUT` set to 1. By default, the processed far-end signal will be routed to this output. However, the secondary I^{2}S line may still be enabled with the above steps if desired.

In order to verify the effect of the DSP, you may wish to send the product of the far-end DSP directly to the DAC without any incident down- or up-sampling. The following commands ensure up-sampling is disabled and route far-end DSP signal to the output:

```
(sudo) xvf_host(.exe) AUDIO_MGR_OP_UPSAMPLE 0 0
(sudo) xvf_host(.exe) AUDIO_MGR_OP_ALL 10 0 10 2 10 4 10 1 10 3 10 5
```

A control command has already been provided which passes a boolean to the *far_end_dsp()* function, which allows the user to enable and disable the far-end DSP and verify its functionality. This may be controlled using:

```
(sudo) xvf_host(.exe) AUDIO_MGR_FAR_END_DSP_ENABLE 1
(sudo) xvf_host(.exe) AUDIO_MGR_FAR_END_DSP_ENABLE 0
```

5.2.6.1 Far-end reference example

For example purposes, a three stage biquad filter has been implemented which boosts bass and treble by 6 dB and cuts mid-range by 6 dB. This filter produces a noticeable effect suitable for demonstration purposes and for verification that the far-end DSP is active. This example has coefficients that have been generated assuming a 48 kHz sample rate. They will not work properly at 16 kHz and will need to be re-calculated.

The code is shown below, transcluded from `sources/app_xvf3800/src/user_dsp/far_end_dsp.c`:

```c
// Copyright 2022-2023 XMOS LIMITED.
// This Software is subject to the terms of the XCORE VocalFusion Licence.

#include "user_dsp.h"

#ifndef USE_FAR_END_DSP
    #define USE_FAR_END_DSP 0
#endif

#if USE_FAR_END_DSP
#include "xmath/xmath.h"

// Simple example DSP that processes far end using an EQ. Note the coeffs are correct for 48kHz only
// Each filter_biquad_s32_t can store (up to) 8 biquad filter sections
#define SECTION_COUNT   3

filter_biquad_s32_t filter[BECLEAR_NUMBER_OF_FAR] = {{
    // Number of biquad sections in this filter block
    .biquad_count = SECTION_COUNT,

    // Filter state, initialized to 0
    .state = {{0}},
```

(continues on next page)

```
    // Filter coefficients
    // Section 0: Frequency = 100 Hz, Q Factor = +1.2,  Gain = +6.0dB
    // Section 1: Frequency = 1000 Hz, Q Factor = +0.2, Gain = -6.0dB
    // Section 2: Frequency = 8000 Hz, Q Factor = +1.3, Gain = +6.0dB
    .coef = {
        { Q30(+1.00286226693868),  Q30(+0.86561763867029),  Q30(+1.58124059575259)},
        { Q30(-1.98608850422494),  Q30(-1.44869047773446),  Q30(-1.32398889950623)},
        { Q30(+0.98346660965693),  Q30(+0.59557353208939),  Q30(+0.56062804987035)},
        { Q30(+1.98614845462853),  Q30(+1.44869047773446),  Q30(+0.45958190453639)},
        { Q30(-0.98626892619203),  Q30(-0.46119117075968),  Q30(-0.27746165065311)}
        }
}};


void far_end_dsp(int32_t far_end_samples[BECLEAR_NUMBER_OF_FAR], bool far_end_dsp_enable)
{
    // See note in user_dsp.h and user documentation about timing constraints
    if(far_end_dsp_enable)
    {
        for(int channel = 0; channel < BECLEAR_NUMBER_OF_FAR; channel++)
        {
            far_end_samples[channel] >>= 1; // Simple 6db pre-attenuate to account for gain in filter
            far_end_samples[channel] = filter_biquad_s32(&filter[channel], far_end_samples[channel]);
        }
    } else {
        for(int channel = 0; channel < BECLEAR_NUMBER_OF_FAR; channel++)
        {
            far_end_samples[channel] >>= 1; // Simple 6db attenuate to account for filter gains to
→give similar apparent volume
        }
    }
}

#else

void far_end_dsp(int32_t far_end_samples[BECLEAR_NUMBER_OF_FAR], bool far_end_dsp_enable)
{
    // Do nothing - samples unmodified
}

#endif
```

Testing the effect on available cycles, we can see a modest drop from the three stage biquad of just 85 x 10 ns = 0.85 us. This small drop is due in part to the use of the Vector Processing Unit (VPU), which is highly efficient for signal processing purposes. Note that the idle time is not calculated until I²S runs:

```
(sudo) xvf_host(.exe) TEST_CORE_BURN 1
(sudo) xvf_host(.exe) I2S_MIN_IDLE_TIME
2083
aplay <short wav>
aplay <short wav>
aplay <short wav>
(sudo) xvf_host(.exe) I2S_MIN_IDLE_TIME
1245
(sudo) xvf_host(.exe) AUDIO_MGR_FAR_END_DSP_ENABLE 1
aplay <short wav>
aplay <short wav>
aplay <short wav>
(sudo) xvf_host(.exe) I2S_MIN_IDLE_TIME
1160
```

5.2.7 Voice Post-Processing DSP

As the name suggests, this DSP hook allows the user to add any required DSP after the microphone signals have passed through the voice pipeline. The rate of the processing is always the rate of the voice pipeline (nominally 16 kHz). A number of audio signals are available including multiple output beams and AEC residuals which are the echo-cancelled only signals for each of the four microphones.

To allow more informed selection of the output beam, the Direction of Arrival (DoA) azimuths and the speech energy are also provided to allow custom logic to choose the desired signal.

Adding processing to this part of the chain will not affect the performance of the core voice pipeline. However, it will add to the total delay through the device from microphones to output interface.

Follow the same steps as per the *Far-end reference example* except when checking timing, please use *AUDIO_MGR_MIN_IDLE_TIME* instead of *I2S_MIN_IDLE_TIME* since the processing cycles are consumed from a different task. The *Testing the Software* section also provides detailed information on how to check timing.

5.2.7.1 Spatial output example

An example is provided in `post_shf_dsp.c` which uses the DoA information to pan the auto select beam onto the left and right output channel. This allows for a stereo output from the device giving audible indication of the location of the speaker. This feature can be enabled using a macro named `appconfSPATIAL` and is enabled in build configs with the `-spatial` suffix. These configs also configure the output multiplexer (mux) to play the left and right outputs back to the host correctly.

To get the best output from this example the macro `LEFT_ANGLE_RADIANS` may need to be updated so that the correct output is heard. For linear microphone arrays this will either be 0 or `M_PI` depending on the microphone geometry. After changing the software, it will need to be recompiled and flashed to the device.

5.3 Modifying Existing Functionality

The XVF3800 provides the ability to customise the initialisation code for any connected hardware at firmware boot time. During run-time, the GPIO pins may be modified via control commands from the host control application. Initialisation typically involves setting GPO pins to control board level features such as LEDs and configuring I^2C connected devices such as an IO expander, a DAC, or a digital amplifier. The code for user hardware initialisation can be found in `sources/app_xvf3800/src/user_config`. The main file to be modified is `user_config.c` which is shown below.

```c
// Copyright 2022-2023 XMOS LIMITED.
// This Software is subject to the terms of the XCORE VocalFusion Licence.

#include "FreeRTOS.h"
#include "app_conf.h"
#include "user_config.h"
#include "io_config_servicer.h"
#include "dac3101.h"

// This file contains the user hardware configuration code. It uses the implementations in dac3101.h
↪(EVK3800)
// and the lower level hardware implementations in dac_port.c

int user_init_hardware(device_control_t *device_control_gpio_ctx)
{
    int errors_encountered = 0;

    rtos_printf("user_init_hardware\n");
```

(continues on next page)

```
#if !defined(MIC_ARRAY_TYPE)
#error
#endif
#if MIC_ARRAY_TYPE == BECLEAR_LINEAR_ARRAY
    write_gpo_pin(device_control_gpio_ctx, GPO_SQ_nLIN_PIN, 0);
#elif MIC_ARRAY_TYPE == BECLEAR_CIRCULAR_ARRAY
    write_gpo_pin(device_control_gpio_ctx, GPO_SQ_nLIN_PIN, 1);
#else
    #error MIC_ARRAY_TYPE invalid
#endif

    // De-assert HOST INTERRUPT line
    write_gpo_pin(device_control_gpio_ctx, GPO_INT_N_PIN, 1); // No interrupt to host asserted when
→high

#if (appconfUSER_CONFIG_ENABLED == 1)
    // Reset the DAC
    dac3101_codec_reset(device_control_gpio_ctx);

    errors_encountered |= dac3101_init(appconfLRCLK_NOMINAL_HZ);
#endif

    // Test that we can turn on a LED by sending a command from this task to the GPO task
    // Note even though LEDs are active low, we have setup the LED pins in gpo_servicer to drive
→negaive logic
    for(int i = 0; i < 5; i++){
        write_gpo_pin(device_control_gpio_ctx, GPO_LED_GREEN_PIN, 1); // Turn the green LED on
        vTaskDelay(pdMS_TO_TICKS(100));
        write_gpo_pin(device_control_gpio_ctx, GPO_LED_GREEN_PIN, 0); // Turn the green LED off
        vTaskDelay(pdMS_TO_TICKS(100));
    }

    return errors_encountered;
}
```

This file is currently configured for the XK-VOICE-SQ66 development kit and its associated hardware set.

In this file we can see the following actions taken:

- Setting of GPO output line on the XK-VOICE-SQ66 development kit to control the microphone array topology.
- Setting of GPO output line to de-assert the host interrupt line.
- Resetting of the DAC. See the next section *Digital to Analogue Converter Configuration* for details.
- Configuring the DAC. See the next section *Digital to Analogue Converter Configuration* for details.
- Flashing the green LED five times to show that booting of the XVF3800 is occurring.

Note: The control plane part of the XVF3800 firmware, responsible for servicing control commands from the host, will not start until the call to *user_init_hardware()* is complete. Any control commands issued by the host before initialisation is complete will not be serviced.

5.3.1 Digital to Analogue Converter Configuration

Each DAC or digital amplifier selected will normally have its own set of registers that need to be configured. There are two parts to the DAC configuration code. The first is the abstraction layer responsible for providing the I^{2}C, GPO, and wait functions. These may need to be modified if the GPO responsible for resetting the DAC or the I^{2}C register access methods needs to be altered. This file can be seen below, transcluded from `sources/app_xvf3800src/user_config/dac_port.c`:

```
// Copyright 2022-2023 XMOS LIMITED.
// This Software is subject to the terms of the XCORE VocalFusion Licence.

// This file contains the implementations of the DAC configuration steps such as which registers to
↪write with which values
// Note there are currently two implementations because in I2C slave we init the DAC pre-RTOS

/* FreeRTOS headers */
#include "FreeRTOS.h"

/* App headers */
#include "xcore/port.h"
#include "rtos_i2c_master.h" // Includes "i2c.h" too
#include "platform/driver_instances.h"
#include "io_config_servicer.h"
#include "user_config.h"
#include "dac3101.h"


void dac3101_wait(uint32_t wait_ms)
{
    vTaskDelay(pdMS_TO_TICKS(wait_ms));
}

#if (appconfUSER_CONFIG_ENABLED == 1) // Some builds use a specific control transport but DO NOT
↪require setting up of DAC HW

int dac3101_reg_write(uint8_t reg, uint8_t val)
{
    rtos_i2c_master_t * i2c_master_ctx = get_i2c_master_ctx();
    i2c_regop_res_t ret = rtos_i2c_master_reg_write(i2c_master_ctx, DAC3101_I2C_DEVICE_ADDR, reg,
↪val);

    if (ret == I2C_REGOP_SUCCESS) {
        return 0;
    } else {
        return -1;
    }
}


void dac3101_codec_reset(void * args)
{
    device_control_t *device_control_gpio_ctx = args;

    write_gpo_pin(device_control_gpio_ctx, GPO_DAC_RST_N_PIN, 0);
    dac3101_wait(1);  /* From DS - The hardware reset pin (RESET) must be pulled low for at least 10ns
↪*/
    write_gpo_pin(device_control_gpio_ctx, GPO_DAC_RST_N_PIN, 1);
    dac3101_wait(1); /* From DS - This initialization takes place within 1 ms after pulling the RESET
↪signal high */
}
#endif
```

The second file, which specifies the sequence of GPO accesses and I^{2}C register writes specific to the chosen DAC, can be found in `sources/modules/fwk_xvf/modules/bsp/dac`. There are two files which may need to be modified: `dac3101.h`, which contains the defines and function prototypes, and `dac3101.c`, which contains the *dac3101_init()* function that is called from `user_config.c` and performs the sequence of operations required to configure the DAC. These sources are not printed here for documentation brevity.

Note: Error detection is included and errors (non-zero return) will be reported back to the application if encountered.

5.3.2 General Purpose Input and Output Operation

Several GPIO ports are provided by the XVF3800 to allow input and output capability. These may be accessed from the firmware at startup via `user_config.c` or by the host application using GPO and GPI commands. The XVF3800 ports contain a single direction register and therefore groups of pins on a single port are all either input or output. The firmware provides functionality to address individual pins within a port.

GPI ports provide the capability to read the current state of pins, invert their logic, and capture an edge (event). GPO ports provide the ability to output a logic level, autonomously flash a 32b serial pattern, or provide a PWM signal suitable for dimming LEDs.

The initial configuration of the roles of each GPO pin can be found in the function *init_gpo()* in `sources/ modules/fwk_xvf/modules/xvf/src/control_plane/gpo_servicer.c`. This contains the GPO setup for the XVF3800 demonstration board including initial level and drive invert. Drive invert can be useful for negative logic hardware such as LEDs connected between the 3v3 rail and the GPO pin.

Note: Because GPO pins support PWM, setting the duty to 100% or 0% is the same as setting a 1 or 0. The *write_gpo_pin()* function hides this functionality by providing a simple logic write; however, the initialisation section in `gpo_servicer.c` initialises a PWM value of 0 or 100. Additionally, the flash mask is set to 0xffffffff so that there is no flash sequence enabled.

Individual bit defines for the individual pins in the default firmware can be found in `sources/app_xvf3800/ src/app_conf.h`.

Once the pin roles and initial values have been configured, they may be accessed using a simple API providing logic level access. An example is shown in the code listing in *Modifying Existing Functionality* which asserts GPO pins during the DAC setup.

Note: In the XVF3800 firmware, the GPI servicer is currently only defined on tile[1]; therefore, ports associated with tile[0] cannot be used as GPI pins without first modifying the firmware to instantiate a GPI servicer on tile[0]. Such a change will require significant alteration of the existing functionality. Consequently, it is not possible to easily use ports on tile[0] as GPI sources within the application. However, depending on desired functionality, it may be possible to use certain ports *outside* of the application layer, for example during boot as sense pins for a custom flash loader as described in the XMOS XTC Tools User Guide.

5.3.3   USB configuration

In the XVF3800-UA device, several settings related to the USB interfaces can be configured. The USB audio sample rate can be configured using the appropriate build configuration as described in the Building the Application section of the XVF3800 User Guide. The remaining USB settings can be updated using the `usb_param_values.yaml` in `sources/app_xvf3800/autogeneration/yaml_files/settings_and_defaults/`. This file contains additional configurable parameters used in the USB descriptors, such as `vendor ID` and `product ID`, and the default data bit depths of the input and output audio. The full list of parameters and their default values are below:

```
VENDOR_ID: 0x20B1
PRODUCT_ID_IO_16KHZ: 0x4F01
PRODUCT_ID_IO_32KHZ: 0x0000
PRODUCT_ID_IO_48KHZ: 0x4F00
MANUFACTURER_STR: "XMOS"
PRODUCT_STR: "XVF3800 Voice Processor"
SERIAL_NUMBER_STR: "000000"
CONTROL_INTERFACE_STR: "XMOS Control"
HID_INTERFACE_STR: "XMOS HID"
DFU_FACTORY_INTERFACE_STR: "XMOS DFU Factory"
DFU_UPGRADE_INTERFACE_STR: "XMOS DFU Upgrade"
DEFAULT_BIT_DEPTH_IN: "16"
DEFAULT_BIT_DEPTH_OUT: "16"
```

5.3.4   Programmatically Rebooting the Device

In both the XVF3800-UA and XVF3800-INT devices, the function `reboot_xvf3800(unsigned delay)` is provided in `tile_common.h` to facilitate rebooting the device from within the application. It is declared as

```
/// @brief reboot the package that called this function after delay ms
/// @param delay a count in ms that will wait until the reboot happens
void reboot_xvf3800(unsigned delay);
```

This will enable the watchdog timer and instruct it to operate after `delay` milliseconds. The function will return when complete, but the reboot will have been scheduled as requested. For more information on the operation of the watchdog timer, see the datasheet for the relevant xcore.ai package.

The device may also be rebooted by use of the `RST_N` pin. Pulling this pin low externally will force a hardware reboot of the device. For more information, including the pin number of the `RST_N` pin on the relevant package, see the Device Datasheet included in the XVF3800 documentation set.

5.3.5   Modifying the HID to GPIO mapping

The `init_hid_button_config` and `init_hid_led_config` functions in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c` can be modified to change the mapping between the HID buttons/LEDs and the GPIO buttons/LEDs. For more details about the HID design, refer to *HID Interface design*

5.3.5.1 Changing button mapping

By default, the code in `init_hid_button_config` maps the HID Mute button to the button on the XK-VOICE-SQ66 development kit.

```
    hid_button_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = BUTTON_MUTE_OFFSET,
↪.size = 1, .gpi_source = GPI_SOURCE_EVK, .gpi_pin_index = EVK_BUTTON_INDEX, .button_type = BUTTON_
↪TYPE_OSC, .button_press_precondition=HOOKSWITCH_BUTTON};
    hid_button_config[MUTE_BUTTON] = config;
```

This mapping can be changed subject to some contraints:

- the Dialpad buttons are not supported.
- the Teams button is not supported.

Any of the other remaining buttons, HookSwitch, Flash, Redial, Volume Increment and Volume Decrement, can be mapped to the EVK button. To do that, change the gpi_source and gpi_pin_index to BUTTON_GPI_UNMAPPED for the Mute button.

```
{
  hid_button_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = BUTTON_MUTE_OFFSET, .
↪size = 1, .gpi_source = BUTTON_GPI_UNMAPPED, .gpi_pin_index = BUTTON_GPI_UNMAPPED, .button_type =
↪BUTTON_TYPE_OSC, .button_press_precondition=HOOKSWITCH_BUTTON};
  hid_button_config[MUTE_BUTTON] = config;
}
```

Then, for the HID button that needs to be mapped to the EVK button, change the gpi_source to GPI_SOURCE_EVK and change the gpi_pin_index to EVK_BUTTON_INDEX. Below is an example of mapping the Volume Increment button to the EVK button.

```
{
  hid_button_config_t config = {.report_id = REPORT_ID_VOLUME_BUTTONS, .offset = BUTTON_VOL_UP_
↪OFFSET, .size = 1, .gpi_source = GPI_SOURCE_EVK, .gpi_pin_index = EVK_BUTTON_INDEX, .button_type =
↪BUTTON_TYPE_RTC, .button_press_precondition = NO_BUTTON_PRESS_PRECONDITION};
  hid_button_config[VOLUME_UP_BUTTON] = config;
}
```

Note: This code change needs to be made for the code that is not in the `#if (IO_EXPANDER_ENABLED)` define block.

5.3.5.2 Changing LED mapping

By default, the code in `init_hid_led_config` maps the HID Off-Hook LED to the Green LED on the XK-VOICE-SQ66 development kit and the HID Mute LED to the Red LED on the XK-VOICE-SQ66 development kit.

```
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_OFFHOOK_OFFSET, .
↪gpo_source = GPO_SOURCE_EVK, .gpo_pin_index = EVK_LED_GREEN, .notify_hid_task = true, .trigger_hid_
↪input_index = HOOKSWITCH_BUTTON, .led_mode=LED_MODE_STEADY};
    hid_led_config[OFFHOOK_LED] = config;
```

```
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_MUTE_OFFSET, .gpo_
↪source = GPO_SOURCE_EVK, .gpo_pin_index = EVK_LED_RED, .notify_hid_task = false, .trigger_hid_
↪input_index = NO_HID_IN_TRIGGER, .led_mode=LED_MODE_STEADY};
    hid_led_config[MUTE_LED] = config;
```

This can be changed and any of the remaining HID LEDs, the Ring and Hold LED, can be mapped to the XK-VOICE-SQ66 development kit LEDs. To make the change, first make sure that the existing mapping is

removed. For example, when mapping the Green LED to something else, make sure that the existing Off-Hook to Green LED mapping is removed. This is done by changing the gpo_source and gpo_pin_index to LED_GPO_UNMAPPED for the OffHook LED.

```
{
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_OFFHOOK_OFFSET, .
→gpo_source = LED_GPO_UNMAPPED, .gpo_pin_index = LED_GPO_UNMAPPED, .notify_hid_task = true, .
→trigger_hid_input_index = HOOKSWITCH_BUTTON, .led_mode=LED_MODE_STEADY};
    hid_led_config[OFFHOOK_LED] = config;
}
```

Then, for the HID LED that needs to be mapped to the Green XK-VOICE-SQ66 development kit LED, set the .gpo_source as GPO_SOURCE_EVK and gpo_pin_index as EVK_LED_GREEN. For example, if mapping the Ring LED to the Green LED, change

```
{
    hid_led_config_t config = {.report_id = REPORT_ID_MISC_BUTTONS, .offset = LED_RING_OFFSET, .gpo_
→source = GPO_SOURCE_EVK, .gpo_pin_index = EVK_LED_GREEN, .notify_hid_task = true, .trigger_hid_
→input_index = NO_HID_IN_TRIGGER, .led_mode=LED_MODE_FAST_FLASH};
    hid_led_config[RING_LED] = config;
}
```

Note: This code change needs to be made for the code that is not in the #if (IO_EXPANDER_ENABLED) define block.

5.3.6 Modifying the HID to GPIO mapping for the IO expander build

Similar to *Modifying the HID to GPIO mapping* the GPIO to HID events mapping can be changed by modifying the code in the init_hid_button_config and init_hid_led_config functions in sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/hid_init.c. For the IO expander build (application_xvf3800_ua-io48-lin-io-exp), the code within #if (IO_EXPANDER_ENABLED) needs to be modified.

The GPI button is defined by the gpi_source and gpi_pin_index fields in the hid_button_config_t structure. To change the GPI button mapped to a given HID button, change the gpi_source and gpi_pin_index fields in the initialisation code for that button in the init_hid_button_config function. The available GPI sources and pin indexes for every source are defined in sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/usb_hid.h

```
/// @brief Buttons sources. The EVK and the IO expander board
typedef enum
{
    GPI_SOURCE_EVK = 0,
    GPI_SOURCE_IO_EXP
}all_gpi_sources_t;

/// @brief Button indexes for the buttons on the EVK.
typedef enum
{
    EVK_BUTTON_INDEX = 0,
    TOTAL_EVK_BUTTONS
}all_evk_buttons_t;

/// @brief Button indexes for the buttons on the IO expander
typedef enum
{
    IO_EXP_MUTE_BUTTON_INDEX = 0,
```

```
    IO_EXP_VOL_UP_BUTTON_INDEX,
    IO_EXP_VOL_DN_BUTTON_INDEX,
    IO_EXP_ACTION_BUTTON_INDEX,
    TOTAL_IO_EXP_BUTTONS
}all_io_exp_buttons_t;
```

The GPO LED is defined by the `gpo_source` and `gpo_pin_index` fields in the `hid_led_config_t` structure. To change the GPO LED mapped to a given HID LED, change the `gpo_source` and `gpo_pin_index` fields in the initialisation code for that LED in `init_hid_led_config` function. The available GPO sources and pin indexes for every source are defined in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/usb_hid.h`

```
/// @brief LED sources. The EVK and the IO expander board
typedef enum
{
    GPO_SOURCE_EVK = 0,
    GPO_SOURCE_IO_EXP
}all_gpo_sources_t;

/// @brief LEDs on the EVK board
typedef enum
{
    EVK_LED_GREEN,
    EVK_LED_RED,
    TOTAL_EVK_LEDS
}all_evk_leds_t;

/// @brief LEDs on the IO Expander board
typedef enum
{
    IO_EXP_PCAL6416A_LED, // Red LED on the IO expander
    IO_EXP_IS31FL3193_RGB_LED,  // IS31FL3193 RGB LED on the IO expander
    TOTAL_IO_EXP_LEDS
}all_io_exp_leds_t;
```

5.3.7   Adding a different I^{2}C Expander

The HID + I^{2}C expander design described in *System Design* mentions the `io_expander_task` that is responsible for responding to buttons and driving LEDs on the I^{2}C expander. The existing `io_expander_task` is written for supporting the PCAL6416A I^{2}C expander and requires modifications when supporting a different I^{2}C expander.

In addition, the definitions for the buttons and LEDs supported on the I^{2}C expander that get exposed to the HID tasks will need to be modified. Section *Defining available GPIO on the IO expander* describes this, followed by *Modifying the IO expander task* which describes the changes required to the `io_expander_task` for supporting a different I^{2}C exapnder.

5.3.7.1 Defining available GPIO on the IO expander

The (`gpi_source`, `gpi_pin_index`) fields in the `hid_button_config_t` structure and the (`gpo_source`, `gpo_pin_index`) fields in the `hid_led_config_t` structure define the GPIO buttons/LEDs that are mapped to HID buttons/LEDs. The available GPIO sources and the buttons/LEDs supported per source are defined as enums in `sources/modules/fwk_xvf/modules/xvf/src/usb/control_plane/usb_hid.h`.

```
/// @brief Buttons sources. The EVK and the IO expander board
typedef enum
{
    GPI_SOURCE_EVK = 0,
    GPI_SOURCE_IO_EXP
}all_gpi_sources_t;

/// @brief Button indexes for the buttons on the EVK.
typedef enum
{
    EVK_BUTTON_INDEX = 0,
    TOTAL_EVK_BUTTONS
}all_evk_buttons_t;

/// @brief Button indexes for the buttons on the IO expander
typedef enum
{
    IO_EXP_MUTE_BUTTON_INDEX = 0,
    IO_EXP_VOL_UP_BUTTON_INDEX,
    IO_EXP_VOL_DN_BUTTON_INDEX,
    IO_EXP_ACTION_BUTTON_INDEX,
    TOTAL_IO_EXP_BUTTONS
}all_io_exp_buttons_t;

/// @brief LED sources. The EVK and the IO expander board
typedef enum
{
    GPO_SOURCE_EVK = 0,
    GPO_SOURCE_IO_EXP
}all_gpo_sources_t;

/// @brief LEDs on the EVK board
typedef enum
{
    EVK_LED_GREEN,
    EVK_LED_RED,
    TOTAL_EVK_LEDS
}all_evk_leds_t;

/// @brief LEDs on the IO Expander board
typedef enum
{
    IO_EXP_PCAL6416A_LED, // Red LED on the IO expander
    IO_EXP_IS31FL3193_RGB_LED,  // IS31FL3193 RGB LED on the IO expander
    TOTAL_IO_EXP_LEDS
}all_io_exp_leds_t;
```

These enums are used in the `init_hid_button_config` and `init_hid_led_config` functions when initialising the `hid_button_config_t` and `hid_led_config_t` structures for all HID buttons and LEDs.

When replacing the existing I^2C expander with a different one, change the `all_io_exp_buttons_t` and `all_io_exp_leds_t` enums to define the available GPIO buttons and LEDs, and change the `init_hid_button_config` and `init_hid_led_config` functions to map the HID buttons/LEDs to these new ones.

5.3.7.2 Modifying the IO expander task

This section describes the modifications required in the `io_expander_task` to support a different I^{2}C expander. It walks through the `io_expander_task` code describing the purpose of each bit and the changes required in it when supporting a different I^{2}C expander.

1. Initialise the IO expander registers. At the start of the `io_expander_task` there is code for initialising the I^{2}C expander registers.

```
res = rtos_i2c_master_reg_write(i2c_master_ctx, addr_ioexp, 0x2, 0x00); // drive mute led and dac_rst
↪low
res |= rtos_i2c_master_reg_write(i2c_master_ctx, addr_ioexp, 0x6, ~(int8_t)0b10010000); // all input
↪except mic_off and dac_rst
res |= rtos_i2c_master_reg_write(i2c_master_ctx, addr_ioexp, 0x7, ~0x00); // all input
res |= rtos_i2c_master_reg_write(i2c_master_ctx, addr_ioexp, 0x44, 0x0f); // Latching on bits 0..3
res |= rtos_i2c_master_reg_write(i2c_master_ctx, addr_ioexp, 0x45, 0x00); // No latching
```

When using a different I^{2}C expander the register initialisation code will need to change accordingly.

2. Call the `init_io_exp_gpo` function to initialise the GPO LED states and initialise the `io_exp_gpo_config` structure. The `io_exp_gpo_config` structure is of type `io_exp_gpo_config_t` and is defined to contain the information for mapping from the LED index exposed to the HID task (`all_io_exp_leds_t`) to the actual LED on the I^{2}C expander. Since the two LEDs added via the I^{2}C expander in the current design are completely different and reside on different I^{2}C addresses, the `io_exp_gpo_config_t` doesn't contain any information other than the location of the LED. Instead, the code in `io_exp_drive_leds` which is the function responsible to drive the LEDs does so by executing different pieces of code selected based on the LED port.

```
if(led_port == IO_EXP_GPO_PORT_PCAL6416A)   // Red LED on the PCAL6416A
{
    // Program the LED on PCAL6416A
}
else if(led_port == IO_EXP_GPO_PORT_IS31FL3193) // RGB LED on the IS31FL3193
{
    // Program the LED on the IS31FL3193
}
```

The GPO LED states initialisation and the structure and initialisation of the `io_exp_gpo_config_t` structure will change when using a different I^{2}C expander.

3. Initialise the `io_exp_gpi_info` structure. This structure is of type `io_exp_gpi_info_t` which contains the information required to map from the I^{2}C expander buttons exposed to the HID tasks (`all_io_exp_buttons_t`) to the actual GPI buttons on the IO expander. It currently contains the GPI pin index in the IO expander Input Port register 00h for the `all_io_exp_buttons_t` buttons and is initialised as follows:

```
io_exp_gpi_info_t io_exp_gpi_info[TOTAL_IO_EXP_BUTTONS] = {
    {.pin = IO_EXP_MUTE_BUTTON_PIN, ._previous_event_time = 0},   // IO_EXP_MUTE_BUTTON_INDEX
    {.pin = IO_EXP_VOL_UP_BUTTON_PIN, ._previous_event_time = 0}, // IO_EXP_VOL_UP_BUTTON_INDEX
    {.pin = IO_EXP_VOL_DN_BUTTON_PIN, ._previous_event_time = 0}, // IO_EXP_VOL_DN_BUTTON_INDEX
    {.pin = IO_EXP_ACTION_BUTTON_PIN, ._previous_event_time = 0}, // IO_EXP_ACTION_BUTTON_INDEX
};
```

The `io_exp_gpi_info_t` structure and its initialisation will change when using a different I^{2}C expander.

4. Create the `io_expander_gpo_servicer` task:

```
// Create task for receiving internal GPO commands from tud_hid_set_report_cb()
xTaskCreate((TaskFunction_t) io_expander_gpo_servicer,
                "io_expander_gpo_task",
                portTASK_STACK_DEPTH(io_expander_gpo_servicer),
                &io_exp_gpo_state,
```

```
                      appconfTEST_TASK_PRIORITY,
                      NULL);
```

The `io_expander_gpo_servicer` responds to the LED control commands from the HID task. The `io_exp_gpo_state` is shared between the `io_expander_gpo_servicer` and the `io_expander_task`. `io_expander_gpo_servicer` receives the `IO_EXPANDER_SERVICER_RESID_INTERNAL_GPO_LED_STATE` control command from `tud_hid_set_report_cb` -> `handle_hid_output_report_bit_change` -> `send_led_command` path. The `IO_EXPANDER_SERVICER_RESID_INTERNAL_GPO_LED_STATE` contains the LED index (`all_io_exp_leds_t`) and LED state (`e_led_mode_t`) that a given LED needs to be set to. The `io_expander_gpo_servicer` updates the `gpo_state->led_state[led_index].led_mode` and `gpo_state->led_state[led_index].counter`. Since the `io_expander_gpo_servicer` doesn't actually program the LEDs on the I^{2}C expander, it shouldn't need to change when modifying the code for a different I^{2}C expander.

5. In a timer driven loop, for every button, read its state and if changed from the previous read, send a `HID_TASK_RESID_INTERNAL_BUTTON_PRESS` command to the `hid_in_servicer` notifying button state change.

```
for(;;){

    // Read the GPI pins logic levels from the input port register 00h over I2C master

    // For every pin with a state change, populate the button_info structure

    //Call send_write_cmd_to_servicer() and send a HID_TASK_RESID_INTERNAL_BUTTON_PRESS command to hid_
→in_servicer over the device_control context.

    vTaskDelay(pdMS_TO_TICKS(IO_EXP_POLL_TIME_MS));
}
```

The code for reading the GPI pin logic level and deducing the button state is I^{2}C expander specific and will change when using a different one. Once the `button_info` structure is populated, the `send_write_cmd_to_servicer` call to send it via a control command to the HID task will remain the same.

6. Configure any LED states that need changing. This in done in the `io_exp_drive_leds` function that is called at the end of the timer driven loop described above.

```
for(;;){

    io_exp_drive_leds(&io_exp_gpo_state, io_exp_gpo_config, i2c_master_ctx);

    vTaskDelay(pdMS_TO_TICKS(IO_EXP_POLL_TIME_MS));
}
```

The `io_exp_drive_leds` function reads the LED modes from the shared `io_exp_gpo_state` structure that the `io_expander_gpo_servicer` updates and configures the LED registers over the I^{2}C interface accordingly. The `io_exp_drive_leds` function will need to be modified when using a different I^{2}C expander.