XVF3610 Voice Processor - User Guide

Release: 5.7.2

Publication Date: 2023/01/23

XMOS

Table of Contents

1 Introduction **2**
- 1.1 Overview . . . 2
- 1.2 Audio processing . . . 4
- 1.3 System Interfaces . . . 4
- 1.4 Booting and Initial configuration . . . 4
- 1.5 Default operation . . . 5

2 Audio Processing Pipeline **6**
- 2.1 Signal flow and processing . . . 6
- 2.2 Signal Routing and Scaling . . . 7
  - 2.2.1 Routing commands . . . 7
  - 2.2.2 Destinations . . . 8
  - 2.2.3 Sources . . . 8
  - 2.2.4 Example Routing Commands . . . 9
  - 2.2.5 PACKED_ALL signals . . . 10
- 2.3 General Purpose Filter . . . 11
- 2.4 PDM microphone interface . . . 13
- 2.5 Automatic Echo Cancellation (AEC) . . . 15
- 2.6 Automatic Delay Estimation Control (ADEC) . . . 17
- 2.7 Interference canceller . . . 20
- 2.8 Noise Suppressor (NS) . . . 21
- 2.9 Automatic Gain Control (AGC) and Loss Control . . . 22
- 2.10 Alternative Architecture mode (ALT_ARCH) . . . 24

3 System Interfaces **29**
- 3.1 General Purpose Input and Output and Peripheral Bridging . . . 29
- 3.2 GPIO . . . 30
- 3.3 General Purpose Inputs . . . 30
- 3.4 General Purpose Outputs . . . 32
- 3.5 I^2C Master peripheral interface (XVF3610-UA Only) . . . 34
- 3.6 I^2C Slave Control interface (XVF3610-INT only) . . . 38
- 3.7 Using I^2C Master to write to a device . . . 38
- 3.8 Using the I^2C master to read from a device . . . 38
- 3.9 SPI Master . . . 39

4 System Boot and Initial Configuration **42**
- 4.1 Boot process . . . 42
- 4.2 Flash storage structure . . . 42
- 4.3 Programming the Factory Boot image and Data Partition . . . 43
- 4.4 Upgrade Images and Data Partitions . . . 44
- 4.5 Generation of Binary Upgrade image . . . 44
- 4.6 Addition of DFU Suffix to Binary files . . . 45
- 4.7 Performing Firmware Updates . . . 46
- 4.8 Factory restore . . . 47
- 4.9 Boot Image and Data Partition Compatibility checks . . . 47
- 4.10 Custom flash memory devices . . . 48
  - 4.10.1 Custom flash definition for factory programming . . . 48
  - 4.10.2 Custom flash definition for Data Partition generation . . . 48

4.11 SPI Slave Boot . . . 48
4.11.1 SPI Boot of XVF3610-INT . . . 49
4.11.2 SPI Boot of XVF3610-UA . . . 49
4.11.3 Implementing a SPI Boot host application . . . 50
4.12 Configuration and the Data Partition . . . 52
4.12.1 Data Partition file structure . . . 52
4.12.2 Item files . . . 53
4.12.3 Generating a Data Partition for custom applications . . . 53

5 Device operation **55**
5.1 Host Utilities . . . 55
5.1.1 Building the host utilities from source code . . . 55
5.2 Command-line interface (vfctrl) . . . 56
5.3 vfctrl Installation . . . 56
5.4 vfctrl syntax . . . 57
5.5 Configuration via Control interface . . . 58
5.5.1 Control operation . . . 59
5.5.2 Host Application . . . 59
5.5.3 Device Application . . . 59
5.6 Configuration via Data Partition . . . 59

6 USB Interface - (XVF3610-UA and XVF3610-UA-HYBRID only) **61**
6.1 USB Interface . . . 61
6.2 USB Configuration . . . 61
6.3 USB HID interface . . . 62
6.4 HID Report configuration . . . 62
6.5 USB HID report format . . . 63
6.6 HID report generation . . . 64
6.7 Serial Number . . . 66
6.8 USB device enumeration . . . 66

7 Reference information **67**
7.1 Base vfctrl command list . . . 67
7.2 Advanced vfctrl command list . . . 68
7.3 Boot status codes (RUN_STATUS) . . . 74
7.4 Example .SPISPEC file format . . . 76
7.5 USB enumeration . . . 76
7.6 General purpose filter example . . . 78
7.6.1 Specification . . . 78
7.6.2 Worked Example . . . 78
7.7 Command transport protocol . . . 79
7.7.1 Transport protocol for control parameters . . . 79
7.7.2 Transporting control parameters over I^2C . . . 80
7.7.3 Transporting control parameters over USB . . . 80
7.7.4 Floating point to fixed point (Q format) conversion . . . 80
7.8 Flash programming and update flow . . . 81
7.9 Capturing packed samples . . . 82
7.9.1 Capturing all pipeline input and output signals over a 48kHz USB interface . . . 82
7.9.2 Capturing all pipeline input and output signals over a 48kHz I^2S interface . . . 86
7.9.3 Packing specific signals . . . 87
7.10 Direct access to DSP Pipeline . . . 87
7.10.1 Injecting a 4-channel, 16kHz test vector into the DSP pipeline over USB . . . 87
7.10.2 Injecting a 4-channel, 16kHz test vector into the DSP pipeline over I^2S . . . 89
7.10.3 Injecting a 4-channel packed input and capturing a 6-channel packed output . . . 91

Steps for XVF3610-UA 92
Steps for XVF3610-UA-HYBRID 92
Steps for XVF3610-INT 93

The XMOS VocalFusion® XVF3610 User Guide is written for system architects and engineers designing Far-field voice systems using the XVF3610 voice processor. The document describes typical usage models, the processor architecture, key feature operation, and interface definitions. In conjunction with the product datasheet, these two documents provide all the information required for system design, from concept to production testing and verification.

It is expected that this document is read in conjunction with the relevant datasheet and that the user is familiar with basic voice processing terminology.

Note: This issue of the user guide covers the functionality supported by version 5.7 and release 5.7.2 of the VocalFusion® XVF3610 application firmware.

1 Introduction

The XMOS VocalFusion® XVF3610 voice processor uses microphone array processing to capture clear, high-quality audio from anywhere in the room. XVF3610 processors use highly optimised digital signal processing algorithms to implement 'barge-in', suppress point noise sources and reduce ambient noise levels increasing the effective Signal to Noise Ratio (SNR) to achieve a reliable voice interface whatever the environment.

1.1 Overview

The processor is designed for seamless integration into consumer electronic products requiring voice interfaces for Automatic Speech Recognition (ASR), or communication and conferencing. In addition to the class-leading voice processing, XVF3610 processor implements specific features and interfaces required for use in closely integrated applications such and incorporated into a TV or set-top box.

Three variants of the XVF3610 are available:

- XVF3610-UA (**U**-SB **A**-ccessory) - Audio and control via a USB2.0 interface
- XVF3610-UA-HYBRID (**U**-SB **A**-ccessory **HYBRID**) - Audio from the host via I^{2}S and audio to the host, and control via a USB2.0 interface
- XVF3610-INT (**INT**-egrated) - Audio via I^{2}S and control over I^{2}C interfaces

The functional block diagram of the XVF3610 is shown in the figures below:

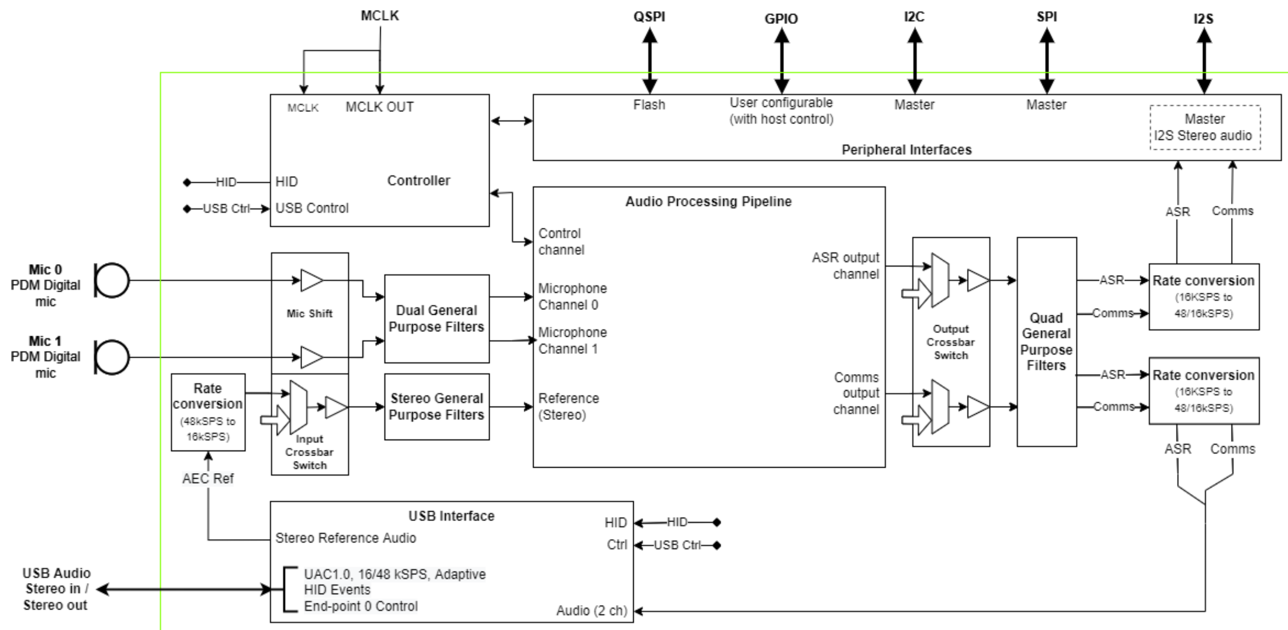

Fig. 1.1: Functional block diagram of XVF3610 in UA and UA-HYBRID configurations

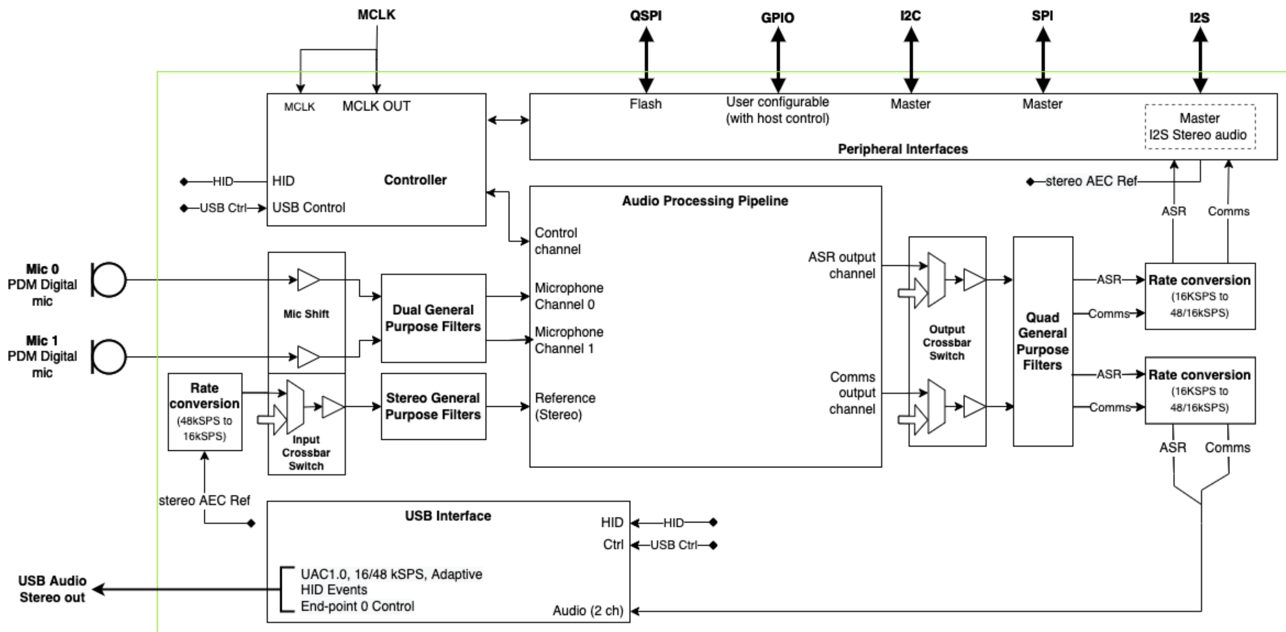

Fig. 1.2: Functional block diagram of XVF3610 in UA-HYBRID configuration

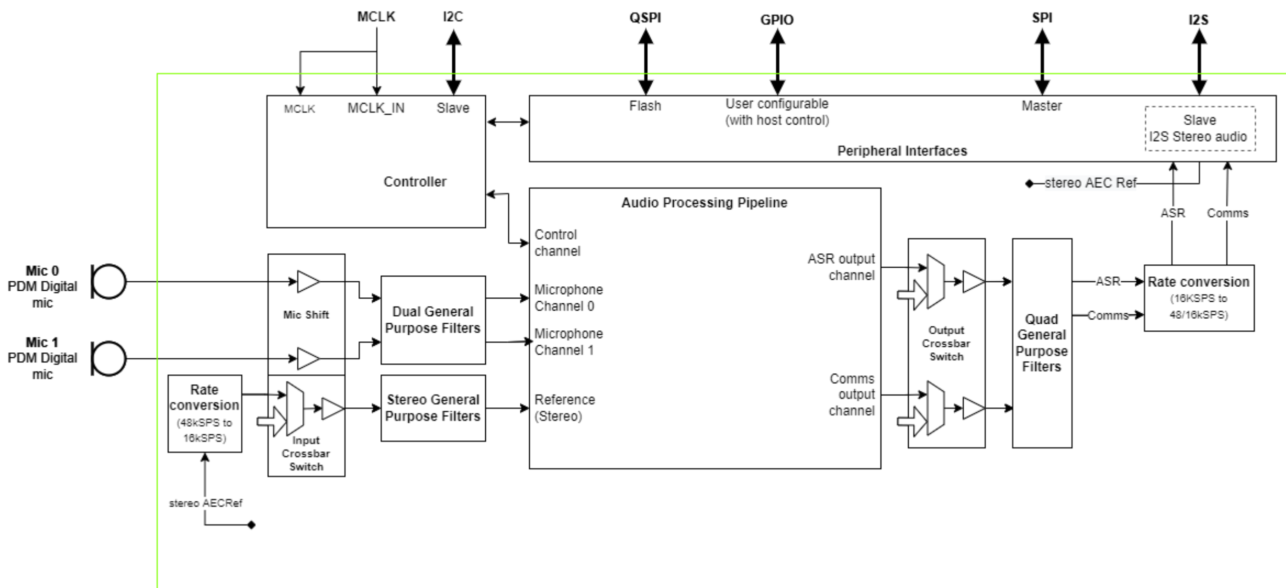

Fig. 1.3: Functional block diagram of XVF3610 in INT configuration

1.2 Audio processing

The VocalFusion® XVF3610 voice processor converts and enhances audio captured using a pair of low-cost digital microphones. Processed audio streams are suitable for use in Automatic Speech Recognition (ASR) or voice communications applications and benefit from a range of configurable audio processing techniques to allow customisation to the use case. The embedded audio processing provides the following features:

- 2 microphone far-field operation.
- Full 360-degree operation in “coffee table” applications or 180-degree for operation in edge-of-room products such as smart TVs.
- 16kHz voice processing, with optional 16kHz and 48kHz interface sample rates.
- Full duplex, Stereo, Acoustic Echo cancellation with a maximum tail length of 225ms accommodating highly reverberant environments. The Reference audio for cancellation can be provided via an I^2S Slave interface (INT variant), or I^2S Master interface (UA-HYBRID variant) or via USB (UA variant).
- Automatic bulk delay insertion, of up to 150ms, to account for positive or negative reference audio delays ensuring optimal echo cancellation with all audio output paths.
- Cancellation of point noise sources via a 256-frequency band Interference Canceller.
- Switchable stationary noise suppressor.
- Adjustable gain over a 60dB range with automatic gain control.
- Audio output filtering and range limiter.
- Independent audio processing paths and control of parameters for communications and ASR audio.

1.3 System Interfaces

The VocalFusion® XVF3610 voice processor provides the following additional interfaces to increase usability and reduce total system cost:

- 4 General Purpose Output pins. These can be configured as simple digital I/O pins, Pulse Width Modulated (PWM) outputs and rate adjustable LED flashers.
- 4 General Purpose Input pins. These can be used as simple logic inputs or event capture (edge detection).
- SPI master interface to control and interrogate an SPI slave device, such as ADCs, DACs or external keyword detection devices.

1.4 Booting and Initial configuration

The VocalFusion® XVF3610 voice processor can be booted over SPI by a local host processor or from a separate, user-supplied, QSPI Flash memory. When operating with flash, the memory can be used for the following functions:

- A default firmware image for power-on operation.
- An upgrade image. Upgrades are provided via I^2C or USB providing a host-controlled upgrade process for over-the-air device management.
- A persistent user information space to allow user-configured data such as board identifiers and serial numbers to be maintained across multiple firmware upgrade cycles.

- An upgradable user command space. Commands stored in this space are executed at boot time allowing the definition of start-up behaviour, VocalFusion® XVF3610 configuration and setup of SPI peripheral devices connected to it.

With the exception of the persistent user information the contents of the flash, and therefore the configuration of the system can be upgraded and configured using the Device Firmware Upgrade (DFU) mechanism from the host processor.

Note: The three XVF3610 variants; one providing I^2S/I^2C interface (XVF3610-INT) and two providing a USB interface (XVF3610-UA and XVF3610-UA-HYBRID) are delivered as separate sets of firmware.

Note: Unless otherwise stated, throughout the remainder of this document, the term XVF3610-UA will refer to both the UA and UA-HYBRID variants.

1.5 Default operation

The following table details the default configuration for the XVF3610-UA and XVF3610-INT firmware version 5.7 after firmware update to the default configuration.

Table 1.1: Default configuration of XVF3610

Parameter	Default UA and UA-HYBRID	Default INT	Can Configure?
Version (x=patch version)	5.7.x	5.7.x	N
Reference input FROM host	USB UAC 1.0 48k samples/s PCM 16-bit resolution	I^2S slave 48k samples/s PCM 32-bit resolution	Y (prior to microphone and I^2S start up
Reference format	1 or 2 channel (Mono / Stereo)	1 or 2 channel (Mono / Stereo)	N
Processed audio output TO host	USB UAC 1.0 48k samples/s PCM 16-bit resolution	I^2S bus 48k samples/s PCM 32-bit resolution	Y (prior to microphone and I^2S start up
Audio format to host	2 channel - two different streams CH[0] - ASR CH[1] - Comms	2 channel - two different streams CH[0] - ASR CH[1] - Comms	Y
USB Product String	XVF3610 (UAC1.0) Adaptive	N/A	Y
USB Vendor ID	0x20B1 (8369)	N/A	Y
USB Product ID	0x0016 (22)	N/A	Y
USB Vendor String	XMOS	N/A	Y
USB Serial Number	null	N/A	Y
I^2C address	N/A	0X2C	N
MCLK	24.576MHz OUTPUT	24.576MHz INPUT	Y
Acoustic Echo Canceller	Enabled	Enabled	Y
Automatic Delay Estimator	Activated once on startup	Activated once on startup	Y
Interference Canceller	Enabled	Enabled	Y
Noise suppressor	Enabled	Enabled	Y

2 Audio Processing Pipeline

The core of the XVF3610 voice processor is a high-performance audio processing pipeline that takes its input from a pair of the microphone and executes a series of signal processing algorithms to extract a voice signal from a complex soundscape. The audio pipeline can accept a reference signal from a host system which is used to perform Acoustic Echo Cancellation (AEC) to remove audio being played by the host. The audio pipeline provides two different output channels - one that is optimized for Automatic Speech Recognition systems and the other for voice communications.

A flexible audio signal routing infrastructure and a range of digital inputs and outputs enables the XVF3610 to be integrated into a wide range of system configurations, that can be configured at start up and during operation through a set of control registers.

2.1 Signal flow and processing

The arrangement of the blocks, with respect to the device Input & Output and the XVF3610 audio processing pipeline, is shown below:

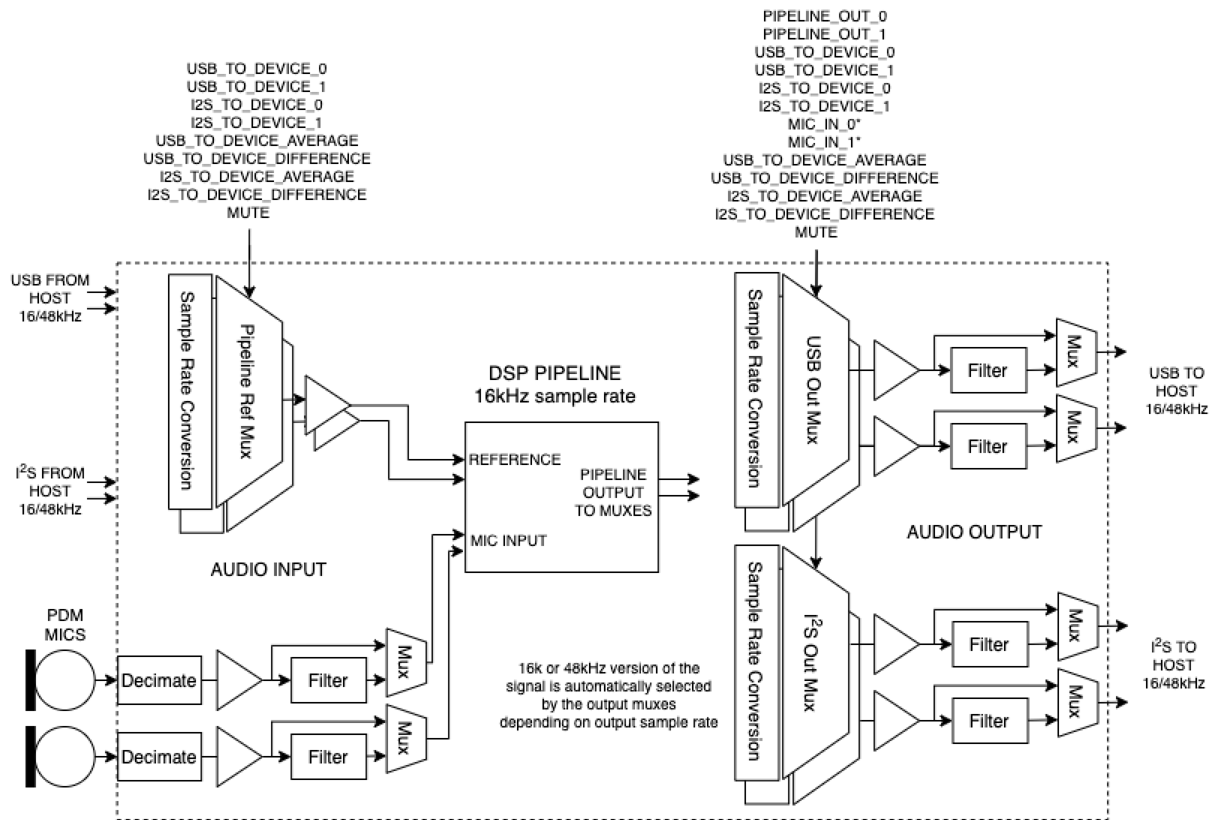

Fig. 2.1: XVF3610 input, output and audio signal routing

The blocks supported are as follows:

- Signal Multiplexers. These allow dynamic selection (switching) of signals. The signals available depend on the multiplexer position.
- Gain Blocks. These are blocks that apply a variable bit shift (left or right) and, in the case of left shift, saturate in the case of overflow. Because they are shifters, the gain applied is a power of two.
- Filter Blocks. The filter blocks consist of two cascaded biquad units. Each of the five coefficients per stage is directly manipulated via the control utility.

The commands to control the audio multiplexes (Mux) blocks and the source and destination index numbers are listed in the following sections.

The XVF3610 provides a flexible routing control scheme to configure signal routing through the pipeline itself, providing flexibility useful in:

- Hardware testing of microphones by monitoring the raw microphone signal.
- Improving pipeline performance by filtering known noise sources at the raw microphone input.
- Monitoring and debugging of reference signals and microphone signals during development.
- Compensating for gain offset in the reference signal.
- Supporting specific audio connectivity requirements such as obtaining the reference signal from I^2S.
- Inserting audio filtering where a loudspeaker is connected downstream of the XVF3610 via I^2S.

2.2 Signal Routing and Scaling

2.2.1 Routing commands

The following controls are provided for configuring the signal control blocks.

Table 2.1: IO Mapping commands

Command	Type	Args	Definition
SET_IO_MAP	uint8	2	Configures the two input switches and four output switches. See Destination and Source index table for valid argument options. arg1 <Destination Index> - arg2 <Source Index>
SET_OUTPUT_SHIFT	int32	2	Sets the gain for each mux block. Select mux block Destination Index followed by shift (+ve is left -ve is right shift). arg1 <Destination Index> - arg2 <shift value>
GET_OUTPUT_SHIFT	uint32	8 x 3	Get all IO_MAP and OUTPUT_SHIFT values for all destinations.
SET_MIC_SHIFT_SATURATE / GET_MIC_SHIFT_SATURATE	uint32	2	Sets the gain on the raw mic signals before entering the Pipeline. arg1 <shift value (left shift)> - arg2 <saturate - enable if =1>

2.2.2 Destinations

The Destination channels available to be mapped are referenced as follows:

Table 2.2: Mapping Destination Indexes

Channel (Destination)	Value	Definition
USB_FROM_DEVICE_0	0	USB channel 0 output from device to host
USB_FROM_DEVICE_1	1	USB channel 1 output from device to host
I2S_FROM_DEVICE_0	2	I^2S channel 0 output from device
I2S_FROM_DEVICE_1	3	I^2S channel 1 output from device
REF_TO_PIPELINE_0	4	reference channel 0 going into the pipeline
REF_TO_PIPELINE_1	5	reference channel 1 going into the pipeline
MIC_TO_PIPELINE_0	6	microphone channel 0 going into the pipeline
MIC_TO_PIPELINE_1	7	microphone channel 1 going into the pipeline

2.2.3 Sources

Sources available to be mapped to destinations are referenced as follows:

Table 2.3: I/O Mapping Source Indexes

Channel (Source)	Value	Definition
MUTE	0	Zeros are sent to the destination if this value is selected - which mutes the channel
USB_TO_DEVICE_AVERAGE	1	Average of USB input from host to device
USB_TO_DEVICE_DIFFERENCE	2	Half of the difference between ch0 and ch1 of USB input from host to device
I2S_TO_DEVICE_AVERAGE	3	Average of I^2S input to device
I2S_TO_DEVICE_DIFFERENCE	4	Half of the difference between ch0 and ch1 of I^2S input to device
PIPELINE_OUT_0	5	Pipeline output channel 0
PIPELINE_OUT_1	6	Pipeline output channel 1
USB_TO_DEVICE_0	7	USB input channel 0 from host to device
USB_TO_DEVICE_1	8	USB input channel 1 from host to device
I2S_TO_DEVICE_0	9	I^2S input channel 0 to device
I2S_TO_DEVICE_1	10	I^2S input channel 1 to device
MIC_IN_0	11	Ch0 Microphone input seen by the pipeline
MIC_IN_1	12	Ch1 Microphone input seen by the pipeline
PACKED_PIPELINE_OUTPUT	13	pack 16kHz pipeline output on 48kHz output
PACKED_MIC	14	pack 16kHz mic input to pipeline on 48kHz output
PACKED_REF	15	pack 16kHz reference input to pipeline on 48kHz output
PACKED_ALL	16	pack 1 channel of 16kHz mic - reference input and pipeline. When this option is used the other channel of the same output also gets PACKED_ALL set in its IO map
PACKED_ALL_INPUT_USB	17	pack 16kHz mic and reference into a 48kHz USB input
PACKED_ALL_INPUT_I2S	18	pack 16kHz mic and reference into a 48kHz I^2S input

Note: The MIC_IN_0 and MIC_IN_1 signals are at 16kHz. If they are routed to a 48kHz output they will be sample repeated three times. No antialiasing filter is applied.

2.2.4 Example Routing Commands

The following section illustrates how to use the IO mapping and scaling commands.

Using the *SET_IO_MAP* command, the user can choose the sources that get routed to the following 3 destinations:

- the USB output from device to host
- the I^{2}S output from the device
- the reference going into the device

For instance, to route I^{2}S channel 0 (= 9 as shown in the Source table) input to the device to USB channel 1 output from the device (= 1 as shown in the destination table), the command is:

```
vfctrl_usb SET_IO_MAP 1 9
```

where the first argument “1” refers to USB_FROM_DEVICE_1 as shown in the destination table and the second argument “9” refers to I2S_TO_DEVICE_0 in the source table.

Signal routing is also useful for hardware debugging of microphone or reference signal connection. As an example, the following command routes USB reference channel 0 from host to the USB audio output channel 0 of the XVF3610:

```
vfctrl_usb SET_IO_MAP 0 7
```

This command sets a loopback of the reference signal given to the XVF3610 to its audio output. By playing a simple reference signal, e.g., a sine wave, the user can verify if the XVF3610 has received the signal properly through its audio output. If the audio signal recorded at host is different from the reference output, the user may check if the problem is caused by hardware connection failure or wrong data format.

Signal routing can also be used for debugging microphone signal:

```
vfctrl_usb SET_IO_MAP 1 12
```

The above command routes microphone channel 1 as the direct signal to the USB audio output of the XVF3610. Microphone signals can the be verified by recording the audio output from the XVF3610.

For XVF3610-UA, its I^{2}S master interface can be used for sending out different signals as shown in the source channel table while having the USB output processed audio. For example, the following command configures the XVF3610 to send microphone, reference and pipeline outputs in 16kHz sampling frequency packed to 48kHz I^{2}S output:

```
vfctrl_usb SET_IO_MAP 2 16
vfctrl_usb SET_IO_MAP 3 16
```

By using Raspberry Pi with I^{2}S slave interface configured, the user can then capture synchronized signals of microphone, reference and pipeline output. Observing these signals can be very useful for debugging. The packed signal can be unpacked to mic, reference and pipeline signal with 2 channels in each of them by using a Python script provided in the Release Package.

The *SET_OUTPUT_SHIFT* command can be used to specify a bit shift that is applied to all samples of a given target. For example, specifying:

```
vfctrl_usb SET_OUTPUT_SHIFT 2 4
```

applies a left shift of 4 bits on all samples output from the device on I2S channel 0 as 2^4=16x of gain. A negative shift value would imply a right bit shift for attenuation.

The *GET_IO_MAP_AND_SHIFT* command displays the IO mapping and the shift values for all targets.

Executing a *GET_IO_MAP_AND_SHIFT* command without having set any mapping or shifts explicitly shows the default mapping that is configured in firmware.

```
vfctrl_usb GET_IO_MAP_AND_SHIFT
```

```
 GET_IO_MAP_AND_SHIFT:
 target: USB_FROM_DEVICE_0, source: PIPELINE_OUT_0 output shift: NONE
 target: USB_FROM_DEVICE_1, source: PIPELINE_OUT_1 output shift: NONE
 target: I2S_FROM_DEVICE_0, source: PIPELINE_OUT_0 output shift: NONE
 target: I2S_FROM_DEVICE_1, source: FAR_END_IN_0 output shift: NONE
 target: REF_TO_PIPELINE_0, source: USB_TO_DEVICE_0 output shift: NONE
 target: REF_TO_PIPELINE_1, source: USB_TO_DEVICE_1 output shift: NONE
```

2.2.5 PACKED_ALL signals

`PACKED_ALL` packs up to six 16kHz channels into a 48kHz stereo signal. When using USB (UA) firmware it uses the bit resolution of the USB output interface (even if you output to I^2S on the UA device) and always assumes 32b it you are using I^2S since the I^2S interface uses a fixed 32b bit width. The packing sequence is as follows:

Table 2.4: Packed audio channels

16kHz channel	PACKED_ALL input	PACKED_ALL output
0	MIC 1	MIC 1
1	MIC 0	MIC 0
2	REF left	REF left
3	REF right	REF right
4	unused (ignored)	ASR pipeline output
5	unused (ignored)	Comms pipeline output

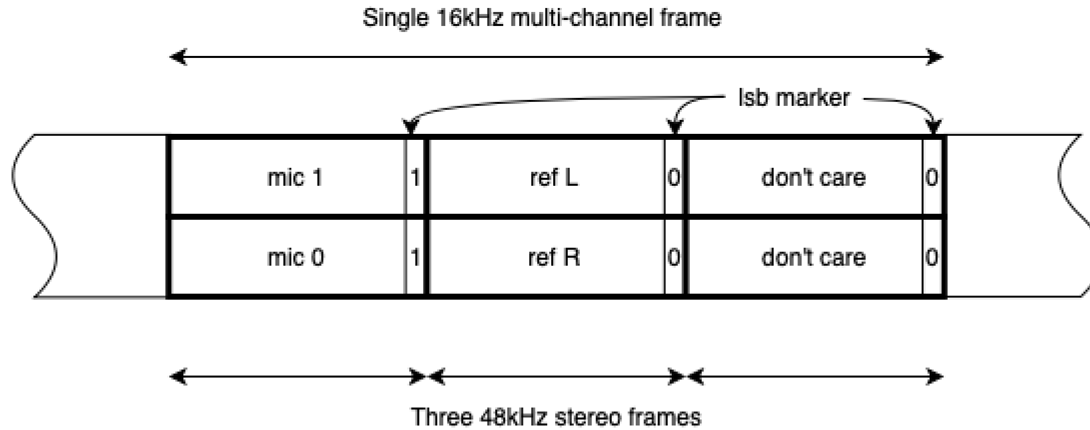

Fig. 2.2: Packing sequence

1. Microphone samples with marker '1' in least significant bit
2. Reference samples with marker '0' in least significant bit
3. Pipeline out sample with marker '0' in least significant bit

The *packer_packed_all.py* script masks off the least significant bit and inserts the packing marker sequence, as well as changing the output format to 48kHz stereo. It can support 16, 24 or 32b resolution although 24b files are saved and read as 32b with 8b LSB padding. It can work on a Mac if you use a 16b or 24b output resolution on the USB interface. Since microphone signal levels are quite low from the output of the decimators, it is recommended to use at least 24b resolution to keep the quantisation noise floor down with respect to signal.

The *unpacker_packed_all.pyscript* looks for 0, 0, 1 for the LS bit to check for a `PACKED_ALL` sequence, else it will report an error. It will try to recover from sequence errors if present. The packing will work with 16b, 24b and 32b sample bit widths although <=24b is recommended.

For more information and use cases for the packed audio please refer to *Capturing Packed Samples* and *Direct access to DSP Pipeline* sections.

2.3 General Purpose Filter

The General Purpose filter blocks each comprise of two cascade biquad filters permitting configuration as band-pass, notch, low-pass, high-pass filters etc. By default, all filters are disabled (bypassed).

Note: A maximum of two output filters may be enabled simultaneously. E.g. Two channels of USB filtering or one I^2S and one USB output. Exceeding this may cause audio glitching.

There is no restriction on input filters (microphone and reference filters).

The filter coefficients are accepted in a floating-point format in a1, a2, b0, b1, b2 order directly from filter design tools such as https://arachnoid.com/BiQuadDesigner/index.html.

Support for the raw 32bit integer write/read is offered which directly accesses the internal representation. When using the raw control method, coefficients should be converted to Q28.4 format first and a_1 and a_2 need to be negated. See configuration parameters for more information.

The sample rate for filters on the input to the pipeline are always 16kHz whereas the output filters match the selected rate which may be either 16kHz or 48kHz, depending on system configuration. Ensure that the filter coefficients have been designed with the correct rate.

Note that, although potential numerical overflows are handled as a saturation, it is up to the designer to ensure no saturation occurs from the coefficients chosen to avoid non-linear behaviour of the filter. The implementation offers three bits of headroom (Q28.4) which is more than sufficient for most filters.

The coefficients are cleared to zero on boot.

The following table describes the commands for the configuration of the filters.

Table 2.5: Filter configuration parameters

Command	Type	Arguments	Definition
SET_FILTER_INDEX	uint8	1	Used as an index to point to which filter block that will be manipulated. output_filter_map_t below defines the filter block IDs.
GET_FILTER_INDEX	uint8	1	Retrieve the current filter index.
SET_FITER_BYPASS	uint8	1	Bypass (1) means filter pointed to by the index is not enabled (default) - 0 means enable the filter.
GET_FILTER_BYPASS	uint8	1	Retrieve the bypass status.
SET_FILTER_COEFF	float	10 (5x2)	Set 5 x 2 biquad coefficients in a floating-point format in the order a1 a2 b0 b1 b2. Coefficient a0 is assumed to be 1.0. If it is not - divide all coefficients by a0.
GET_FILTER_COEFF	float	10 (5x2)	Retrieve the floating-point representation of the coefficients in the order a1 a2 b0 b1 b2.
SET_FILTER_COEFF_RAW	int32	10 (5x2)	Set 5 x 2 biquad coefficients in Q28.4 format for the filter pointed to by the index. See note above in Filter Blocks section about the format.
GET_FILTER_COEFF_RAW	int32	10 (5x2)	Retrieve the Q28.4 representation of the coefficients. See note above in Filter Blocks section about the format.

Filter output indexes available to be used with filter setting commands (output_filter_map_t):

Table 2.6: Output Indexes

Channel	Value	Definition
FILTER_USB_FROM_DEVICE_0	0	USB channel 0 from device to host (Left)
FILTER_USB_FROM_DEVICE_1	1	USB channel 1 from device to host (Right)
FILTER_I2S_FROM_DEVICE_0	2	I^2S channel 0 from device (Left)
FILTER_I2S_FROM_DEVICE_1	3	I^2S channel 1 output from device (Right)
FILTER_MIC_TO_PIPELINE_0	4	16kHz mic channel 0 going into the pipeline
FILTER_MIC_TO_PIPELINE_1	5	16kHz mic channel 1 going into the pipeline
FILTER_REF_TO_PIPELINE_1	6	16kHz reference channel 0 going into the pipeline (Left)
FILTER_REF_TO_PIPELINE_1	7	16kHz reference channel 1 going into the pipeline (Right)

While setting the index or bypass control will always be safe, there is a small chance that the coefficients may be partially updated halfway through a filter operation. For this reason, the filter state is also cleared following

updating to ensure that any possibility of instability is reduced. It is up to the user to ensure that the coefficients provided result in a stable filter configuration.

A *worked example* is provided in the reference section.

2.4 PDM microphone interface

The PDM microphone interface converts Pulse Density Modulation (PDM) audio input from the microphones to Pulse Code Modulation (PCM) format allowing further processing. The PDM microphone interface consists of the physical pins connecting to the two microphones and a series of filters resulting in a 16kHz PCM, two-channel output stream suitable for far-field voice processing. Please refer to the datasheet for the physical and electrical details of the PDM pins.

The processing consists of four filter stages:

- Decimate by 8 FIR filter to 384kHz
- Decimate by 4 FIR filter to 96kHz
- Decimate by 6 FIR filter to 16kHz
- DC Blocking, single-pole IIR filter

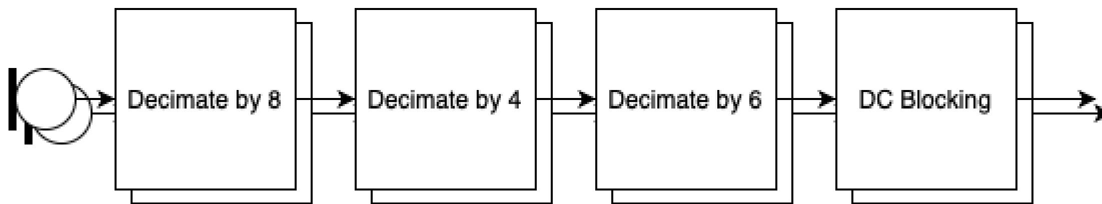

Fig. 2.3: PDM microphone processing steps

The PDM microphone interface uses 32-bit internal processing to provide very low distortion with a specification exceeding -110dB THD+N with a 140dB dynamic range.

The frequency response of the FIR filter has a stopband attenuation of at least 70dB with a passband ripple of less than 0.9dB and a passband of 6.8kHz. The total group delay from pin to the XVF3610 audio pipeline input is 1.125 milliseconds.

A DC blocking filter is placed at the end of the PDM microphone interface pipeline and is tuned to have a 5Hz -6dB point and removes any DC offset present in the PDM input.

The output from the PDM microphone interface may optionally be shifted or attenuated providing a 'power of two' gain control. Saturation may be applied in the case that the gain is greater than one.

By default, the gain block shift is set to zero (a gain of $2^0 = 1$) and this is the recommended setting for normal use.

The PDM interface control parameters are shown below:

Table 2.7: Microphone commands

Command	Type	Value	Description
SET_MIC_SHIFT_SATURATE	uint32	arg1 <shift value (left shift)> arg2 <saturate - enable if !=0>	Write the gain (power of 2) on the raw mic signals before entering the audio pipeline.
GET_MIC_SHIFT_SATURATE	uint32		Read the gain (power of 2) on the raw mic and Saturate Enable signals before entering the audio pipeline.

2.5 Automatic Echo Cancellation (AEC)

This process uses the stereo audio from the product as a reference signal to model the echo characteristics between each loudspeaker and microphone, caused by the acoustic environment of the device and room.

The AEC uses four models to continuously remove echoes in the microphone audio input created in the room by the loudspeakers. The models continually adapt to the acoustic environment to accommodate changes in the room created by events such as doors opening or closing and people moving about.

An illustration of echo paths in two sizes of room are shown below.

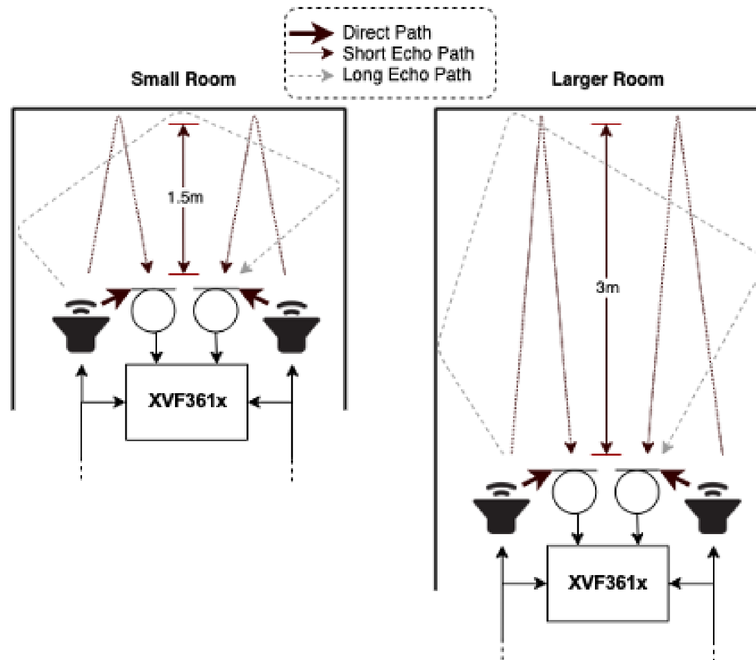

Fig. 2.4: Echo paths from the speakers to the microphones

After reset, or when echo paths change due to a change in the environment, the AEC will re-converge. Echo Return Loss Enhancement (ERLE) can be used to indicate the degree of convergence on the AEC filters as shown below.

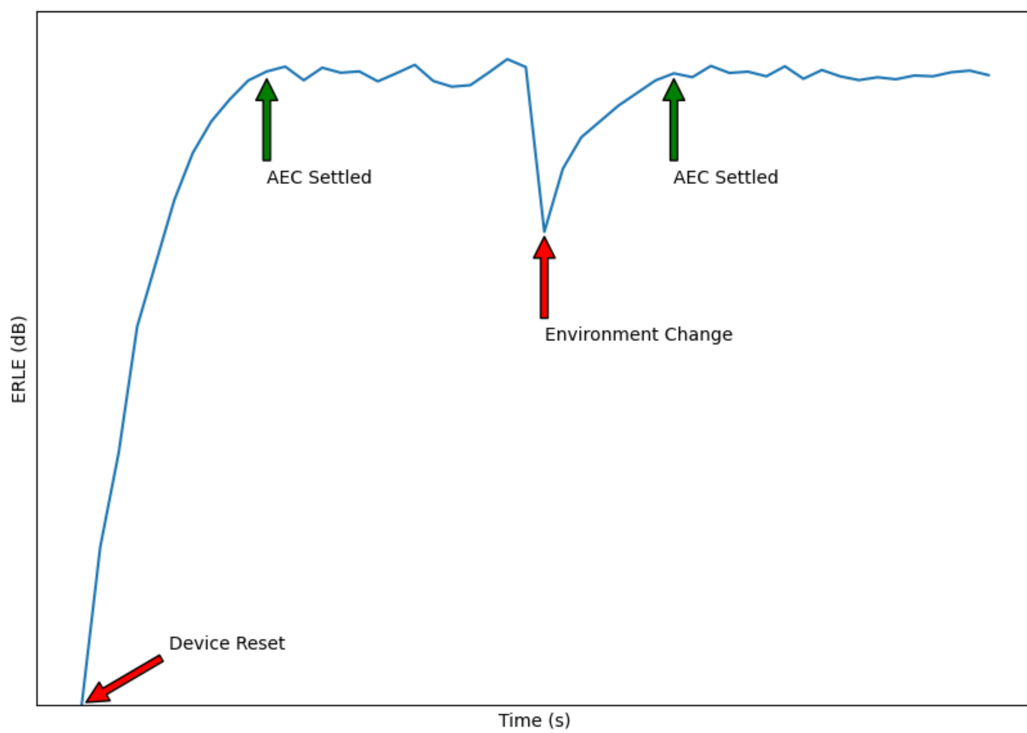

Fig. 2.5: Settling time of the AEC shown using an ERLE plot

For optimal AEC settling-time performance, the volume of the loudspeakers must be linearly proportional to the level of the reference audio sent to the XVF3610. If the volume of the loudspeakers changes without the level of the reference changing by the same linear factor, the AEC will respond as if the environment has changed such that all echo paths have increased/decreased energy. The AEC and therefore incur a settling time.

The Alternative Architecture (described in the *Alternative Architecture mode (ALT_ARCH)* section*)* selectively extends the AEC filters to accommodate highly reverberant environments.

The configuration parameters for the AEC are shown below:

Table 2.8: Useful Automatic Echo Canceller (AEC) commands

Command	Type	Value	Description	Notes
GET_BYPASS_AEC / SET_BYPASS_AEC	uint32	[0 - 1]	Get / set AEC bypass parameter. If set to one AEC processing is disabled	A
GET_ADAPTATION_CONFIG_AEC / SET_ADAPTATION_CONFIG_AEC	uint32	[0 - 2]	Get / set AEC adaptation configuration: 0 = Auto adapt (default) 1 = Force adaptation ON 2 = Force adaptation OFF. If AEC is set to bypass then setting the adaptation config has no effect	B
GET_ERLE_CH0_AEC	float		Get AEC ERLE for channel 0	
GET_ERLE_CH1_AEC	float		Get AEC ERLE for channel 1	C
RESET_FILTER_AEC			This command resets all AEC filters	C

Notes:

[A] When the Alternative Architecture (ALT_ARCH) mode is enabled (default), AEC bypass state will be overwritten and so should not be used. The GET command remains functional. For more information see the *Alternative Architecture (ALT_ARCH)* section.

[B] If Automatic Delay Estimation is enabled, these parameters will be overwritten and so should not be used. The GET commands remain functional. For more information see the *Automatic Delay Estimation Control (ADEC)* section.

[C] When the ALT_ARCH mode is enabled, there is only valid ERLE data available on CH0. In this mode the GET_ERLE_CH1_AEC will report NaN.

Note: The AEC operates on acoustic paths modelled in the AEC tail length. The Automatic Delay Estimation Control module handles delays between microphone and loudspeaker introduced by the equipment, for instance receiving the reference ahead of it actually being played out of the loudspeakers.

2.6 Automatic Delay Estimation Control (ADEC)

The ADEC module automatically corrects for possible delay offsets between the reference and the loudspeakers.

Echo cancellation is an adaptive filtering process which compares the reference audio to that received from the microphones. It models the reverberant time of a room, i.e. the time it takes for acoustic reflections to decay to insignificance. This is shown in the figure below (the red "Acoustic echo path delay").

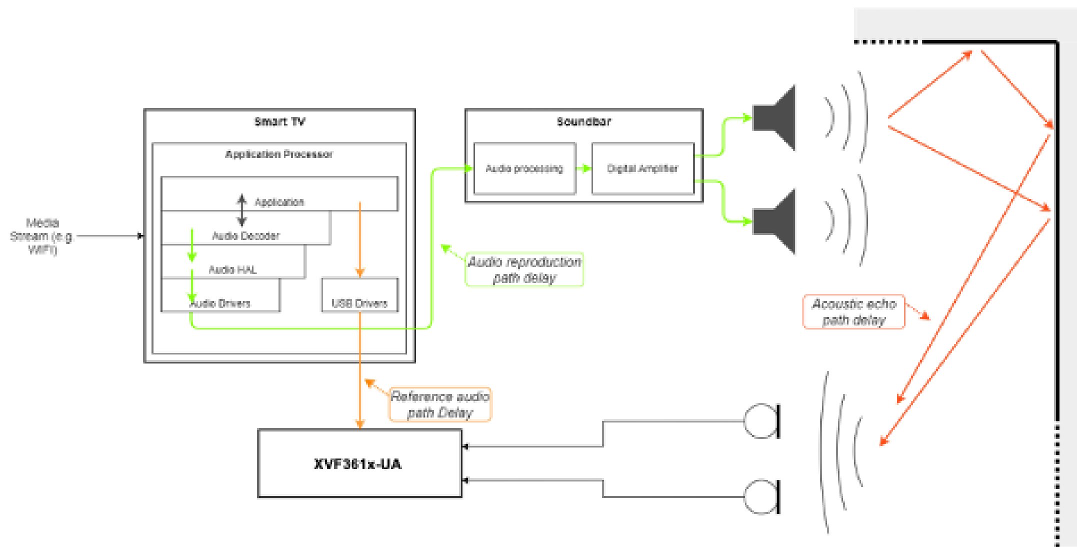

Fig. 2.6: ADEC use case diagram

The time window modelled by the Acoustic Echo Canceller (AEC) is finite (filter tail length), and to maximise its performance it is important to ensure that the reference audio is presented to the AEC time aligned to the audio being reproduced by the loudspeakers. The diagram below highlights how the reference audio path delay and the audio reproduction path may be significantly different, therefore requiring additional delay to be inserted into one of the two paths, correcting this delay difference.

The functional blocks in the ADEC are shown below:

The ADEC may apply a delay to either the microphone or the reference path. When the loudspeaker signal lags behind the reference signal, the ADEC places a delay into the reference channel. When the reference signal lags behind the loudspeaker speaker, the ADEC places a delay into the microphone channel.

Automatic delay estimation is triggered at power-up, or if the host system configuration changes. The process will not begin until the reference signal is present and has sufficient energy.

The delay estimation process re-purposes the AEC to detect larger delays. During estimation, the AEC does not perform cancellation. Once the delay is detected and delay correction made, the AEC restarts and converges based on the delayed signals.

Possible causes that may trigger an estimation cycle (where automatic mode is enabled):

- Host changing applications causing a delay change between loudspeakers and reference.
- Large volume changes between the reference and the loudspeaker play-back.
- User equipment changes, such as switching from TV audio output to playing the audio through a sound bar.

The characteristics and specification of the ADEC function is shown below:

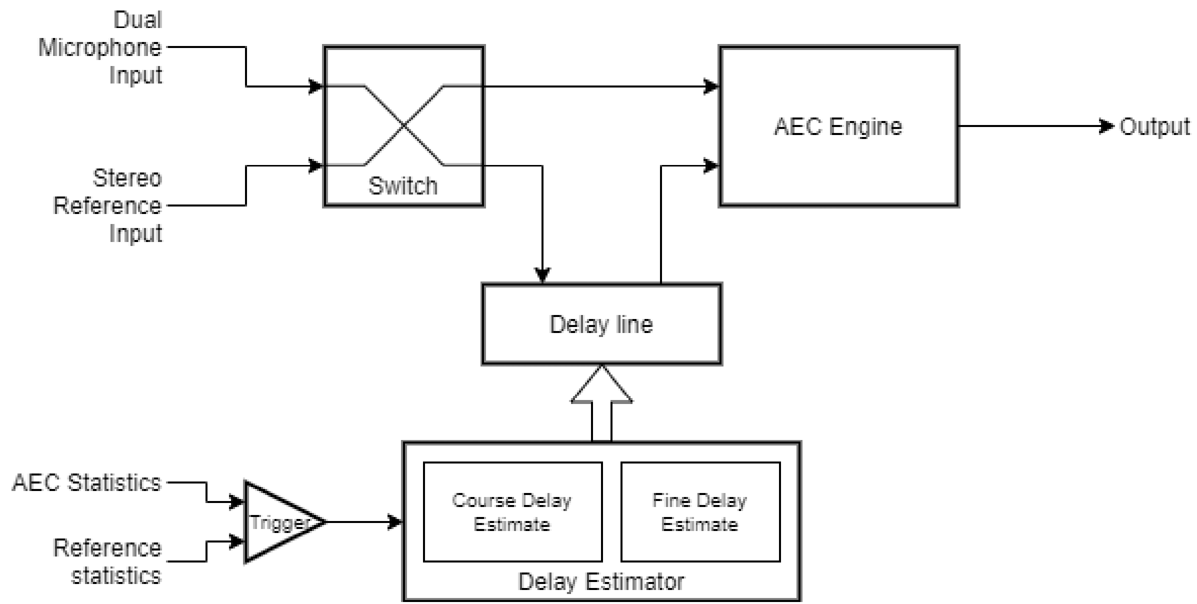

Fig. 2.7: ADEC block diagram

Table 2.9: ADEC characteristics

Name	Value	Description
Maximum delay correction	$\pm$ 150ms	The maximum delay that can be added to either the microphone channel or the reference channel
Estimation time	With good reference SNR: 2-5 seconds	During this time AEC is disabled. Note that estimation will not start unless reference is available and loudspeakers are playing back

The configuration commands are shown below:

Table 2.10: Automatic Delay Estimator parameters

Command	Type	Value	Description	Notes
GET_DELAY_SAMPLES SET_DELAY_SAMPLES	uint32	[0 - 2399]	Change the number of samples of input delay at the sample rate 16kHz. The delay is applied to either the reference or the microphone input according to the delay direction. This provides a maximum delay of +/- 150mS.	A
GET_DELAY_DIRECTION SET_DELAY_DIRECTION	uint32	[0 - 1]	Select the direction of input delay. 0: Delay the reference input (default) - 1: Delay the microphone input.	A
GET_DELAY_ESTIMATE	uint32	[0 - 7200]	Get an estimate of the number of samples of delay on the reference input at a sample rate of 16kHz. This value is valid only when a delay estimation is in progress and is offset by the maximum length of the delay buffer (2400 samples). Add 2400 samples to this value to get the absolute delay estimate.	
SET_ADEC_ENABLED GET_ADEC_ENABLED	uint32	[0 - 1]	Enable automatic delay control: 0: ADEC disabled - 1: ADEC enabled.	
GET_ADEC_MODE	uint32	[0 - 1]	Get the status of delay estimation: 0: Normal AEC mode - 1: delay estimation in progress.	
SET_ADEC_INITIAL_CYCLE _ENABLED	uint32	[0 - 1]	Trigger a delay estimation cycle at startup. The default behavior in firmware is to trigger a delay estimation cycle when the far end reference is detected for the first time after device reset. This is done irrespective of whether automatic delay control is enabled or disabled. To disable this initial delay estimation set SET_ADEC_INITIAL_CYCLE_ENABLED = 0 in the data partition.	
SET_MANUAL_ADEC_CYCLE _TRIGGER	uint32	[0 - 1]	Trigger a delay estimation cycle. If delay estimation is disabled the SET_MANUAL_ADEC_CYCLE_TRIGGER can be used to force a delay estimation cycle at any time.	

NOTES:

[A] When the ADEC is enabled, this value will be overwritten, therefore the SET commands should not be used. GET commands remain valid.

2.7 Interference canceller

The Interference Canceller (IC) suppresses static noise from point sources such as cooker hoods, washing machines, or radios for which there is no reference audio signal available. When an internal Voice Activity Detector (VAD) indicates the absence of voice, the IC adapts to remove noise from point sources in the environment. When the VAD detects voice, the IC suspends adaptation which maintains suppression of the interfering noise sources previously adapted to.

The IC only operates on the ASR channel from the pipeline output.

The following table describes the configuration parameters for the Interference Canceller.

Table 2.11: Interference Canceller (IC) parameters

Command	Type	Value	Description	Notes
SET_BYPASS_IC GET_BYPASS_IC	uint32	[0 - 1]	Set IC bypass parameter: 0 = IC bypass disabled (default) - 1 = IC bypass enabled	A
SET_CH1_BEAMFORM_ENABLE GET_CH1_BEAMFORM_ENABLE	uint32	[0 - 1]	Enable beamformed output on IC output channel index 1 - 0 = Passthrough IC input channel 1 onto IC output channel 1 - 1 = Beamformed output on IC output channel 1 (default)	
RESET_FILTER_IC			This command resets the IC filter	

[A] If Alternative architecture mode (ALT_ARCH) is enabled (default), the IC bypass state will be dynamically changed by the firmware. Do not use the `SET_` commands. The `GET_` commands remains functional.

2.8 Noise Suppressor (NS)

The Noise Suppressor (NS) suppresses noise from sources whose frequency characteristics do not change rapidly over time. This includes diffuse background noise and stationary noise sources.

The following table describes the settings for the Noise Suppressor.

Table 2.12: Noise Suppressor (NS) commands

Command	Type	Value	Description
GET_BYPASS_SUP SET_BYPASS_SUP	uint32	[0 - 1]	Get / set suppressor bypass parameter. If set to one the suppressor which contains the noise suppression stages is bypassed. 0: suppressor bypass disabled (default) 1: suppressor bypass enabled
GET_ENABLED_NS SET_ENABLED_NS	uint32	[0 - 1]	Get / set noise suppression enabled parameter within the suppressor. If set to one - the noise suppression stage within suppressor is enabled. Changing this parameter only takes effect if the suppressor is not bypassed. 0: noise suppression disabled - 1: noise suppression enabled (default)

2.9 Automatic Gain Control (AGC) and Loss Control

The Automatic Gain Control (AGC) can dynamically adapt the audio gain, or apply a fixed gain such that voice content maintains a desired output level. The AGC uses an internal Voice Activity Detector to normalise voice content and avoid amplifying noise sources and applies a soft limiter to avoid clipping on the output. The design is based on standard modern AGC techniques as detailed in '*Acoustic Echo and Noise Control*', by Hansler and Schmidt.

The desired output level of voice content is defined by an upper and lower threshold. If a voice signal is outside of the upper and lower threshold then the gain will adapt accordingly. If the voice signal is within the upper and lower threshold then the gain will remain constant.

The rate at which the gain increases or decreases per audio frame can also be configured. The gain increment value must be greater than 1, whilst the gain decrement value must be below 1. When the gain is adapting, the current gain value is multiplied by either the increment or decrement value to calculate the gain value to be applied on the next audio frame. Voice activity is monitored and included in the algorithm to avoid the noise floor being amplified during silent periods. In addition, maximum and minimum levels may be set to keep the gain within a certain range.

The following table details the configuration parameters for the AGC. Both GET_ and SET_ operations are supported for these parameters.

Table 2.13: Automatic Gain Control (AGC) parameters

Parameter	Type	Value	Description
LC_ENABLED_CH0_AGC LC_ENABLED_CH1_AGC	uint32	[0 - 1]	Set Loss Control to be enabled in the AGC for channel 0 or 1. 0 - Loss Control disabled for the channel 1 - Loss Control enabled for the channel
LC_GAINS_CH0_AGC LC_GAINS_CH1_AGC	Q16.16	[0..32767]	Loss control gains: arg1: max arg2: double-talk arg3: silence arg4: min
LC_N_FRAMES_CH0_AGC LC_N_FRAMES_CH1_AGC	Q16.16	[0..32767]	Number of frames in loss control for nearend and far-end activity arg1: nearend arg2: far-end
LC_GAMMAS_CH0_AGC LC_GAMMAS_CH1_AGC	Q16.16	[0..32767]	Loss control gamma coefficients: background power increment and decrement arg1: background power arg2: increment arg3: decrement
LC_DELTAS_CH0_AGC LC_DELTAS_CH1_AGC	Q16.16	[0..32767]	Loss control delta coefficients: arg1: farend only arg2: near-end only arg3: both far-end and near-end
LC_CORR_THRESHOLD_CH0_AGC LC_CORR_THRESHOLD_CH1_AGC	Q1.31	[0..1]	Loss control correlation threshold for channel. Values are linear. Default: 1000
MIN_GAIN_CH0_AGC MIN_GAIN_CH1_AGC	Q16.16	[0..32767]	Set the minimum gain threshold in the AGC for channel 0 or 1. Values are linear. Default: 0
SOFT_CLIPPING_CH0_AGC SOFT_CLIPPING_CH1_AGC	uint32	[0 - 1]	Enable soft clipping on the output of channel 0 or 1. 0: Soft clipping disabled for the channel - 1: Soft clipping enabled for the channel
UPPER_THRESHOLD_CH0_AGC UPPER_THRESHOLD_CH1_AGC	Q1.31	[0..1]	Set the upper threshold for desired voice level. Values are in range 0 to 1 (full-scale) and must be greater than the lower threshold of the channel
LOWER_THRESHOLD_CH0_AGC LOWER_THRESHOLD_CH1_AGC	Q1.31	[0..1]	Set the lower threshold for desired voice level. Values are in range 0 to 1 (full-scale) and must be lower than the upper threshold of the channel
INCREMENT_GAIN_STEPSIZE_CH0_AGC INCREMENT_GAIN_STEPSIZE_CH1_AGC	Q16.16	[0..32767]	Set the rate at which the gain increases. This value is applied on a per-frame basis when voice content is detected
DECREMENT_GAIN_STEPSIZE_CH0_AGC DECREMENT_GAIN_STEPSIZE_CH1_AGC	Q16.16	[0..32767]	Set the rate at which the gain decreases. This value is: applied on a per-frame basis when voice content is detected

The Loss Control process improves the subjective audio quality by attenuating any residual echo of the reference far-end audio. It is designed to be used on the communications channel. In cases where there is both far-end echo and near-end audio then the attenuation is reduced, allowing listeners to interrupt each other. The Loss Control relies on the Automatic Echo Canceller to classify and attenuate residual far-end echo.

The following table details the configuration parameters for the Loss Control process. Both GET_ and SET_ operations are supported for these parameters.

Table 2.14: Loss Control (LC) parameters

Parameter	Type	Value	Description
LC_ENABLED_CH0_AGC LC_ENABLED_CH1_AGC	uint32	[0 - 1]	Set Loss Control to be enabled in the AGC for channel 0 or 1. 0: Loss Control disabled for the channel 1: Loss Control enabled for the channel
LC_GAINS_CH0_AGC LC_GAINS_CH1_AGC	Q16.16	[0 - 32767]	Loss control gains: arg1: max arg2: double-talk arg3: silence arg4: min
LC_N_FRAMES_CH0_AGC LC_N_FRAMES_CH1_AGC	Q16.16	[0 - 32767]	Number of frames in loss control for near-end and far-end activity arg1: near-end arg2: far-end
LC_GAMMAS_CH0_AGC LC_GAMMAS_CH1_AGC	Q16.16	[0 - 32767]	Loss control gamma coefficients: background power increment and decrement arg1: background power arg2: increment arg3: decrement
LC_DELTAS_CH0_AGC LC_DELTAS_CH1_AGC	Q16.16	[0 - 32767]	Loss control delta coefficients: arg1: far-end only arg2: near-end only arg3: both far-end and near-end
LC_CORR_THRESHOLD_CH0_AGC LC_CORR_THRESHOLD_CH1_AGC	Q1.31	[0-1]	Loss control correlation threshold for channel

2.10 Alternative Architecture mode (ALT_ARCH)

The Alternative Architecture mode, when enabled, improves Echo Cancellation performance in reverberate environments. It operates by re-configuring the audio pipeline by switching out either the Acoustic Echo Canceller (AEC) or the Interference Canceller (IC), depending on the energy in the AEC reference signal, to recover resources for use by the rest of the pipeline.

The two audio pipeline configurations are summarised below:

- **ALT_ARCH disabled** ALWAYS apply echo-cancelling AND interference cancelling; or
- **ALT_ARCH enabled** apply ONLY echo-cancelling when a reference signal is available, otherwise ONLY apply interference cancelling.

Multiplexers permit the AEC and/or the IC to be bypassed. When bypassing the IC, the XVF3610 reconfigures the AEC to use a single channel which results in the AEC cancelling echos received later in time. An internal module which collects statistics about the reference is used to dynamically control these multiplexers and memory allocation during runtime.

Note: Manually bypassing the IC using the Control Interface does not apply the memory reallocation.

The figure below highlights the audio signal path when the Alternative Architecture is disabled (ie. standard operation).

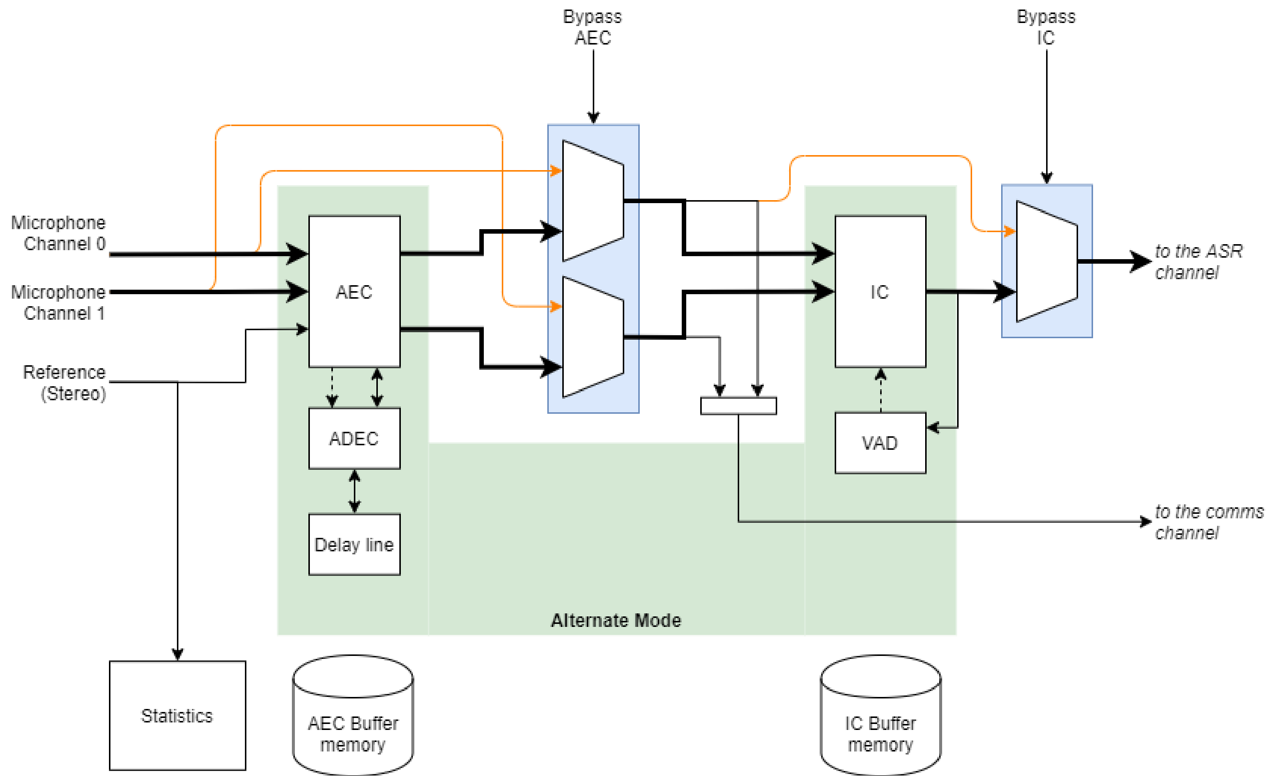

Fig. 2.8: Audio pipeline configuration, [ALT_ARCH=0] mode

Whenever ALT_ARCH=1, then the pipeline dynamically switches between AEC alone, or IC alone. In this condition the AEC is able to make use of additional memory increasing the echo cancelling period, and making it more resilient to echo in highly reverberant conditions.

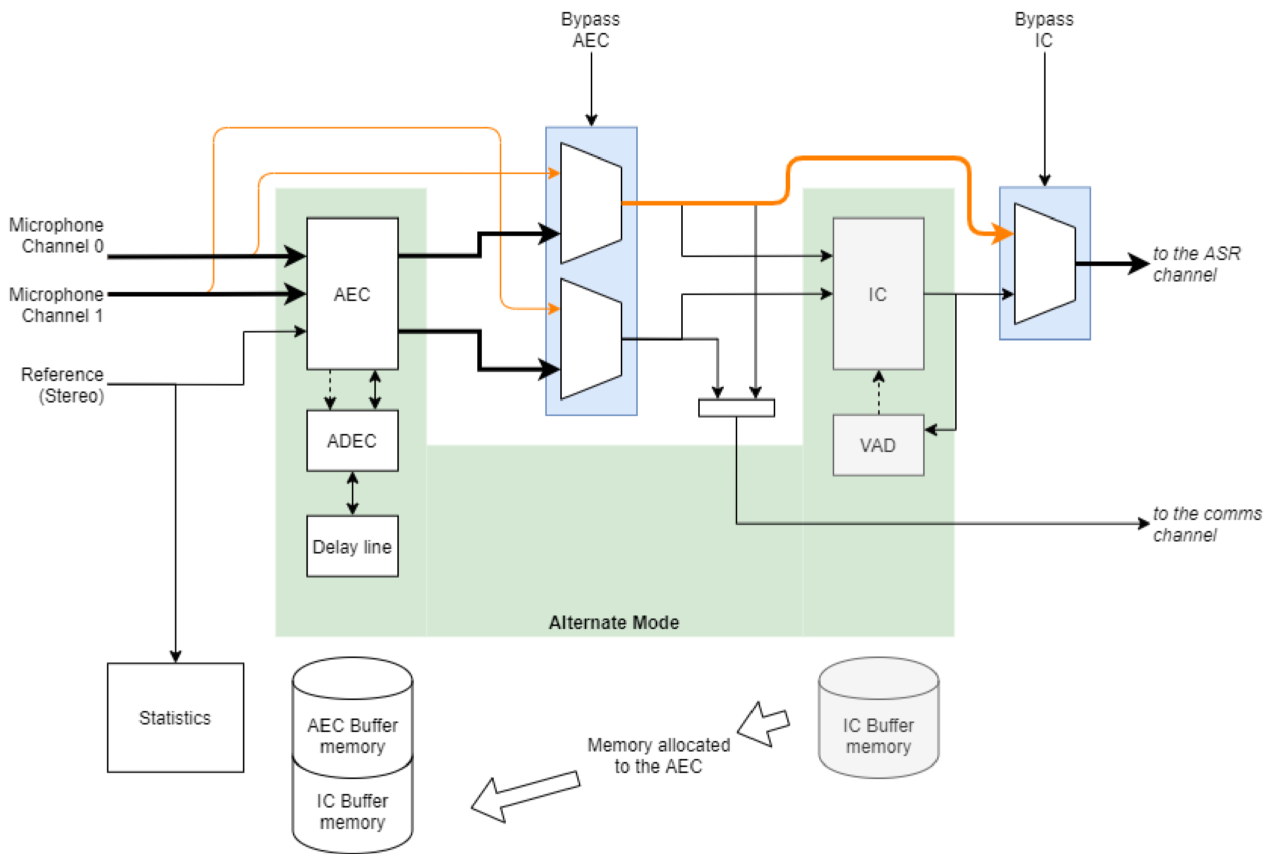

Fig. 2.9: Audio pipeline configuration, [ALT_ARCH=1] when reference signal is present

The dynamic switching uses statistics collected from the reference signal to establish if echo cancelling is required.

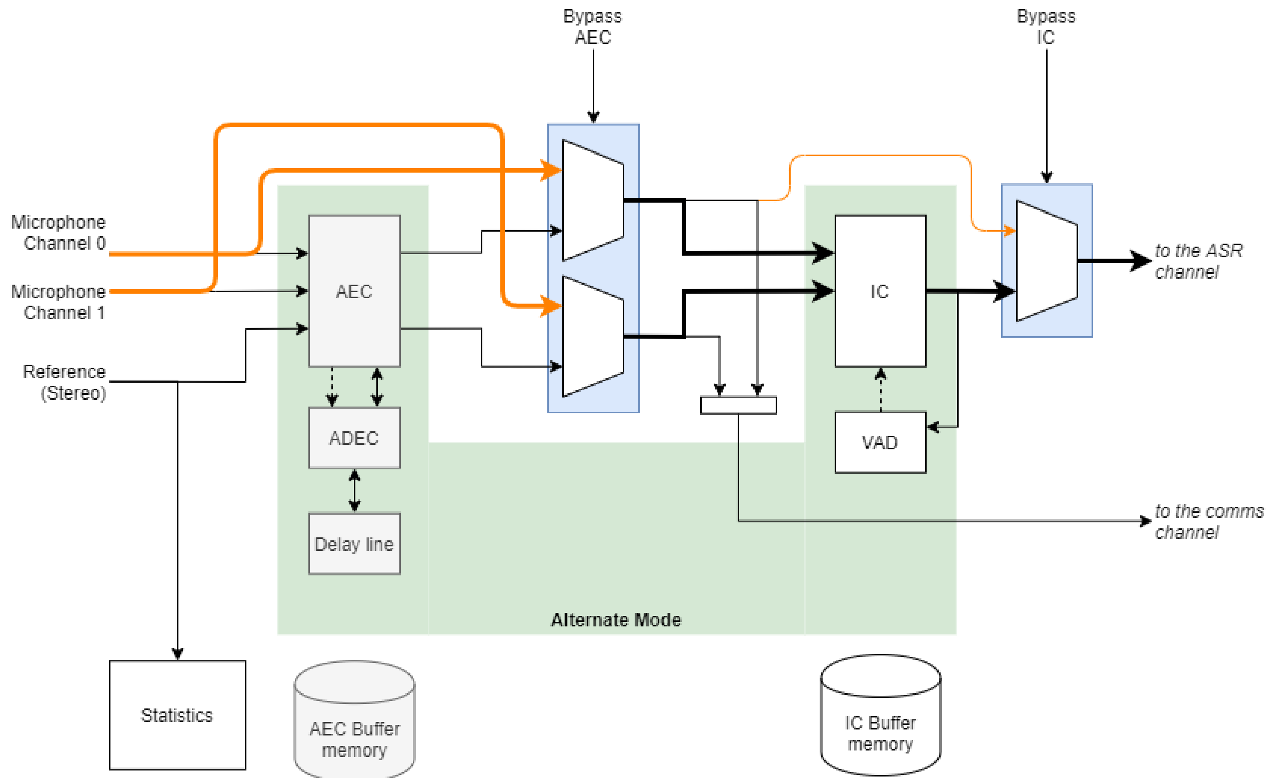

Fig. 2.10: Audio pipeline configuration, [ALT_ARCH=1] when reference signal is absent

The following table summarises the audio characteristics for standard and alternative architectures.

Table 2.15: Alternative pipeline mode characteristics

Pipeline configuration	Far-end audio (AEC Ref) status	Pipeline functionality	AEC characteristics
ALT_ARCH = 0	With and without Far-end audio present	IC enabled / AEC enabled	Max echo delay = 150ms
ALT_ARCH = 1	No far-end audio	IC enabled / AEC disabled	No cancellation
ALT_ARCH = 1	Far-end audio present	IC disabled / AEC enabled	Max echo delay = 225ms

The following table describes the configuration parameters for the Alternative Architecture.

Table 2.16: Alternative pipeline mode configuration parameters

Command	Type	Value	Description
SET_ALT_ARCH_ENABLED	uint32	[0 - 1]	Enable or disable alternate architecture (alt arch). 0: Alt arch is disabled - 1: Alt arch is enabled. When alt arch is enabled. The system works in either AEC mode (when far end signal is detected) or IC mode (when far end signal is not detected). When in AEC mode in Alt arch AEC processing happens on only one Mic channel with 15 phases per mic-ref AEC filter.

3 System Interfaces

The XVF3610 voice processor provides the following additional interfaces to increase usability and reduce total system cost:

- 4 General Purpose Output pins. These can be configured as simple digital I/O pins, Pulse Width Modulated (PWM) outputs and rate adjustable LED flashers.
- 4 General Purpose Input pins. These can be used as simple logic inputs or event capture (edge detection).
- I^2C and SPI master to control external devices such as ADCs, DACs or external keyword detection devices.

3.1 General Purpose Input and Output and Peripheral Bridging

The XVF3610 supports I/O expansion and protocol bridging over USB and I^2C for the XVF3610-UA and XVF3610-INT respectively. This allows peripheral devices such as audio hardware connected to XVF3610 to be configured and monitored by the host.

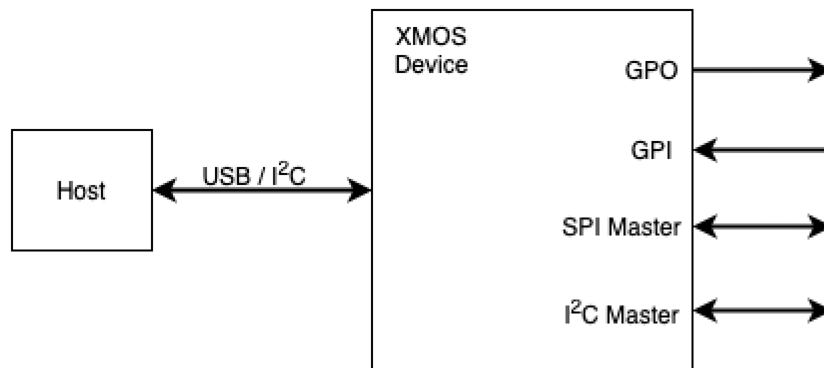

Fig. 3.1: Device GPIO interfaces

- Four GPI channels (pins)
- Direct read of port value
- Rising, falling or both edge capture with “sticky” bit which is cleared on read
- Mode configurable per pin
- Four GPO channels (pins)
- Direct write of entire port or pin
- Active high or Active low
- 500Hz PWM configurable between 0 and 100% duty cycle
- Blinking control supporting a sequence of 32, 100ms states
- SPI Master
- 1Mbps SPI clock
- Up to 128 Bytes SPI write

- Up to 56 Bytes SPI read
- I^{2}C Master (XVF3610-UA only)
- 100kbps SCL clock speed
- Register read/write (byte)
- Up to 56 byte I^{2}C read/write

The following sections describe the configuration and usage of each peripheral interface.

3.2 GPIO

There are four general input and four general output pins provided on the XVF3610.

Table 3.1: GPIO pin table

Name	Description	I/O
IP_0	General purpose input	I
IP_1	General purpose input	I
IP_2	General purpose input	I
IP_3	General purpose input	I
OP_0	General purpose output	O
OP_1	General purpose output	O
OP_2	General purpose output	O
OP_3	General purpose output	O

3.3 General Purpose Inputs

The following commands are available to read and control GPIs. Some read commands use the SET_GPI_READ_HEADER command to select the GPI port and pin.

Note that interrupt registers are set to 1 when an edge has been detected and 0 when no event has occurred. All interrupt registers are initialised to 0 on boot.

The following parameters are available to interrogate and configure the GPI behaviour.

Table 3.2: General Purpose Input commands

Command	Type	Dir	Args	Description
GET_GPI_PORT	uint32	READ	1	Read current state of all pins in the selected GPI port
GET_GPI_PIN	uint32	READ	1	Read current state of the selected GPI pin
GET_GPI_INT_PENDING _PIN	uint32	READ	1	Read whether interrupt was triggered for selected pin. The interrupt pending register for the selected pin is cleared by this command
GET_GPI_INT_PENDING _PORT	uint32	READ	1	Read whether interrupt was triggered for all pins on selected port. The interrupt pending register for the whole port is cleared by this command
SET_GPI_PIN_ACTIVE _LEVEL	uint8	WRITE	3	Set the active level for a specific GPI pin. Arguments are <Port Index> <Pin Index> <Level> - 0=active low - 1=active high. By default - all GPI pins are set to active high
SET_GPI_INT_CONFIG	uint8	WRITE	3	Sets the interrupt config for a specific pin. Arguments are <Port Index> <Pin Index> <Interrupt type> - 0=None - 1=Falling - 2=Rising - 3=Both
SET_GPI_READ_HEADER	uint8	WRITE	2	Sets the selected port and pin for the next GPI read. Arguments are <Port Index> <Pin Index>
GET_GPI_READ_HEADER	uint8	READ	2	Gets the currently selected port and pin set by a previous SET_GPI_READ_HEADER command
SET_KWD_INTERRUPT_PIN	uint8	WRITE	1	Set gpi pin index to receive kwd interrupt on
GET_KWD_INTERRUPT_PIN	uint8	READ	1	Read gpi pin index to receive kwd interrupt on

3.4 General Purpose Outputs

The following commands are available to write and control GPOs:

Table 3.3: General Purpose output commands

Command	Type	Args	Description
SET_GPO_PORT	uint32	2	Write a value to all pins of a GPO port. Arguments are <Port Index> <Value>
SET_GPO_PIN	uint8	3	Write to a specific GPO pin. Arguments are <Port Index> <Pin Index> <Value>
SET_GPO_PIN_ACTIVE_LEVEL	uint8	3	Set the active logic level for a specific GPO pin. Arguments are <Port Index> <Pin Index> <Level> - 0=active low - 1=active high. By default all GPO pins are active high
SET_GPO_PWM_DUTY	uint8	3	Set the PWM duty for a specific pin. Value given as an integer percentage. Arguments are <Port Index> <Pin Index> <Duty in percent>
SET_GPO_FLASHING	uint32	3	Set the serial flash mask for a specific pin. Each bit in the mask describes the GPO state for a 100ms interval. Arguments are <Port Index> <Pin Index> <Flash mask>

Note: All GPOs have a weak pull-down (~30kΩ) during reset, initialised to logic low on device boot and will always drive the pin thereafter.

To illustrate usage of the GPOs the following section considers four common examples. Writing to a GPO pin, configuring a PWM output, generating a blink sequence and driving a three colour (RGB) LED.

The following commands toggle OP_2 high then low (XVF3610-UA shown for example):

```
vfctrl_usb SET_GPO_PIN 0 2 1
vfctrl_usb SET_GPO_PIN 0 2 0
```

To set all GPOs high and then low:

```
vfctrl_usb SET_GPO_PORT 0 15
vfctrl_usb SET_GPO_PORT 0 0
```

The PWM runs at a fixed 500Hz frequency designed to minimise visible flicker when dimming LEDs and supports 100 discrete duty settings to permit gradual off to fully-on control.

The following commands illustrate setting individual PWM frequencies on each output by setting GPO pins 0, 1, 2 and 3 to output 25%, 50%, 75% and 100% duty cycles respectively:

```
vfctrl_usb SET_GPO_PWM_DUTY 0 0 25
vfctrl_usb SET_GPO_PWM_DUTY 0 1 50
vfctrl_usb SET_GPO_PWM_DUTY 0 2 75
vfctrl_usb SET_GPO_PWM_DUTY 0 3 100
```

Setting a pin duty to 100% is the same as setting that pin to high.

Each GPO is driven from the LSB of an internal 32bit register, which is rotated by one bit every 100mS.

The figure below shows how the blinking sequence works:

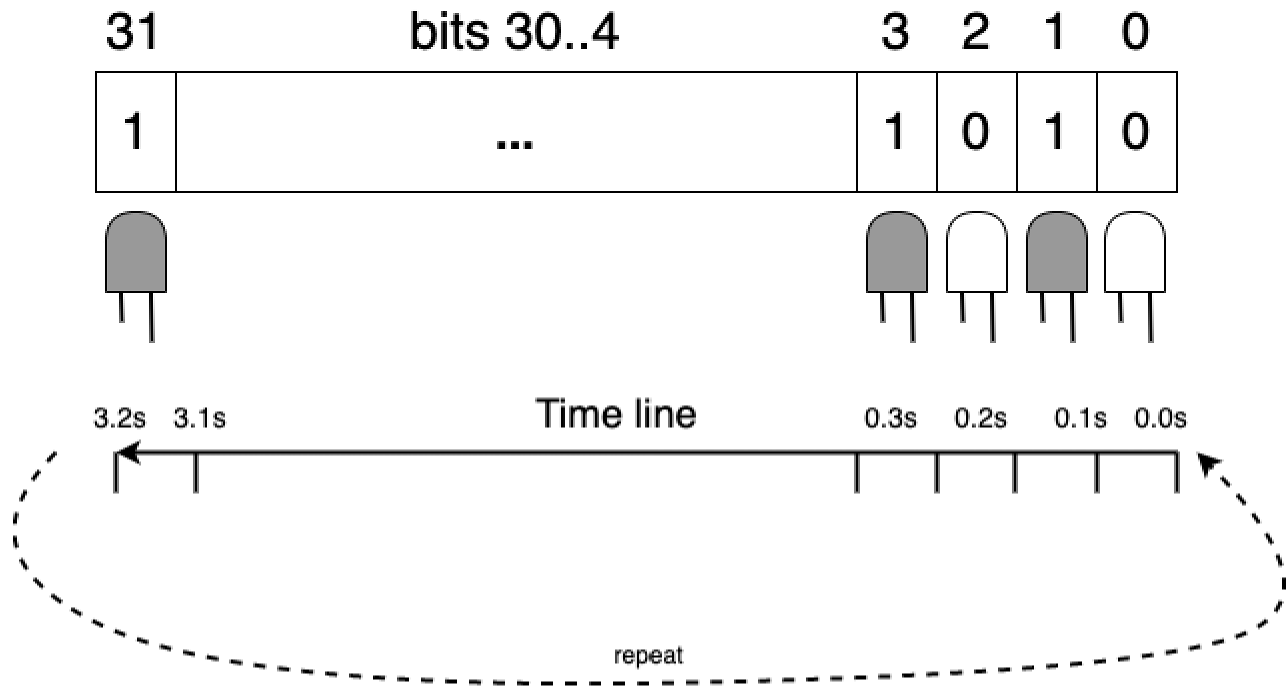

Fig. 3.2: Use of 32 bit word is used to define the blinking function of GPO

The following commands generate these LED effects:

- GPO pin 0 blinking, ON for 1.6 seconds, then OFF for 1.6 seconds, i.e. a period of 3.2 seconds;
- GPO pin 1 blinking, ON for 0.8 seconds, then OFF for 0.8 seconds, i.e. a period of 1.6 seconds;
- GPO pin 2 blinking, ON for 0.1 seconds, then OFF for 0.1 seconds, i.e. a period of 0.2 seconds;

```
vfctrl_usb SET_GPO_FLASHING 0 0 4294901760 # equivalent to pattern: xFFFF0000
vfctrl_usb SET_GPO_FLASHING 0 1 4042322160 # equivalent to pattern: xFF00FF00
vfctrl_usb SET_GPO_FLASHING 0 2 2863311530 # equivalent to pattern: xAAAAAAAA
```

Note: A GPO pin can be set to both a PWM duty cycle, and to flashing by issuing both a GPO_SET_PWM_DUTY instruction and a SET_GPO_FLASHING instruction for the same port and pin.

Where RGB LEDs are connected to three GPO pins (0 = Red, 1 = Green, 2 = Blue) automated colour sequencing can be programmed. For example, to colour cycle between Red-Yellow-Green-Cyan-Blue every 3.2 seconds:

```
vfctrl_usb SET_GPO_FLASHING 0 0 65535 # 0 x0000FFFF
vfctrl_usb SET_GPO_FLASHING 0 1 16776960 # 0 x00FFFF00
vfctrl_usb SET_GPO_FLASHING 0 2 4294901760 # 0 xFFFF0000
```

3.5 I²C Master peripheral interface (XVF3610-UA Only)

The XVF3610-UA variants provide an I²C master interface which can be used as:

- A bridge from the USB interface, i.e. VFCTRL_USB commands can be used from the host to read and write devices connected to the I²C Peripheral Port;
- A mechanism to initialise devices connected to the I²C Peripheral Port by incorporating commands into the Data Partition (in the external flash), which are executed at boot time.

The interface supports:

- 100kbps fixed speed
- 7bit addressing only
- Byte I²C register read/writes

The following table shows the commands for the configuration of the I^2C Master interface:

Table 3.4: I^2C peripheral interface commands

Command	Type	Num args	Num values	Definition
SET_I2C_READ_HEADER	uint8	3	0	Set the parameters to be used by the next GET_I2C or GET_I2C_WITH_REG command. Arguments: 1: The 7-bit I^2C slave device address. 2: The register address within the device. 3: The number of bytes to read.
GET_I2C_READ _HEADER	uint8	0	3	Get the parameters to be used by the next GET_I2C or GET_I2C_WITH_REG command. Returned values: 1: The 7-bit I^2C slave device address. 2: The register address within the device. 3: The number of bytes to read.
GET_I2C	uint8	0	56	Read from an I^2C device defined by the SET_I2C_READ_HEADER command. Returned values: 1 to 56: The number of bytes read as defined by the SET_I2C_READ_HEADER command followed by additional undefined values. The number of bytes read from the I^2C device when executing GET_I2C is set using SET_I2C_READ_HEADER.
GET_I2C_WITH_REG	uint8	0	56	Read from the register of an I^2C device as defined by the SET_I2C_READ_HEADER command. Returned values: 1 to 56: The number of bytes read as defined by the SET_I2C_READ_HEADER command followed by additional undefined values. The number of bytes read from the I^2C device when executing GET_I2C is set using SET_I2C_READ_HEADER.
SET_I2C	uint8	56	0	Write to an I^2C slave device. Arguments: 1: The 7-bit I^2C slave device address. 2: The number of data bytes to write (n). 3 to 56: Data bytes. All 54 values must be given but only n will be sent.
SET_I2C_WITH_REG	uint8	56	0	Write to a specific register of an I^2C slave device. Arguments: 1: The 7-bit I^2C slave device address. 2: The register address within the device. 3: The number of data bytes to write (n). 4 to 56: Data bytes. All 53 values must be given but only n will be sent.

The figures below shows the signals and messages for reading and writing registers. Raw I²C read/writes may be performed.

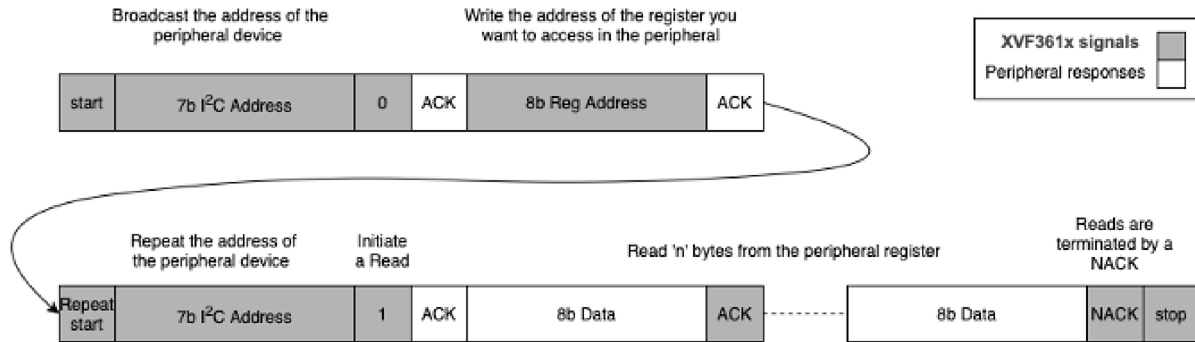

Fig. 3.3: I²C protocol for register reads

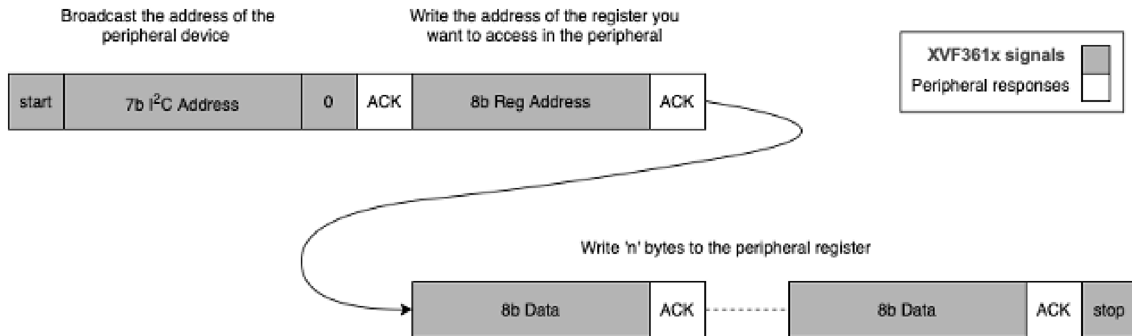

Fig. 3.4: I^{2}C protocol for register writes

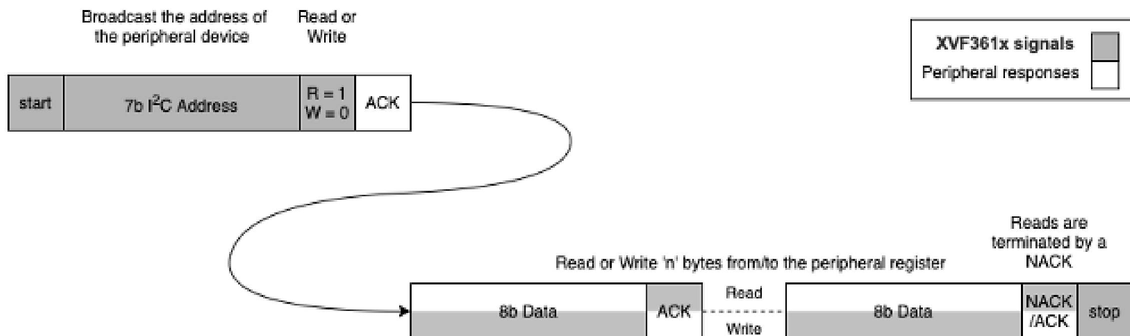

Fig. 3.5: I^{2}C protocol for raw reads and writes

3.6 I²C Slave Control interface (XVF3610-INT only)

The XVF3610-INT implements an I2C slave interface for Control and Setup of the device. The interface conforms to the following specifications.

Table 3.5: I²C interface specification

Specification	Value
Maximum I²C operation speed	100kbps
I²C Slave Address	0x2C

3.7 Using I²C Master to write to a device

Typically byte register read/writes are used to configure external I²C controlled hardware.

As an example, assume there is a device connected at address 0x40 (64) with three, single byte, registers. The following commands will write 77 to register 0, 48 to register 1 and 33 to register 2.

```
vfctrl_usb SET_I2C_WITH_REG 64 0 1 77 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vfctrl_usb SET_I2C_WITH_REG 64 1 1 48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vfctrl_usb SET_I2C_WITH_REG 64 2 1 33 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Note: The control protocol does not support variadic (variable number of) arguments. Hence, even when writing a single byte, the full number of arguments must be passed. Unwritten values are ignored.

3.8 Using the I²C master to read from a device

To verify the previous I²C register write to register number 0 at address 0x40 (64), an I²C register read can be performed as follows:

```
vfctrl_usb SET_I2C_READ_HEADER 64 0 1
vfctrl_usb GET_I2C_WITH_REG
> 77 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The byte read is the first of the 56 return values, which in this case, is 77. The following 55 values are undefined since the command only performed a read of 1 register.

3.9 SPI Master

The XVF3610-UA and XVF3610-INT variants provide an SPI master interface which can be used as:

- A bridge from the USB interface, i.e. *vfctrl_usb* commands can be used from the host to read and write devices connected to the SPI Peripheral Port; and
- A mechanism to initialise devices connected to the SPI Peripheral Port by incorporating commands into the Data Partition (in the external flash), which are executed at boot time.

Note: From Version 4.1 the SPI Master peripheral interface is not available on XVF3610-UA and XVF3610-INT devices that have been SPI booted to prevent possible bus contention issues.

The SPI master peripheral supports the following fixed specifications:

- Single chip select line
- 1Mbps fixed clock speed
- Supports either reads or writes. Duplex read/writes are not supported.
- Most significant bit transferred first
- Mode 0 transfer (CPOL = 0, CPHA = 0)

Note: The chip select is asserted a minimum of 20ns before the start of the transfer and de-asserted a minimum of 20ns after the transfer ends.

The SPI Master is controlled using the following commands.

Table 3.6: SPI peripheral interface commands

Command	Type	Args	Description
GET_SPI	uint8	56	Gets the contents of the SPI read buffer.
GET_SPI_READ_HEADER	uint8	2	Get the address and count of next SPI read.
SET_SPI_PUSH	uint8	56	Push SPI command data onto the execution queue.
SET_SPI_PUSH_AND_EXEC	uint8	56	Push SPI command data and execute the command from the stack. Data will then be sent to SPI device.
SET_SPI_READ_HEADER	uint8	2	Set address and count of next SPI read.

Reads of up to 56 Bytes at a time may be performed but writes of 128 Bytes at a time can be made by pushing multiple commands into a command stack and executing them in one go. The transaction is performed within a single chip select assertion.

Fig. 3.6: SPI peripheral, read sequence

Fig. 3.7: SPI peripheral, write sequence

The control protocol does not support variadic (variable number of) arguments. Hence, even when writing a single byte, the total number of arguments passed must be the maximum. Unwritten values are ignored.

See below examples.

The following example writes one byte of data (with value 122) to a control register as address 6.

```
vfctrl_i2c SET_SPI_PUSH_AND_EXEC 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 1 122
```

Note: All numbers are decimal. It is necessary to pad the payload to 56 bytes, which includes the address, length and data values. This is a requirement of the *vfctrl* tool, the SPI interface itself will only transmit the valid data.

Transmitting more than 54 bytes of data is possible using the SET_SPI_PUSH command to queue up data, using multiple commands before the push is executed. The following example writes values 0 to 69 to address 100 (70 bytes in total) using command to push 56 data values into the queue, followed by a push the remaining 14 data words and then execute the transfer:

```
vfctrl_i2c SET_SPI_PUSH 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34␣
↪33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1␣
↪0
vfctrl_i2c SET_SPI_PUSH_AND_EXEC 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0␣
↪0 0 0 0 0 0 0 0 0 0 0 100 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56
```

To read one byte at address 6, which contains the value 122, do the following:

```
vfctrl SET_SPI_READ_HEADER 6 1
vfctrl GET_SPI
> GET_SPI: 122 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

To read 16 bytes from address 0, which all contain the value 33, do the following:

```
vfctrl SET_SPI_READ_HEADER 0 16
vfctrl GET_SPI
> GET_SPI: 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 0 0 0 0
-> 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

4 System Boot and Initial Configuration

The standard mechanism for booting the XVF3610 is from an attached QSPI Flash device. This provides standalone operation, and persistent storage for configuration data. The XVF3610 supports device firmware upgrade (DFU) over USB (UA product variants) and I^2C (INT product variant).

Pre-compiled host utilities, and source code for reference, are supplied for performing DFU operations.

In addition, when using flash to boot the XVF3610 processor, a Data Partition in the flash memory can be used to store commands which are executed immediately after boot-up to configure and define the functionality of the device.

An alternative boot mode, using a executable image supplied by a host processor over a SPI interface is also available for the XVF3610.

The following sections describe the boot process and the Data Partition, including customisation for specific applications.

4.1 Boot process

The standard mechanism for booting is from an attached QSPI Flash device. This provides standalone operation, and persistent storage for configuration data. VocalFusion® XVF3610 supports device firmware upgrade (DFU) over USB (UA product variants) and I^2C (INT product variant). Pre-compiled host utilities, and source code for reference, are supplied for performing DFU operations.

The following sections discuss the structure of data within the flash memory, and operation of DFU.

Warning: While the functionality of the DFU is similar to the USB DFU specification, it has diverged to accommodate both USB and I^2C operation and therefore is not compatible with compliant USB DFU tools.

4.2 Flash storage structure

The structure of data within the VocalFusion® XVF3610 is arranged to contain a factory image, a single upgrade image, device serial numbers and Data Partitions for both the factory and upgrade image. This is shown below.

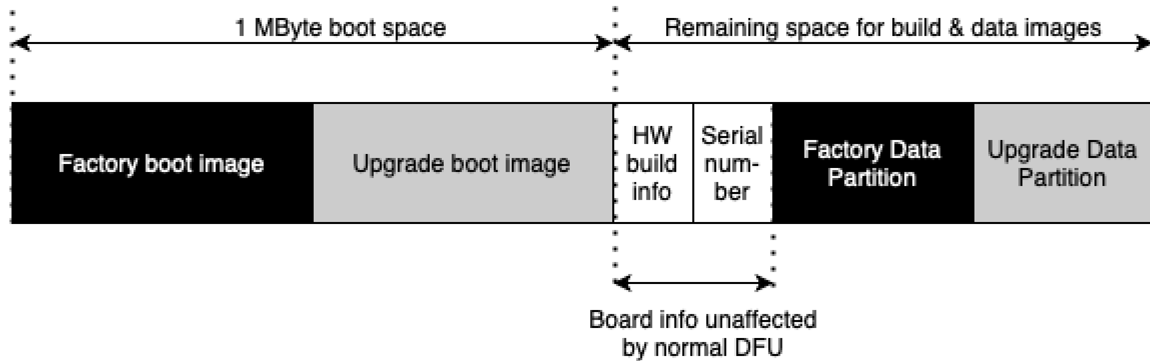

Fig. 4.1: Flash data structure for VocalFusion® XVF3610

- The **factory boot image** is the executable code for VocalFusion® and is supplied in the Release Package in the bin directory. The file format is xe, which refers to XMOS Executable. This is written to the device via the XTAG debugger or through a bulk flash programming operation.
- The **upgrade boot image**, if present, is the executable code written to the flash memory via a DFU operation. Generation of the upgrade boot image is covered below.
- The **HW build info** is specified in the .json Data Partition file for the factory image and is written at the same time as the factory image and Data Partition. It is a unique identifier which is unaffected by subsequent DFU upgrade operations.
- The **Serial Number** is a custom field which can be programmed via USB and I^2S control interfaces and remains untouched by the subsequent DFU operations.
- The **Factory and Upgrade Data Partitions** are the associated Data Partitions for the Factory and Upgrade images (where upgrade is present). They are written to flash in the same operation as the boot images. For more information on the generation and usage of Data Partitions see the *Configuration and the Data Partition* section.

Warning: Storage of only a single upgrade boot image and Data Partition pair is supported. Therefore, any Upgrade image applied will overwrite any existing upgrade image present.

A summary of the factory programming and field update process for flash-based systems is shown in the *reference* section.

4.3 Programming the Factory Boot image and Data Partition

The XVF3610 Voice Processor is provided in two pre-compiled builds (UA and INT) and as such only requires the usage of the XTC Tools programming tools, specifically xFLASH. This operates as a command-line application, to create the boot image, and if using flash, program the boot image to the attached device.

An XTAG debugger must be connected to the XVF3610 for flash programming operations. Refer to the Development Kit User Guide for information on using XTAG connections to the XVF3610 development kits.

The basic form of the xFLASH command for flash image creation and programming with a data partition is as follows.

```
xflash --boot-partition-size 0x100000 --factory [Application executable (.xe)] --data [Data␣
↪partition description (.bin)]
```

where

- Application executable (.xe) The .xe file is a boot image provided with a VocalFusion® release package in one of the supported configurations (UA or INT product variants).
- Data partition description (.bin) The .bin file is a Data Partition description either supplied in the release package (UA or INT) or customised as described later in this section.

Note: Boot over SPI from a host processor uses a specific image which is supplied in the release package. No data partition is included as configuration command are assumed to be supplied by the host controller used.

4.4 Upgrade Images and Data Partitions

In order to be able to apply an Upgrade image to the device it must be programmed with a Factory Image and Data Partition.

The Device Firmware Update (DFU) process requires the use of two utilities, `dfu_usb` or `dfu_i2c`, depending on the firmware variant, and `dfu_suffix_generator`.

Precompiled versions of these utilities are provided as part of the Release Package in the appropriate platform directory in `\host` (eg. `\host\Win32\bin`), and the source code for the DFU utility is provided in the `\host\src\dfu` directory.

For more information on building the host applications refer to the build instructions in `\host\how_to_build_host_apps.rst` in the Release Package.

In addition to the DFU utilities, the Upgrade image and Data Partition are required. These are provided in the Release Package in the `\bin` and `\data-partition\images` directories.

Generation of custom Data Partitions is detailed in the *Data Partition* section.

There are a number of stages required to prepare and execute a DFU to ensure a safe and successful update. These are detailed in the next section.

4.5 Generation of Binary Upgrade image

The Upgrade Image (.xe) needs to be converted to a binary format. Use `xflash` and the following command to convert the .xe image into a binary form:

```
xflash --noinq --factory-version 15.0 --upgrade [UPGRADE_VERSION] [UPGRADE_EXECUTABLE] -o␣
↪[OUTPUT_BINARY_NAME]
```

Specify `--factory-version` value of 15.0 for all 15.x.x releases of the XTC tools. (The 15.0 value refers to boot loader API for the XTC tool chain).

Note: Should a different version of the XTC tools be used in a future firmware release, the version number should be noted such that an update image of compatible format can be created.

The upgrade version number is specified with `--upgrade <version>`. Use the 16-bit format``0xJJMP`` where

- J is major
- M is minor
- P is patch

For example, to create an upgrade Binary image for a UA system, from the v5.7.2 Release Package use the following command:

```
xflash --noinq --factory-version 15.0 --upgrade 0x0570 app_xvf3610_ua_v5.7.2.xe -o app_
↪xvf3610_ua_v5.7.2.bin
```

4.6 Addition of DFU Suffix to Binary files

To prevent accidental upgrade of an incompatible image both the binary Upgrade image and the Data Partition binary must be signed using the provided `dfu_suffix_generator` which can be found pre-compiled in the host platform directory of the release package eg. `/host/MAC/bin`.

This mechanism embeds a structure into the binary files which can be read by the Device Firmware Update (DFU) tool to check that the binary data is appropriate for the connected device, prior to executing.

The general form of usage for the `dfu_suffix_generator` is as follows:

```
dfu_suffix_generator VENDOR_ID PRODUCT_ID [BCD_DEVICE] BINARY_INPUT_FILE BINARY_OUTPUT_FILE
```

`VENDOR_ID`, `PRODUCT_ID` and `BCD_DEVICE` are non-zero 16bit values in decimal or hexadecimal format, with the value of `0xFFFF` bypassing verification of this field.

When building Upgrade images for XVF3610-UA devices, the USB Vendor Identifier (VID) and USB Product Identifier (PID) are added to the header and then checked by the DFU utility to ensure that the connected device matches. An error is reported by the tool if there is no match with the connected device.

For XVF3610-INT devices both Vendor ID and Product ID fields should be set to `0xFFFF` for the generation of the Upgrade image and Data Partition binary. This instructs the DFU to bypass the checking as there is no equivalent to the USB identifiers for I2C systems. However, even though the checking is bypassed for the XVF3610-INT the suffix must be added to both Upgrade and Data partition files as the DFU utility checks the integrity of the binaries based on this information.

The following examples show how to add DFU Suffix to Update binaries for both XVF3610-INT and XVF3610-UA products.

For XVF3610-UA (default XMOS Vendor and XVF3610-UA product identifiers are used for illustration):

```
dfu_suffix_generator.exe 0x20B1 0x0016 app_xvf3610_ua_v<release_version>.bin boot.dfu
```

```
dfu_suffix_generator.exe 0x20B1 0x0016 data_partition_upgrade_ua_v<release_version>.bin␣
↪data.dfu
```

For XVF3610-INT:

```
dfu_suffix_generator.exe 0xFFFF 0xFFFF app_xvf3610_int_v<release_version>.bin boot.dfu
```

```
dfu_suffix_generator.exe 0xFFFF 0xFFFF data_partition_upgrade_int_v<release_version>.bin␣
↪data.dfu
```

Warning: Extreme care must be taken if modifying the default Vendor ID or default Product ID through a Data Partition. If configuration from Data Partition fails, the USB VID and PID will remain at their default values (VID=0x20B1, PID=0x0016) and DFU requests with the modified ID's will not be allowed.

4.7 Performing Firmware Updates

The pre-compiled firmware update utility is provided in the Release Package in the host architecture directory eg. `/host/Linux/bin`. For MAC, Linux and Windows the `DFU_USB` is provided, and for Raspberry Pi `DFU_I2C` is provided. The source code can be used to rebuild either version on the required platform.

The general form of dfu_usb utility is as follows:

```
dfu_usb [OPTIONS] write_upgrade BOOT_IMAGE_BINARY DATA_PARTITION_BIN
```

OPTIONS: `--quiet`

`--vendor-id` 0x20B1 (default)

`--product-id` 0x0016 (default)

`--bcd-device` 0xFFFF (default)

`--block-size` 128 (default)

and the general form of the `dfu_i2c` utility is shown below:

```
dfu_i2c [OPTIONS] write_upgrade BOOT_IMAGE_BINARY DATA_PARTITION_BIN
```

OPTIONS: `--quiet`

`--i2c-address` 0x2c (default)

`--block-size` 128 (default)

The two binary files passed to the utility, the boot image and Data Partition, must have the DFU suffix present otherwise the DFU utility will generate an error. Example DFU utility usage is shown for both XVF3610-UA and XVF3610-INT below.

For XVF3610-UA:

```
dfu_usb --vendor-id <USB_VENDOR_ID> --product-id <USB_PRODUCT_ID> write_upgrade boot.dfu␣
↪data.dfu
```

where the default values of `<USB_VENDOR_ID>` and `<USB_PRODUCT_ID>` are 0x20B1 and 0x0016, and for XVF3610-INT:

```
dfu_i2c write_upgrade boot.dfu data.dfu
```

Once complete the following message will be returned and the device will reboot. In the case of XVF3610-UA the device will re-enumerate.

```
write upgrade successful
```

For verification that DFU has succeeded as planned, the *vfctrl* utility can be used to query the firmware version before and after update. For example, to query the version of XVF3610-UA the following command is used:

```
vfctrl_usb GET_VERSION
```

Note: The *vfctrl* utilities check the version number of the connected device to ensure correct operation. To suppress an error caused by a disparity in the version of *vfctrl* and upgraded firmware the `--no-check-version` option can be used with the utility.

4.8 Factory restore

To restore the device to its factory configuration, effectively discarding any upgrades made, the same process as outlined above is followed but using a blank Boot Image and Data Partition.

This is the only way a restore can be initiated as the device does not have the ability to restore itself.

The same blank file can be used for both Boot Image and Data partition and can be generated using `dd` on MAC and Linux, and `fsutil` on Windows. A blank image can be created with a file of zeroes the size of one flash sector. In the normal case of 4KB sectors on a UNIX-compatible platform, this can be created as follows:

```
dd bs=4096 count=1 < /dev/zero 2>/dev/null blank.dfu
```

and for Windows systems:

```
fsutil file createNew blank.dfu 4096
```

The process outlined in the *Generation Upgrade Image and Data Partition* section can now be followed using the blank.dfu file for both Boot Image and Data Partition.

4.9 Boot Image and Data Partition Compatibility checks

The format of Data Partitions and Boot Images may change between version increments. Therefore to prevent incompatible Boot and Data Partitions from running and causing undefined behaviour, a field called compatibility version is embedded into the Data Partition. A running Boot Image checks its own version, against the compatibility version in the Data Partition before reading the partition data.

The version of the firmware should also be specified in the `--upgrade` argument of `xflash` when generating the Upgrade Image as described previously.

If the compatibility check fails the booted image, which could be a factory image or an upgrade image, will not read the Data Partition and will operate with its default settings (described in Default Operation section above). The Boot status is reported in the `RUN_STATUS` register which can be accessed via the *vfctrl* utility, for example:

```
vfctrl_usb.exe GET_RUN_STATUS
```

Successful Boot status is reported by either `FACTORY_DATA_SUCCESS` or `UPGRADE_DATA_SUCCESS` depending on which Boot Image was executed.

If unsuccessful the device will revert to a fail-safe mode of operation. The `RUN_STATUS` register can be queried for further debug information. The full list of `RUN_STATUS` codes are described in the the reference section.

Note: Fail safe mode uses the default vendor ID of 0x20B1 (XMOS) and product ID of 0x0016. In this event, host needs to be equipped with the ability to locate USB device under different IDs.

4.10 Custom flash memory devices

The majority of QSPI flash devices conform to the same set of parameters which define the access and usage of flash devices. However, to support instances when the flash interface parameters are different, the following section explains how to define a custom flash interface.

Details of the flash device used to store the Boot Image and Data Partition data must be specified in both the factory Boot Image and in any Data Partition files to ensure successful Factory programming and the ability to execute a Device Firmware Upgrade (DFU) to upgrade the firmware.

4.10.1 Custom flash definition for factory programming

During the Factory programming procedure, using the XMOS XTAG debugger, the specification of the flash device is used to create the loader which is responsible for downloading the Boot Image from flash and to the device.

The flash specification is provided to `xflash`, as described in the *Updating the Firmware* section, using a `.spispec` file. A representative `.spispec` file, which supports the majority of QSPI flash devices and the Development Kits, is provided in the Release Package here:

`\data-partition\16mbit_12.5mhz_sector_4kb.spispec`

This is a text file and must be modified with any differing parameters. An example `.spispec` file is shown in the reference section.

4.10.2 Custom flash definition for Data Partition generation

The `.spispec` file must also be included in the Data Partition, along with the sector size, so that DFU operations can be executed correctly.

Warning: Due to the nature of the DFU function, it is critically important to test the execution of the DFU process in a target system prior to production manufacturing.

4.11 SPI Slave Boot

Both UA and INT configurations of XVF3610 have a SPI slave boot mode, in addition to the boot from flash mode. The SPI slave boot downloads the boot image in binary form. The release package includes the necessary SPI boot images for all the variants.

Following an SPI boot the XVF3610 will not read any Data Partition that may be present in flash memory. If the default values set in the SPI boot image must be updated, the necessary parameters must be configured using the *vfctrl* host application after the device is booted.

The following section illustrates the use of the SPI boot files provided in the release packages.

4.11.1 SPI Boot of XVF3610-INT

When the XVF3610-INT boots via SPI, some parameters must be configured using the *vfctrl_i2c* application. Two of these parameters are the divider between the MCLK_IN to PDM clock and the I^{2}S sample rate. These values must be set before the audio processing is started, and this can be done using a special configuration, called *delayed mode*.

When using the *delayed mode*, the BOOTSEL pin is released before the firmware image is transferred to the device. After the XVF3610-INT boots up, it checks the status of the BOOTSEL pin, if the pin is enabled it proceeds with initialising the audio pipelines, if it is not, it waits for some specific control messages before starting the audio processing. The messages to be used in *delayed mode* to start the audio processing and interfaces are:

- `SET_MIC_START_STATUS`
- `SET_I2S_START_STATUS`

Below you can find an example of booting in *delayed mode* using the XVF3610-INT Release Package available on the Raspberry Pi:

1. Using a terminal console on the Raspberry Pi, navigate to the location of the XVF3610-INT Release Package.
2. Use the following command to transfer to the device the image of the XVF3610-INT firmware in the Release Package (replacing vX_X_X with the appropriate version number):

   ```
   python3 host/Pi/scripts/send_image_from_rpi.py bin/spi_boot/app_xvf3610_int_spi_boot_
   ↪vX_X_X.bin --delay
   ```

   The device should be ready within 3 seconds.
3. Update the divider from input master clock to 6.144MHz DDR PDM microphone clock using the Control Utility *vfctrl_i2c*:

   ```
   ./host/Pi/bin/vfctrl_i2c SET_MCLK_IN_TO_PDM_CLK_DIVIDER 1
   ```

4. Update the I^{2}S rate if necessary, default value is 48000 Hz, using the Control Utility *vfctrl_i2c*:

   ```
   ./host/Pi/bin/vfctrl_i2c SET_I2S_RATE 16000
   ```

5. Configure any system specific settings using the Control Utility *vfctrl_i2c*.
6. Start the audio processing and interfaces by issuing the following commands over the VocalFusion® control utility:

   ```
   ./host/Pi/bin/vfctrl_i2c SET_MIC_START_STATUS 1
   ```

   ```
   ./host/Pi/bin/vfctrl_i2c SET_I2S_START_STATUS 1
   ```

4.11.2 SPI Boot of XVF3610-UA

Using the XVF3610-UA Release Package available on the Raspberry Pi, a SPI boot can be executed by following the steps below:

1. Using a terminal console on the Raspberry Pi, navigate to the location of the XVF3610-UA Release Package.
2. Use the following command to execute the SPI boot process booting the XVF3610-UA firmware in the Release Package (replacing vX_X_X with the appropriate version number):

   ```
   python3 host/Pi/scripts/send_image_from_rpi.py bin/spi_boot/app_xvf3610_ua_spi_boot_vX_
   ↪X_X.bin
   ```

The device should be ready within 3 seconds.

Note: The *delayed mode* is not available for XVF3610-UA. This means that settings pertaining to the USB interfaces, such as sample rates, bit widths, HID report descriptor and endpoint descriptor, cannot be modified.

4.11.3 Implementing a SPI Boot host application

The SPI boot process shown in the diagram below should be adhered to:

Read SPI boot image to buffer

SPI Mode 0
SPI CLK 5MHz
BOOTSEL = H
RES_N = L

Max SPI clock depends on xcore.ai core clock
SPI_CLK<=85MHz

This step forces xcore to reset

RES_N = H

Pause

Wait for xcore.ai to exit reset.
2mS

SPI_CS_N = L

Send buffer

First Block sent

1st stage boot-loader executing from xcore.ai Boot ROM loads the 2nd stage loader contained in the first block.

Pause

Wait for PLL and network setup.
100 mS
Note - setting a slow application clock in the XN file will dictate the speed of the SPI interface from this point on.

Send next block

Pause

Inter-block delay
1mS

Last block?

No

Yes

Tri-state SPI_CLK, SPI_CS_N and SPI_MOSI

Fig. 4.2: SPI Boot process and timing requirements

Note: The phase locked loop (PLL) and netword setup needs enough time to settle after sending the first block over SPI.

4.12 Configuration and the Data Partition

When using flash to boot the XVF3610 processor, the Data Partition can be used to store commands which are executed immediately after boot-up to configure and define the functionality of the device. The following sections describe the definition of the Data Partition, how to generate, and the customisation for specific applications.

4.12.1 Data Partition file structure

The contents of a Data Partition are defined in a .json file which is passed to a generation script which forms the binary files used when flashing the device. The generation process is described below, after the definition .json file is described.

For the purpose of explanation consider the following example for a custom XVF3610-UA Data Partition:

```
{
  "comment": "Example data partition definition",
  "spispec_path": "16mbit_12.5mhz_sector_4kb.spispec",
  "regular_sector_size": "4096",
  "hardware_build": "0xFFFFFFFF",
  "item_files": [
      { "path": "input/usb_to_device_rate_48k.txt", "comment": "" },
      { "path": "input/device_to_usb_rate_48k.txt", "comment": "" },
      { "path": "input/usb_mclk_divider.txt", "comment": "" },
      { "path": "input/xmos_usb_params.txt", "comment": "" },
      { "path": "input/i2s_rate_16k.txt", "comment": "" },
      { "path": "input/led_after_boot.txt", "comment":"" }
  ]
}
```

Comment pairs are provided for the .json configuration, but also the individual item files:

```
{ "comment": "Example Comment" }
```

A running VocalFusion® device needs to know size and geometry of its external QSPI flash in order to write firmware upgrades to it. This is added to a Data Partition in the form of a flash specification or SPI specification.

```
{ "spispec_path": "16mbit_12.5mhz_sector_4kb.spispec" }
```

The Data Partition generation process aligns various sections onto flash sectors, and needs to know the sector size (this can be found in the flash device datasheet):

```
{ "regular_sector_size": "4096" }
```

Hardware build is a custom-defined, 32bit identifier written to flash along with the application firmware. It can be used to define a unique identifier for the hardware revision or other information which cannot be overwritten by subsequent updates:

```
{ "hardware_build": "0xFFFFFFFF" }
```

Item files which contain the commands to execute (format of item files described below). An optional comment field is provided:

```
{ "path": "input/usb_to_device_rate_48k.txt", "comment": "" }
```

Note: Because the generator is a Python script, the paths use forward slashes irrespective of platform.

4.12.2 Item files

The item files contain the commands used to configure the system. The commands are simply added to the file in the same format as the command line control utility. For clarity, multiple item files can be included in the .json definition, each specifying a sub-set of commands relating to a particular function or aspect. Example item files for common configurations are provided in the `data-partition/input` directory of the release package. For example, the `agc_bypass.txt` item file bypasses the AGC for both output channels and contains the following commands:

```
SET_ADAPT_CH0_AGC 0
SET_ADAPT_CH1_AGC 0
SET_GAIN_CH0_AGC 1
SET_GAIN_CH1_AGC 1
```

4.12.3 Generating a Data Partition for custom applications

It is recommended that in order to create a custom Data Partition, an existing set of .json and item files is used as a template and modified as required. The release package contains example .json and item files for this purpose.

The required additional control commands should be stored in an appropriately named text file inside the `data-partition/input` subdirectory. For example, a file named `aec_bypass.txt` could be added containing the collected commands:

```
SET_BYPASS_AEC 1
```

Note: Only commands which are required to be set with non-default values need to be included in the item file list.

These text files are then included in the custom JSON description.

In the above example, the `aec_bypass.txt` is added to a JSON description, `bypass_AEC.json` as shown below:

```
...
"item_files": [
  ...
  {
    "path": "input/aec_bypass.txt",
    "comment": ""
  }
```

(continues on next page)

(continued from previous page)

```
    ...
]
...
```

Note: The execution order of the commands and input files can affect the behaviour of the device. Commands to configure USB and I^{2}S should be added at the beginning of the data image.

Finally, to generate the custom data partition, the command below should be run from the data-partition directory:

```
python3 xvf361x_data_partition_generator.py <build_type>.json
```

The generator script produces two data image files; one for factory programming and one for device upgrade in a directory named `output`.

For the above example these files will be called:

```
data_partition_<build_type>_factory_v<release_version>.bin
```

and

```
data_partition_<build_type>_upgrade_v<release_version>.bin
```

These two binary files can be used to factory program or upgrade as described in Updating the firmware and *Generation of Upgrade Image* sections respectively.

A .json file is also produced for debugging purposes.

5 Device operation

To facilitate control of the XVF3610 and to allow the specification of the default behaviour, the XVF3610 implements two mechanisms for control and parameterisation. The first is the Control Interface which is a direct connection between the host and the XVF3610 and is operational at runtime. The second is the Data Partition which is held in flash and contains configuration data to parameterise the XVF3610 on boot up. Both mechanisms can be used by a host application to control the behaviour of the device.

5.1 Host Utilities

There are seven host utilities provided in the XVF3610 Release Package as pre-compiled utilities and also as source code to allow rebuilding for other system architectures. The utilities are summarised below:

vfctrl_usb, *vfctrl_i2c* - Vocal Fusion Control Utilities for the XVF3610-UA, XVF3610-INT respectively

data_partition_generator, *vfctrl_json* - Uses .json configuration definition and generates binary Data Partitions for download to flash memory. *vfctrl_json* is used internally by the *data_partition_generator* but is referenced here for completeness.

dfu_suffix_generator - Adds DFU suffix to binary Boot Images and binary Data Partitions to protect the device from accidental DFU of incompatible image partition pair.

dfu_usb, *dfu_i2c* - DFU utilities for XVF3610-UA, XVF3610-INT respectively

The pre-compiled versions are found in the following platform sub-directories within the 'host' directory of the Firmware Release Package:

- `./Linux` for Linux based systems
- `./MAC` for macOS
- `./Pi` for Raspbian based Raspberry Pi systems
- `.\Win32` for Windows platforms

Note: For cross-platform support *vfctrl_usb* uses *libusb*. While this is natively supported in macOS and most Linux distributions, it requires the installation of a driver for use on a Windows host. Driver installation should be done using a third-party installation tool like Zadig (https://zadig.akeo.ie/).

5.1.1 Building the host utilities from source code

The source code for these utilities is provided in the following directory:

`/host/src`

The steps to build each utility are described in the Release Package here:

`/host/how_to_build_host_apps.rst`

5.2 Command-line interface (vfctrl)

To allow command-line access to the control interface on the XVF3610 processor, the **vfctrl** (**V**ocal**F**usion **C**on**tr**o**l**) utility is provided as part of the release package.

Two versions of this utility are provided for control of the device (a third is used internally by the Data Partition generation process):

Table 5.1: vfctrl versions and platforms

Version	Function	Host platforms supported
vfctrl_usb	Control of XVF3610-UA over a USB interface	Windows - MacOS - Linux - Raspberry Pi OS
vfctrl_i2c	Control of XVF3610-INT over I^2C interface	Raspberry Pi OS

Source code for the utility is also provided for compilation for other host devices if required.

5.3 vfctrl Installation

Control and configuration of the XVF3610-UA are achieved via the control interface implemented over USB. A VocalFusion® Host Control application, *vfctrl_usb*, is provided pre-compiled and as source code for this purpose.

For cross-platform support *vfctrl_usb* uses *libusb*. While this is natively supported in macOS and most Linux distributions, it requires the installation of a driver for use on a Windows host. Driver installation should be done using a third-party installation tool like Zadig (https://zadig.akeo.ie/).

The following steps show how to install the libusb driver using Zadig:

1. Connect the XVF3610 board to the host PC using a USB cable.
2. Open Zadig and select *XMOS Control (Interface 3)* from the list of devices. If the device is not present, ensure Options -> List All Devices is checked.
3. Select *libusb-win32* from the list of drivers.
4. Click the *Reinstall Driver* button

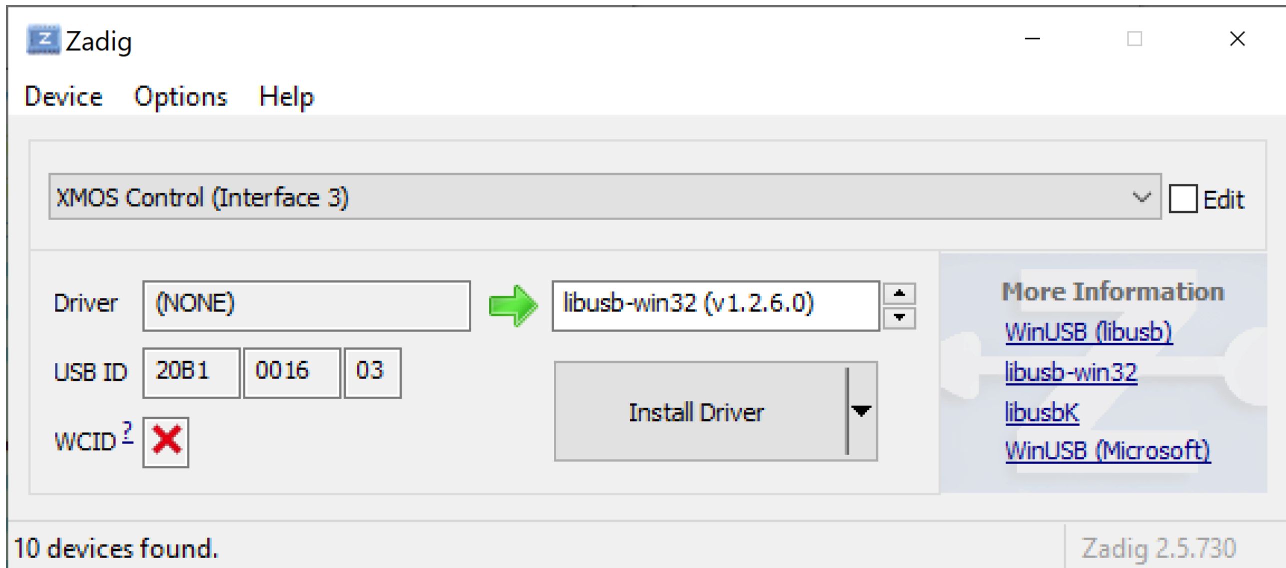

Fig. 5.1: Selecting the libusb driver in Zadig

Once installed the *vfctrl_usb* utility is ready to use. The following steps explain how to use the host control utility.

1. Copy the host directory of the Firmware Release Pack to the host platform.
2. Navigate, from a terminal window, to the copied host directory and execute one of the commands, depending on the specific platform, as described in the next section.

5.4 vfctrl syntax

Note: The examples below are written for the UA variants, if using the INT variant, please replace *vfctrl_usb* with *vfctrl_i2c*

The general syntax of the command line tool, when used for device control, is as follows:

```
vfctrl_usb <COMMAND> [ arg 1] [arg 2]....[arg N] ['# Comment']
```

The `<COMMAND>` is required and is used to control the parameters of the device. Each command either reads or writes parameters to the XVF3610 device. Commands that read parameters begin with `GET_`. Commands that write parameters begin with `SET_`.

Note: The *<COMMAND>* verb is case insensitve, e.g. `GET_VERSION` and `get_version` are equivalent and both supported

The available commands are described in detail in the *command reference section* of the user guide, and a summary table of all the parameters is provided.

If the `<COMMAND>` is a `GET_` command, the output of the operation is printed to the terminal as in the example below:

```
vfctrl_usb GET_GPI
GET_GPI: 13
```

The number and type of arguments, [arg 1]..[arg N], depend on the command and these are detailed in the command tables. All arguments are integer or floating-point numbers separated by a space. All the values are transferred to the device as integers and the host utility converts the floating-point values to the appropriate Q format.

The specification of the Q format for representing floating-point numbers is given in *Q format conversion section* section of the User Guide.

A secondary form of *vfctrl_usb* is also available which provides information for developers:

```
vfctrl_usb [options]
```

Where [options] can be:

```
-h, --help : Print help menu

-H, --help-params: Print help menu and the list of all available commands

-d, --dump-params : Print the values of the parameters configured in the device

-n, --no-check-version : Do not check version of firmware image

-f, --cmd-list <filename> : Execute the commands in the given <filename>
```

Note: For the last option the filename should be a text file with one command per line. The format is the same as the data partition generator input files.

5.5 Configuration via Control interface

The XVF3610 Voice Processor contains parameters which can be read and written by the host processor at run time. For information about writing parameters at boot time for initial configuration, please see *Configuration via Data Partition*

The XVF3610 firmware is provided as two pre-compiled builds, UA and INT, which provide a parameter control mechanism over USB endpoint 0 and I^{2}C respectively.

Device functions have controllable parameters for the audio pipeline, GPIO, sample rate settings, audio muxing, timing and general device setup and adjustment. Commands support either read using the `GET_` prefix or write using the `SET_` prefix. Controllable parameters may either be readable and writeable, read-only or write-only. Various data types are supported including signed/unsigned integer of either 8b or 32b, fixed point signed/unsigned and floating-point.

In addition, the UA and UA-HYBRID builds include volume controls for input (processed microphone from XVF3610). The UA build has similar controls for the output (far-end reference signal) too. These are USB Audio Class 1.0 compliant controls and are accessed via the host OS audio control panel instead of the XVF3610 control interface. The volumes are initialised to 100% (0dB attenuation) on device power up, which is the recommended setting.

It is recommended that the USB Audio input and output volume controls on the host are set to 100% (no attenuation) to ensure proper operation of the device. Some host OS (eg. Windows) may store volume setting in between device connections.

For a comprehensive list of parameters, their data types and an understanding of their function within the device please consult the User Guide section relevant to the function of interest, or the command list in the *command reference section* of this guide which summarises all the commands.

The full list of commands can also be obtained through the use of the -H or –help-params option of the control utility.

```
vfctrl_usb --help-params
```

This dumps a list of commands to the console along with a brief description of the function of each command. The remainder of this section will cover the generic operation of the control interface.

5.5.1 Control operation

The control interface works by sending a message from the host to the control process within the XVF3610 device. The time required to execute commands can vary, but most will respond within 30ms. Since the commands are fully acknowledged, by design, the control utility blocks until completion. This interface is designed to allow real-time tuning and adjustment but may stall due to bus access or data retrieval.

The control interface consists of two parts, a host side application and the device application. These are briefly summarised below.

5.5.2 Host Application

The example host applications, found in the `/host` directory in the Release Package, are command-line utilities that accept text commands and, in the case of a read, provides a text response containing the read parameter(s). Full acknowledgement is included in the protocol and an error is returned in the case of the command not being executed properly or handled correctly by the device.

Example host source code and makefiles are provided in the release package for x86 Linux, ARM Linux (Raspberry Pi), Windows and Mac platforms along with pre-compiled executables to allow fast evaluation and integration. For more information refer to the *Building the host utilities from source code* section.

5.5.3 Device Application

The device is always ready to receive commands. The device includes command buffering and an asynchronous mechanism which means that Endpoint 0, NACKing for USB or clock stretching for I^2C is not required. This simplifies the host requirements particularly in the cases where clock stretching is not supported by the host I^2C peripheral.

5.6 Configuration via Data Partition

The XVF3610 device flash firmware configuration comprises a Boot image and a Data Partition.

- The **Boot image** in the form of an .xe archive is the executable code. It is provided as part of the XVF3610-UA or XVF3610-INT Release Package. This configures the underlying operation of the device.
- The **Data Partition** configures a running Boot image instance at startup with a set of commands which are customisable for the specific application. This contains any command that can be issued at run-time via USB or I^2C, plus some more that are boot-time only. Pre-configured Data Partitions are supplied in the release packages for default operation.

This combination of Boot image and Data Partition allow the functionality of the processor to be configured and defined without requiring any modification or recompilation of base firmware. The commands stored in the Data Partition are executed at startup redefining the default operation of the device. More information about the data partition can be found in *Configuration and the Data Partition*.

6 USB Interface - (XVF3610-UA and XVF3610-UA-HYBRID only)

The XVF3610-UA variants support a standard USB PHY interface which supports a UAC1.0 audio device and device control over USB.

In addition the product supports a standard USB HID (Human Interface Device) that can be used to signal the host device when events are detected on the XVF3610 device inputs.

6.1 USB Interface

The following section details aspects that relate to the USB interface configuration and usage. This section only pertains to the XVF3610-UA and XVF3610-UA-HYBRID variants of the processor.

The USB interface provides the host three end points:

- Adaptive USB Audio Class 1.0 endpoint for the transfer of far-field voice to the host (for both XVF3610-UA and XVF3610-UA-HYBRID variants) and AEC reference audio from the host (for XVF3610-UA only).
- Vendor Specific Control allowing the host to control and parameterise the processor.
- Human Interface Device (HID) interrupt endpoint to signal the detection of events which have occurred on the GPIOs.

The USB Audio interface supports class compliant volume controls on both the input (processed microphone from XVF3610) and output (AEC reference) interfaces. These controls are accessed via the host OS audio control panels. They are initialised to 100% (0dB attenuation) on boot and this is the recommended setting for normal device operation.

By default the device will enumerate with the VID and PID shown below, but these can be configured using the Data Partition.

Table 6.1: Default XVF3610 USB Identification

USB Identification	Value
Vendor Identification (VID)	0x20B1
Product Identification (PID)	0x0016

Warning: If XVF3610 users change the PID value in the data partition, they must also change the XMOS VID to their own VID to avoid clashes with other XMOS products.

The following section describes the parameters available to configure the USB interface behaviour.

6.2 USB Configuration

Due to the nature of the USB enumeration process, USB setup must be done using a Data Partition so that the configuration is complete prior to enumeration. The following table summarises the USB interface parameters which can be configured.

Table 6.2: USB configuration parameters

Command	Type	Arguments	Definition
GET_USB_VENDOR_ID SET_USB_VENDOR_ID	uint32	1	Get / set USB Vendor ID. See notes A and B
GET_USB_PRODUCT_ID SET_USB_PRODUCT_ID	uint32	1	Get / set USB Product ID. See notes A and B
GET_USB_BCD_DEVICE SET_USB_BCD_DEVICE	uint32	1	Get / set USB Device Release Number (bcdDevice). See notes A and B
GET_USB_VENDOR_STRING SET_USB_VENDOR_STRING	uint8	25	Get / set USB Vendor string. See notes A and B
GET_USB_PRODUCT_STRING SET_USB_PRODUCT_STRING	uint8	25	Get / set USB Product string. See notes A and B
GET_USB_SERIAL_NUMBER SET_USB_SERIAL_NUMBER	uint32	1	Get / set write-only register to set the behaviour of iSerialNumber field in USB descriptor. See notes A and B
GET_USB_TO_DEVICE_RATE SET_USB_TO_DEVICE_RATE	uint32	1	Get / set sampling frequency of USB reference from USB host. Default usb_to_device_rate is 48000 samples/sec. See notes A and B
GET_DEVICE_TO_USB_RATE SET_DEVICE_TO_USB_RATE	uint32	1	Get / set sampling frequency of audio output to USB host. Default device_to_usb_rate is 48000 samples/sec. See notes A and B
GET_USB_TO_DEVICE_BIT_RES SET_USB_TO_DEVICE_BIT_RES	uint32	1	Get / set bit depth of USB reference from USB host. Default usb_to_device_bit_res is 16 bits. See notes A and B
GET_DEVICE_TO_USB_BIT_RES SET_DEVICE_TO_USB_BIT_RES	uint32	1	Get / set bit depth of audio output to USB host. Default device_to_usb_bit_res is 16 bits. See notes A and B
GET_USB_START_STATUS SET_USB_START_STATUS	uint8	1	Get / set start USB flag. Set as 1 as the last USB item in Data Partition. See note A

A: Command supported for Data Partition use only

B: Command must occur before SET_USB_START_STATUS 1

6.3 USB HID interface

A Human Interface Device (HID) is an electronic device with an interface which a human can use for control. Examples include a Personal Computer with a keyboard and mouse or a consumer appliance with control knobs, push buttons or a voice interface.

The XVF3610-UA uses the HID interface to inform the host system of events which have occurred on the General Purpose Inputs (GPI). The following section describes the setup of the GPI HID triggers.

6.4 HID Report configuration

The XVF3610 can send a HID Report when a GPI pin detects an interrupt (logic edge transition event). The HID features are described below:

- The XVF3610 supports three HID Reports, one each for keyboard, consumer and telephony events
- Each HID Report supports detection and reporting of multiple events
- Control commands allow configurable mapping between each GPI pin and each non-reserved HID Report bit
- Control commands allow boot-time configuration of the meaning (USB HID usage) assigned to each non-reserved HID Report bit

Table 6.3: USB HID Report 1

USB HID Usage Page	Bit Byte	7	6	5	4	3	2	1	0
Keyboard	0	Rsvd	Rsvd	Rsvd	Rsvd	F24	F23	Rsvd	't'

Table 6.4: USB HID Report 2

USB HID Usage Page	Bit Byte	7	6	5	4	3	2	1	0
Consumer	0	Volume-	Volume+	Rsvd	Voice Command	Rsvd	Mute	AC Search	AC Stop

Table 6.5: USB HID Report 3

USB HID Usage Page	Bit Byte	7	6	5	4	3	2	1	0
Telephony	0	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	Phone mute	Hook switch

6.5 USB HID report format

A USB HID device describes each field in each HID Report by sending a HID Report Descriptor to the USB host when requested. Information sent to the USB host for each field establishes the field's contents and its method of operation. This information includes a code, known as the Usage ID, which defines the field's exact meaning. To allow for reuse of the limited number of code values across many different types of devices, the information also includes another code, known as the Usage Page ID, which qualifies the meaning of the Usage ID.

The XVF3610 defines a default Usage Page ID and Usage ID for each bit in the HID Report as seen in the default HID Report tables. It also supports changes to the Usage ID for each non-reserved bit at boot-time via the Data Partition. Changes to the Usage Page ID are not supported, and each byte in a HID Report pertains to a specific Usage Page ID as shown in the table below.

Table 6.6: USB HID report usage page

HID Report Id	Usage Page	Usage Page ID
1	Keyboard	7
2	Consumer Control	12
3	Telephony	11

To change the meaning of a bit in a HID Report, include the **SET_HID_USAGE_HEADER`** and **SET_HID_USAGE** commands in the Data Partition. For example, the following commands issued from the Data Partition change the meaning of byte 0 bit 1 in HID Report 2 from Application Control (AC) Search to Media Select Telephone:

```
vfctrl_usb SET_HID_USAGE_HEADER 2 0 1 12
vfctrl_usb SET_HID_USAGE 9 140
```

The **SET_HID_USAGE_HEADER** command establishes HID Report 2, byte 0, bit 1 as the target location for the subsequent operation. Its first, second and third arguments make the subsequent **SET_HID_USAGE** command target HID Report 2, byte 0 and bit 1, respectively. Its fourth argument specifies the Consumer Control Usage Page as the qualifier for the subsequent **SET_HID_USAGE** command. The **SET_HID_USAGE** command changes the meaning of the HID Report byte and bit targeted by the most recent **SET_HID_USAGE_HEADER** command, byte 0 and bit 1 in this example. Its first argument determines the number of bytes required to store the second argument. The value 9 specifies a one-byte value (less than 256). The number 10 specifies a two-byte value (less than 65536). The XVF3610 does not support length values greater than two bytes. The second argument to the **SET_HID_USAGE** command specifies the Usage ID to associate with the targeted HID Report byte and bit. In this example, the one-byte value 140 (0x8C) changes the meaning of HID Report 2, byte 0, bit 1 to Media Select Telephone.

Note: Changes to HID usage can occur only at boot-time via the Data Partition. The **SET_HID_USAGE** command has no effect when received after the boot process completes and USB operation begins.

6.6 HID report generation

The XVF3610 can send a HID Report when a GPI pin detects an interrupt (logic edge transition event). When interrupts are enabled using **SET_GPI_INT_CONFIG**, the HID Report generator automatically services the interrupt generating a new report. The HID generation features are listed below:

- The HID Report changes upon assertion (positive edge) or de-assertion (negative edge) of a GPI pin
- Each HID Report supports detection and reporting of multiple events
- Control commands allow configurable mapping between each GPI pin and each non-reserved HID Report bit
- Each GPI pin positive edge asserts a bit in a HID Report; each negative edge enables de-assertion of the bit; sending the HID Report to the USB host de-asserts each bit previously enabled for de-assertion
- When no event has occurred, depending on “set idle” configuration by the host, the XVF3610 will either reply with a de-assert report (default) or NAK (set to idle by the host)

Note: HID idle behaviour is platform-specific and rarely does the high-level application code have any control over the settings. Linux, for example, typically silences the devices by issuing an indefinite idle (NAK report if

no change). Other platforms such as MacOS, on the other hand, leave the device verbose by not issuing an idle (report always sent).

The HID Report generator requires configuration of each GPI pin mapped to a HID Report bit to generate interrupts on both edges.

The default mapping between GPI pins and HID Report bits is:

Table 6.7: GPI pins and HID Report bits mapping

GPI Pin	HID Report bit
0	F23
1	F24
2	None
3	None

Use the `SET_GPI_INT_CONFIG` command to configure a GPI pin that triggers a change in a HID Report. For example, the following command configures GPI pin 0 to generate interrupts on both edges, which enables the HID Report logic:

```
vfctrl_usb SET_GPI_INT_CONFIG 0 0 3
```

The first argument is a reserved value and should be set to 0. The second argument makes the command target pin IP_0. The third argument selects both edges for the interrupt.

Use the `SET_GPI_PIN_ACTIVE_LEVEL` command to configure the interpretation of signal edges for the GPI pin. For example, the following command configures GPI pin 0 so that the XVF3610 interprets a falling edge as the positive (asserting) edge and a raising edge the negative (de-asserting) edge:

```
vfctrl_usb SET_GPI_PIN_ACTIVE_LEVEL 0 0 0
```

The first argument is a reserved value and should be set to 0. The second argument makes the command target pin IP_0. The third argument configures the XVF3610 to treat IP_0 as active low (use 1 to configure as active high).

The Data Partition allows changes to the mapping of GPI pins to HID Report bits at boot-time using the `SET_HID_MAP_HEADER` and `SET_HID_MAP` commands. They may also be used with *vfctrl* to change the mapping after the boot process completes and USB operation begins. For example, the following commands change the mapping so that an interrupt on pin IP_1 results in the XVF3610 reporting a Voice Command instead of an F24:

```
vfctrl_usb SET_HID_MAP_HEADER 0 1
vfctrl_usb SET_HID_MAP 2 0 4
```

The `SET_HID_MAP_HEADER` command establishes the GPI pin for subsequent mapping operations. Its first argument is reserved and should be set to 0. Its second argument makes the subsequent `SET_HID_MAP` command target pin IP_1. The `SET_HID_MAP` command changes the association between the GPI pin targeted by the most recent `SET_HID_MAP_HEADER` command, IP_1 in this example, and the control bits in a HID Report. Its three arguments identify the HID Report and state the byte and bit within that report, to associate with the targeted GPI pin. In this example, HID Report 2, byte 0, bit 4 associates the Voice Command control bit with IP_1 as can be seen in the default HID Report table.

Note: The XVF3610 will ignore a `SET_HID_MAP` command that specifies a reserved bit in the HID Report.

6.7 Serial Number

The XVF3610 allows a 24 ASCII character long serial number to be stored in the external flash memory. This can be accessed using the VocalFusion Control application using the following commands. To write to the SERIAL_NUMBER field use:

```
vfctrl_usb SET_SERIAL_NUMBER "DEADBEEF"
```

and to read use:

```
vfctrl_usb GET_SERIAL_NUMBER
```

6.8 USB device enumeration

The XVF3610-UA additionally allows the Serial Number to be copied into the iSerialNumber field of the USB descriptor. As the host reads the USB descriptor on enumeration the command to copy the serial number must be present in the Data Partition. To illustrate this process the following commands must be incorporated into a Data Partition in the specified order (example assumes SERIAL_NUMBER field is already populated).

To set the USB configuration to use the serial number in the descriptor add the following lines, in this order, to the Data Partition:

```
SET_USB_SERIAL_NUMBER 1
```

To set the USB configuration, then to start enumeration:

```
SET_USB_START_STATUS 1
```

7 Reference information

7.1 Base vfctrl command list

The table below summarises the XVF3610 parameters which are programmable via the control interfaces or flash data partition. These parameters allow the setup of the XVF3610 processor's interfaces and tuning of the internal signal processing.

To aid quick reference of the key parameters the summary is split into two sections. This section details the most frequently used parameters which are required for interface configuration and basic control, and the second details *advanced parameters* which will not generally need to be modified.

The XVF3610 version 5.7 supports the following commands for configuration, control and diagnostics.

Group	Command	Number of Values	UA Default	INT Default
ADMIN	GET_HARDWARE_BUILD	1	-	-
	GET_RUN_STATUS	1	-	-
	GET_SERIAL_NUMBER	26	-	-
	SET_SERIAL_NUMBER	26	-	-
	GET_VERSION	1	v5.7.2	v5.7.2
AEC	GET_BYPASS_AEC	1	0	0
	SET_BYPASS_AEC	1	-	-
AGC	GET_ADAPT_CH0_AGC	1	1	1
	GET_ADAPT_CH1_AGC	1	1	1
	GET_GAIN_CH0_AGC	1	53.8407	71.1364
	GET_GAIN_CH1_AGC	1	422.3719	432.6752
AUDIO	GET_ADEC_ENABLED	1	0	0
	SET_ADEC_ENABLED	1	-	-
	GET_ADEC_MODE	1	-	-
	GET_ALT_ARCH_ENABLED	1	0	0
	SET_ALT_ARCH_ENABLED	1	-	-
	GET_BYPASS_IC	1	0	0
	SET_BYPASS_IC	1	-	-
	GET_BYPASS_SUP	1	0	0
	SET_BYPASS_SUP	1	-	-
	SET_DELAY_DIRECTION	1	-	-
GPIO	GET_GPI_PIN	1	0	0
	SET_GPI_PIN_ACTIVE_LEVEL	3	-	-
	GET_GPI_PORT	1	0	0
	GET_GPI_READ_HEADER	2	0 0	0 0
	SET_GPI_READ_HEADER	2	-	-
	SET_GPO_FLASHING	3	-	-
	SET_GPO_PIN	3	-	-
	SET_GPO_PIN_ACTIVE_LEVEL	3	-	-
	SET_GPO_PORT	2	-	-
	SET_GPO_PWM_DUTY	3	-	-

continues on next page

Table 7.1 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	SET_IO_MAP	2	-	-
	GET_IO_MAP_AND_SHIFT	24	-	-
	SET_OUTPUT_SHIFT	2	-	-
USB	GET_DEVICE_TO_USB_BIT_RES	1	16	32
	SET_DEVICE_TO_USB_BIT_RES	1	-	-
	GET_DEVICE_TO_USB_RATE	1	48000	48000
	SET_DEVICE_TO_USB_RATE	1	-	-
	GET_USB_PRODUCT_ID	1	22	22
	SET_USB_PRODUCT_ID	1	-	-
	GET_USB_PRODUCT_STRING	26	XVF3610 Voice Processor	XVF3610 Voice Processor
	SET_USB_PRODUCT_STRING	26	-	-
	SET_USB_SERIAL_NUMBER	1	-	-
	GET_USB_TO_DEVICE_BIT_RES	1	16	32
	SET_USB_TO_DEVICE_BIT_RES	1	-	-
	GET_USB_TO_DEVICE_RATE	1	48000	48000
	SET_USB_TO_DEVICE_RATE	1	-	-
	GET_USB_VENDOR_ID	1	8369	8369
	SET_USB_VENDOR_ID	1	-	-
	GET_USB_VENDOR_STRING	26	XMOS	XMOS
	SET_USB_VENDOR_STRING	26	-	-

7.2 Advanced vfctrl command list

The XVF3610 version 5.7 supports the following additional commands for advanced configuration and diagnostics.

Group	Command	Number of Values	UA Default	INT Default
ADMIN	GET_BLD_HOST	30	-	-
	GET_BLD_MODIFIED	6	-	-
	GET_BLD_MSG	50	-	-
	GET_BLD_REPO_HASH	7	-	-
	GET_BLD_XGIT_HASH	7	-	-
	GET_BLD_XGIT_VIEW	50	-	-
	GET_DELAY_SAMPLES	1	0	0
	SET_DELAY_SAMPLES	1	-	-
	GET_STATUS	1	-	-
AEC	GET_ADAPTATION_CONFIG_AEC	1	0	0
	SET_ADAPTATION_CONFIG_AEC	1	-	-
	GET_COEFF_INDEX_AEC	1	0	0
	SET_COEFF_INDEX_AEC	1	-	-
	GET_ERLE_CH0_AEC	2	-	-

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	GET_ERLE_CH1_AEC	2	-	-
	GET_F_BIN_COUNT_AEC	1	-	-
	GET_FILTER_COEFFICIENTS_AEC	14	-	-
	GET_FORCED_MU_VALUE_AEC	1	1	1
	SET_FORCED_MU_VALUE_AEC	1	-	-
	GET_FRAME_ADVANCE_AEC	1	-	-
	GET_MU_LIMITS_AEC	2	1.0000 0.0001	1.0000 0.0001
	SET_MU_LIMITS_AEC	2	-	-
	GET_MU_SCALAR_AEC	1	2	2
	SET_MU_SCALAR_AEC	1	-	-
	RESET_FILTER_AEC	1	-	-
	GET_SIGMA_ALPHAS_AEC	3	5 5 11	5 5 11
	SET_SIGMA_ALPHAS_AEC	3	-	-
	GET_X_CHANNEL_PHASES_AEC	10	-	-
	GET_X_CHANNELS_AEC	1	-	-
	GET_X_ENERGY_DELTA_AEC	1	0.000000 dB	0.000000 dB
	SET_X_ENERGY_DELTA_AEC	1	-	-
	GET_X_ENERGY_GAMMA_LOG2_AEC	1	6	6
	SET_X_ENERGY_GAMMA_LOG2_AEC	1	-	-
	GET_Y_CHANNELS_AEC	1	-	-
AGC	SET_ADAPT_CH0_AGC	1	-	-
	SET_ADAPT_CH1_AGC	1	-	-
	GET_ADAPT_ON_VAD_CH0_AGC	1	1	1
	SET_ADAPT_ON_VAD_CH0_AGC	1	-	-
	GET_ADAPT_ON_VAD_CH1_AGC	1	1	1
	SET_ADAPT_ON_VAD_CH1_AGC	1	-	-
	GET_DECREMENT_GAIN_STEPSIZE_CH0_AGC	1	0.87	0.87
	SET_DECREMENT_GAIN_STEPSIZE_CH0_AGC	1	-	-
	GET_DECREMENT_GAIN_STEPSIZE_CH1_AGC	1	0.988	0.988
	SET_DECREMENT_GAIN_STEPSIZE_CH1_AGC	1	-	-
	SET_GAIN_CH0_AGC	1	-	-
	SET_GAIN_CH1_AGC	1	-	-
	GET_INCREMENT_GAIN_STEPSIZE_CH0_AGC	1	1.197	1.197
	SET_INCREMENT_GAIN_STEPSIZE_CH0_AGC	1	-	-
	GET_INCREMENT_GAIN_STEPSIZE_CH1_AGC	1	1.0034	1.0034
	SET_INCREMENT_GAIN_STEPSIZE_CH1_AGC	1	-	-
	GET_LC_CORR_THRESHOLD_CH0_AGC	1	0	0
	SET_LC_CORR_THRESHOLD_CH0_AGC	1	-	-
	GET_LC_CORR_THRESHOLD_CH1_AGC	1	0.993	0.993
	SET_LC_CORR_THRESHOLD_CH1_AGC	1	-	-
	GET_LC_DELTAS_CH0_AGC	3	0.0000 0.0000 0.0000	0.0000 0.0000 0.0000
	SET_LC_DELTAS_CH0_AGC	3	-	-

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	GET_LC_DELTAS_CH1_AGC	3	299.9954 49.9992 99.9985	299.9954 49.9992 99.9985
	SET_LC_DELTAS_CH1_AGC	3	-	-
	GET_LC_ENABLED_CH0_AGC	1	0	0
	SET_LC_ENABLED_CH0_AGC	1	-	-
	GET_LC_ENABLED_CH1_AGC	1	1	1
	SET_LC_ENABLED_CH1_AGC	1	-	-
	GET_LC_GAINS_CH0_AGC	4	0.0000 0.0000 0.0000 0.0000	0.0000 0.0000 0.0000 0.0000
	SET_LC_GAINS_CH0_AGC	4	-	-
	GET_LC_GAINS_CH1_AGC	4	1.0000 0.9000 0.1000 0.0224	1.0000 0.9000 0.1000 0.0224
	SET_LC_GAINS_CH1_AGC	4	-	-
	GET_LC_GAMMAS_CH0_AGC	3	0.0000 0.0000 0.0000	0.0000 0.0000 0.0000
	SET_LC_GAMMAS_CH0_AGC	3	-	-
	GET_LC_GAMMAS_CH1_AGC	3	1.0020 1.0050 0.9950	1.0020 1.0050 0.9950
	SET_LC_GAMMAS_CH1_AGC	3	-	-
	SET_LC_N_FRAMES_CH0_AGC	2	-	-
	GET_LC_N_FRAMES_CH1_AGC	2	17 34	17 34
	SET_LC_N_FRAMES_CH1_AGC	2	-	-
	GET_LOWER_THRESHOLD_CH0_AGC	1	0.1905	0.1905
	SET_LOWER_THRESHOLD_CH0_AGC	1	-	-
	GET_LOWER_THRESHOLD_CH1_AGC	1	0.4	0.4
	SET_LOWER_THRESHOLD_CH1_AGC	1	-	-
	GET_MAX_GAIN_CH0_AGC	1	999.9847	999.9847
	SET_MAX_GAIN_CH0_AGC	1	-	-
	GET_MAX_GAIN_CH1_AGC	1	999.9847	999.9847
	SET_MAX_GAIN_CH1_AGC	1	-	-
	GET_MIN_GAIN_CH0_AGC	1	0	0
	SET_MIN_GAIN_CH0_AGC	1	-	-
	GET_MIN_GAIN_CH1_AGC	1	0	0
	SET_MIN_GAIN_CH1_AGC	1	-	-
	GET_SOFT_CLIPPING_CH0_AGC	1	1	1
	SET_SOFT_CLIPPING_CH0_AGC	1	-	-
	GET_SOFT_CLIPPING_CH1_AGC	1	1	1
	SET_SOFT_CLIPPING_CH1_AGC	1	-	-
	GET_UPPER_THRESHOLD_CH0_AGC	1	0.7079	0.7079
	SET_UPPER_THRESHOLD_CH0_AGC	1	-	-
	GET_UPPER_THRESHOLD_CH1_AGC	1	0.4	0.4

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	SET_UPPER_THRESHOLD_CH1_AGC	1	-	-
AUDIO	GET_ADEC_FAR_THRESHOLD	1	0.000002 dB	0.000002 dB
	SET_ADEC_FAR_THRESHOLD	1	-	-
	GET_ADEC_PEAK_TO_AVERAGE_GOOD_AEC	1	4.000000 dB	4.000000 dB
	SET_ADEC_PEAK_TO_AVERAGE_GOOD_AEC	1	-	-
	GET_ADEC_TIME_SINCE_RESET	1	-	-
	GET_AEC_PEAK_TO_AVERAGE_RATIO	1	-	-
	GET_AEC_RESET_TIMEOUT	1	-1	-1
	SET_AEC_RESET_TIMEOUT	1	-	-
	GET_AGM	1	-	-
	GET_DELAY_DIRECTION	1	0	0
	GET_DELAY_ESTIMATE	1	-	-
	GET_DELAY_ESTIMATOR_ENABLED	1	0	0
	SET_DELAY_ESTIMATOR_ENABLED	1	-	-
	GET_ERLE_BAD_BITS	1	-0.066	-0.066
	SET_ERLE_BAD_BITS	1	-	-
	GET_ERLE_BAD_GAIN	1	0.0625	0.0625
	SET_ERLE_BAD_GAIN	1	-	-
	GET_ERLE_GOOD_BITS	1	2	2
	SET_ERLE_GOOD_BITS	1	-	-
	SET_MANUAL_ADEC_CYCLE_TRIGGER	1	-	-
	GET_MAX_CONTROL_TIME_STAGE_A	1	-	-
	GET_MAX_CONTROL_TIME_STAGE_B	1	-	-
	GET_MAX_CONTROL_TIME_STAGE_C	1	-	-
	GET_MAX_DSP_TIME_STAGE_A	1	-	-
	GET_MAX_DSP_TIME_STAGE_B	1	-	-
	GET_MAX_DSP_TIME_STAGE_C	1	-	-
	GET_MAX_IDLE_TIME_STAGE_A	1	-	-
	GET_MAX_IDLE_TIME_STAGE_B	1	-	-
	GET_MAX_IDLE_TIME_STAGE_C	1	-	-
	GET_MAX_RX_TIME_STAGE_A	1	-	-
	GET_MAX_RX_TIME_STAGE_B	1	-	-
	GET_MAX_RX_TIME_STAGE_C	1	-	-
	GET_MAX_TX_TIME_STAGE_A	1	-	-
	GET_MAX_TX_TIME_STAGE_B	1	-	-
	GET_MAX_TX_TIME_STAGE_C	1	-	-
	GET_MIC_SHIFT_SATURATE	2	0 0	0 0
	SET_MIC_SHIFT_SATURATE	2	-	-
	GET_MIN_CONTROL_TIME_STAGE_A	1	-	-
	GET_MIN_CONTROL_TIME_STAGE_B	1	-	-
	GET_MIN_CONTROL_TIME_STAGE_C	1	-	-
	GET_MIN_DSP_TIME_STAGE_A	1	-	-
	GET_MIN_DSP_TIME_STAGE_B	1	-	-
	GET_MIN_DSP_TIME_STAGE_C	1	-	-
	GET_MIN_IDLE_TIME_STAGE_A	1	-	-
	GET_MIN_IDLE_TIME_STAGE_B	1	-	-
	GET_MIN_IDLE_TIME_STAGE_C	1	-	-

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	GET_MIN_RX_TIME_STAGE_A	1	-	-
	GET_MIN_RX_TIME_STAGE_B	1	-	-
	GET_MIN_RX_TIME_STAGE_C	1	-	-
	GET_MIN_TX_TIME_STAGE_A	1	-	-
	GET_MIN_TX_TIME_STAGE_B	1	-	-
	GET_MIN_TX_TIME_STAGE_C	1	-	-
	GET_PEAK_PHASE_ENERGY_TREND_GAIN	1	3	3
	SET_PEAK_PHASE_ENERGY_TREND_GAIN	1	-	-
	GET_PHASE_POWER_INDEX	1	0	0
	SET_PHASE_POWER_INDEX	1	-	-
	GET_PHASE_POWERS	5	-	-
	RESET_TIME_STAGE_A	1	-	-
	RESET_TIME_STAGE_B	1	-	-
	RESET_TIME_STAGE_C	1	-	-
FILTER	GET_FILTER_BYPASS	1	1	1
	SET_FILTER_BYPASS	1	-	-
	GET_FILTER_COEFF	10	-0.00000000 -0.00000000 0.00000000 0.00000000 0.00000000 -0.00000000 -0.00000000 0.00000000 0.00000000 0.00000000	-0.00000000 -0.00000000 0.00000000 0.00000000 0.00000000 -0.00000000 -0.00000000 0.00000000 0.00000000 0.00000000
	SET_FILTER_COEFF	10	-	-
	GET_FILTER_COEFF_RAW	10	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0
	SET_FILTER_COEFF_RAW	10	-	-
	GET_FILTER_INDEX	1	0	0
	SET_FILTER_INDEX	1	-	-
GPIO	SET_GPI_INT_CONFIG	3	-	-
	GET_GPI_INT_PENDING_PIN	1	-	-
	GET_GPI_INT_PENDING_PORT	1	-	-
	GET_MAX_UBM_CYCLES	1	-	-
	GET_MCLK_IN_TO_PDM_CLK_DIVIDER	1	2	2
	SET_MCLK_IN_TO_PDM_CLK_DIVIDER	1	-	-
	GET_MIC_START_STATUS	1	2	2
	SET_MIC_START_STATUS	1	-	-
	RESET_MAX_UBM_CYCLES	1	-	-
	GET_SYS_CLK_TO_MCLK_OUT_DIVIDER	1	12	12
	SET_SYS_CLK_TO_MCLK_OUT_DIVIDER	1	-	-

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
I2C	GET_I2C	56	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	-
	SET_I2C	56	-	-
	GET_I2C_READ_HEADER	3	0 0 0	-
	SET_I2C_READ_HEADER	3	-	-
	GET_I2C_WITH_REG	56	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	-
	SET_I2C_WITH_REG	56	-	-
I2S	GET_I2S_RATE	1	48000	48000
	SET_I2S_RATE	1	-	-
	GET_I2S_START_STATUS	1	2	2
	SET_I2S_START_STATUS	1	-	-
IC	GET_ADAPTATION_CONFIG_IC	1	0	0
	SET_ADAPTATION_CONFIG_IC	1	-	-
	GET_CH1_BEAMFORM_ENABLE	1	1	1
	SET_CH1_BEAMFORM_ENABLE	1	-	-
	GET_COEFFICIENT_INDEX_IC	1	0	0
	SET_COEFFICIENT_INDEX_IC	1	-	-
	GET_FILTER_COEFFICIENTS_IC	14	-	-
	GET_FORCED_MU_VALUE_IC	1	0.999	0.6021
	SET_FORCED_MU_VALUE_IC	1	-	-
	GET_PHASES_IC	1	-	-
	GET_PROC_FRAME_BINS_IC	1	-	-
	RESET_FILTER_IC	1	-	-
	GET_SIGMA_ALPHA_IC	1	11	11
	SET_SIGMA_ALPHA_IC	1	-	-
	GET_X_ENERGY_DELTA_IC	1	0.000070 dB	0.000070 dB
	SET_X_ENERGY_DELTA_IC	1	-	-
	GET_X_ENERGY_GAMMA_LOG2_IC	1	2	2
	SET_X_ENERGY_GAMMA_LOG2_IC	1	-	-
KWD	GET_KWD_BOOT_STATUS	1	-	-
	GET_KWD_HID_EVENT_CNT	1	0	0
	SET_KWD_HID_EVENT_CNT	1	-	-
	GET_KWD_INTERRUPT_PIN	1	4	4
	SET_KWD_INTERRUPT_PIN	1	-	-
SPI	GET_SPI	56	-	-
	SET_SPI_PUSH	56	-	-
	SET_SPI_PUSH_AND_EXEC	56	-	-

continues on next page

Table 7.2 – continued from previous page

Group	Command	Number of Values	UA Default	INT Default
	GET_SPI_READ_HEADER	2	0 0	0 0
	SET_SPI_READ_HEADER	2	-	-
USB	GET_HID_MAP	2	0 2	-
	SET_HID_MAP	2	-	-
	GET_HID_MAP_HEADER	2	0 0	-
	SET_HID_MAP_HEADER	2	-	-
	GET_HID_USAGE	2	9 23	-
	SET_HID_USAGE	2	-	-
	GET_HID_USAGE_HEADER	3	0 0 7	-
	SET_HID_USAGE_HEADER	3	-	-
	GET_USB_BCD_DEVICE	1	1	1
	SET_USB_BCD_DEVICE	1	-	-
	GET_USB_START_STATUS	1	2	-
	SET_USB_START_STATUS	1	-	-

7.3 Boot status codes (RUN_STATUS)

The following table describes the Boot Status codes returned by the startup processes accessible though the `GET_RUN_STATUS` control utility command.

Table 7.3: XVF3610 Boot Status Codes

Code	Label	Note
0	INIT	Reserved initial value. Decline attempts to initiate DFU.
1	DATA_PARTITION_NOT_FOUND	Not used.
2	FACTORY_DATA_SUCCESS	Normal operation.
3	UPGRADE_DATA_SUCCESS	Normal operation.
4	FACTORY_DATA_IN_PROGRESS	Image scanning in progress. Decline attempts to initiate DFU.
5	UPGRADE_DATA_IN_PROGRESS	Image scanning in progress. Decline attempts to initiate DFU.
6	DFU_IN_PROGRESS	Enough DFU commands received to establish a connection to on-board flash memory. Not cleared until reboot.
7	HW_BUILD_READ_SUCCESS	Reserved intermediate value. Normally never returned.
8	HW_BUILD_PARTITION_SIZE_ERROR	Problem reading data partition header. Check factory programming.
9	HW_BUILD_PARTITION_BASE_ERROR	Problem reading data partition header. Check factory programming.
10	HW_BUILD_READ_ERROR	Problem reading data partition header. Check factory programming.
11	HW_BUILD_CRC_ERROR	Problem reading data partition header. Check factory programming. May indicate that no data partition is present or a flash wear issue.
12	HW_BUILD_TAG_ERROR	Problem reading data partition header. Check factory programming.
13	FACTORY_VERSION_ERROR	No valid upgrade image found. A factory image did not match running version. This can indicate fail-safe mode.
14	UPGRADE_VERSION_ERROR	Valid upgrade boot and data images found but data image version does not match running version. Check correct version of deployed field upgrade.
15	FACTORY_ITEM_READ_ERROR	Problem reading configuration items from data image. Unexpected error.
16	UPGRADE_ITEM_READ_ERROR	Problem reading configuration items from data image. Unexpected error.
17	FACTORY_ITEM_INVALID_TYPE	Last item encountered is not of terminator type. Should never happen with script generated data images. Check generation procedure.
18	UPGRADE_ITEM_INVALID_TYPE	Last item encountered is not of terminator type. Should never happen with script generated data images. Check generation procedure.
19	DFU_FLASH_CONNECT_FAILED	Failed to establish on-board flash connection. Check factory programming. Check flash specification (see section below).
20	DFU_FLASH_SPEC_UNSUITABLE	Flash specification unsuitable for DFU. Check flash specification (see section below).

7.4 Example .SPISPEC file format

SPISPEC file for 64Mbit Winbond W25Q64JV (used on XK-VOICE-L71 kit).

This file is required to run the *xflash* command to program the firmware into the flash memory device.

Comments are inserted with */* .. */*.

```
0,                      /* W25Q64JV - Just specify 0 as flash_id */
256,                    /* page size */
32768,                  /* num pages */
3,                      /* address size */
4,                      /* log2 clock divider */
0x9F,                   /* QSPI_RDID */
0,                      /* id dummy bytes */
3,                      /* id size in bytes */
0,                      /* device id (leave zero) */
0x20,                   /* QSPI_SE */
4096,                   /* Sector erase is always 4KB */
0x06,                   /* QSPI_WREN */
0x04,                   /* QSPI_WRDI */
PROT_TYPE_SR,           /* Protection via SR */
{{0x18,0x00},{0,0}},    /* QSPI_SP, QSPI_SU */
0x02,                   /* QSPI_PP */
0xEB,                   /* QSPI_READ_FAST */
1,                      /* 1 read dummy byte */
SECTOR_LAYOUT_REGULAR,  /* mad sectors */
{4096,{0,{0}}},         /* regular sector sizes */
0x05,                   /* QSPI_RDSR */
0x01,                   /* QSPI_WRSR */
0x01,                   /* QSPI_WIP_BIT_MASK */
```

7.5 USB enumeration

The XVF3610 includes a Human Interface Device (HID) endpoint to enable the XVF3610 to signal interrupts caused by GPIO events. The table below shows how the XVF3610 HID appears on Windows using USB view.

Table 7.4: USB HID Endpoint

Device Name	Description	Device Type	Vendor ID	Product ID	USB Class	USB Sub-Class	USB Protocol	Service Name	USB Version	Driver Description
XVF3610 (UAC1.0) Adaptive	USB Composite Device	Unknown	20b1	0x0016	0	0	0	usbccgp	2	USB Composite Device
XVF3610 (UAC1.0) Adaptive	USB Audio Device	Audio	20b1	0x0016	1	1	0	usbaudio	2	USB Audio Device
XVF3610 (UAC1.0) Adaptive	XMOS Control	Vendor Specific	20b1	0x0016	ff	ff	ff		2	
XVF3610 (UAC1.0) Adaptive	USB Input Device	HID (Human Interface Device)	20b1	0x0016	3	0	0	HidUsb	2	USB Input Device

During USB enumeration, the XVF3610 HID produces three descriptors. The listing below shows them as recorded on Windows using USB View. For details of the structure and meaning of these descriptors, see the USB Specification v2.0 sections 9.6.5 and 9.6.6 and the Device Class Definition for Human Interface Devices (HID) v1.11 section 6.2.1.

```
===>Interface Descriptor<===
bLength: 0x09
bDescriptorType: 0x04
bInterfaceNumber: 0x04
bAlternateSetting: 0x00
bNumEndpoints: 0x01
bInterfaceClass: 0x03 -> HID Interface Class
bInterfaceSubClass: 0x00
bInterfaceProtocol: 0x00
iInterface: 0x00
===>HID Descriptor<===
bLength: 0x09
bDescriptorType: 0x21
bcdHID: 0x0110
bCountryCode: 0x00
bNumDescriptors: 0x01
bDescriptorType: 0x22 (Report Descriptor)
wDescriptorLength: 0x006E
===>Endpoint Descriptor<===
bLength: 0x07
bDescriptorType: 0x05
bEndpointAddress: 0x82 -> Direction: IN - EndpointID: 2
bmAttributes: 0x03 -> Interrupt Transfer Type
wMaxPacketSize: 0x0040 = 0x40 bytes
bInterval: 0x08
```

Note: If the SET_HID_USAGE command has been used to change the meaning of a bit in the HID Report, the wDescriptorLength field of the HID Descriptor may contain a different value.

7.6 General purpose filter example

7.6.1 Specification

This page illustrates the process of defining an audio filter block in the XVF3610.

The example below routes a USB audio signal through a filer block and sends the output back to the USB host.

Steps to set up this configuration are:

1. Set the stereo USB output to listen to the stereo USB input (loopback, skipping audio processing pipeline completely)
2. Apply a stereo 500Hz high-pass and 4kHz low-pass cascaded biquad filter
3. The 500Hz high-pass filter coefficients are:

   a1 = -1.90748889
   a2 = 0.91158173
   b0 = 0.95476766
   b1 = -1.90953531
   b2 = 0.95476766

4. The 4kHz low-pass filter coefficients are:

   a1 = -1.27958194
   a2 = 0.47753396
   b0 = 0.04948800
   b1 = 0.09897601
   b2 = 0.04948800

5. Enable the filter and hear the effect of the filter on a signal when the filters are enabled

7.6.2 Worked Example

This example assumes that the input and output sample rate is 48kHz.

First, connect the USB output to the USB input:

```
vfctrl_usb SET_IO_MAP 0 7 # (USB output left outputs USB input left)
vfctrl_usb SET_IO_MAP 1 8 # (As above for right channel)
```

Now configure the filter:

```
vfctrl_usb SET_FILTER_INDEX 2 (USB output left filter)
vfctrl_usb SET_FILTER_COEFF -1.90748889 0.91158173 0.95476766 -1.90953531 0.95476766 -1.
↪27958194 0.47753396 0.04948800 0.09897601 0.04948800
vfctrl_usb SET_FILTER_INDEX 3 (USB output right filter)
```

(continues on next page)

(continued from previous page)

```
vfctrl_usb SET_FILTER_COEFF -1.90748889 0.91158173 0.95476766 -1.90953531 0.95476766 -1.
↪27958194 0.47753396 0.04948800 0.09897601 0.04948800
```

Now enable the filter:

```
vfctrl_usb SET_FILTER_INDEX 0
vfctrl_usb SET_FILTER_BYPASS 0
vfctrl_usb SET_FILTER_INDEX 1
vfctrl_usb SET_FILTER_BYPASS 0
```

Play a white noise source from the USB device and record the input. Use a spectrogram to show the band limited signal due to the effect of the filters. The effect should also be audible.

7.7 Command transport protocol

7.7.1 Transport protocol for control parameters

Control parameters are converted to an array of bytes in network byte order (big endian) before they're sent over the transport protocol. For example, to set a control parameter to integer value 305419896 which corresponds to hex 0x12345678, the array of bytes sent over the transport protocol would be {0x12, 0x34, 0x56, 0x78}. Similarly, a 4 byte payload {0x00, 0x01, 0x23, 0x22} read over the transport protocol is interpreted as an integer value 0x00012322.

In addition to the control parameters values, commands include Resource ID, the Command ID and Payload Length fields that must be communicated from the host to the device. The Resource ID is an 8-bit identifier that identifies the resource within the device that the command is for. The Command ID is an 8-bit identifier used to identify a command for a resource in the device. Payload length is the length of the data in bytes that the host wants to write to the device or read from the device.

The payload length is interpreted differently for GET_ and SET_ commands. For SET_ commands, the payload length is simply the number of bytes worth of control parameters to write to the device. For example, the payload length for a SET_ command to set a control parameter of type int32 to a certain value, would be set to 4. For GET_ commands the payload length is 1 more than the number of bytes of control parameters to read from the device. For example, a GET_ command to read a parameter of type int32, payload length would be set to 5. The one extra byte is used for status and is the first byte (payload[0]) of the payload received from the device. In the example above, payload[0] would be the status byte and payload[1]..payload[4] would be the 4 bytes that make up the value of the control parameter.

The table below lists the different values of the status byte and the action the user is expected to take for each status:

Table 7.5: Values for returned status byte

Return code	Values	Description
ctrl_done	0	Read command successful. The payload bytes contain valid payload returned from the device.
ctrl_wait	1	Read command not serviced. Retry until ctrl_done status returned.
ctrl_invalid	3	Error in read command. Abort and debug.

The GET_commands need the extra status byte since the device might not return the control parameter value imimagestely due to timing constraints. If that is the case the status byte would indicate the status as ctrl_wait

and the user would need to retry the command. When returned a ctrl_wait, the user is expected to retry the GET_ command until the status is returned as ctrl_done. The first GET_command is placed in a queue and it will be serviced by the end of each 15ms audio frame. Once the status byte indicates ctrl_done, the rest of the bytes in the payload indicate the control parameter value.

7.7.2 Transporting control parameters over I^{2}C

This section describes the I^{2}C command sequence when issuing read and write commands to the device.

The first byte sent over I^{2}C after start contains the device address and information about whether this is an I^{2}C read transaction or a write transaction. This byte is 0x58 for a write command or 0x59 for a read command. These values are derived by left shifting the device address (0x2c) by 1 and doing a logical OR of the resulting value with 0 for an I^{2}C write and 1 for an I^{2}C read.

The bytes sequence sent between I^{2}C start and stop for SET_ commands is shown in the figure below:

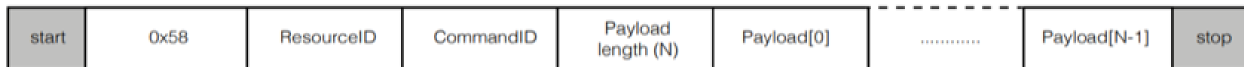

Fig. 7.1: Bytes sequence for I^{2}C SET_ commands

For GET_ commands, the I^{2}C commands sequence consists of a write command followed by a read command with a repeated start between the 2 commands. The write command writes the resource ID, command ID and the expected data length to the device and the read command reads the status byte followed by the rest of the payload that makes up the control parameter value. The figure below shows the I^{2}C bytes sequence sent and received for a GET_ command.

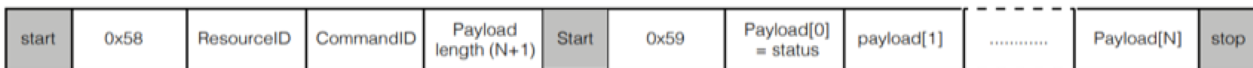

Fig. 7.2: Bytes sequence for I^{2}C GET_ commands

7.7.3 Transporting control parameters over USB

Use the vendor_id 0x20b1, product_id 0x0016 and interface number 3 to initialize for USB. The API function libusb_control_transfer() is used for transporting over USB. When calling libusb_control_transfer(), wIndex corresponds to the Resource ID, wValue is the Command ID and wLength is the payload length.

7.7.4 Floating point to fixed point (Q format) conversion

Numbers with fractional parts can be represented as floating-point or fixed-point numbers. Floating point formats are widely used but carry performance overheads. Fixed point formats can improve system efficiency and are used extensively within the XVF3610. Fixed point numbers have the position of the decimal point fixed and this is indicated as a part of the format description.

In this document, Q format is used to describe fixed point number formats, with the representation given as Q*m*.*n* format where *m* is the number of bits reserved for the sign and integer part of the number and *n* is the number of bits reserved for the fractional part of the number. The position of the decimal point is a trade-off between the range of values supported and the resolution provided by the fractional bits.

The dynamic range of Q*m*.*n* format is -2^{m-1} and $2^{m-1}-2^{-n}$ with a resolution of 2^{-n}

To convert a floating-point format number to Q*m*.*n* format fixed-point number:

- Multiply the floating-point number by 2^m
- Round the result to the nearest integer
- The resulting integer number is the Q*m*.*n* fixed-point representation of the initial floating-point number

To convert a Q*m*.*n* fixed-point number to floating-point:

- Divide the fixed-point number by 2^m
- The resulting decimal number is a floating-point representation of the fixed-point number.

Converting a number into fixed point format and then back to a floating point number may introduce an error of up to $\pm 2^{-(n+1)}$

Example:

To represent a floating-point number 14.765467 in Q8.24 format, the equivalent fixed-point number would be 14.765467×2^{24} = 247723429.2 which rounds to 247723429.

To get back the floating-point number given the Q8.24 number 247723429, calculate 247723429 / 2^{24} and get back the floating-point number as 14.76546699. The difference of 0.00000001 is correct to with the error bounds of $\pm 2^{-25}$ which is ± 0.00000003

7.8 Flash programming and update flow

The flows to program the flash and to update the device are shown in the diagram below:

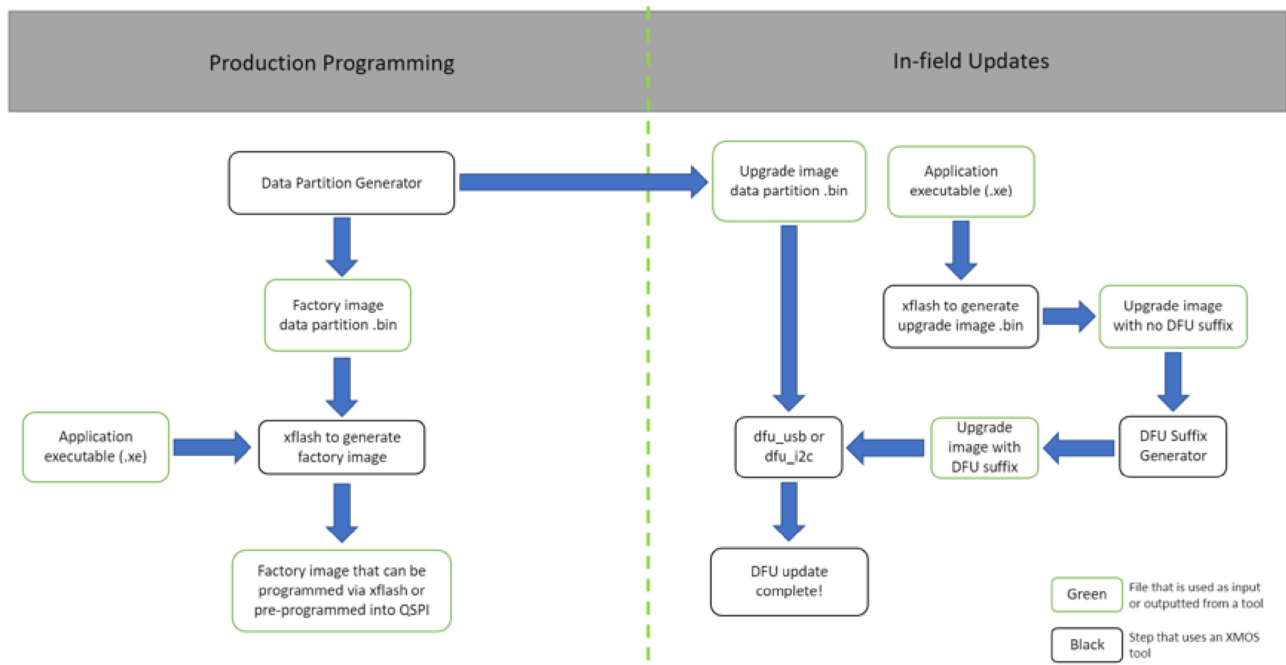

Fig. 7.3: Flash programming and update flow

The first steps for both flows consists of generating the data partition as described in *Configuration and the Data Partition*. When programming the factory image, the data partition and the boot image must be used as reported in *Programming the Factory Boot image and Data Partition*. In case of a firmware update, the list of tools and steps required can be found in *Upgrade Images and Data Partitions*.

7.9 Capturing packed samples

To assist with system integration, the XVF3610 provides the ability to pack multiple 16kHz channels into a 48kHz output. The following section describes the usage of packed signals.

Note: All packed functions provide a snapshot of a 16kHz signals over a 48kHz output. If the output stream is not 48kHz, it will not work because the 3x bandwidth is needed for packing the 16kHz signals. They all also require that no volume scaling be applied on the host otherwise it will break the marker sequence resulting in the captured audio being unable to be unpacked.

There are two packing mechanisms however for typical usage where a full capture of the pipeline is needed, *PACKED_ALL* is recommended.

7.9.1 Capturing all pipeline input and output signals over a 48kHz USB interface

The goal here is to capture the pipeline input and output to provide visibility on what signals are actually entering the pipeline and what processed output was generated. This can be useful when checking the microphone and reference signals are correctly routed, as well as checking signal delay issues causing poor AEC performance.

This procedure is described in the figure below:

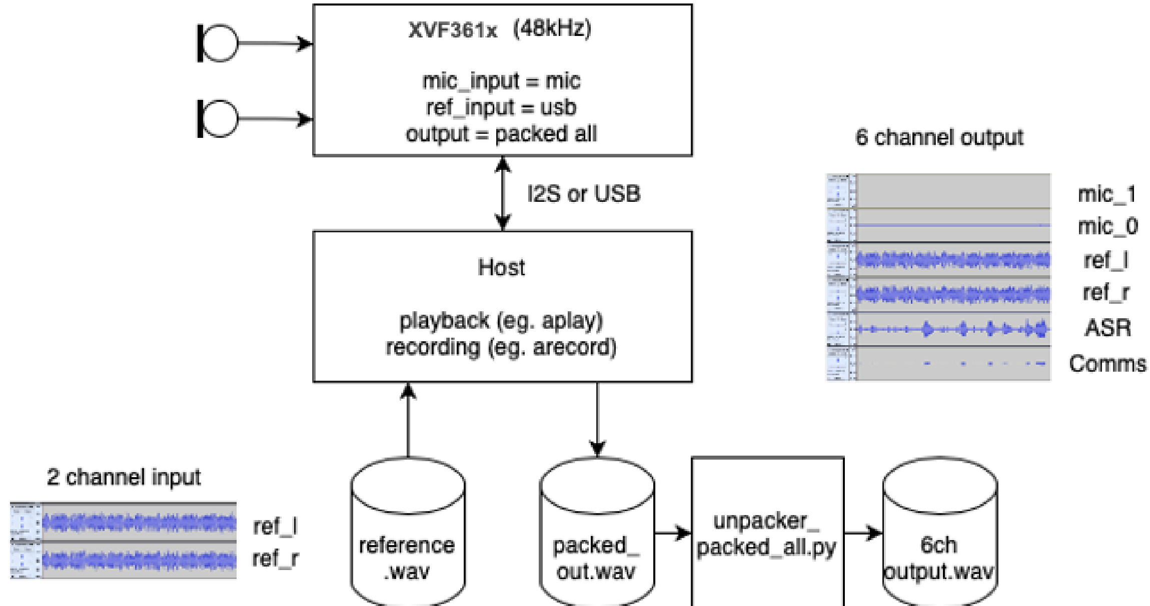

Fig. 7.4: System overview to capture of DSP pipeline signals over a 48kHz USB interface

First, set the USB output interface resolution to 24b. This is important because microphone signals in a quiet room (35dBA) may be quantised away in a 16b audio capture. Also, 24b audio has been found to work on most hosts.

Second, configure the audio crossbar to output `PACKED_ALL` on USB output channels 0 and 1. More information about the output channels can be found in *the signal routing section*.

This can be done by setting the parameters in the data partition, as described in *the data partition section*.

To configure the packed output for USB, add in the file `input/set_packed_all_usb_output.txt` the following contents:

```
SET_IO_MAP 0 16
SET_IO_MAP 1 16
```

Note: The IO map source 16 is set for both USB output channels. Source 16 automatically resolves the channel indices so this will result in a stereo output containing a packed capture of all six discrete channels of interest.

Next, add the following sections to the .json configuration file item section and save it:

```
{
"path": "input/set_packed_all_usb_output.txt",
"comment": ""
},

{
"path": "input/device_to_usb_bit_res_24.txt",
```

(continues on next page)

(continued from previous page)

```
"comment": ""
},
```

Now generate the data partition from the updated .json configuration file and flash the device with the newly generated data partition as described in *the data partition section*.

Once the firmware has booted following the flashing operation, it can be verified in the sound control panel that the USB input stream from the XVF3610-UA to the host is now set to 24b.

Next the audio of interest is captured. Do this with a wav capture utility to capture the stereo output from the USB input from the XVF3610 device at 48kHz. Ensure the file is saved as 32b Signed Integer which is needed for the next step.

An example command line for Linux (with ALSA tools installed and XVF3610 as device 1) is shown below:

```
arecord -c 2 -f S24_LE -r 48000 packed_capture.wav -D plughw:1
```

Note: Viewing/listening to the packed wav is non-sensical because it contains packed/multiplexed signals and will sound noisy.

Finally convert these packed files into unpacked, 16kHz, six-channel audio files.

```
python3 host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 24
```

The output file unpacked_6ch_16kHz.wav may now be inspected. The channel assignment is as follows:

1. Microphone Ch 0
2. Microphone Ch 1
3. Reference input Left
4. Reference input Right
5. Pipeline Output Ch 0 (nominally ASR)
6. Pipeline Output Ch 1 (nominally Comms)

Below is a visualisation of a six-channel audio capture. Note the relatively quiet microphone signals compared with the reference. This is typical and allows for loud near-end signals without distortion.

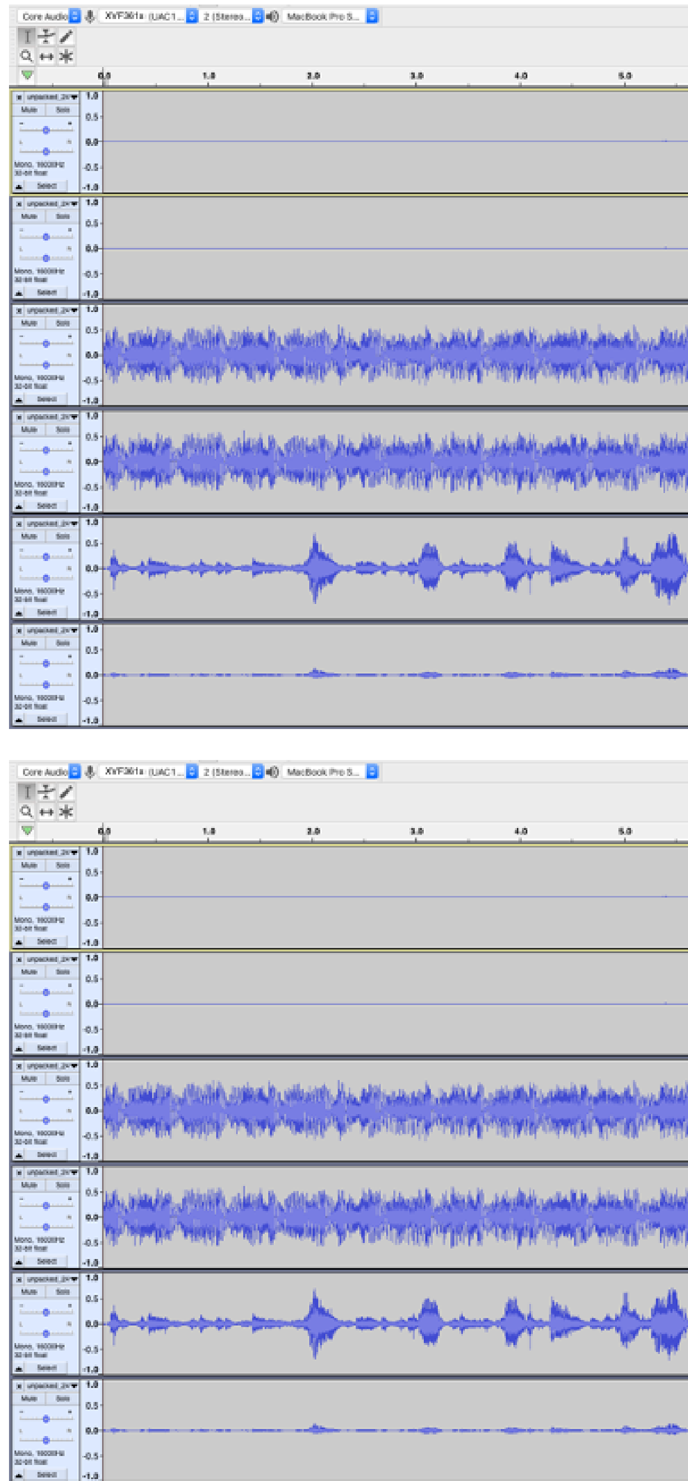

Fig. 7.5: Example of six-channel audio capture

7.9.2 Capturing all pipeline input and output signals over a 48kHz I²S interface

The same *procedure described for the USB interface* can be adapted for the I²S interface. The only differences are that:

- I²S interface is always 32 bits
- I²S has different output channels than USB

To configure the packed output for I²S, add in the file `input/set_packed_all_i2s_output.txt` the following contents:

```
SET_IO_MAP 2 16
SET_IO_MAP 3 16
```

Note: The IO map source 16 is set for both I²S output channels. Source 16 automatically resolves the channel indices so this will result in a stereo output containing a packed capture of all six discrete channels of interest.

Next, add the following sections to .json configuration file item section and save it:

```
{
"path": "input/set_packed_all_i2s_output.txt",
"comment": ""
},
```

Now generate the data partition from the updated .json file and flash it on the device using the same instructions as described in *the data partition section*.

Next the audio of interest is captured. Do this with a wav capture utility to capture the stereo output from the I²S input from the XVF3610 device at 48kHz. Ensure the file is saved as 32b Signed Integer which is needed for the next step.

An example command line for Linux (with ALSA tools installed and XVF3610 as device 0) is shown below:

```
arecord -c 2 -f S24_LE -r 48000 packed_capture.wav -D plughw:0
```

Note: Viewing/listening to the packed wav is non-sensical because it contains packed/multiplexed signals and will sound noisy.

Finally convert these packed files into unpacked, 16kHz, six-channel audio files.

```
python3 host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 32
```

The output file unpacked_6ch_16kHz.wav may now be inspected. The channel assignment is as follows:

1. Microphone Ch 0
2. Microphone Ch 1
3. Reference input Left
4. Reference input Right
5. Pipeline Output Ch 0 (nominally ASR)
6. Pipeline Output Ch 1 (nominally Comms)

7.9.3 Packing specific signals

`PACKED_PIPELINE_OUTPUT`, `PACKED_MIC`, `PACKED_REF` all use the same underlying packing function. They pack 2 channels (pipeline outpout 0/1 or microphone0/1 or reference left/right) into a single audio channel. They require that the output interface, including host processing, be capable of bit-perfect 32b audio. The underlying function packs the two 16kHz samples into three 48kHz samples as follows:

- Top 24b of sample[0] with 8b LSB marker '0x00'
- Top 24b of sample[1] with 8b LSB marker '0x01'
- The bottom 8b of sample[0], the bottom 8b of sample[1], 0x00, 8b LSB marker '0x02'

The `unpacker.py` script then looks for 0x00, 0x01, 0x02 in the LSByte to check for a packed sequence. So inspecting the wav in a hex editor should make it clear when it is captured properly.

It will capture bit-perfect data.

Warning: Packing specific signals will not work on a Mac because it only supports 24b audio due to core audio representing audio using single-precision floating-point. It has been tested and works well on Linux (x86/RPI) which supports bit-perfect 32b audio.

7.10 Direct access to DSP Pipeline

The XVF3610 supports a mode where the DSP pipeline can be fed directly from a 4-channel test vector which may either be pre-generated or even pre-captured by recording from an XVF3610 device. This can be helpful when recreating a previous scenario or when tuning the system via the control interface in the presence of a fixed and repeatable test vector.

The vector injection mode works by packing 4-channel 16kHz input data (dual microphones and stereo reference) into a 48kHz stereo input signal. The device then unpacks the 48kHz wav into 16kHz multi-channel input and feeds it to the front end of the pipeline.

7.10.1 Injecting a 4-channel, 16kHz test vector into the DSP pipeline over USB

The goal here is to provide a 4-channel test vector directly into the DSP pipeline instead of using the microphones and a separate reference signal. The packed input feature is supported by both I^{2}S (INT) and USB (UA and UA-HYBRID) connected XVF3610 variants.

First the procedure for the UA variant is described.

This procedure is represented in the figure below:

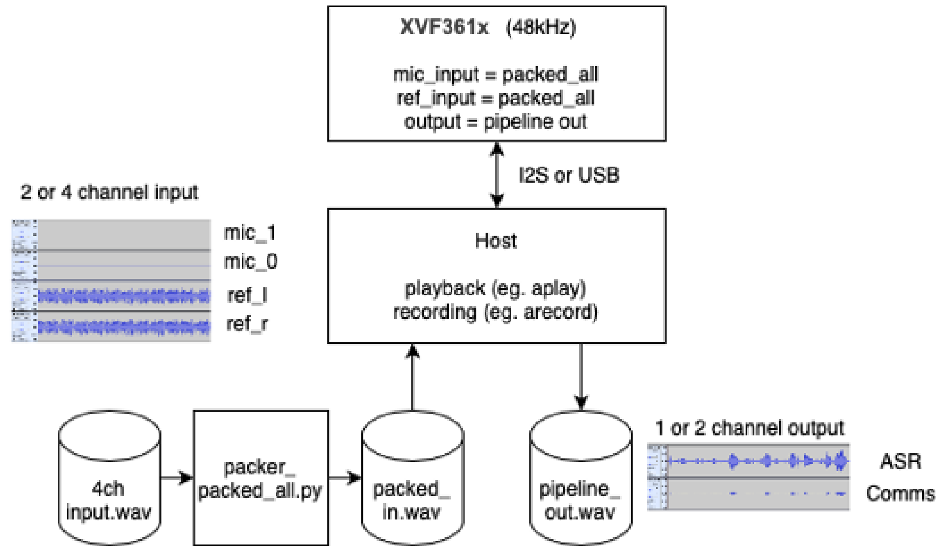

Fig. 7.6: System overview to inject a 4-channel, 16kHz test vector into the DSP pipeline

First, the resolution must be set to 24b (default is 16b). No need exists to adjust the sample rate as this is set to 48kHz by default. The resolution is important because microphone signals in a quiet room (35dBA) may be quantised away in a 16b audio sample. Also, the packing process uses the least significant bit of the audio to carry the frame markers and hence 1 bit of audio resolution is lost. Using 24b audio has been found to work on most popular hosts and OSs.

Second, the audio crossbar switch must be configured to input *PACKED_ALL_INPUT_USB* on input channels *MIC_TO_PIPELINE_0*, *MIC_TO_PIPELINE_1*, *REF_TO_PIPELINE_0* and *REF_TO_PIPELINE_1*. More information about the input channels can be found in *the signal routing section*.

Both of these settings will be set by configuring parameters in a custom data partition. The audio crossbar can be configured at runtime whereas USB parameters can only be set in the data partition.

To configure the packed output for USB add in the file `input/set_packed_all_usb_input.txt` the following contents:

```
SET_IO_MAP 4 17
SET_IO_MAP 5 17
SET_IO_MAP 6 17
SET_IO_MAP 7 17
```

Note: The IO map source 17 is set for both microphone and reference channels. Source 17 automatically resolves the channel indices.

Next, add the following sections to the .json configuration file item section and save it:

```
"item_files": [
```

(continues on next page)

(continued from previous page)

```
    {
        "path": "input/set_packed_all_usb_input.txt",
        "comment": ""
    },
    {
        "path": "input/usb_to_device_bit_res_24.txt",
        "comment": ""
    }
]
```

Now generate the data partition from the updated .json configuration file and flash the device with the newly generated data partition as described in *the data partition section*.

Once the firmware has booted following the flashing operation you may verify in the sound control panel that the USB stream from host to the XVF3610 is now set to 24b.

Next the 4-channel audio test file to be injected must be converted into a packed, 48kHz, stereo audio file.

Note: Viewing/listening to the packed wav is non-sensical because it contains packed/multiplexed signals and will sound noisy. If you provide a 2-channel 16KHz input vector, the two channels are treated as microphone inputs and the reference channels are set to zero.

```
python3 ../python/packer_packed_all.py my_4ch_test_vector.wav packed_input.wav 24
```

The output file `packed_input.wav` can now be fed into the XVF3610. Do this with your favourite wav playback utility to inject the test file across the USB input to the XVF3610 device at 48kHz.

An example command to play the audio on a Linux host (with ALSA tools installed, assuming XVF3610 is device 1) is:

```
aplay -c 2 -f S24_LE -r 48000 packed_input.wav-D plughw:1
```

Note: Ensure the XVF3610 input audio device setting is set to 100% which will allow samples to be passed through without scaling or breaking the marker sequence. Do this in your OS Audio control panel; it is not a control supported by the *vfctrl* mechanism. If the device receives an invalid marker sequence it mutes the inputs.

7.10.2 Injecting a 4-channel, 16kHz test vector into the DSP pipeline over I^{2}S

The same *procedure described for the USB interface* can be adapted for the I^{2}S interface used by INT and UA-HYBRID variants. The only differences are that:

- I^{2}S interface is always 32 bits
- I^{2}S has different input channels than USB

The audio crossbar switch must be configured to input `PACKED_ALL_INPUT_I2S` on input channels `MIC_TO_PIPELINE_0`, `MIC_TO_PIPELINE_1`, `REF_TO_PIPELINE_0` and `REF_TO_PIPELINE_1`. More information about the input channels can be found in *the signal routing section*.

To configure the packed output for I^{2}S, add in the file `input/set_packed_all_i2s_input.txt` the following contents:

```
SET_IO_MAP 4 18
SET_IO_MAP 5 18
SET_IO_MAP 6 18
SET_IO_MAP 7 18
```

Note: The IO map source 18 is set for both microphone and reference channels. Source 18 automatically resolves the channel indices.

Next, add the following sections to the .json configuration file item section and save it:

```
"item_files": [
    {
        "path": "input/set_packed_all_i2s_input.txt",
        "comment": ""
    }
]
```

Now generate the data partition from the updated .json configuration file and flash the device with the newly generated data partition as described in *the data partition section*.

Next the 4-channel audio test file to be injected must be converted into a packed, 48kHz, stereo audio file.

Note: Viewing/listening to the packed wav is non-sensical because it contains packed/multiplexed signals and will sound noisy. If you provide a 2-channel 16KHz input vector, the two channels are treated as microphone inputs and the reference channels are set to zero.

```
python3 ../python/packer_packed_all.py my_4ch_test_vector.wav packed_input.wav 32
```

The output file `packed_input.wav` can now be fed into the XVF3610. Do this with your favourite wav playback utility to inject the test file across the I^{2}S input to the XVF3610 device at 48kHz.

An example command to play the audio on a Linux host (with ALSA tools installed, assuming XVF3610 is device 0) is:

```
aplay -c 2 -f S24_LE -r 48000 packed_input.wav-D plughw:0
```

Note: Ensure the XVF3610 input audio device setting is set to 100% which will allow samples to be passed through without scaling or breaking the marker sequence. Do this in your OS Audio control panel; it is not a control supported by the *vfctrl* mechanism. If the device receives an invalid marker sequence it mutes the inputs.

7.10.3 Injecting a 4-channel packed input and capturing a 6-channel packed output

The full packed-input, packed-output system combines the behaviours of capturing and injecting the packed audio samples on the XVF3610 device.

A system overview of this procedure for XVF3610-UA is below:

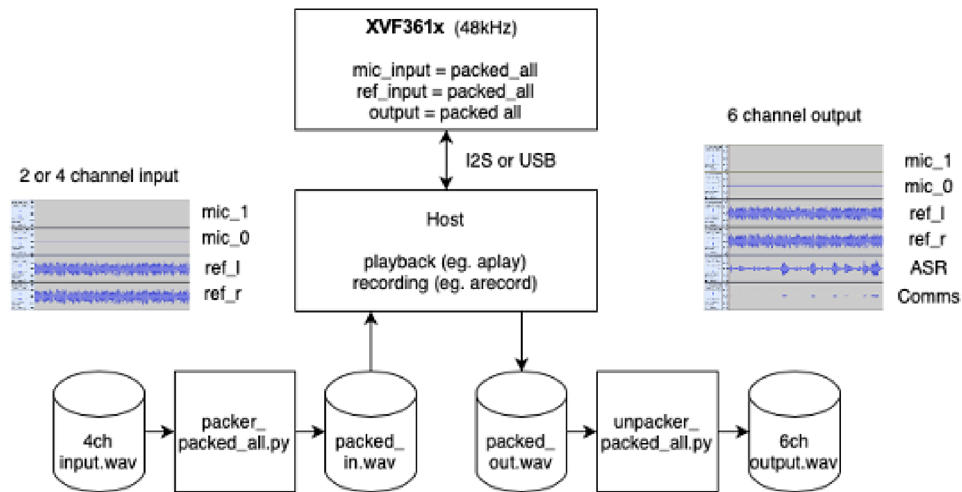

Fig. 7.7: System overview to inject a 4-channel packed input and capture a 6-channel packed output

To enable 4-channel input and 6-channel output simultaneously, create a json configuration file using the `item_files` sections used in the previous chapters, the sections must be included in the same .json configuration file.

The configurations described below are recommended for Hardware-In-Loop implementations because the integrity of the packing and unpacking process can easily be checked visually by inspecting the 6-channel output capture.

Steps for XVF3610-UA

Following the instructions in *the data partition section*, generate and flash a data partition including the commands to *capture the packed audio over USB* and *inject the packed audio over USB*.

Now start capturing the packed output audio file. Do this with your favourite wav capture utility to capture the stereo output from the USB input from the XVF3610 device at 48kHz. Ensure you save the file as 32b Signed Integer which is needed for the next step.

An example command line for Linux (with ALSA tools installed and XVF3610 as device 1) is shown below:

```
arecord -c 2 -f S24_LE -r 48000 packed_capture.wav -D plughw:1
```

Next the 4-channel audio test file to be injected must be converted into a packed, 48kHz, stereo audio file.

```
python3 ../python/packer_packed_all.py my_4ch_test_vector.wav packed_input.wav 24
```

The output file `packed_input.wav` can now be fed into the XVF3610. Do this with your favourite wav playback utility to inject the test file across the USB input to the XVF3610 device at 48kHz.

An example command to play the audio on a Linux host (with ALSA tools installed, assuming XVF3610 is device 1) is:

```
aplay -c 2 -f S24_LE -r 48000 packed_input.wav-D plughw:1
```

When the audio file is played into the device, the recording can be stopped and you can now convert the packed file into an unpacked, 16kHz, 6-Channel audio file.

```
python3 host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 24
```

Steps for XVF3610-UA-HYBRID

Following the instructions in *the data partition section*, generate and flash a data partition including the commands to *capture the packed audio over USB* and *inject the packed audio over* I^{2}S.

Now start capturing the packed output audio file. Do this with your favourite wav capture utility to capture the stereo output from the USB input from the XVF3610 device at 48kHz. Ensure you save the file as 32b Signed Integer which is needed for the next step.

An example command line for Linux (with ALSA tools installed and XVF3610 as device 1) is shown below:

```
arecord -c 2 -f S24_LE -r 48000 packed_capture.wav -D plughw:1
```

Next the 4-channel audio test file to be injected must be converted into a packed, 48kHz, stereo audio file.

```
python3 ../python/packer_packed_all.py my_4ch_test_vector.wav packed_input.wav 32
```

The output file `packed_input.wav` can now be fed into the XVF3610. Do this with your favourite wav playback utility to inject the test file across the I2S input to the XVF3610 device at 48kHz.

An example command to play the audio on a Linux host (with ALSA tools installed, assuming XVF3610 is device 0) is:

```
aplay -c 2 -f S24_LE -r 48000 packed_input.wav-D plughw:0
```

When the audio file is played into the device, the recording can be stopped and you can now convert the packed file into an unpacked, 16kHz, 6-Channel audio file.

```
python3 host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 24
```

Steps for XVF3610-INT

Following the instructions in *the data partition section*, generate and flash a data partition including the commands to *capture the packed audio over* I2S and *inject the packed audio over* I2S.

Now start capturing the packed output audio file. Do this with your favourite wav capture utility to capture the stereo output from the USB input from the XVF3610 device at 48kHz. Ensure you save the file as 32b Signed Integer which is needed for the next step.

An example command line for Linux (with ALSA tools installed and XVF3610 as device 1) is shown below:

```
arecord -c 2 -f S24_LE -r 48000 packed_capture.wav -D plughw:1
```

Next the 4-channel audio test file to be injected must be converted into a packed, 48kHz, stereo audio file.

```
python3 ../python/packer_packed_all.py my_4ch_test_vector.wav packed_input.wav 32
```

The output file `packed_input.wav` can now be fed into the XVF3610. Do this with your favourite wav playback utility to inject the test file across the I2S input to the XVF3610 device at 48kHz.

An example command to play the audio on a Linux host (with ALSA tools installed, assuming XVF3610 is device 1) is:

```
aplay -c 2 -f S24_LE -r 48000 packed_input.wav-D plughw:1
```

When the audio file is played into the device, the recording can be stopped and you can now convert the packed file into an unpacked, 16kHz, 6-Channel audio file.

```
python3 host/unpacker_packed_all.py packed_capture.wav unpacked_6ch_16kHz.wav 32
```

Copyright © 2023, All Rights Reserved.

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

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