XMOS XVF3800 - Programming Guide

Release: 1.0.0
Publication Date: 2023/03/31

Table of Contents

1 Overview

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

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

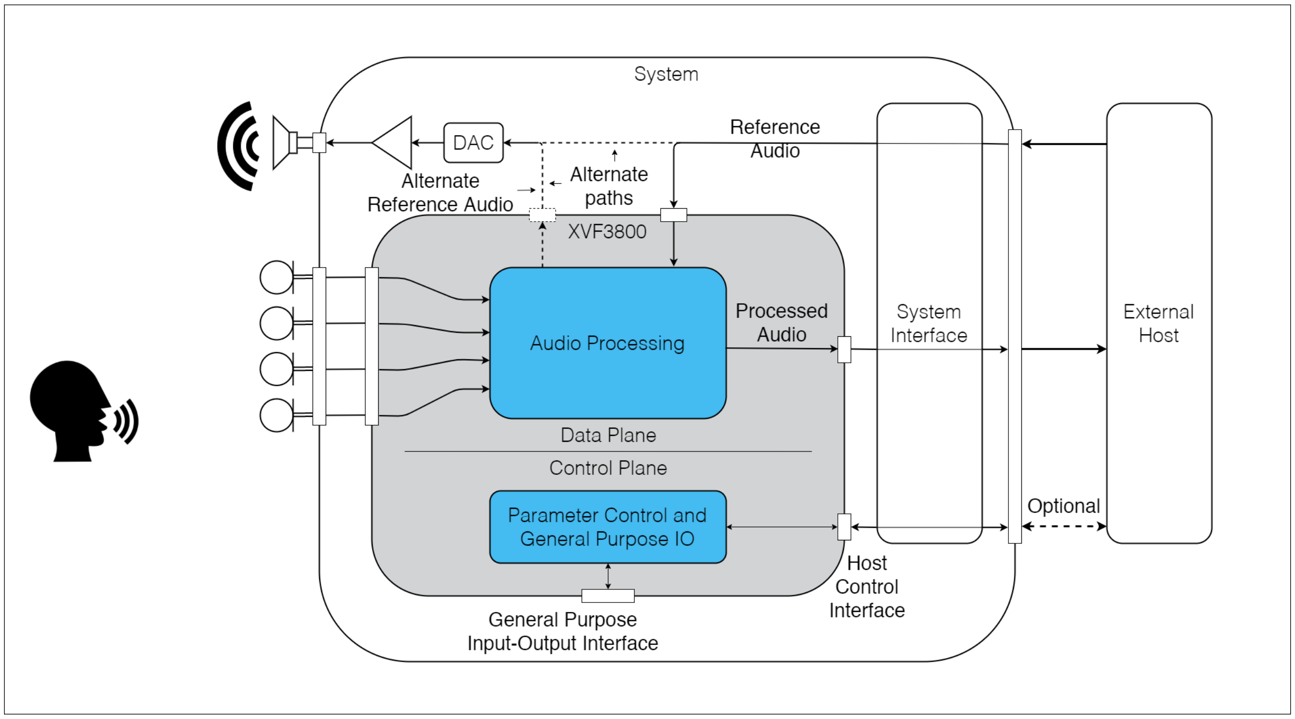

Fig. 1.1: Context Diagram

This document provides information on:

- The XVF3800's theory of operation,
- Advanced options for configuring and building the firmware,
- Methods for testing changes to the XVF3800 firmware, and
- Areas within the XVF3800 firmware intended for customisation.

2 Theory of Operation

2.1 System Architecture Overview

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

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

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

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

2.1.1 Control Plane Modules

The Control Plane includes the following modules:

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

2.1.2 Data Plane Modules

The Data Plane includes the following modules:

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

- Software Phase-Locked Loop (SW PLL)

2.2 Control Plane Module Responsibilities

2.2.1 Device Control

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

2.2.2 Device Firmware Update Controller

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

2.2.3 General Purpose Input Output

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

2.2.4 Human Interface Device

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

2.2.5 Input Output Configuration

The Input Output Configuration module configures GPIO devices and an attached Digital to Analogue Converter (DAC).

2.2.6 Inter-Integrated Circuit Master

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

2.2.7 Inter-Integrated Circuit Slave

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

2.2.8 Quad Serial Peripheral Interface

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

2.2.9 Serial Peripheral Interface Slave

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

2.2.10 Servicers

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

2.3 Data Plane Module Responsibilities

2.3.1 Acoustic Echo Cancellation

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

2.3.2 Audio Manager

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

2.3.3 Beamforming and Post-processing

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

2.3.4 Customer DSP

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

2.3.5 Inter-IC Sound

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

2.3.6 Microphone Array

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

2.3.7 Software Phase-Locked Loop

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

2.4 Product Configurations

The XVF3800 supports three primary use cases:

- Integrated device,
- Integrated host, and
- USB accessory.

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

The integrated host use case embeds the XVF3800 within a system that does not include a separate, primary microcontroller. Instead, the system includes a component capable of converting the reference and processed microphone signals between I^{2}S and some desired transport protocol such as Bluetooth or WiFi. The customer adds functionality to the XVF3800 to perform any necessary system tasks beyond the audio processing already provided. Such functionality may include power-on configuration of additional components, responding to signals on the general-purpose input pins and generating signals for the general-purpose output pins.

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

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

Table 2.1: Use Case Interface Variations

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

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

A system diagram for each use case appears in Fig. 2.1, Fig. 2.2, and Fig. 2.3.

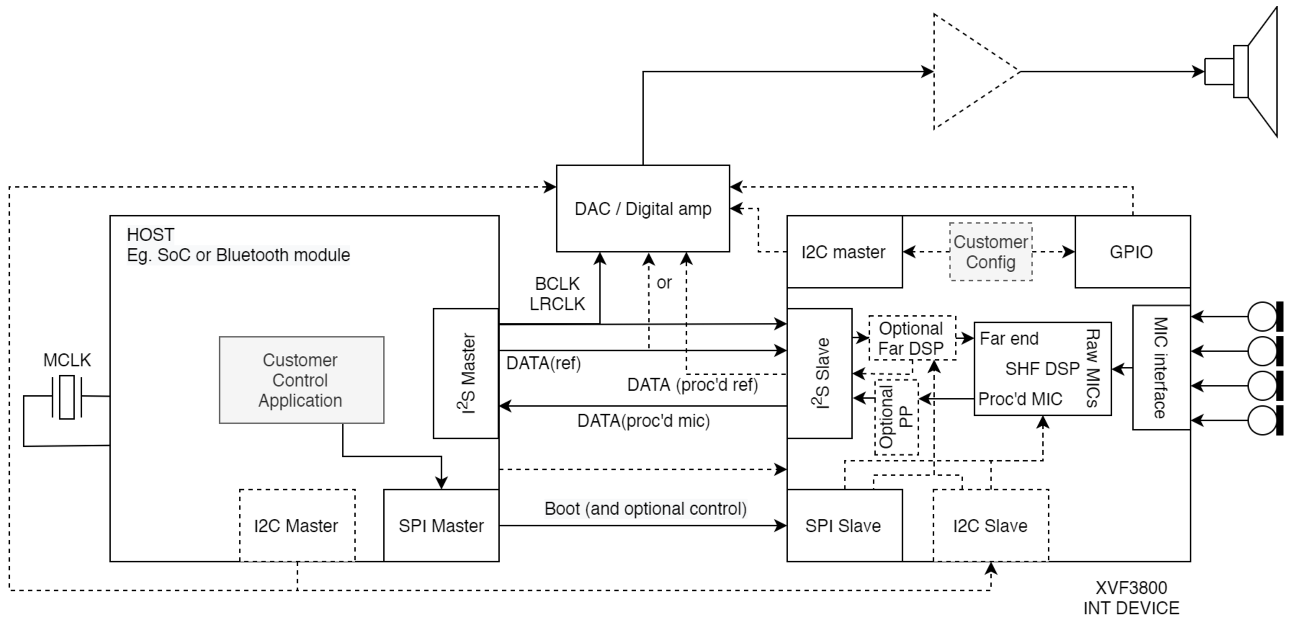

Fig. 2.1: XVF3800 Integrated Device System Diagram

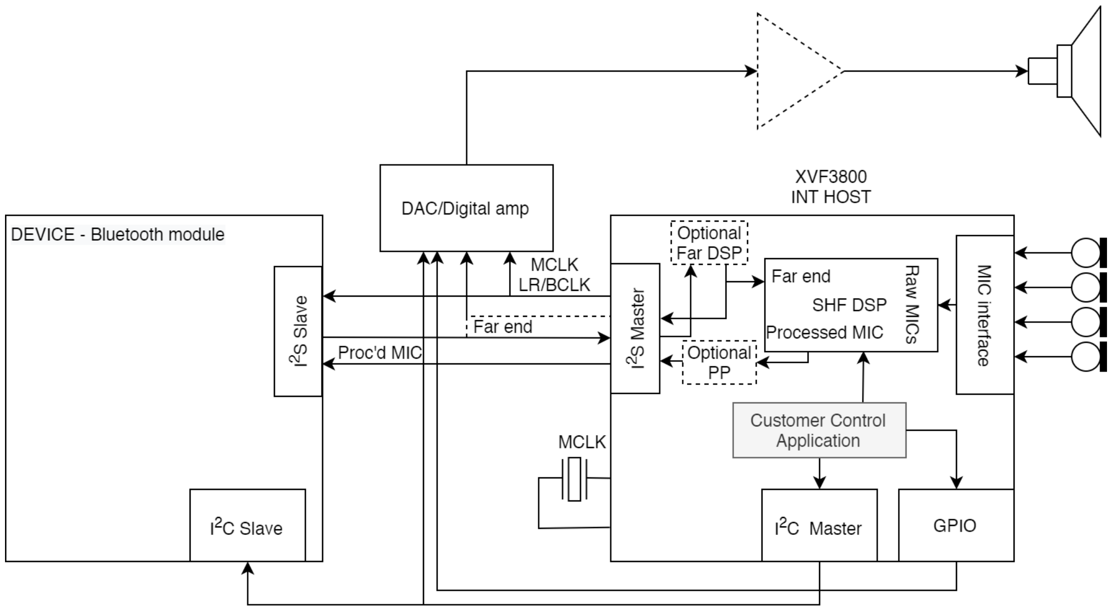

Fig. 2.2: XVF3800 Integrated Host System Diagram

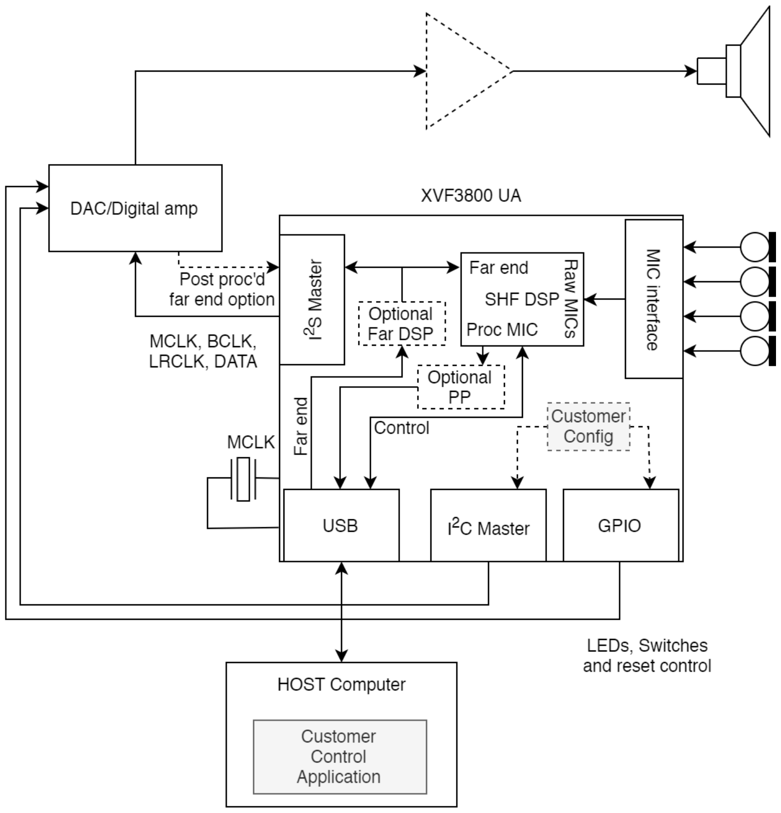

Fig. 2.3: XVF3800 USB Accessory System Diagram

2.5 Module Placement and Interconnection

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

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

One diagram each appears for the Integrated Host (Fig. 2.6) and USB Accessory (Fig. 2.7) use cases. The Integrated Device use case supports a control data transport over either SPI or I^{2}C, so two diagrams (Fig. 2.4 and Fig. 2.5) appear for it.

2.5.1 Integrated Device with SPI Control

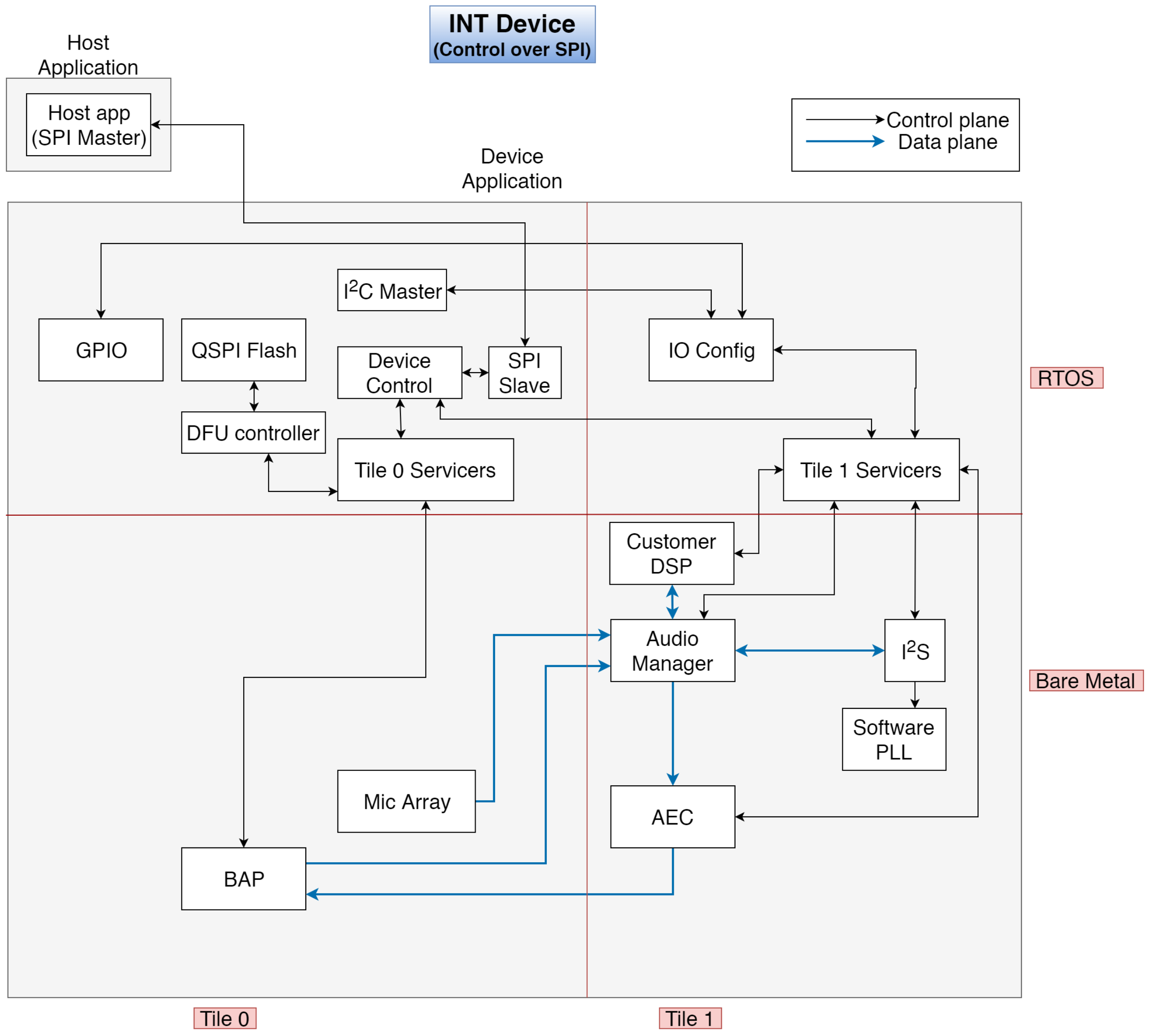

Fig. 2.4: XVF3800 Integrated Device (SPI control) Location Diagram

2.5.2 Integrated Device with I^{2}C Control

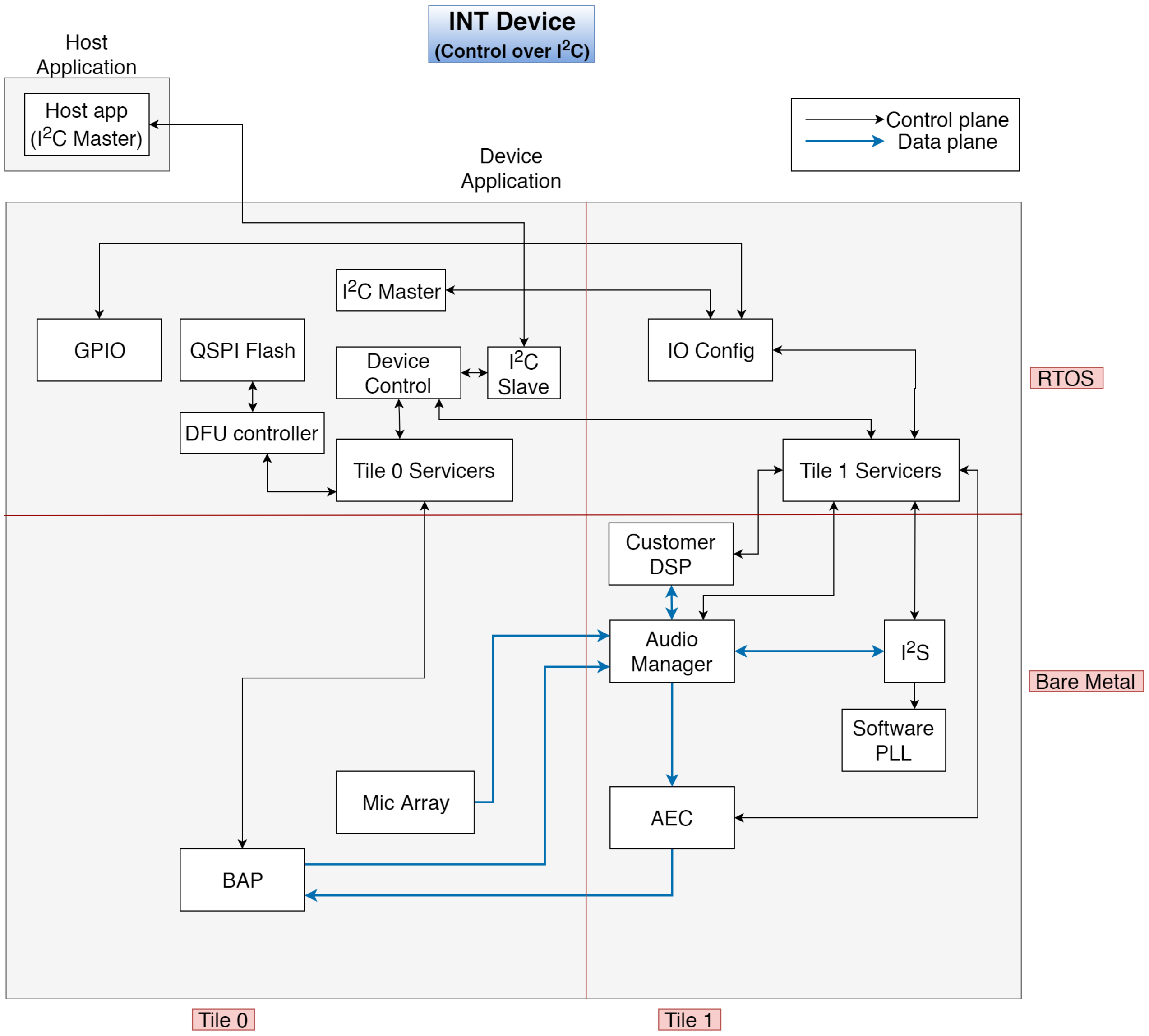

Fig. 2.5: XVF3800 Integrated Device (I^{2}C control) Location Diagram

2.5.3 Integrated Host

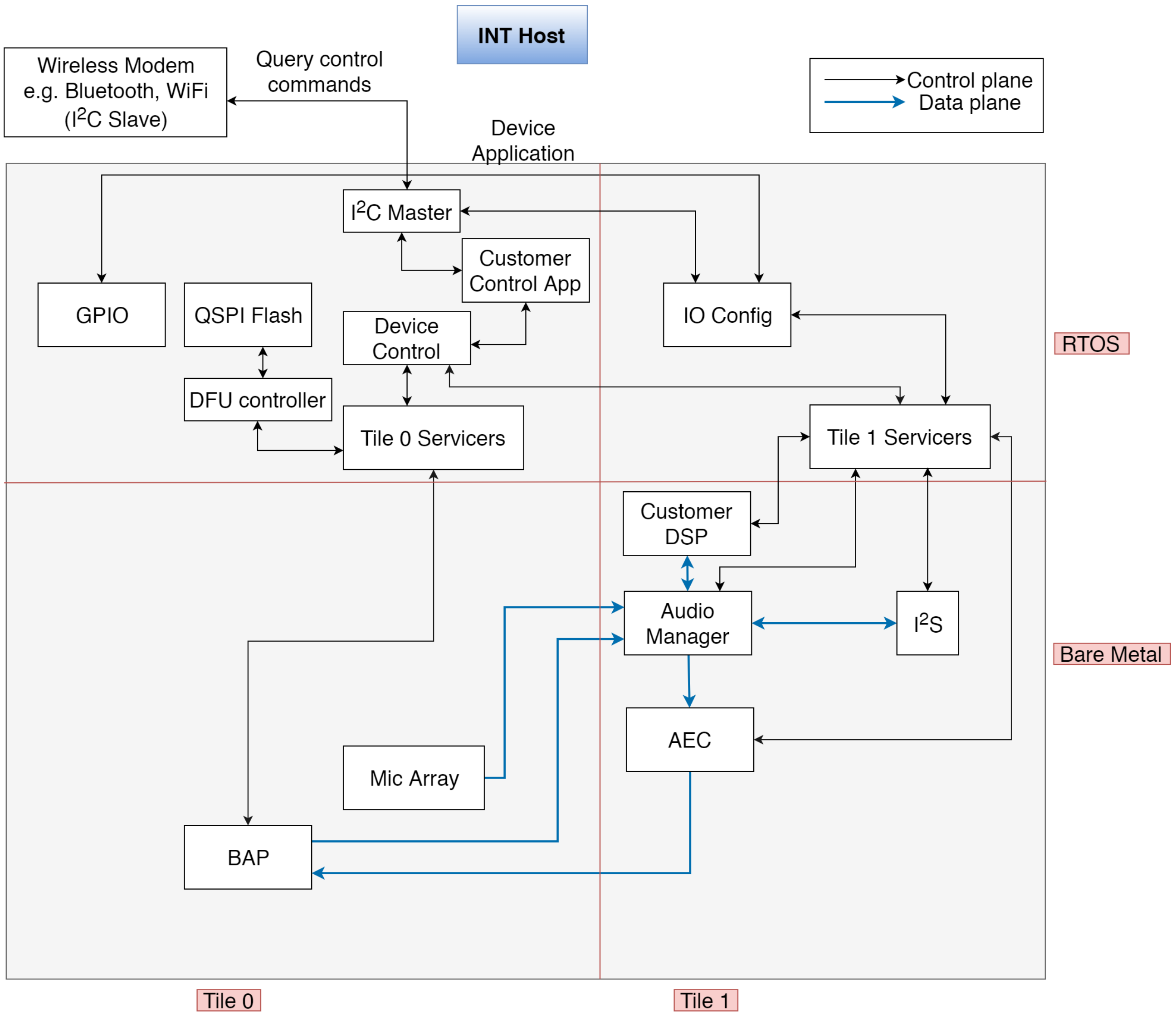

Fig. 2.6: XVF3800 Integrated Host Location Diagram

Note: The firmware design for the Integrated Host use case is still under development and subject to change.

2.5.4 USB Accessory

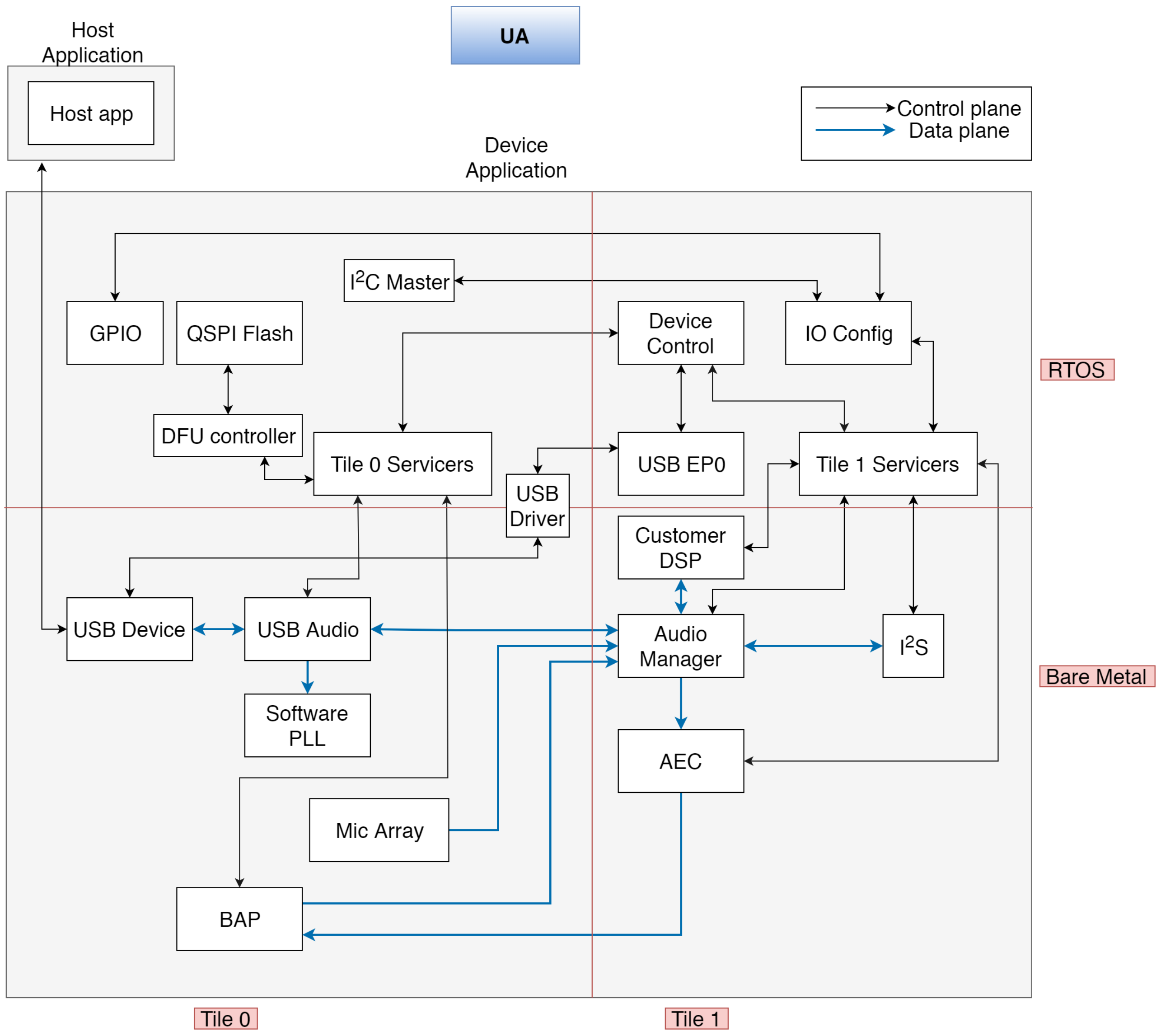

Fig. 2.7: XVF3800 USB Accessory Location Diagram

Note: The firmware design for the USB Accessory use case is still under development and subject to change.

2.6 Control Plane Detailed Design

2.6.1 Control Plane Structure and Operation

Fig. 2.8 shows the modules involved in processing control commands. In order to concentrate on their processing, it does not include Control Plane modules, such as the DFU controller, HID, I^{2}C Master or QSPI, that are not directly involved with control command processing.

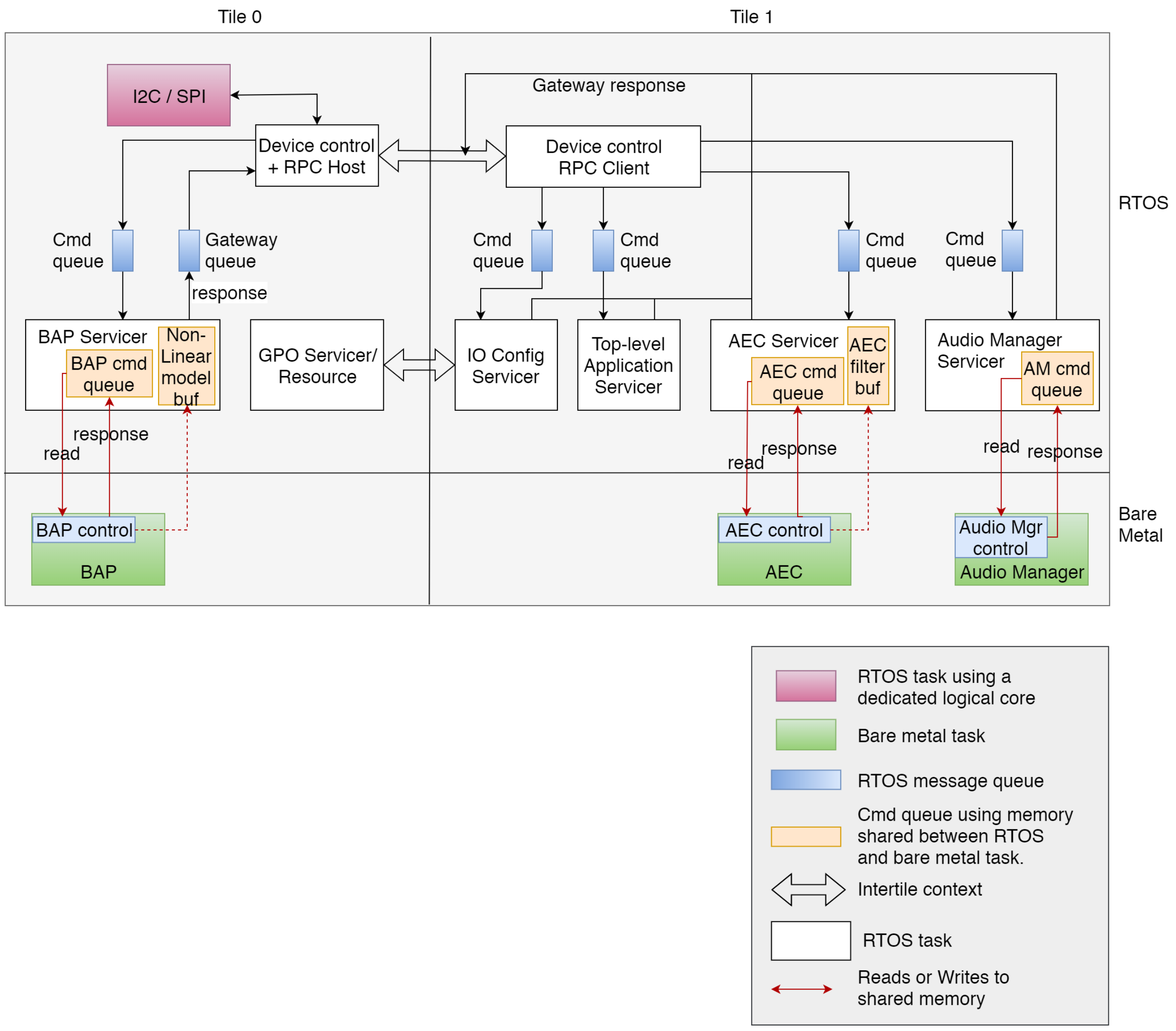

Fig. 2.8: XVF3800 Control Plane Components Diagram

Fig. 2.9 shows the interaction between the Device Control module and a Servicer. In this diagram, boxes with the same colour reside in the same RTOS task.

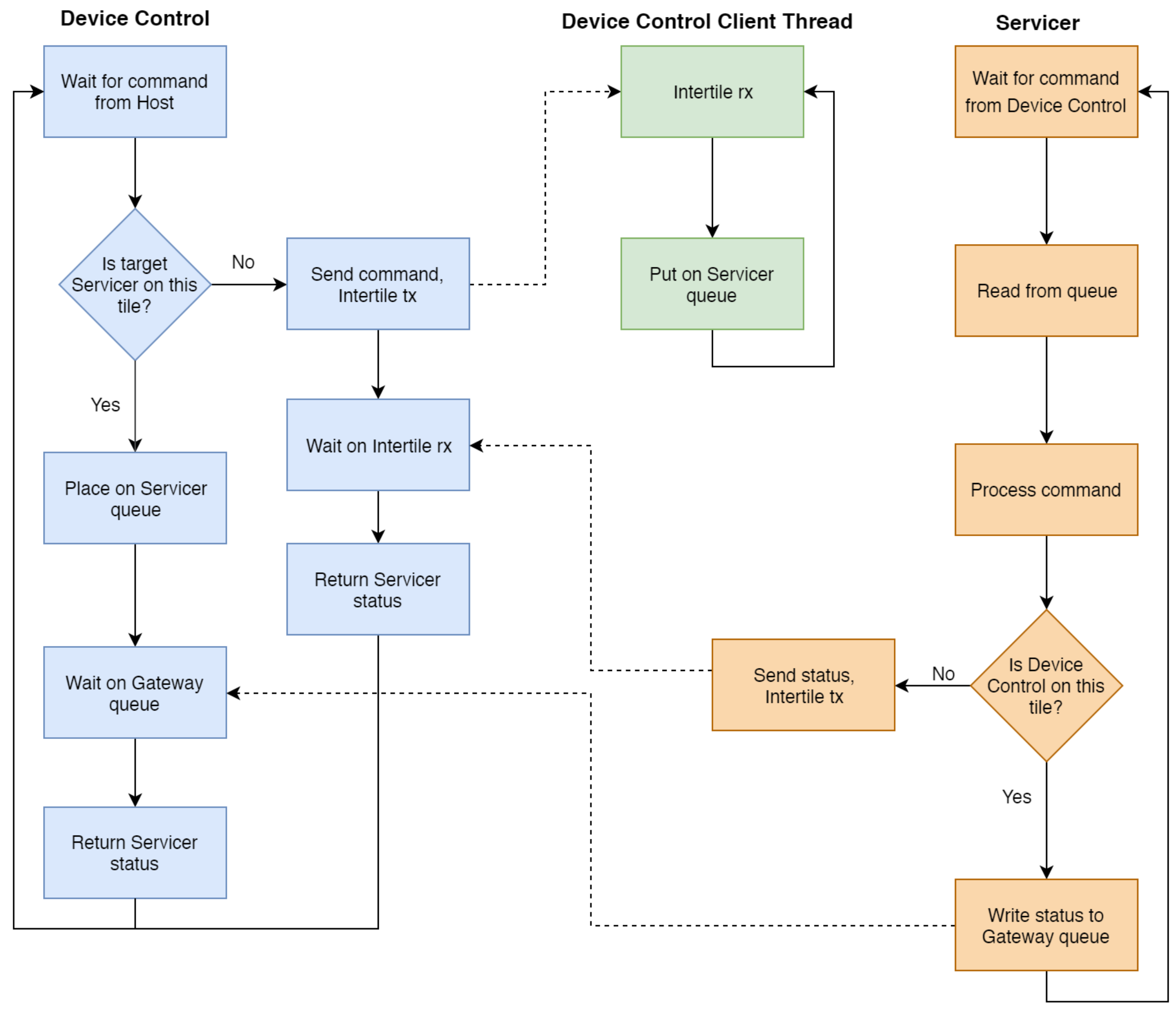
Device Control
Device Control Client Thread
Servicer
Wait for command from Host
Is target Servicer on this tile?
No
Yes
Send command, Intertile tx
Wait on Intertile rx
Return Servicer status
Place on Servicer queue
Wait on Gateway queue
Return Servicer status
Intertile rx
Put on Servicer queue
Wait for command from Device Control
Read from queue
Process command
Is Device Control on this tile?
No
Yes
Send status, Intertile tx
Write status to Gateway queue

Fig. 2.9: XVF3800 Device Control – Servicer Flow Chart

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

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

2.6.2 Control Protocol

The XVF3800 uses a packet protocol to receive control commands and send each corresponding response. Because packet transmission occurs over a very short-haul transport, e.g. I^2C or SPI, or as the payload within a USB packet, the protocol does not include fields for error detection or correction such as start-of-frame and end-of-frame symbols, a cyclical redundancy check or an error correcting code. Fig. 2.10 depicts the structure of each packet.

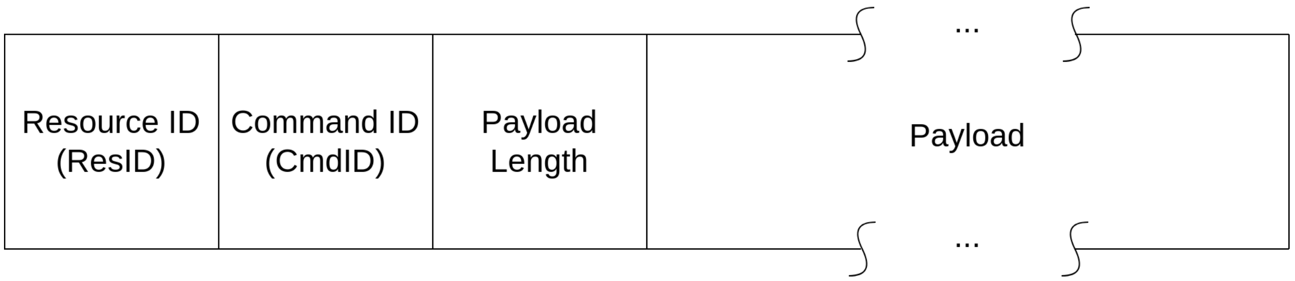

Fig. 2.10: XVF3800 Control Plane Packet Diagram

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

2.7 Data Plane Detailed Design

Fig. 2.11 shows the activities within each Data Plane logical core.

The portion of the Customer DSP module that allows processing of the reference signal prior to use by the Acoustic Echo Cancellation module appears in the I^{2}S logical core. The other portion of the Customer DSP module, which allows further processing of the audio produced by the Beamforming and Post-processing module, appears in the Audio Manager logical core.

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

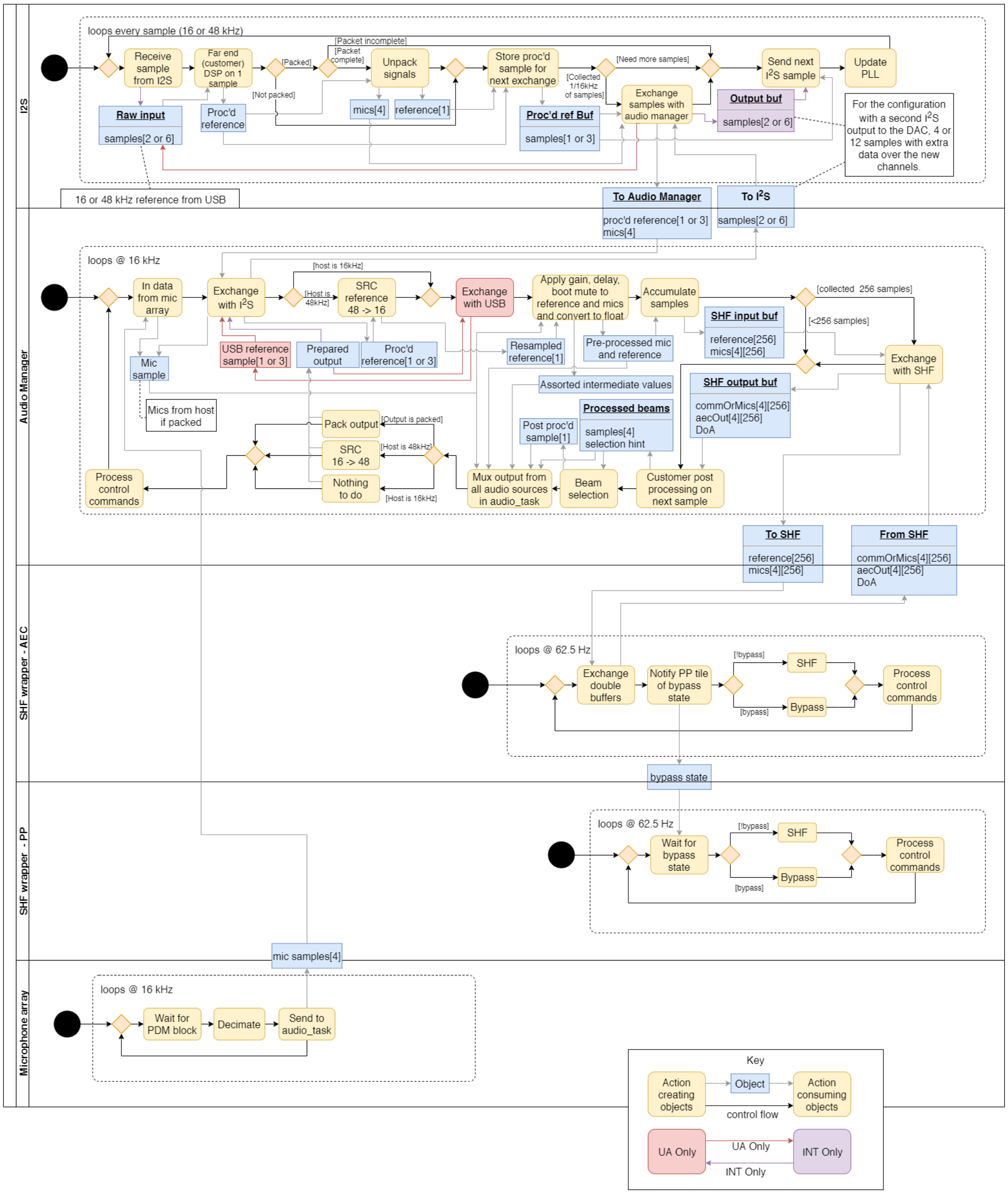

Fig. 2.11: XVF3800 Data Plane Activity Diagram

3 Building the Software

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

3.1 Building the Firmware

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

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

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

The CMake build process follows two stages. The first is configuration; in this stage the script in the top level `CMakeLists.txt` is run, which in turn runs many other `CMakeLists.txt` and CMake scripts. To run the configuration a number of parameters need to be passed into CMake. In order to ensure the correct flags are used, a `CMakePresets.json` has been included which provides a configuration preset named "rel_app_xvf3800". Configuration will generate the Makefiles for building the executable in the directory specified by the preset. The XVF3800 build process is designed so that one CMake configuration defines all target executables so managing multiple configurations is not required.

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

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

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

3.1.1 Adding or Modifying Build Configurations

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

1. Define a variable named `BUILD_PROFILES` which is a list of the names of all the targets.
2. Iterate through the build profiles, defining a list of definitions, source files and libraries for each one. A set of patterns are checked for common build flags; this is documented as comments in `CMakeLists.txt`.
3. Iterate through them again defining the libraries and executables required for the target. Including the extra flags defined in the previous step.

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

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

# ...

foreach(PROFILE ${BUILD_PROFILES})

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

    # 2. Add build flags for the target with the new name.
    #    Note that this must be *above* the check for EXTRA_BUILD_INCLUDED.
    if(PROFILE STREQUAL "my-custom-target")
        add_to_build_opts("INT_DEVICE=1")
        add_to_build_opts("appconfLRCLK_NOMINAL_HZ=48000")
        add_to_build_opts("appconfSPI_CTRL_ENABLED=0")
        add_to_build_opts("appconfI2C_CTRL_ENABLED=1")
        add_to_build_opts("appconfUSER_CONFIG_ENABLED=0")
        add_to_build_sources(src/default_params/product_defaults.c)
        add_to_build_libs(sw_xvf3800::bsp_config::xk_voice_sq66_evk)

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

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

3.1.2 Adding New Files and Compilation Flags to the Build

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

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

```
sources/applications/app_xvf3800
├── CMakeLists.txt
└── my_custom_lib
    ├── CMakeLists.txt
    ├── my_custom_source.c
    ├── my_other_source.c
    └── my_custom_source.h
```

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

```
add_subdirectory(my_custom_lib)
```

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

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

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

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

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

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

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

```
add_to_build_libs(my_custom_lib)
```

3.2 Building the Host Control App

The host control application is distributed with the release as a binary alongside its associated shared libraries, targeting the Raspberry Pi. The control app requires awareness of all the control parameters in the software.

The control parameters are defined in YAML files that are located at `sources/applications/app_xvf3800/cmd_map_gen/yaml_files` in the source release. When one of these files is modified it is not necessary to rebuild the whole host control application; only `libcommand_map.so` needs to be updated.

`libcommand_map.so` is compiled from source files that are generated based on the YAML files. Updating `libcommand_map.so` requires a Raspberry Pi with:

- the XVF3800 source release, including all sub-modules
- CMake with minimum version 3.13
- Python with the non-standard packages *pyyaml* and *jinja2*.

CMake can be installed using from any location:

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

The non-standard Python packages can be installed using from any location:

```
pip install pyyaml jinja2
```

To build the command map, run the following commands on the Raspberry Pi from the source release package:

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

The compiled `libcommand_map.so` should then be copied to the same directory as `xvf_host`, and it will be used automatically. The build process is defined by `sources/host_cmd_map/CMakeLists.txt`. This includes finding the YAML files, generating the source code and finally building the shared library.

4 Testing the Software

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

4.1 Test Capabilities

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

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

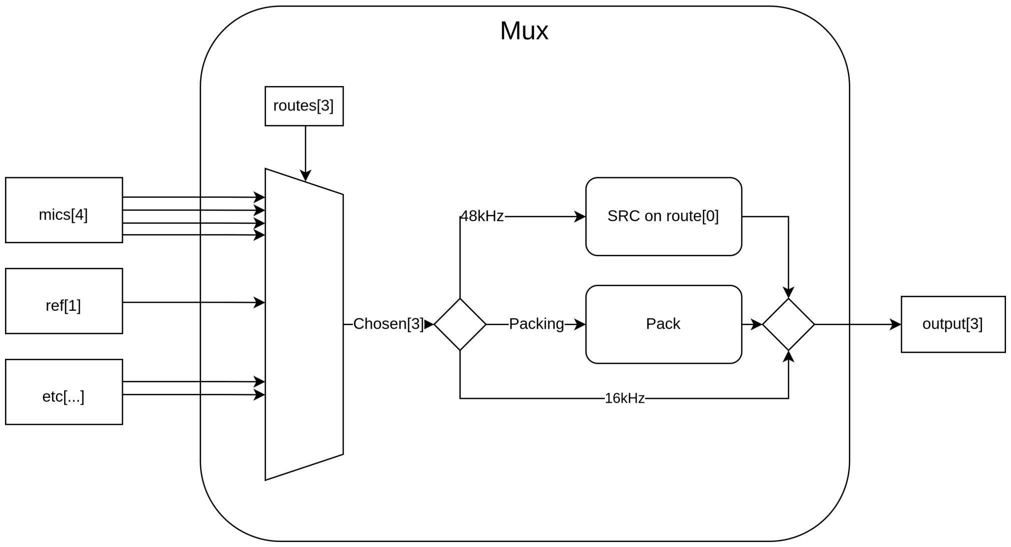

Fig. 4.1: Representation of a single output channel's mux block

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

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

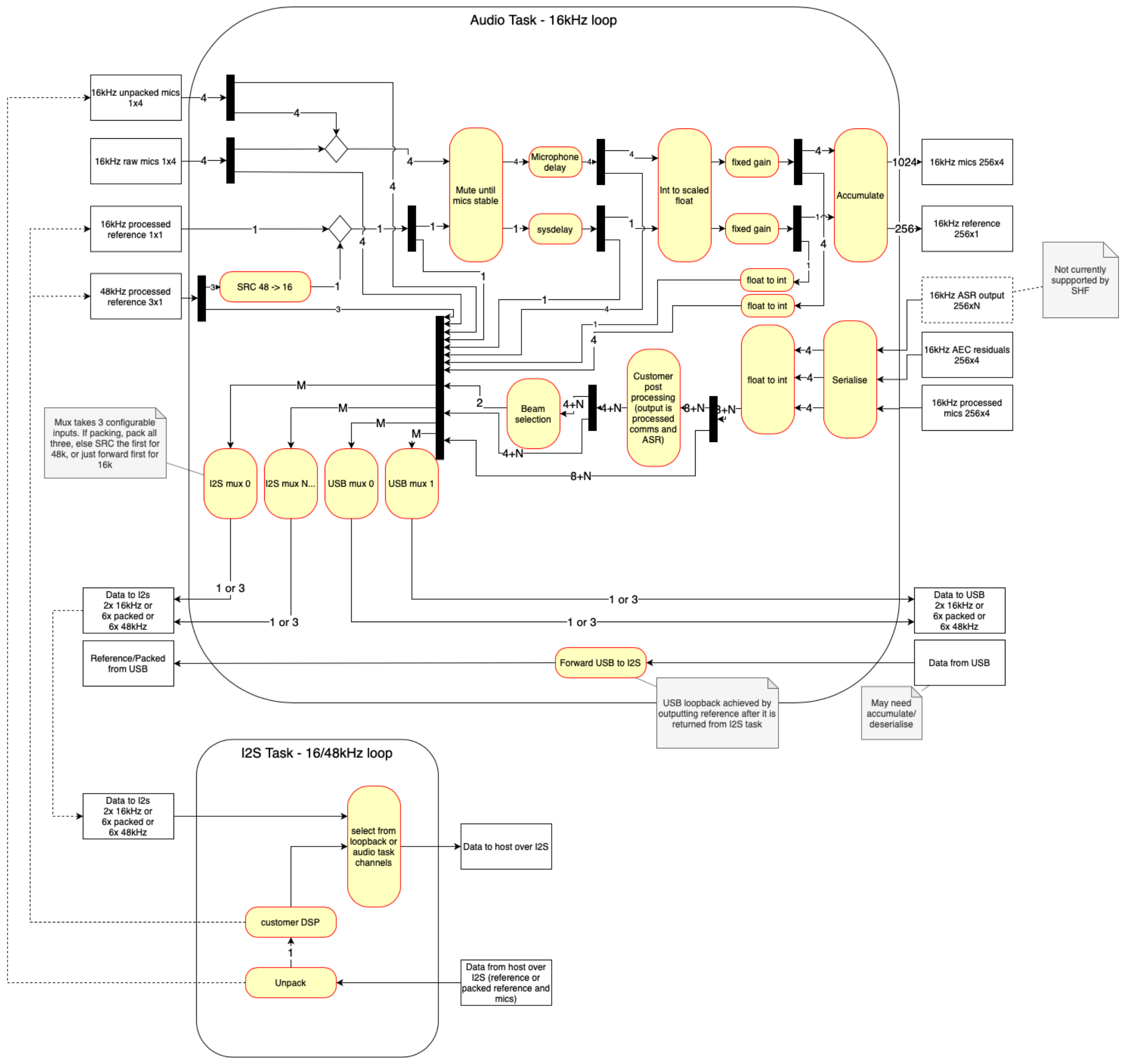
Audio Task - 16kHz loop
16kHz unpacked mics 1x4
16kHz raw mics 1x4
16kHz processed reference 1x1
48kHz processed reference 3x1
SRC 48 -> 16
Mute until mics stable
Microphone delay
sysdelay
Int to scaled float
fixed gain
fixed gain
Accumulate
16kHz mics 256x4
16kHz reference 256x1
16kHz ASR output 256xN
Not currently suppported by SHF
16kHz AEC residuals 256x4
16kHz processed mics 256x4
float to int
float to int
float to int
Serialise
Customer post processing (output is processed comms and ASR)
Beam selection
Mux takes 3 configurable inputs. If packing, pack all three, else SRC the first for 48k, or just forward first for 16k
I2S mux 0
I2S mux N...
USB mux 0
USB mux 1
Data to I2s 2x 16kHz or 6x packed or 6x 48kHz
Reference/Packed from USB
Data to USB 2x 16kHz or 6x packed or 6x 48kHz
Data from USB
Forward USB to I2S
USB loopback achieved by outputting reference after it is returned from I2S task
May need accumulate/ deserialise
I2S Task - 16/48kHz loop
Data to I2s 2x 16kHz or 6x packed or 6x 48kHz
select from loopback or audio task channels
Data to host over I2S
customer DSP
Unpack
Data from host over I2S (reference or packed reference and mics)

Fig. 4.2: Complete audio path through the XVF3800

4.1.1 Loopbacks

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

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

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

In this example, the mux will be modified so that the far-end reference is looped back to the audio output stream. This can be useful for evaluating propagation delays and/or volume scaling within the complete system. The first command ensures that the native signal is used (instead of being down-sampled and then up-sampled) and the second command routes the raw far-end DSP signal to the output mux. Note that any far-end DSP used will be included in this signal path and so it can also be used to check its operation. See *Modifying the Software* for an example of adding far-end DSP.

```
./xvf_host AUDIO_MGR_OP_UPSAMPLE 0 0
./xvf_host AUDIO_MGR_OP_ALL 10 0 10 2 10 4 10 1 10 3 10 5
```

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

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

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

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

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

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

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

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

```
./xvf_host AUDIO_MGR_OP_UPSAMPLE 1 1
# Set the left and right outputs to silence
./xvf_host AUDIO_MGR_OP_L 0 0
./xvf_host AUDIO_MGR_OP_R 0 0
```

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

4.1.2 Signal Capture

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

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

To capture the voice-DSP pipeline input, use the following commands which capture the mono far-end signal (with delay and gain if enabled), the four amplified (and optionally delayed) raw-mic signals and the processed output (autoselect beam in this case):

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

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

```
# Play the desired far-end reference signal in the background
aplay <my_48kHz_stereo_far_end_reference_signal.wav> &
# Run a stereo audio capture at 48 kHz with 32b bit depth for 60 sec
arecord -r 48000 -f S32_LE -c 2 -d 60 <capture_48k_2ch.wav>
```

(continues on next page)

(continued from previous page)

```
# Unpack the 48 kHz stereo packed file into a 6ch 16 kHz unpacked wav using 32b sample␣
↪depth
./packing.py unpack <capture_48k_2ch.wav> <unpacked_16k_6ch.wav> -b 32
```

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

- Channel 1 is the left far-end reference, post delay and amplification
- Channel 2 is the processed output (autoselect beam)
- Channel 3 is the amplified raw MIC 0, post delay
- Channel 4 is the amplified raw MIC 1, post delay
- Channel 5 is the amplified raw MIC 2, post delay
- Channel 6 is the amplified raw MIC 3, post delay

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

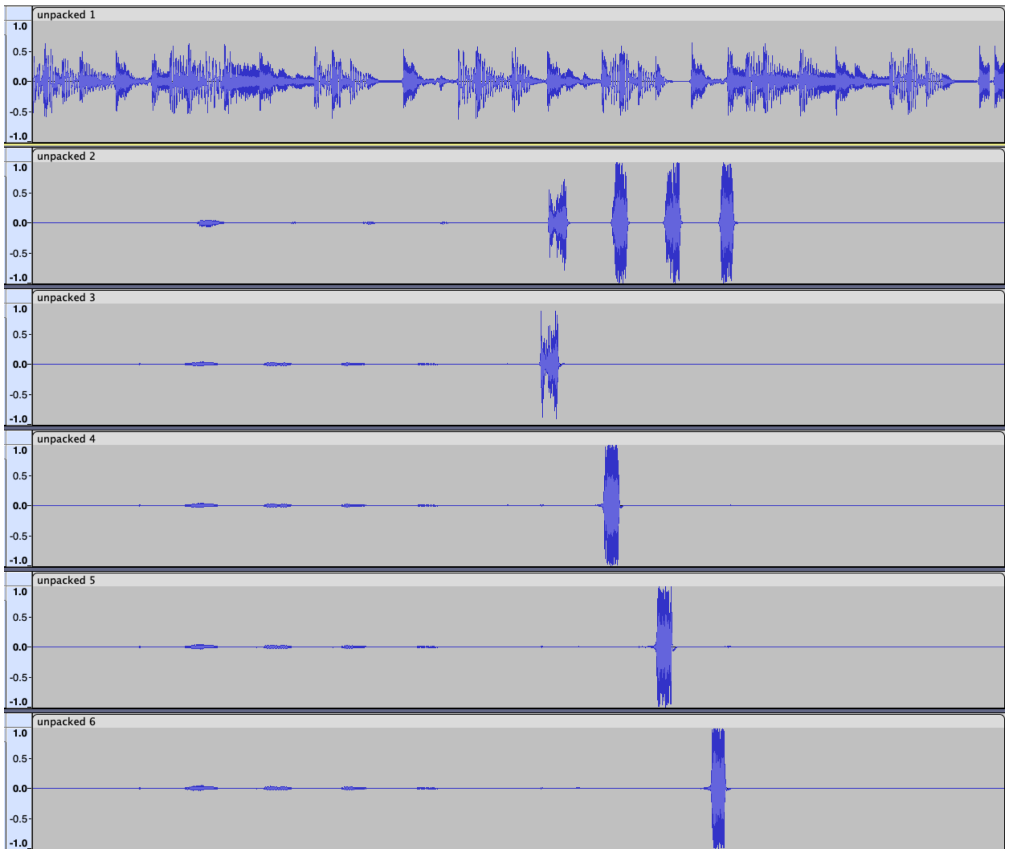

Fig. 4.3: Screenshot of the unpacked output from the XVF3800

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

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

If the following output is observed (with no output file generated), it means that either the audio mux in the XVF3800 has not been configured to properly produce a packed output, the sample resolution is incorrect or the capture process has been corrupted (perhaps by volume scaling). Re-check the mux configuration commands and host system controls.

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

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

4.1.3 Signal Injection

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

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

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

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

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

- Channel 1 is the far-end reference signal
- Channel 2 is ignored
- Channel 3 is the amplified raw MIC 0
- Channel 4 is the amplified raw MIC 1
- Channel 5 is the amplified raw MIC 2
- Channel 6 is the amplified raw MIC 3

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

Next, turn the six channel 16 kHz test vector into a 48 kHz packed file:

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

With the firmware freshly booted and running, configure the input to accept a packed signal:

```
# Configure I2S to unpack a 48 kHz packed input
./xvf_host I2S_INPUT_PACKED 1
# Set mic gain to 1.0. This is needed if the original vector is captured after amplifying
# raw mics; otherwise, skip this command
./xvf_host AUDIO_MGR_MIC_GAIN 1.0
```

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

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

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

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

4.1.4 Signal Injection and Capture Simultaneously

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

- The raw inputs to the voice-DSP to ensure correct transport and injection/capture.
- The delayed mic/far-end inputs to check that the input to the AEC is causal (the far-end reference must arrive before the acoustically coupled echo).
- The amplified mics to ensure that the microphone amplifier gain has been tuned correctly.
- Multiple beam outputs to help determine which of the beams performs best in the desired application.
- The AEC residual signals to determine how much to tune the post-processing stages to trade off echo cancellation versus double-talk performance
- Processed far-end DSP to ensure that the far-end DSP is performing as expected.
- . . . and many more

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

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

(continues on next page)

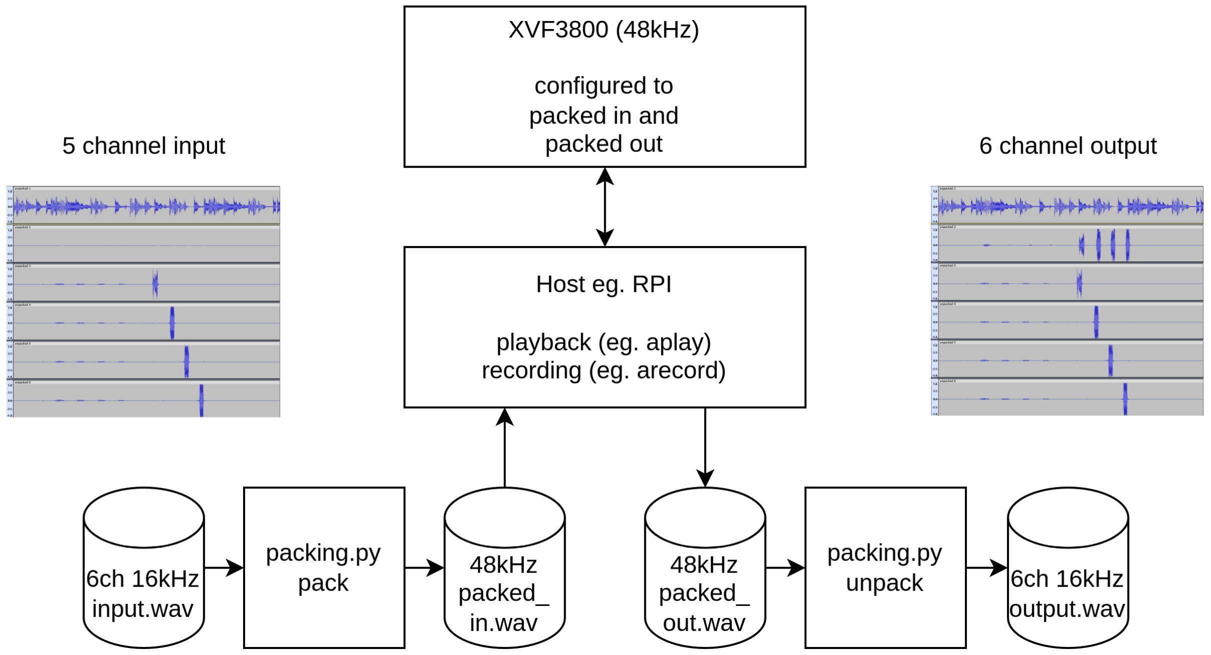

Fig. 4.4: Representation of a fully HIL workflow with the XVF3800

(continued from previous page)

```
# Configure I2S to unpack a 48 kHz packed input
./xvf_host I2S_INPUT_PACKED 1
# Set mic gain to 1.0. This is needed if the vector is capture after post-amplified raw
# mics; otherwise, skip this command
./xvf_host AUDIO_MGR_MIC_GAIN 1.0

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

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

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

4.2 Measuring Resources

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

4.2.1 Measuring Available Cycles

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

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

A number of control commands are available to assist with verifying that timing has not been violated. These commands report idle time, which is the amount of free time available within the audio loops. As this number approaches zero there is an increasing risk of violating timing. A basic command help string is available by executing `./xvf_host --list-commands`. A more detailed description of the function of these commands can be seen below. Note that the *TEST_CORE_BURN* command is hidden from --list-commands since it is used only in test situations.

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

The commands in Table 4.1 provide information and control mechanisms relevant to achieving the necessary timing constraints.

Table 4.1: Control commands relevant to measuring timing

Function	Comment
I2S_CURRENT_IDLE_TIME	This command provides the current idle time of the last I^{2}S loop executed in system timer ticks of 10 ns. This reports the real-time value and should only be used as an indication of how much the processing varies. Note that the amount of idle time heavily depends on the frequency of the I^{2}S interface. For example, a 16 kHz I^{2}S interface will offer significantly more idle time than the 48 kHz setting due to a 3x shorter cycle.
I2S_MIN_IDLE_TIME	This command provides the minimum idle time of all I^{2}S loops executed in system timer ticks of 10 ns. This reports the worst case value since boot or since the idle time metric was last reset. It is the value that should be used to close timing when adding DSP.
I2S_RESET_MIN_IDLE_TIME	Use this control to reset the minimum idle time.
AUDIO_MGR_CURRENT _IDLE_TIME	This command provides the current idle time of the last audio manager loop executed in system timer ticks of 10 ns. This reports the real-time value and should only be used as an indication of how much the processing varies.
AUDIO_MGR_MIN _IDLE_TIME	This command provides the minimum idle time of all audio manager loops executed in system timer ticks of 10 ns. This reports the worst case value since the idle time metric was last reset. It is the value that should be used to close timing when adding DSP.
AUDIO_MGR_RESET_MIN _IDLE_TIME	Use this control to reset the minimum idle time.
SHF_BYPASS	The bypass command causes the far-field voice pipeline to be bypassed and the raw (but amplified) microphone signals to be passed to the output. In addition, it adds poll loops to burn processor cycles up to the maximum available for the voice pipeline section of the design. This means it exercises a tougher timing case compared with running the normal voice-DSP operation and so should be used when ascertaining the worst-case.
TEST_CORE_BURN	This hidden command places the firmware in 'burn' mode. Note this resets the firmware and consequently subsequent commands may be ignored while the firmware initialises. Because it performs a reset, all previously written parameters will be lost and all internal state will be set to the defaults that would be expected following power-on-reset. For this reason, it is recommended that this command be executed at the beginning of the timing-closure session.

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

This means a typical sequence of commands to ascertain worst-case timing would be:

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

(continues on next page)

(continued from previous page)

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

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


# Enable the voice-DSP and reset the minimum idle times
./xvf_host SHF_BYPASS 0
./xvf_host I2S_RESET_MIN_IDLE_TIME 1
./xvf_host AUDIO_MGR_RESET_MIN_IDLE_TIME 1

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


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

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

4.2.2 Measuring Available Memory

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

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

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

```
Constraint check for tile[0]:
Memory available:       524288,   used:       6980 .  OKAY
(Stack: 356, Code: 4064, Data: 2560)
```

(continues on next page)

(continued from previous page)

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

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

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

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

5 Modifying the Software

5.1 Adding a Control Command

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

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

- `application_cmds.yaml` - This is where build information is accessed and some test features. The application servicer does not directly connect to any peripherals, however commands requiring internal storage or calling a user API may be added here.
- `audio_cmds.yaml` - This is where high-level aspects of the audio framework including SRC, packing, I^{2}S and user-DSP are accessed. If extending the DSP capabilities of the design it is likely that commands may be added here, for example to control user-DSP. See *Adding Custom Digital Signal Processing*. Note that two tasks are controlled by this servicer (Audio Manager and I^{2}S) with the I^{2}S task being accessed via a shared-memory structure.
- `io_config_cmds.yaml` - This is where GPIO parameters are accessed. Commands are already provided for manipulating many aspects of these pins, although any custom requirements involving GPIO access may be added here.
- `aec_cmds.yaml` - This is where high level voice-DSP parameters are accessed as well as AEC information. The voice-DSP is not modifiable other than the published API. It is not expected that this file will need to be modified.
- `pp_cmds.yaml` - This is where high level voice-DSP parameters are accessed as well as post processing information. The voice-DSP is not modifiable other than the published API. It is not expected that this file will need to be modified.

5.1.1 Adding a new control command

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

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

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

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

The following access permissions may be assigned to parameters:

```
CMD_READ_ONLY
CMD_WRITE_ONLY
CMD_READ_WRITE
```

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

1. list of intervals - each interval is listed using the syntax *[A .. B]*
   - the syntax is the same for both integers and float values
   - multiple intervals can be specified, for example *[0 .. 5, 10 .. 15]*
   - all the intervals must be closed, meaning that they include all the limit points
   - if only one value is valid, the range can be specified as *[E .. E]*
2. any value is valid - this is declared using the word *any* and the range depends on the maximum and minimum values of the specific type. For example, *TYPE_UINT8* can have values from 0 to 255.

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

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

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

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

```
- cmd: MY_INTERNAL_REGISTER
  number_of_values: 1
  type: CMD_READ_WRITE
  help: A simple example of setting / getting a variable in the firmware
  value_type: TYPE_UINT32
```

Next, in the appropriate servicer C file, add the handlers for the command. In this case we are adding the following code to `applications/app_xvf3800/src/control_plane/application_servicer.c`:

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

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

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

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

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

Next, build the firmware and host app; see *Building the Software* for instructions on this. Test the new command:

```
./xvf_host MY_INTERNAL_REGISTER
0
./xvf_host MY_INTERNAL_REGISTER 1066
./xvf_host MY_INTERNAL_REGISTER
1066
```

5.2 Adding Custom Digital Signal Processing

The XVF3800 supports the addition of user DSP at two parts of the signal path. These points are:

- Far-end DSP between the far-end (reference) input and the start of the far-field voice pipeline. This allows the far-end signal to be processed to allow for speaker/amplifier imperfections and a copy of the pre-processed far-end be sent to the voice pipeline (and optionally over I^2S to the DAC) to ensure optimum AEC performance.
- Voice post-processing. While the voice processing offers a wide range of typically needed functions such as AGC, high-pass filtering and automatic beam selection, some users may wish to augment these functions.

Fig. 5.1 shows the audio paths for the far-end DSP as well as where up/down-sampling may occur. Voice post-processing occurs immediately after the voice pipeline and before the processed microphone signals are sent to the host.

Note: USB functionality is not currently supported in the XVF3800.

Both DSP hooks provide a sample-based processing API. This means a single sample is processed at each time. The reason for this is to reduce latency (block based algorithms introduce a minimum latency of the block size) and to simplify the integration into the main firmware framework. Various DSP functions are available in lib_xcore_math including FIR filters and Biquad IIR filters. An example of the latter is given further on.

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

The API for the user-DSP functions is transcluded below from `applications/app_xvf3800/src/user_dsp/user_dsp.h`:

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

#ifndef __USER_DSP_H_
#define __USER_DSP_H_

#include "aec_cmds.h"
#include "shf_wrapper.h"
#include <stddef.h>
```

(continues on next page)

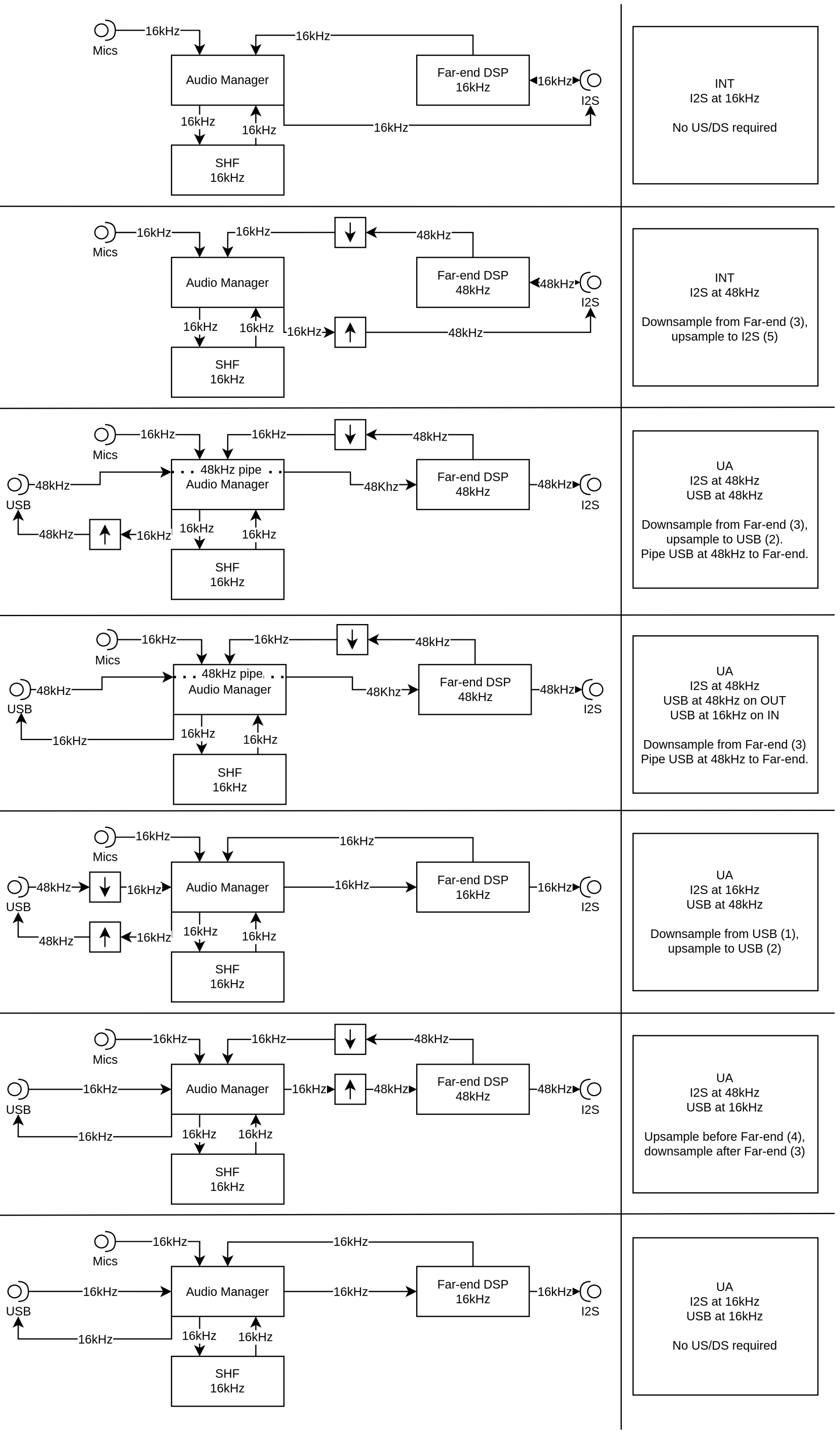
Mics
Audio Manager
SHF 16kHz
Far-end DSP 16kHz
Far-end DSP 48kHz
48kHz pipe
USB
I2S
16kHz
48kHz
48Khz
INT I2S at 16kHz No US/DS required
INT I2S at 48kHz Downsample from Far-end (3), upsample to I2S (5)
UA I2S at 48kHz USB at 48kHz Downsample from Far-end (3), upsample to USB (2). Pipe USB at 48kHz to Far-end.
UA I2S at 48kHz USB at 48kHz on OUT USB at 16kHz on IN Downsample from Far-end (3) Pipe USB at 48kHz to Far-end.
UA I2S at 16kHz USB at 48kHz Downsample from USB (1), upsample to USB (2)
UA I2S at 48kHz USB at 16kHz Upsample before Far-end (4), downsample after Far-end (3)
UA I2S at 16kHz USB at 16kHz No US/DS required

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

(continued from previous page)

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


/// @brief callback to pre-process one sample of far end before outputting to DAC/SHF DSP␣
↪input
/// Note that this callback runs at the I2S rate (16 or 48 kHz).
/// @param far_end_sample      input and output (sample is processed in place)
/// @param far_end_dsp_enable  Set to 1 to enable, 0 to disable. This is handled by the␣
↪user.
void far_end_dsp(int32_t far_end_samples[BECLEAR_NUMBER_OF_FAR], bool far_end_dsp_enable);

/// @brief callback to post-process one sample of audio after the SHF voice DSP stage
/// Note that this callback runs at the SHF sample rate 16kHz.
/// @param out processed output buffer
/// @param post_shf_processed_mic_samples input
/// @param azimuth direction of arrival data on the 3 tracking beams (not auto select),␣
↪Note
/// that this only gets updated every 256 samples.
/// @param aec_residuals output of the BeClear AEC stage
void post_shf_dsp(int32_t out[BECLEAR_NUMBER_OF_OUTPUTS],
                  int32_t post_shf_processed_mic_samples[BECLEAR_NUMBER_OF_OUTPUTS],
                  int32_t aec_residuals[BECLEAR_NUMBER_OF_MICS],
                  float azimuths[AEC_RESID_AEC_AZIMUTH_VALUES_NUM_VALUES]);

#define USER_DSP_NUM_OUTPUT_CHANNELS 2

/// @brief called immediately after post_shf_dsp and will be used to determine the
/// channels that MUX_USER_CHOSEN_CHANNELS will consist of. It also sets the azimuth
/// of each chosen channel so that it can be requested via control command.
///
/// @param[in,out] out_idx 2 chosen channels from `out` which were written by
///    the function `post_shf_dsp`. Before being passed to beam_selection,
///    out_idx will be populated by the suggested indices.
/// @param[out] out_azimuths azimuths of the two channels that have been
///    selected. Most likely a copy of the correct input from above.
void beam_selection(uint8_t out_idx[USER_DSP_NUM_OUTPUT_CHANNELS],
                    float out_azimuths[USER_DSP_NUM_OUTPUT_CHANNELS]);

#endif
```

5.2.1 Meeting Timing

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

5.2.2 Adding control to user DSP

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

The Far-end reference DSP already has a built-in control which is *AUDIO_MGR_FAR_END_DSP_ENABLE*. It is possible to add further controls if needed.

5.2.3 Far-end Reference

The far-end DSP offers the opportunity to add processing between the host audio signal and the DAC output. This is particularly important if adding non-linear processing (eg. bass-enhancement, dynamic-range compression) which will cause a degradation in AEC performance if the far-end reference to the voice pipeline differs from what is being played through the speaker.

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

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

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

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

- Increase `appconfNUM_I2S_PINS_OUT` in `app_conf.h` from 1 to 2. This will enable PORT_I2S_DATA2 as an I^{2}S output pin. The default is to set the second pin to output the post-processed far-end input with sample rate conversion disabled.
- Set `USE_FAR_END_DSP` in `far_end_dsp.c` to 1. Far-end DSP is kept out of the standard build to avoid using extra memory and processing cycles, and to avoid modifying the far-end when not needed.
- Make sure the DAC input is connected to the far-end DSP processed signal. If using USB, it is possible that the hardware will not easily support routing the second I^{2}S line to the DAC. Therefore, in the USB build, the pin formerly used as the I^{2}S input to the device may be repurposed as an I^{2}S output. To take advantage of this, keep `appconfNUM_I2S_PINS_OUT` set to 1 and instead use the audio MUX command to route the post-processed far-end signal to the output. The following commands ensure up-sampling is disabled and route far-end DSP signal to the output:

    ./xvf_host AUDIO_MGR_OP_UPSAMPLE 0 0 ./xvf_host AUDIO_MGR_OP_ALL 10 0 10 2 10 4 10 1 10 3 10 5

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

```
./xvf_host AUDIO_MGR_FAR_END_DSP_ENABLE 1
./xvf_host AUDIO_MGR_FAR_END_DSP_ENABLE 0
```

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

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

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

#include "user_dsp.h"

#ifndef USE_FAR_END_DSP
    #define USE_FAR_END_DSP 0
#endif

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

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

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

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

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


void far_end_dsp(int32_t far_end_samples[BECLEAR_NUMBER_OF_FAR], bool far_end_dsp_enable)
{
    // See note in user_dsp.h and user documentation about timing constraints
    if(far_end_dsp_enable)
    {
```

(continues on next page)

(continued from previous page)

```
        for(int channel = 0; channel < BECLEAR_NUMBER_OF_FAR; channel++)
        {
            far_end_samples[channel] >>= 1; // Simple 6db pre-attenuate to account for gain␣
↪in filter
            far_end_samples[channel] = filter_biquad_s32(&filter[channel], far_end_
↪samples[channel]);
        }
    } else {
        for(int channel = 0; channel < BECLEAR_NUMBER_OF_FAR; channel++)
        {
            far_end_samples[channel] >>= 1; // Simple 6db attenuate to account for filter␣
↪gains to give similar apparent volume
        }
    }
}

#else

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

#endif
```

Testing the effect on available cycles, we can see a modest drop from the three stage biquad of just 85 x 10 ns = 0.85 us; this is due in part to the use of the Vector Processing Unit (VPU), which is highly efficient for signal processing purposes. Note that the idle time is not calculated until I^{2}S runs:

```
./xvf_host TEST_CORE_BURN 1
./xvf_host I2S_MIN_IDLE_TIME
2083
aplay <short wav>
aplay <short wav>
aplay <short wav>
./xvf_host I2S_MIN_IDLE_TIME
1245
./xvf_host AUDIO_MGR_FAR_END_DSP_ENABLE 1
aplay <short wav>
aplay <short wav>
aplay <short wav>
./xvf_host I2S_MIN_IDLE_TIME
1160
```

5.2.4 Voice post-processing

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

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

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

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

5.3 Modifying Existing Functionality

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

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

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

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

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

    rtos_printf("user_init_hardware\n");

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

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

(continues on next page)

(continued from previous page)

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

    errors_encountered |= dac3101_init(appconfLRCLK_NOMINAL_HZ);
#endif

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

    return errors_encountered;
}
```

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

In this file we can see the following actions taken:

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

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

5.3.1 Digital to Analogue Converter Configuration

Each DAC or digital amplifier selected will normally have its own set of registers that need to be configured. There are two parts to the DAC configuration code. The first is the abstraction layer responsible for providing the I^{2}C, GPO, and wait functions. These may need to be modified if the GPO responsible for resetting the DAC or the I^{2}C register access methods needs to be altered. This file can be seen below, transcluded from `applications/app_xvf3800/src/user_config/dac_port.c`:

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

(continues on next page)

(continued from previous page)

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

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

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


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

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

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

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


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

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

The second file, which specifies the sequence of GPO accesses and I^{2}C register writes specific to the chosen DAC, can be found in `applications/bsp_config/dac`. There are two files which may need to be modified: `dac3101.h`, which contains the defines and function prototypes, and `dac3101.c`, which contains the *dac3101_init()* function that is called from `user_config.c` and performs the sequence of operations required to configure the DAC. These sources are not printed here for documentation brevity. Note that error detection is included and errors (non-zero return) will be reported back to the application if encountered.

5.3.2 General Purpose Input and Output Operation

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

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

The initial configuration of the roles of each GPO pin can be found in the function *init_gpo()* in `applications/app_xvf3800/src/control_plane/gpo_servicer.c`. This contains the GPO setup for the XVF3800 demonstration board including initial level and drive invert. Drive invert can be useful for negative logic hardware such as LEDs connected between the 3v3 rail and the GPO pin.

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

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

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