XCORE-VOICE SOLUTION - Programming Guide

Publication Date: 2024/12/18
Document Number: v2.3.0

XMOS

IN THIS DOCUMENT

1 Product Description

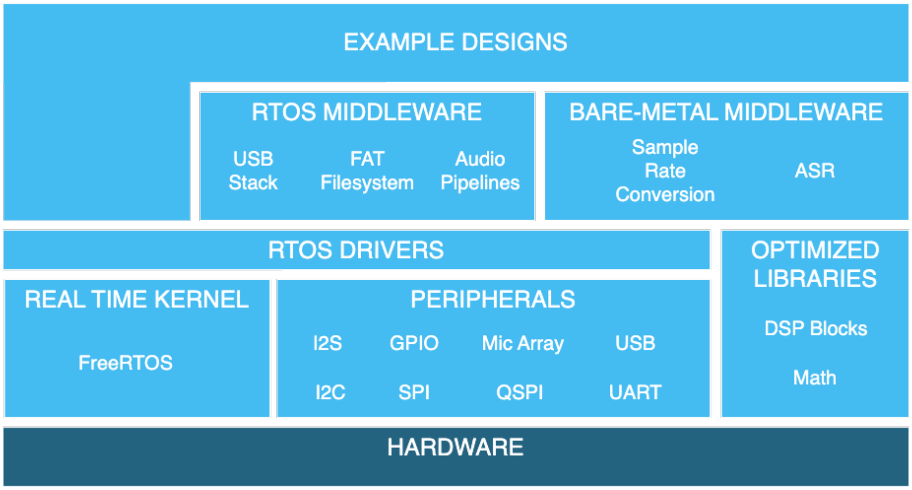
The XCORE-VOICE Solution consists of example designs and a C-based SDK for the development of audio front-end applications to support far-field voice use cases on the xcore.ai family of chips (XU316). The XCORE-VOICE examples are currently based on FreeRTOS or bare-metal, leveraging the flexibility of the xcore.ai platform and providing designers with a familiar environment to customize and develop products.

XCORE-VOICE example designs include turn-key solutions to enable easier product development for smart home applications such as light switches, thermostats, and home appliances. xcore.ai's unique architecture providing powerful signal processing and accelerated AI capabilities combined with the XCORE-VOICE framework allows designers to incorporate keyword, event detection, or advanced local dictionary support to create a complete voice interface solution. Bridging designs including PDM microphone to host aggregation are also included showcasing the use of xcore.ai as an interfacing and bridging solution for deployment in existing systems.

The C SDK is composed of the following components:

- Peripheral IO libraries including; UART, I^{2}C, I^{2}S, SPI, QSPI, PDM microphones, and USB. These libraries support bare-metal and RTOS application development.

- Libraries core to DSP applications, including vectorized math and voice processing DSP. These libraries support bare-metal and RTOS application development.
- Libraries for speech recognition applications. These libraries support bare-metal and RTOS application development.
- Libraries that enable multi-core FreeRTOS development on xcore including a wide array of RTOS drivers and middleware.
- Pre-build and validated audio processing pipelines.
- Code Examples - Examples showing a variety of xcore features based on bare-metal and FreeRTOS programming.
- Documentation - Tutorials, references and API guides.



2 Key Features

The XCORE-VOICE Solution takes advantage of the flexible software-defined xcore-ai architecture to support numerous far-field voice use cases through the available example designs and the ability to construct user-defined audio pipeline from the SW components and libraries in the C-based SDK.

These include:

Voice Processing components

- Two PDM microphone interfaces
- Digital signal processing pipeline
- Full duplex, stereo, Acoustic Echo Cancellation (AEC)
- Reference audio via I^{2}S with automatic bulk delay insertion
- Point noise suppression via interference canceller
- Switchable stationary noise suppressor
- Programmable Automatic Gain Control (AGC)

- Flexible audio output routing and filtering
- Support for Sensory, Cyberon or other 3rd party Automatic Speech Recognition (ASR) software

Device Interface components

- Full speed USB2.0 compliant device supporting USB Audio Class (UAC) 2.0
- Flexible Peripheral Interfaces
- Programmable digital general-purpose inputs and outputs

Example Designs utilizing above components

- Far-Field Voice Local Command
- Low Power Far-Field Voice Local Command
- Far-Field Voice Assistance

Firmware Management

- Boot from QSPI Flash
- Default firmware image for power-on operation
- Option to boot from a local host processor via SPI
- Device Firmware Update (DFU) via USB or I^2C

Power Consumption

- FFD/FFVA: 300-350mW (Typical)
- Low Power FFD: 110mW (Full-Power), 54mW (Low-Power), <50mW possible with Sensory's LPSD under certain conditions.

3 Obtaining the Hardware

The XK-VOICE-L71 DevKit and Hardware Manual can be obtained from the XK-VOICE-L71 product information page.

The XK-VOICE-L71 is based on the: XU316-1024-QF60A

The XCORE-AI-EXPLORER DevKit and Hardware Manual used in the *Microphone Aggregation* example can be obtained from the XK-VOICE-L71 product information page.

Learn more about the The XMOS XS3 Architecture

4 Obtaining the Software

4.1 Development Tools

It is recommended that you download and install the latest release of the XTC Tools. XTC Tools 15.3.0 or newer are required. If you already have the XTC Toolchain installed, you can check the version with the following command:

```
xcc --version
```

4.2 Application Demonstrations

If you only want to run the example designs, pre-built firmware and other software can be downloaded from the XCORE-VOICE product information page.

4.3 Source Code

If you wish to modify the example designs, a zip archive of all source code can be downloaded from the XCORE-VOICE product information page.

See the *Programming Guide* for information on:

- Prerequisites
- Instructions for building, running, and debugging the example designs
- Details on the software design and source code

4.3.1 Cloning the Repository

Alternatively, the source code can be obtained by cloning the public GitHub repository.

Note: Cloning requires a GitHub account configured with SSH key authentication.

Run the following *git* command to clone the repository and all submodules:

```
git clone --recurse-submodules git@github.com:xmos/sln_voice.git
```

If you have previously cloned the repository or downloaded a zip file of source code, the following commands can be used to update and fetch the submodules:

```
git pull
git submodule update --init --recursive
```

5 Prerequisites

It is recommended that you download and install the latest release of the XTC Tools. XTC Tools 15.3.0 or newer are required for building, running, flashing and debugging the example applications.

CMake 3.21 or newer and Git are also required for building the example applications.

5.1 Windows

A standard C/C++ compiler is required to build applications for the host PC. Windows users may use Build Tools for Visual Studio command-line interface.

It is recommended to use *Ninja* as the build system for native Windows firmware builds. To install *Ninja* follow install instructions at https://ninja-build.org/ or on Windows install with `winget` by running the following commands in *PowerShell*:

```
# Install
winget install Ninja-build.ninja
# Reload user Path
$env:Path=[System.Environment]::GetEnvironmentVariable("Path","User")
```

XCORE-VOICE host builds should also work using other Windows GNU development environments like GNU Make, MinGW or Cygwin.

5.1.1 libusb

The DFU feature of XCORE-VOICE requires dfu-util.

5.2 macOS

A standard C/C++ compiler is required to build applications for the host PC. Mac users may use the Xcode command-line tools.

6 Example Designs

6.1 Far-field Voice Local Command

6.1.1 Overview

This is the far-field voice local command (FFD) example design. Three examples are provided: all examples include speech recognition and a local dictionary. One example uses the Sensory TrulyHandsfree™ (THF) libraries, and the other ones use the Cyberon DSPotter™ libraries. The two examples with the Cyberon DSPotter™ libraries differ in the audio source fed into the intent engine. One example uses the audio source from the microphone array, and the other uses the audio source from the I^{2}S interface.

The examples using the microphone array as the audio source include an audio pipeline with the following stages:

1. Interference Canceler (IC) + Voice To Noise Ratio Estimator (VNR)
2. Noise Suppressor (NS)
3. Adaptive Gain Control (AGC)

The FFD examples provide several options to inform the host of a possible intent detected by the intent engine. The device can notify the host by:

- sending the intent ID over a UART interface upon detecting the intent
- sending the intent ID over an I^{2}C master interface upon detecting the intent
- allowing the host to poll the last detected intent ID over the I^{2}C slave interface
- listening to an audio message over an I^{2}S interface

When a wakeword phrase is detected followed by a command phrase, the application will output an audio response and a discrete message over I^{2}C and UART.

Sensory's THF and Cyberon's DSpotter™ libraries ship with an expiring development license. The Sensory one will suspend recognition after 11.4 hours or 107 recognition events, and the Cyberon one will suspend recognition after 100 recognition events. After the maximum number of recognitions is reached, a device reset is required to resume normal operation. To perform a reset, either power cycle the device or press the SW2 button.

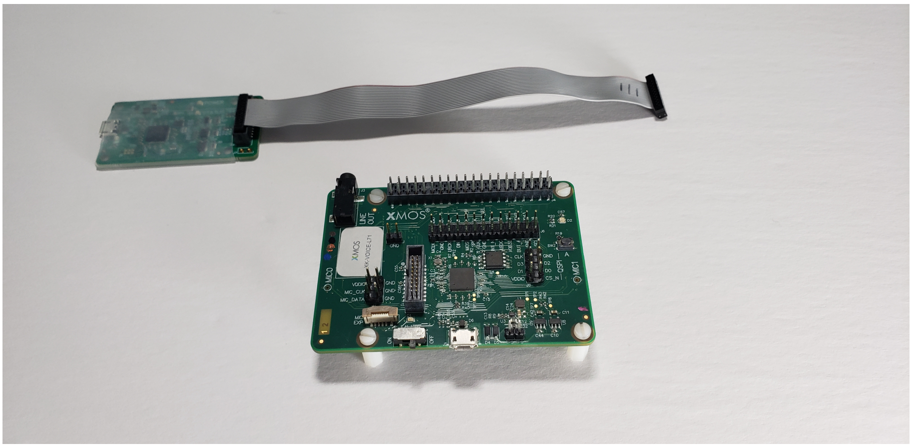
More information on the Sensory speech recognition library can be found here: *Speech Recognition - Sensory*.

More information on the Cyberon speech recognition library can be found here: *Speech Recognition - Cyberon*

6.1.2 Supported Hardware

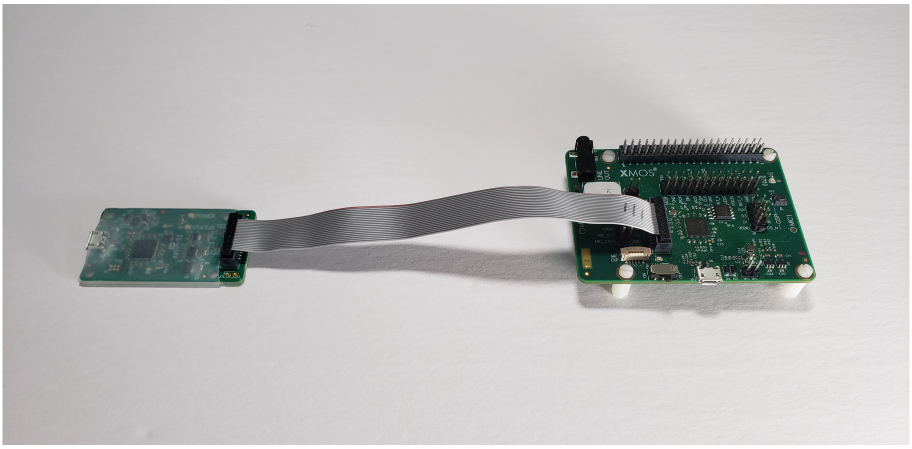
This example application is supported on the XK-VOICE-L71 board.

6.1.2.1 Setting up the Hardware This example design requires an XTAG4 and XK-VOICE-L71 board.

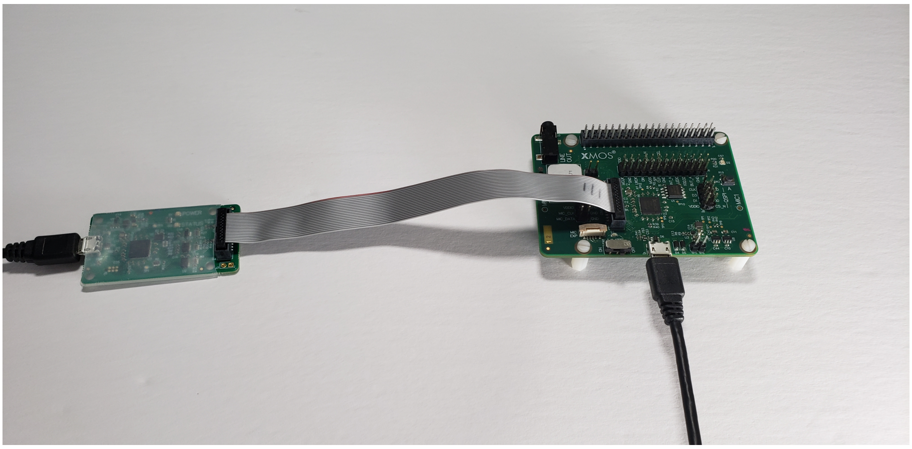


xTAG The xTAG is used to program and debug the device

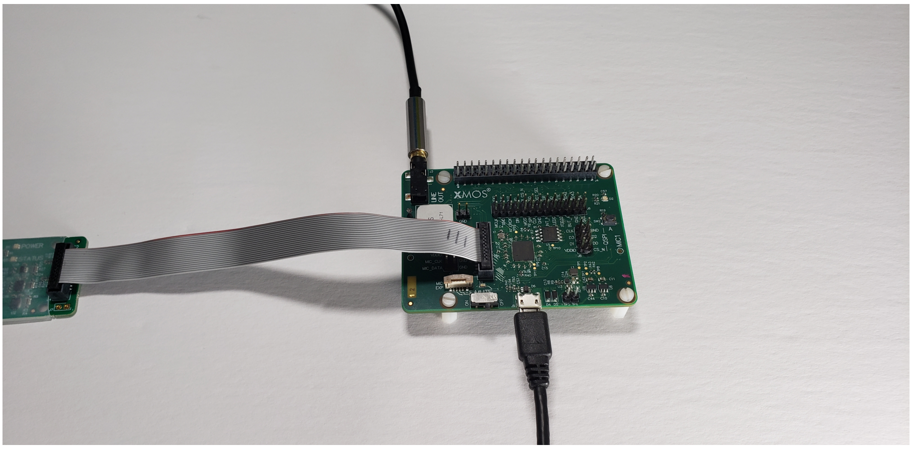
Connect the xTAG to the debug header, as shown below.



Connect the micro USB XTAG4 and micro USB XK-VOICE-L71 to the programming host.



Speakers (OPTIONAL) This example application features audio playback responses. Speakers can be connected to the LINE OUT on the XK-VOICE-L71.



6.1.3 Configuring the Firmware

The default application performs as described in the *Overview*. There are numerous compile time options that can be added to change the example design without requiring code changes. To change the options explained in the table below, add the desired configuration variables to the APP_COMPILE_DEFINITIONS cmake variable in the `.cmake` file located in the `examples/ffd/` folder.

If options are changed, the application firmware must be rebuilt.

Table 1: FFD Compile Options

Compile Option	Description	Default Value
appconfINTENT_ENABLED	Enables/disables the intent engine, primarily for debug.	1

continues on next page

Table 1 – continued from previous page

Compile Option	Description	Default Value
appconfINTENT_RESET_DELAY_MS	Sets the period after the wake up phrase has been heard for a valid command phrase	5000
appconfINTENT_RAW_OUTPUT	Set to 1 to output all keywords found, skipping the internal wake up and command state machine	0
appconfAUDIO_PLAYBACK_ENABLED	Enables/disables the audio playback command response	1
appconfINTENT_UART_OUTPUT_ENABLED	Enables/disables the UART intent message	1
appconfINTENT_UART_DEBUG_INFO_ENABLED	Enables/disables the UART intent debug information	0
appconfI2C_MASTER_DAC_ENABLED	Enables/disables configuring the DAC over I^2C master	1
appconfINTENT_I2C_MASTER_OUTPUT_ENABLED	Enables/disables sending the intent message over I^2C master	1
appconfINTENT_I2C_MASTER_DEVICE_ADDR	Sets the address of the I^2C device receiving the intent via the I^2C master interface	0x01
appconfINTENT_I2C_SLAVE_POLLED_ENABLED	Enables/disables allowing another device to poll the intent message via I^2C slave	0
appconfI2C_SLAVE_DEVICE_ADDR	Sets the address of the I^2C device receiving the intent via the I^2C slave interface	0x42
appconfINTENT_I2C_REG_ADDRESS	Sets the address of the I^2C register to store the intent message, this value can be read via the I^2C slave interface	0x01
appconfUART_BAUD_RATE	Sets the baud rate for the UART tx intent interface	9600
appconfUSE_I2S_INPUT	Replace I^2S audio source instead of the microphone array audio source.	0
appconfI2S_MODE	Select I^2S mode, supported values are appconfI2S_MODE_MASTER and appconfI2S_MODE_SLAVE	master
appconfI2S_AUDIO_SAMPLE_RATE	Select the sample rate of the I^2S interface, supported values are 16000 and 48000	16000
appconfRECOVER_MCLK_I2S_APP_PLL	Enables/disables the recovery of the MCLK from the Software PLL application; this removes the need to use an external MCLK.	0

continues on next page

Table 1 – continued from previous page

Compile Option	Description	Default Value
appconfINTENT_TRANSPORT_DELAY_MS	Sets the delay between host wake up requested and I²C and UART keyword code transmission	50
appconfINTENT_QUEUE_LEN	Sets the maximum number of detected intents to hold while waiting for the host to wake up	10
appconfINTENT_WAKEUP_EDGE_TYPE	Sets the host wake up pin GPIO edge type. 0 for rising edge, 1 for falling edge	0
appconfAUDIO_PIPELINE_SKIP_IC_AND_VNR	Enables/disables the IC and VNR	0
appconfAUDIO_PIPELINE_SKIP_NS	Enables/disables the NS	0
appconfAUDIO_PIPELINE_SKIP_AGC	Enables/disables the AGC	0

Note: The `example_ffd_i2s_input_cyberon` has different default values from the ones in the table above. The list of updated values can be found in the `APP_COMPILE_DEFINITIONS` list in `examples\ffd\ffd_i2s_input_cyberon.cmake`.

6.1.3.1 Configuring the I²C interfaces

The I²C interfaces are used to configure the DAC and to communicate with the host. The I²C interface can be configured as a master or a slave. The DAC must be configured at bootup via the I²C master interface. The I²C master is used when the FFD example asynchronously sends intent messages to the host. The I²C slave is used when the host wants to read intent messages from the FFD example through polling.

Note: The I²C interface cannot operate as both master and slave simultaneously. The FFD example design uses the I²C master interface to configure the DAC at device initialisation. However, if the host reads intent messages from the FFD example using the I²C slave interface, the I²C master interface will be disabled after the DAC configuration is complete.

To send the intent ID via the I²C master interface when a command is detected, set the following variables:

- `appconfINTENT_I2C_MASTER_OUTPUT_ENABLED` to 1.
- `appconfINTENT_I2C_MASTER_DEVICE_ADDR` to the desired address used by the I²C slave device.
- `appconfINTENT_I2C_SLAVE_POLLED_ENABLED` to 0, this will disable the I²C slave interface.

To configure the FFD example so that the host can poll for the intent via the I²C slave interface, set the following variables:

- `appconfINTENT_I2C_SLAVE_POLLED_ENABLED` to 1.
- `appconfI2C_SLAVE_DEVICE_ADDR` to the desired address used by the I^2C master device.
- `appconfINTENT_I2C_REG_ADDRESS` to the desired register read by the I^2C master device.
- `appconfINTENT_I2C_MASTER_OUTPUT_ENABLED` to 0, this will disable the I^2C master interface after initialization.

The handling of the I^2C slave registers is done in the `examples\ffd\src\i2c_reg_handling.c` file. The variable `appconfINTENT_I2C_REG_ADDRESS` is used in the callback function `read_device_reg()`.

6.1.3.2 Configuring the I^2S interface

The I^2S interface is used to play the audio command response to the DAC, and/or to receive the audio samples from the host. The I^2S interface can be configured as either a master or a slave. To configure the I^2S interface, set the following variables:

- `appconfI2S_ENABLED` to 1.
- `appconfI2S_MODE` to the desired mode, either `appconfI2S_MODE_MASTER` or `appconfI2S_MODE_SLAVE`.
- `appconfI2S_AUDIO_SAMPLE_RATE` to the desired sample rate, either 16000 or 48000.
- `appconfRECOVER_MCLK_I2S_APP_PLL` to 1 if an external MCLK is not available, otherwise set it to 0.
- `appconfAUDIO_PLAYBACK_ENABLED` to 1, if the intent audio is to be played back.
- `appconfUSE_I2S_INPUT` to 1, if the I^2S audio source is to be used instead of the microphone array audio source.

6.1.4 Deploying the Firmware with Linux or macOS

This document explains how to deploy the software using *CMake* and *Make*.

Note: In the commands below `<speech_engine>` can be either `sensory` or `cyberon`, depending on the choice of the speech recognition engine and model.

Note: The Cyberon speech recognition engine is integrated in two examples. The `example_ffd_cyberon` use the microphone array as the audio source, and the `example_ffd_i2s_input_cyberon` uses the I^2S interface as the audio source. In the rest of this section, we use only the `example_ffd_<speech_engine>` as an example.

6.1.4.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native Toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -B build_host
cd build_host
make install
```

The host applications will be installed at `/opt/xmos/bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.1.4.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
pip install -r requirements.txt
cmake -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffd_<speech_engine>
```

6.1.4.3 Running the Firmware

Before running the firmware, the filesystem and model must be flashed to the data partition.

Within the root of the build folder, run:

```
make flash_app_example_ffd_<speech_engine>
```

After this command completes, the application will be running.

After flashing the data partition, the application can be run without reflashing. If changes are made to the data partition components, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_ffd_<speech_engine>.xe
```

6.1.4.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_ffd_<speech_engine>.xe
```

6.1.5 Deploying the Firmware with Native Windows

This document explains how to deploy the software using *CMake* and *Ninja*. If you are not using native Windows MSVC build tools and instead using a Linux emulation tool such as WSL, refer to *Deploying the Firmware with Linux or macOS*.

To install *Ninja* follow install instructions at https://ninja-build.org/ or on Windows install with `winget` by running the following commands in *PowerShell*:

```
# Install
winget install Ninja-build.ninja
# Reload user Path
$env:Path=[System.Environment]::GetEnvironmentVariable("Path","User")
```

Note: In the commands below `<speech_engine>` can be either `sensory` or `cyberon`, depending on the choice of the speech recognition engine and model.

Note: The Cyberon speech recognition engine is integrated in two examples. The `example_ffd_cyberon` use the microphone array as the audio source, and the `example_ffd_i2s_input_cyberon` uses the I^2S interface as the audio source. In the rest of this section, we use only the `example_ffd_<speech_engine>` as an example.

6.1.5.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native Toolchain:

Note: Permissions may be required to install the host applications.

Note: A C/C++ compiler, such as Visual Studio or MinGW, must be included in the path.

Before building the host application, you will need to add the path to the XTC Tools to your environment.

```
set "XMOS_TOOL_PATH=<path-to-xtc-tools>"
```

Then build the host application:

```
cmake -G Ninja -B build_host
cd build_host
ninja install
```

The host applications will be installed at `%USERPROFILE%\.xmos\bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.1.5.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
pip install -r requirements.txt
cmake -G Ninja -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
ninja example_ffd_<speech_engine>
```

6.1.5.3 Running the Firmware

Before running the firmware, the filesystem and model must be flashed to the data partition.

Within the root of the build folder, run:

```
ninja flash_app_example_ffd_<speech_engine>
```

After this command completes, the application will be running.

After flashing the data partition, the application can be run without reflashing. If changes are made to the data partition components, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_ffd_<speech_engine>.xe
```

6.1.5.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_ffd_<speech_engine>.xe
```

6.1.6 Modifying the Software

The FFD example design is highly customizable. This section describes how to modify the application.

6.1.6.1 Host Integration

Overview This section describes the connections that would need to be made to an external host for plug and play integration with existing devices.

When an intent is found, the XCORE device will check if the host is awake, by checking the Host Status GPIO pin. If the host is awake the intent code will be transmitted over I^2C and/or UART.

If the host is not awake, the XCORE device will trigger a transition of the Wakeup GPIO pin. This can be configured to be a rising or falling edge. The XCORE device will then wait for a fixed period of time, set at compile time, before transmitting the intent over the I^2C and/or UART interface. This behavior can be changed as desired by modifying the intent handling code.

UART

Table 2: UART Connections

FFD Connection	Host Connection
J4:24	UART RX
J4:20	GND

I^2C

Table 3: I^2C Connections

FFD Connection	Host Connection
J4:3	SDA
J4:5	SCL
J4:9	GND

GPIO

Table 4: GPIO Connections

FFD Connection	Host Connection
J4:19	Wake up input
J4:21	Host Status output

6.1.6.2 Audio Pipeline The audio pipeline in FFD processes two channel PDM microphone input into a single output channel, intended for use by an ASR engine.

The audio pipeline consists of 3 stages.

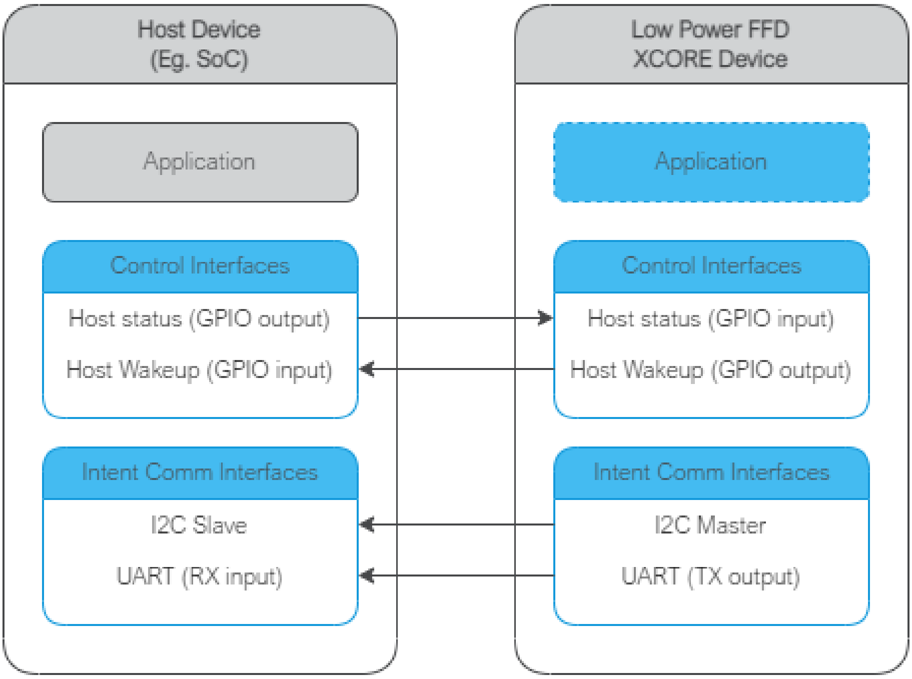


Table 5: FFD Audio Pipeline

Stage	Description	Input Channel Count	Output Channel Count
1	Interference Canceller and Voice Noise Ratio	2	1
2	Noise Suppression	1	1
3	Automatic Gain Control	1	1

See the Voice Framework User Guide for more information.

6.1.6.3 Software Description

Overview The estimated power usage of the example application varies from 100-141 mW. This will vary based on component tolerances and any user added code and/or user added compile options.

Table 6: FFD Resources

Resource	Tile 0	Tile 1
Total Memory Free	145k	208k
Runtime Heap Memory Free	38k	42k

Table 7: FFD CPU Usage

Core ID	Typical Mean CPU Usage (%)	Standard Deviation CPU Usage (%)	Typical Min CPU usage (%, 10ms rolling)	Typical Max CPU usage (%, 10ms rolling)
tile[0], core[0]	0.006	0.345	0.000	21.030
tile[0], core[1]	0.072	2.031	0.000	80.690
tile[0], core[2]	0.082	2.287	0.000	100.000
tile[0], core[3]	1.666	2.906	0.000	54.560
tile[0], core[4]	65.925	27.828	0.000	91.220
tile[1], core[0]	0.014	0.540	0.000	27.440
tile[1], core[1]	99.990	0.505	74.000	100.000
tile[1], core[2]	99.990	0.507	73.870	100.000
tile[1], core[3]	18.272	13.259	0.000	98.220
tile[1], core[4]	17.231	11.048	0.000	37.260

Note that these are typical usage statistics for a representative run of the application on hardware. Core allocations may shift run-to-run in a scheduled RTOS. These statistics are generated by slicing the representative run into 10 ms chunks and calculating % time per chunk not spent in the FreeRTOS IDLE tasks. Therefore, the underlying distribution of these 10 ms bins should not be assumed to be Normal; this has implications on e.g. the interpretation of the Standard Deviation given here.

Table 8: FFD Power Usage

Power State	Power (mW)
Always	114

The description of the software is split up by folder:

Table 9: FFD Software Description

Folder	Description
examples/ffd/bsp_config	Board support configuration setting up software based IO peripherals
examples/ffd/filesystem_support	Filesystem contents for application
examples/ffd/src	Main application
modules/asr/intent_engine	Intent engine integration
modules/asr/intent_handler	Intent engine output integration

examples/ffd/bsp_config This folder contains bsp_configs for the FFD application. More information on bsp_configs can be found in the RTOS Framework documentation.

Table 10: FFD bsp_config

Filename/Directory	Description
dac directory	DAC ports for supported bsp_configs
XCORE-AI-EXPLORER directory	experimental bsp_config, not recommended for general use
XCORE-AI-EXPLORER_EXT directory	experimental bsp_config, not recommended for general use
XK_VOICE_L71 directory	default FFD application bsp_config
XK_VOICE_L71_EXT directory	USB debug extension FFD application bsp_config
bsp_config.cmake	cmake for adding FFD bsp_configs

examples/ffd/filesystem_support This folder contains filesystem contents for the FFD application.

Table 11: FFD filesystem_support

Filename/Directory	Description
50.wav	Playback for intent ID 50
1.wav	Playback for intent ID 1
3.wav	Playback for intent ID 3
4.wav	Playback for intent ID 4
5.wav	Playback for intent ID 5
6.wav	Playback for intent ID 6
7.wav	Playback for intent ID 7
8.wav	Playback for intent ID 8
9.wav	Playback for intent ID 9
10.wav	Playback for intent ID 10
11.wav	Playback for intent ID 11
12.wav	Playback for intent ID 12
13.wav	Playback for intent ID 13
14.wav	Playback for intent ID 14
15.wav	Playback for intent ID 15
16.wav	Playback for intent ID 16
17.wav	Playback for intent ID 17
18.wav	Playback for intent ID 18

examples/ffd/src This folder contains the core application source.

Table 12: FFD src

Filename/Directory	Description
gpio_ctrl directory	contains general purpose input handling and LED handling tasks
intent_engine directory	contains intent engine code
intent_handler directory	contains intent handling code
rtos_conf directory	contains default FreeRTOS configuration headers
app_conf_check.h	header to validate app_conf.h
app_conf.h	header to describe app configuration
config.xscope	xscope configuration file
ff_appconf.h	default fatfs configuration header
main.c	main application source file
xcore_device_memory.c	model loading from filesystem source file
xcore_device_memory.h	model loading from filesystem header file

Audio Pipeline The audio pipeline module provides the application with three API functions:

Listing 1: Audio Pipeline API (audio_pipeline.h)

```
void audio_pipeline_init(
        void *input_app_data,
        void *output_app_data);

void audio_pipeline_input(
        void *input_app_data,
        int32_t **input_audio_frames,
        size_t ch_count,
        size_t frame_count);

int audio_pipeline_output(
        void *output_app_data,
        int32_t **output_audio_frames,
        size_t ch_count,
        size_t frame_count);
```

audio_pipeline_init This function has the role of creating the audio pipeline, with two optional application pointers which are provided to the application in the audio_pipeline_input() and audio_pipeline_output() callbacks.

In FFD, the audio pipeline is initialized with no additional arguments, and instantiates a 3 stage pipeline on tile 1, as described in: *Audio Pipeline*

audio_pipeline_input This function has the role of providing the audio pipeline with the input frames.

In FFD, the input is received from the rtos_mic_array driver.

audio_pipeline_output This function has the role of receiving the processed audio pipeline output.

In FFD, the output is sent to the intent engine.

Main The major components of main are:

Listing 2: Main components (main.c)

```
void startup_task(void *arg)
void vApplicationMinimalIdleHook(void)
void tile_common_init(chanend_t c)
void main_tile0(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
void main_tile1(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
```

startup_task This function has the role of launching tasks on each tile. For those familiar with XCORE, it is comparable to the main par loop in an XC main.

vApplicationMinimalIdleHook This is a FreeRTOS callback. By calling "waiteu" without events configured, this has the effect of both MIPs and power savings on XCORE.

Listing 3: vApplicationMinimalIdleHook (main.c)

```
asm volatile("waiteu");
```

tile_common_init This function is the common tile initialization, which initializes the bsp_config, creates the startup task, and starts the FreeRTOS kernel.

main_tile0 This function is the application C entry point on tile 0, provided by the SDK.

main_tile1 This function is the application C entry point on tile 1, provided by the SDK.

modules/asr/intent_engine This folder contains the intent engine module for the FFD and FFVA applications.

Table 13: ASR Intent Engine

Filename/Directory	Description
intent_engine_io.c	contains additional io intent engine code
intent_engine_support.c	contains general intent engine support code
intent_engine.c	contains the implementation of default intent engine code
intent_engine.h	header for intent engine code

Major Components The intent engine module provides the application with two API functions:

Listing 4: Intent Engine API (intent_engine.h)

```
int32_t intent_engine_create(uint32_t priority, void *args);
void intent_engine_ready_sync(void);
int32_t intent_engine_sample_push(asr_sample_t *buf, size_t frames);
```

If replacing the existing model, these are the only two functions that are required to be populated.

intent_engine_create This function has the role of creating the model running task and providing a pointer, which can be used by the application to handle the output intent result. In the case of the default configuration, the application provides a FreeRTOS Queue object.

The ASR engine is on tile 0 in both FFD and FFVA, but the audio pipeline output is on tile 1 for FFD and on tile 0 for FFVA.

Listing 5: intent_engine_create snippet (intent_engine_io.c)

```
#if ASR_TILE_NO == AUDIO_PIPELINE_OUTPUT_TILE_NO
    intent_engine_task_create(priority);
#else
    intent_engine_intertile_task_create(priority);
#endif
```

The call to intent_engine_intertile_task_create() will create two threads on tile 0. One thread is the ASR engine thread. The other thread is an intertile rx thread, which will interface with the audio pipeline output.

intent_engine_ready_sync This function is called by both tiles and serves to ensure that tile 0 is ready to receive audio samples before starting the audio pipeline. This is a preventative measure to avoid dropping samples at startup.

Listing 6: intent_engine_create snippet (intent_engine_io.c)

```
    int sync = 0;
#if ON_TILE(AUDIO_PIPELINE_OUTPUT_TILE_NO)
    size_t len = rtos_intertile_rx_len(intertile_ctx, appconfINTENT_ENGINE_READY_SYNC_PORT, RTOS_OSAL_WAIT_
↪FOREVER);
    xassert(len == sizeof(sync));
    rtos_intertile_rx_data(intertile_ctx, &sync, sizeof(sync));
#else
    rtos_intertile_tx(intertile_ctx, appconfINTENT_ENGINE_READY_SYNC_PORT, &sync, sizeof(sync));
#endif
```

intent_engine_sample_push This function has the role of sending the ASR output channel from the audio pipeline to the intent engine.

The ASR engine is on tile 0 in both FFD and FFVA, but the audio pipeline output is on tile 1 for FFD and on tile 0 for FFVA.

Listing 7: intent_engine_create snippet (intent_engine_io.c)

```
#if appconfINTENT_ENABLED && ON_TILE(AUDIO_PIPELINE_OUTPUT_TILE_NO)
#if ASR_TILE_NO == AUDIO_PIPELINE_OUTPUT_TILE_NO
    intent_engine_samples_send_local(
            frames,
            buf);
#else
    intent_engine_samples_send_remote(
            intertile_ap_ctx,
            frames,
            buf);
#endif
#endif
```

The call to intent_engine_samples_send_remote() will send the audio samples to the previously configured intertile rx thread.

intent_engine_process_asr_result This function can be replaced by the application to handle the intent in a completely different manner.

Miscellaneous Functions The following helper functions are provided for supporting the command processing features that are unique to the default FFD application:

- intent_engine_keyword_queue_count
- intent_engine_keyword_queue_complete
- intent_engine_stream_buf_reset

- intent_engine_play_response

modules/asr/intent_handler This folder contains ASR output handling modules for the FFD and FFVA applications.

Table 14: ASR Intent handler

Filename/Directory	Description
audio_response directory	include folder for handling audio responses to keywords
intent_handler.c	contains the implementation of default intent handling code
intent_handler.h	header for intent handler code

Major Components The intent handling module provides the application with one API function:

Listing 8: Intent Handler API (intent_handler.h)

```
int32_t intent_handler_create(uint32_t priority, void *args);
```

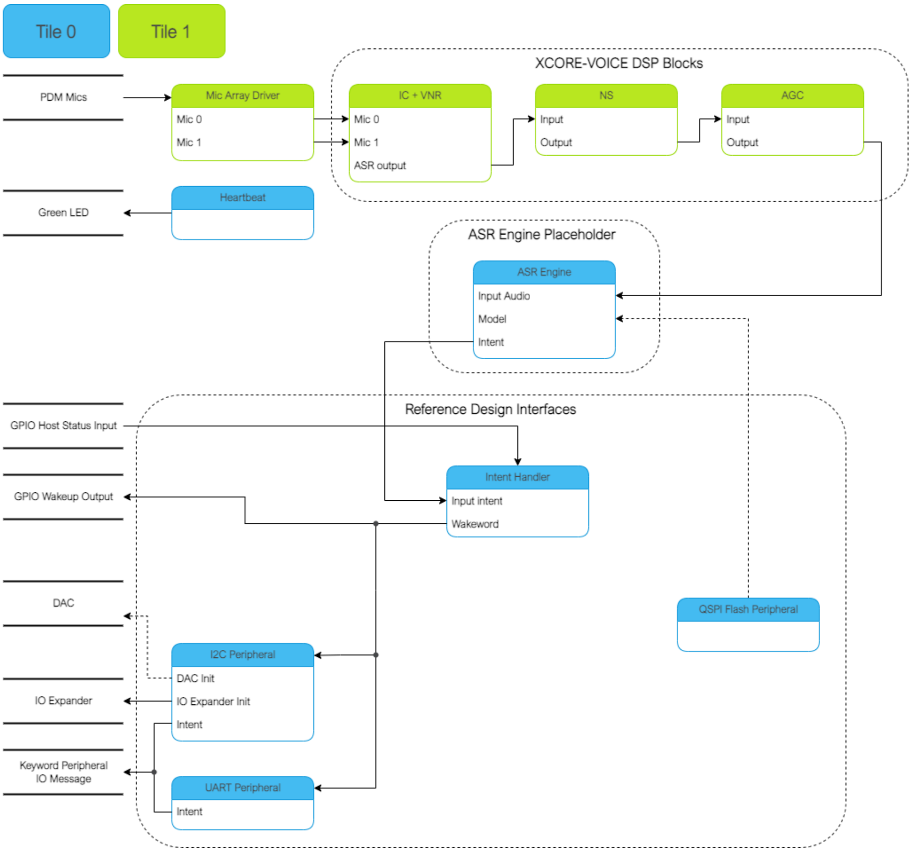
If replacing the existing handler code, this is the only function that is required to be populated.

intent_handler_create This function has the role of creating the keyword handling task for the ASR engine. In the case of the Sensory and Cyberon models, the application provides a FreeRTOS Queue object. This handler is on the same tile as the speech recognition engine, tile 0.

The call to intent_handler_create() will create one thread on tile 0. This thread will receive ID packets from the ASR engine over a FreeRTOS Queue object and output over various IO interfaces based on configuration.

6.1.6.4 Software Modifications

The FFD example design consists of three major software blocks, the audio pipeline, keyword spotter, and keyword handler. This section will go into detail on how to replace each/all of these subsystems.



It is highly recommended to be familiar with the application as a whole before attempting replacing these functional units. This information can be found here: *Software Description*

See *Software Description* for more details on the memory footprint and CPU usage of the major software components.

Replacing XCORE-VOICE DSP Block

The audio pipeline can be replaced by making changes to the *audio_pipeline.c* file.

It is up to the user to ensure that the input and output frames of the audio pipeline remain the same, or the remainder of the application will not function properly.

This section will walk through an example of replacing the XMOS NS stage, with a custom stage foo.

Declaration and Definition of DSP Context

Replace:

Listing 9: XMOS NS (audio_pipeline.c)

```
typedef struct ns_stage_ctx {
    ns_state_t state;
} ns_stage_ctx_t;

static ns_stage_ctx_t ns_stage_state = {};
```

With:

Listing 10: Foo (audio_pipeline.c)

```
typedef struct foo_stage_ctx {
    /* Your required state context here */
} foo_stage_ctx_t;

static foo_stage_ctx_t foo_stage_state = {};
```

DSP Function Replace:

Listing 11: XMOS NS (audio_pipeline.c)

```
static void stage_ns(frame_data_t *frame_data)
{
#if appconfAUDIO_PIPELINE_SKIP_NS
    (void) frame_data;
#else
    int32_t ns_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    configASSERT(NS_FRAME_ADVANCE == appconfAUDIO_PIPELINE_FRAME_ADVANCE);
    ns_process_frame(
                &ns_stage_state.state,
                ns_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, ns_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
#endif
}
```

With:

Listing 12: Foo (audio_pipeline.c)

```
static void stage_foo(frame_data_t *frame_data)
{
    int32_t foo_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    foo_process_frame(
                &foo_stage_state.state,
                foo_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, foo_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
}
```

Runtime Initialization Replace:

Listing 13: XMOS NS (audio_pipeline.c)

```
ns_init(&ns_stage_state.state);
```

With:

Listing 14: Foo (audio_pipeline.c)

```
foo_init(&foo_stage_state.state);
```

Audio Pipeline Setup Replace:

Listing 15: XMOS NS (audio_pipeline.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_ns,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_ns),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

With:

Listing 16: Foo (audio_pipeline.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_foo,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_foo),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

It is also possible to add or remove stages. Refer to the RTOS Framework documentation on the generic pipeline sw_service.

Replacing Example Design Interfaces It may be desired to have a different output interface to talk to a host, or not have a host at all and handle the intent local to the XCORE device.

Different Peripheral IO To add or remove a peripheral IO, modify the bsp_config accordingly. Refer to documentation inside the RTOS Framework on how to instantiate different RTOS peripheral drivers.

Direct Control In a single controller system, the XCORE can be used to control peripherals directly.

The proc_keyword_res task can be modified as follows:

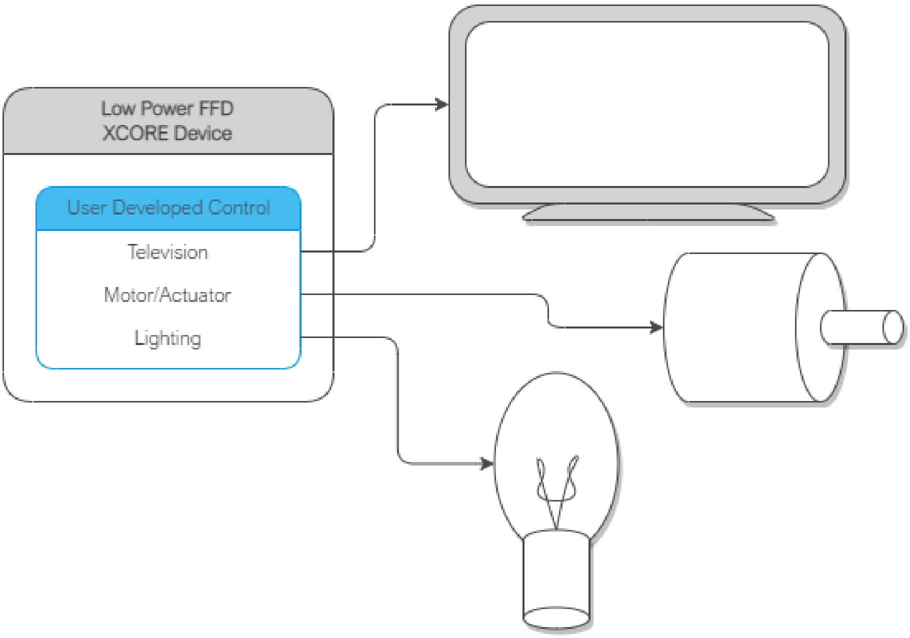
Listing 17: Intent Handler (intent_handler.c)

```
static void proc_keyword_res(void *args) {
    QueueHandle_t q_intent = (QueueHandle_t) args;
    int32_t id = 0;

    while(1) {
        xQueueReceive(q_intent, &id, portMAX_DELAY);

        /* User logic here */
    }
}
```

This code example will receive the ID of each intent, and can be populated by any user application logic. User logic can use other RTOS drivers to control various peripherals, such as screens, motors, lights, etc, based on the intent engine outputs.



6.1.6.5 Speech Recognition - Sensory

License The Sensory TrulyHandsFree™ (THF) speech recognition library is *Copyright (C) 1995-2022 Sensory Inc., All Rights Reserved*.

Sensory THF software requires a commercial license granted by Sensory Inc. This software ships with an expiring development license. It will suspend recognition after 11.4 hours or 107 recognition events.

Overview The Sensory THF speech recognition engine runs proprietary models to identify keywords in an audio stream. Models can be generated using VoiceHub.

Two models are provided - one in US English and one in Mainland Mandarin. The US English model is used by default. To modify the software to use the Mandarin model, see the comment at the top of the `ffd_sensory.cmake` file. Make sure run the following commands to rebuild and re-flash the data partition:

```
make clean
make flash_app_example_ffd_sensory -j
```

Dictionary command table

Table 15: English Language Demo

Utterances	Type	Return code (decimal)
Hello XMOS	keyword	1
Switch on the TV	command	3
Switch off the TV	command	4
Channel up	command	5
Channel down	command	6
Volume up	command	7
Volume down	command	8
Switch on the lights	command	9
Switch off the lights	command	10
Brightness up	command	11
Brightness down	command	12
Switch on the fan	command	13
Switch off the fan	command	14
Speed up the fan	command	15
Slow down the fan	command	16
Set higher temperature	command	17
Set lower temperature	command	18

Application Integration In depth information on out of the box integration can be found here: *Host Integration*

6.1.6.6 Speech Recognition - Cyberon

License Cyberon DSpotter™ software requires a commercial license granted by Cyberon Corporation. This software ships with an expiring development license. It will suspend recognition after 100 recognition events.

Production versions of the DSpotter™ library are unrestricted when running on a specially licensed XMOS device. Please contact Cyberon or XMOS sales for further information.

Overview The Cyberon DSpotter™ speech recognition engine runs proprietary models to identify keywords in an audio stream.

One model for US English is provided. For any technical questions or additional models please contact Cyberon.

Dictionary command table

Table 16: English Language Demo

Utterances	Type	Return code (decimal)
Hello XMOS	keyword	1
Hello Cyberon	keyword	1
Switch on the TV	command	2
Switch off the TV	command	3
Channel up	command	4
Channel down	command	5
Volume up	command	6
Volume down	command	7
Switch on the lights	command	8
Switch off the lights	command	9
Brightness up	command	10
Brightness down	command	11
Switch on the fan	command	12
Switch off the fan	command	13
Speed up the fan	command	14
Slow down the fan	command	15
Set higher temperature	command	16
Set lower temperature	command	17

Application Integration In depth information on out of the box integration can be found here: *Host Integration*

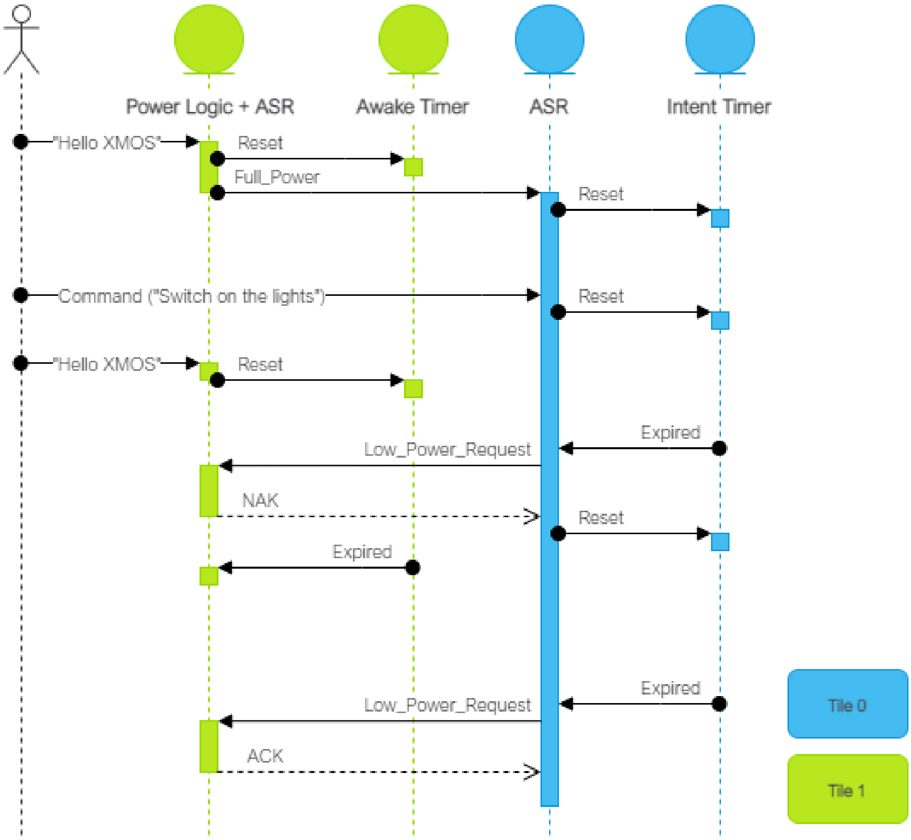
6.2 Low Power Far-field Voice Local Command

6.2.1 Overview

The low power far-field voice local command (Low Power FFD) example design targets low power speech recognition using Sensory's TrulyHandsfree™ (THF) speech recognition and local dictionary.

When the small wake word model running on tile 1 recognizes a wake word utterance, the device transitions to full power mode where tile 0's command model begins receiving audio samples, continuing the command recognition process. On command recognition, the application outputs a discrete message over I^2C and UART.

Tile 0's command model, in combination with a timer, determines when to request a transition to low power. Tile 1 may accept or reject this request based on its own timer that is reset on wake word recognitions and potentially other application-specific events. The figure below illustrates the general behavior.



When in low power mode, tile 0 is effectively disabled along with any peripheral/IO associated with that tile.

Sensory's THF software ships with an expiring development license. It will suspend recognition after 11.4 hours or 107 recognition events; after which, a device reset is required to resume normal operation. To perform a reset, either power cycle the device or press the SW2 button. Note that SW2 is only functional while in full power mode (this application is configured to hold the device in full-power mode on such license expiration events).

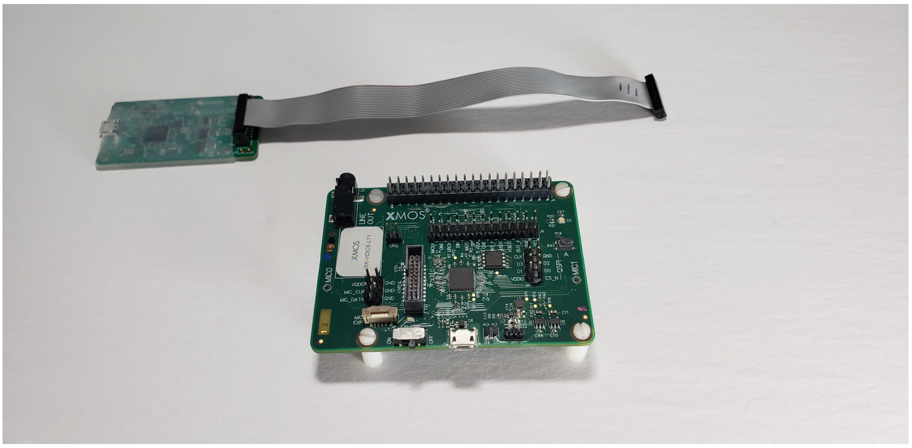
More information on the Sensory speech recognition library can be found here: *Speech Recognition*

6.2.2 Supported Hardware

This example application is supported on the XK-VOICE-L71 board.

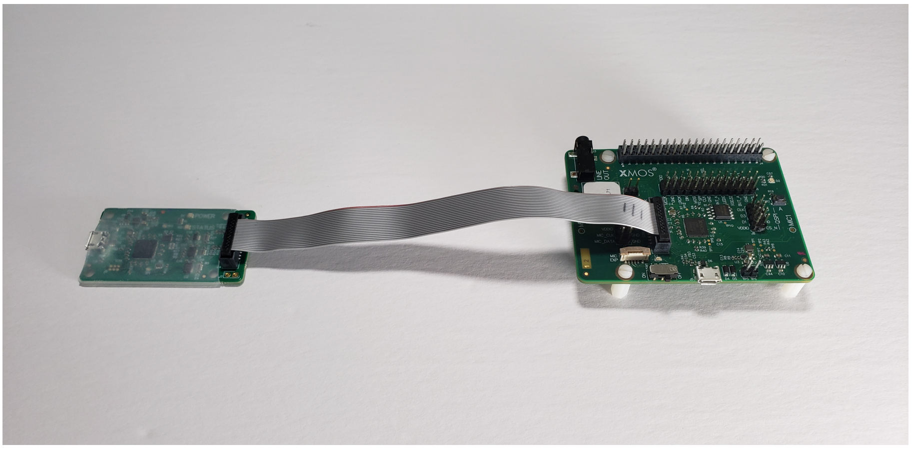
6.2.2.1 Setting up the Hardware

This example design requires an XTAG4 and XK-VOICE-L71 board.

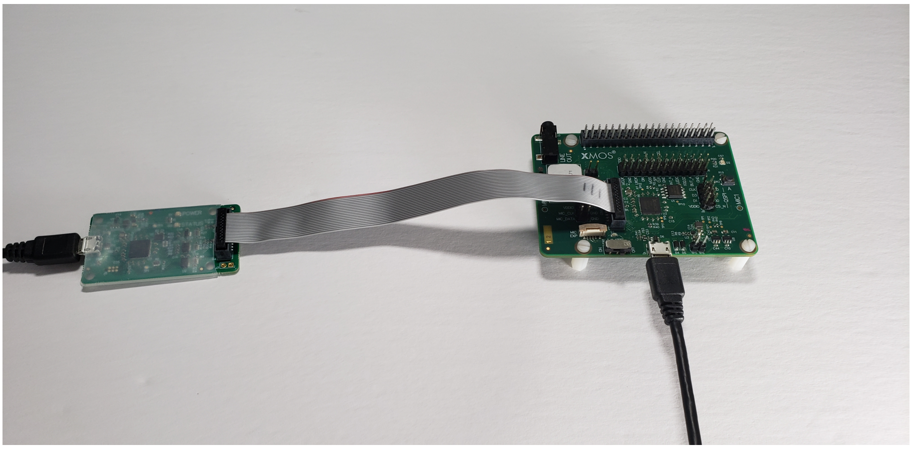


xTAG The xTAG is used to program and debug the device

Connect the xTAG to the debug header, as shown below.



Connect the micro USB XTAG4 and micro USB XK-VOICE-L71 to the programming host.



6.2.3 Configuring the Firmware

The default application performs as described in the *Overview*. There are numerous compile time options that can be added to change the example design without requiring code changes. To change the options explained in the table below, add the desired configuration variables to the APP_COMPILE_DEFINITIONS CMake variable located in the example's CMake file here.

If options are changed, the application firmware must be rebuilt.

Table 17: Low Power FFD Compile Options

Compile Option	Description	Default Value
appconfINTENT_RESET_DELAY_MS	Sets the period after the wake word phrase or subsequent command/wake word phrase has been heard for a valid command phrase	4000
appconfINTENT_UART_OUTPUT_ENABLED	Enables/disables the UART intent message	1
appconfINTENT_I2C_MASTER_OUTPUT_ENABLED	Enables/disables sending the intent message over I^2C master	1
appconfUART_BAUD_RATE	Sets the baud rate for the UART tx intent interface	9600
appconfINTENT_I2C_MASTER_DEVICE_ADDR	Sets the I^2C slave address to transmit the intent to	0x01
appconfINTENT_TRANSPORT_DELAY_MS	Sets the delay between host wake up requested and I^2C and UART keyword code transmission	50
appconfINTENT_QUEUE_LEN	Sets the maximum number of detected intents to hold while waiting for the host to wake up	10
appconfINTENT_WAKEUP_EDGE_TYPE	Sets the host wake up pin GPIO edge type. 0 for rising edge, 1 for falling edge	0
appconfAUDIO_PIPELINE_SKIP_IC_AND_VNR	Enables/disables the IC and VNR	0
appconfAUDIO_PIPELINE_SKIP_NS	Enables/disables the NS	0
appconfAUDIO_PIPELINE_SKIP_AGC	Enables/disables the AGC	0

6.2.4 Deploying the Firmware with Linux or macOS

This document explains how to deploy the software using *CMake* and *Make*.

6.2.4.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -B build_host
cd build_host
make install
```

The host applications will be installed at `/opt/xmos/bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.2.4.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
pip install -r requirements.txt
cmake -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_low_power_ffd_sensory
```

6.2.4.3 Running the Firmware

Before running the firmware, the filesystem and *command* model must be flashed to the data partition.

Within the root of the build folder, run:

```
make flash_app_example_low_power_ffd_sensory
```

After this command completes, the application will be running.

After flashing the data partition, the application can be run without reflashing. If changes are made to the data partition components, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_low_power_ffd_sensory.xe
```

6.2.4.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_low_power_ffd_sensory.xe
```

6.2.5 Deploying the Firmware with Native Windows

This document explains how to deploy the software using *CMake* and *Ninja*. If you are not using native Windows MSVC build tools and instead using a Linux emulation tool such as WSL, refer to *Deploying the Firmware with Linux or macOS*.

To install *Ninja* follow install instructions at https://ninja-build.org/ or on Windows install with `winget` by running the following commands in *PowerShell*:

```
# Install
winget install Ninja-build.ninja
# Reload user Path
$env:Path=[System.Environment]::GetEnvironmentVariable("Path","User")
```

6.2.5.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native toolchain:

Note: Permissions may be required to install the host applications.

Note: A C/C++ compiler, such as Visual Studio or MinGW, must be included in the path.

Before building the host application, you will need to add the path to the XTC Tools to your environment.

```
set "XMOS_TOOL_PATH=<path-to-xtc-tools>"
```

Then build the host application:

```
cmake -G Ninja -B build_host
cd build_host
ninja install
```

The host applications will be installed at `%USERPROFILE%\.xmos\bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.2.5.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
pip install -r requirements.txt
cmake -G Ninja -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
ninja example_low_power_ffd_sensory
```

6.2.5.3 Running the Firmware

Before running the firmware, the filesystem and *command* model must be flashed to the data partition.

Within the root of the build folder, run:

```
ninja flash_app_example_low_power_ffd_sensory
```

After this command completes, the application will be running.

After flashing the data partition, the application can be run without reflashing. If changes are made to the data partition components, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_low_power_ffd_sensory.xe
```

6.2.5.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_low_power_ffd_sensory.xe
```

6.2.6 Modifying the Software

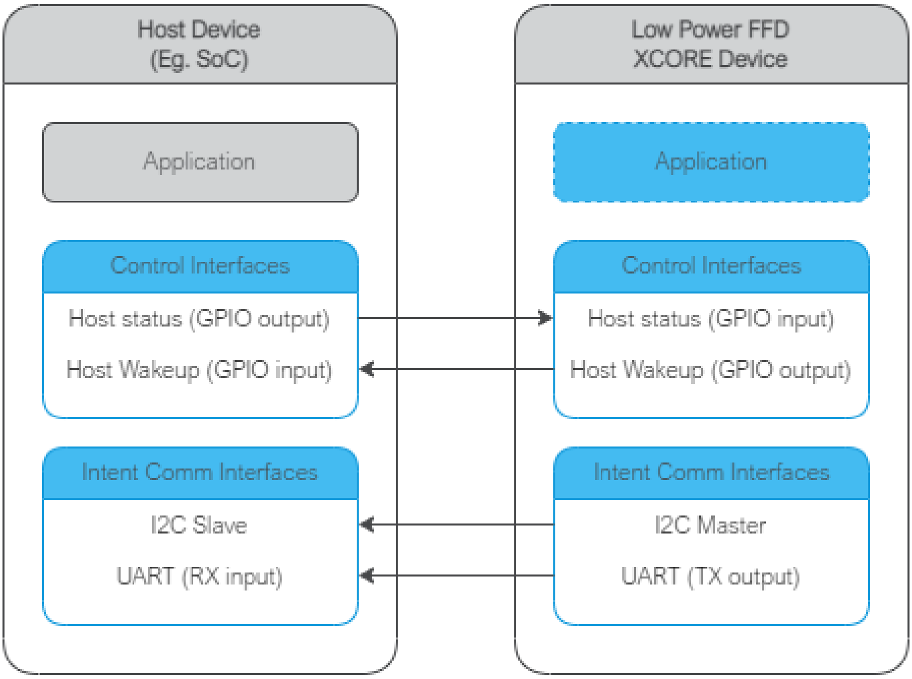
The low-power FFD example design is highly customizable. This section describes how to modify the application.

6.2.6.1 Host Integration

Overview This section describes the connections that would need to be made to an external host for plug and play integration with existing devices.

When an intent is found, the XCORE device will check if the host is awake, by checking the Host Status GPIO pin. If the host is awake the intent code will be transmitted over I^2C and/or UART.

If the host is not awake, the XCORE device will trigger a transition of the Wakeup GPIO pin. This can be configured to be a rising or falling edge. The XCORE device will then wait for a fixed period of time, set at compile time, before transmitting the intent over the I^2C and/or UART interface. This behavior can be changed as desired by modifying the intent handling code.



UART

Table 18: UART Connections

Low Power FFD Connection	Host Connection
J4:24	UART RX
J4:20	GND

I²C

Table 19: I^2C Connections

Low Power FFD Connection	Host Connection
J4:3	SDA
J4:5	SCL
J4:9	GND

GPIO

Table 20: GPIO Connections

Low Power FFD Connection	Host Connection
J4:19	Wake up input
J4:21	Host Status output

6.2.6.2 Audio Pipeline The audio pipeline in Low Power FFD processes two channel PDM microphone input into a single output channel, intended for use by an ASR engine.

The audio pipeline consists of 3 stages.

Table 21: FFD Audio Pipeline

Stage	Description	Input Channel Count	Output Channel Count
1	Interference Canceller and Voice Noise Ratio	2	1
2	Noise Suppression	1	1
3	Automatic Gain Control	1	1

See the Voice Framework User Guide for more information.

6.2.6.3 Software Description

Overview The approximate resource utilizations for Low Power FFD are shown in the table below.

Table 22: Low Power FFD Resources

Resource	Tile 0	Tile 1
Unused CPU Time (600MHz | 200MHz)	50%	10%
Total Memory Free	19.1k	5.3k
Runtime Heap Memory Free	219k	12.4k

The estimated (core) power usage for Low Power FFD are shown in the table below. Additional power savings may be possible using Sensory's Low Power Sound Detect (LPSD) option which approaches sub-50mW operation in Low Power mode. These measurements will vary based on component tolerances and any user added code and/or user added compile options.

Table 23: Low Power FFD Power Usage

Power State	Core Power (mW)
Low Power	54
Full Power	110

The description of the software is split up by folder:

Table 24: Low Power FFD Software Description

Folder	Description
bsp_config	Board support configuration setting up software based IO peripherals
filesystem_support	Filesystem contents for application
model	Wake word and command model files
src	Main application
src/gpio_ctrl	GPIO and LED related functions
src/intent_engine	Intent engine integration
src/intent_handler	Intent engine output integration
src/power	Low power control logic
src/wakeword	Wake word engine integration

bsp_config This folder contains bsp_configs for the Low Power FFD application. More information on bsp_configs can be found in the RTOS Framework documentation.

Table 25: Low Power FFD bsp_config

Filename/Directory	Description
dac directory	DAC ports for supported bsp_configs (not used in example, disabled)
XK_VOICE_L71 directory	default Low Power FFD application bsp_config
bsp_config.cmake	cmake for adding Low Power FFD bsp_configs

filesystem_support This folder contains filesystem contents for the Low Power FFD application.

Table 26: Low Power FFD filesystem_support

Filename/Directory	Description
demo.txt	A file for demonstrative purposes containing the text "Hello World!". This file is not used or interacted with in this application.

model This folder contains the Sensory wake word and command model files the Low Power FFD application.

Note: Only a subset of the files below are used. See `low_power_ffd.cmake` for the files used by the application. Also note the nibble-swapped net-file is manually generated, via the `nibble_swap` tool found in `lib_qspi_fast_read`.

Table 27: Low Power FFD model

Filename/Directory	Description
command-pc62w-6.1.0-op10-prod-net.bin	The command model's net-file, in binary-form
command-pc62w-6.1.0-op10-prod-net.bin.nibble_swapped	The command model's net-file, in binary-form (nibble swapped, for supporting fast flash reads)
command-pc62w-6.1.0-op10-prod-net.c	The command model's net-file, in source form
command-pc62w-6.1.0-op10-prod-search.bin	The command model's search-file, in binary form
command-pc62w-6.1.0-op10-prod-search.c	The command model's search-file, in source form
command-pc62w-6.1.0-op10-prod-search.h	The command model's search header-file
command.snsr	The command model's Sensory THF/TNL SDK "snsr" file
wakeword-pc60w-6.1.0-op10-prod-net.bin	The wake word model's net-file, in binary-form
wakeword-pc60w-6.1.0-op10-prod-net.c	The wake word model's net-file, in source form
wakeword-pc60w-6.1.0-op10-prod-search.bin	The wake word model's search-file, in binary form
wakeword-pc60w-6.1.0-op10-prod-search.c	The wake word model's search-file, in source form
wakeword-pc60w-6.1.0-op10-prod-search.h	The wake word model's search header-file
wakeword.snsr	The wake word model's Sensory THF/TNL SDK "snsr" file

src This folder contains the core application source.

Table 28: FFD src

Filename/Directory	Description
gpio_ctrl directory	contains general purpose input handling and LED handling tasks
intent_engine directory	contains intent engine code
intent_handler directory	contains intent handling code
power directory	contains low power control logic and related audio buffer
rtos_conf directory	contains default FreeRTOS configuration headers
wakeword directory	contains wake word detection code
app_conf_check.h	header to validate app_conf.h
app_conf.h	header to describe app configuration
config.xscope	xscope configuration file
ff_appconf.h	default fatfs configuration header
main.c	main application source file
device_memory_impl.c	contains XCORE device memory functions for supporting ASR functionality
device_memory_impl.h	header for the device memory implementation

Audio Pipeline The audio pipeline module provides the application with three API functions:

Listing 18: Audio Pipeline API (audio_pipeline.h)

```
void audio_pipeline_init(
        void *input_app_data,
        void *output_app_data);

void audio_pipeline_input(
        void *input_app_data,
        int32_t **input_audio_frames,
        size_t ch_count,
        size_t frame_count);

int audio_pipeline_output(
        void *output_app_data,
        int32_t **output_audio_frames,
        size_t ch_count,
        size_t frame_count);
```

audio_pipeline_init This function has the role of creating the audio pipeline, with two optional application pointers which are provided to the application in the audio_pipeline_input() and audio_pipeline_output() callbacks.

In Low Power FFD, the audio pipeline is initialized with no additional arguments, and instantiates a 3 stage pipeline on tile 1, as described in: *Audio Pipeline*

audio_pipeline_input This function has the role of providing the audio pipeline with the input frames.

In Low Power FFD, the input is received from the rtos_mic_array driver.

audio_pipeline_output This function has the role of receiving the processed audio pipeline output.

In Low Power FFD, the output is sent to both the wake word handler and the intent engine. Because the intent engine will be suspended in low power mode and that there is a finite time that it takes to resume full power operation, there is a ring buffer placed between the audio output received from this routine and the intent engine's stream buffer.

Main The major components of main are:

Listing 19: Main components (main.c)

```
void startup_task(void *arg)
void vApplicationMinimalIdleHook(void)
void tile_common_init(chanend_t c)
void main_tile0(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
void main_tile1(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
```

startup_task This function has the role of launching tasks on each tile. For those familiar with XCORE, it is comparable to the main par loop in an XC main.

vApplicationMinimalIdleHook This is a FreeRTOS callback. By calling "waiteu" without events configured, this has the effect of both MIPs and power savings on XCORE.

Listing 20: vApplicationMinimalIdleHook (main.c)

```
asm volatile("waiteu");
```

tile_common_init This function is the common tile initialization, which initializes the bsp_config, creates the startup task, and starts the FreeRTOS kernel.

main_tile0 This function is the application C entry point on tile 0, provided by the SDK.

main_tile1 This function is the application C entry point on tile 1, provided by the SDK.

src/gpio_ctrl This folder contains the GPIO and LED related functionality for the Low Power FFD application.

Table 29: Low Power FFD gpio_ctrl

Filename/Directory	Description
gpi_ctrl.c	The general purpose input control source file. Implements SW2 reset logic.
gpi_ctrl.h	The general purpose input control header file.
leds.c	The LED task source file. Handles the applications LED indications.
leds.h	The LED task header file.

src/intent_engine This folder contains the intent engine module for the low power FFD application.

Table 30: Low Power FFD Intent Engine

Filename/Directory	Description
intent_engine_io.c	contains additional io intent engine code
intent_engine_support.c	contains general intent engine support code
intent_engine.c	contains the implementation of default intent engine code
intent_engine.h	header for intent engine code

Major Components The intent engine module provides the application with the following primary API functions:

Listing 21: Intent Engine API (intent_engine.h)

```
int32_t intent_engine_create(uint32_t priority, void *args);
void intent_engine_ready_sync(void);
int32_t intent_engine_sample_push(asr_sample_t *buf, size_t frames);
```

These APIs provide the functionality needed to feed audio pipeline samples into the ASR engine.

intent_engine_create This function has the role of creating the model running task and providing a pointer, which can be used by the application to handle the output intent result. In the case of the default configuration, the application provides a FreeRTOS Queue object.

In Low Power FFD, the audio pipeline output is on tile 1 and the ASR engine on tile 0.

Listing 22: intent_engine_create snippet (intent_engine_io.c)

```
intent_engine_intertile_task_create(priority);
```

The call to intent_engine_intertile_task_create() will create two threads on tile 0. One thread is the ASR engine thread. The other thread is an intertile RX thread, which will interface with the audio pipeline output.

intent_engine_ready_sync This function is called by both tiles and serves to ensure that tile 0 is ready to receive audio samples before starting the audio pipeline. This is a preventative measure to avoid dropping samples at startup.

Listing 23: intent_engine_create snippet (intent_engine_io.c)

```
    int sync = 0;
#if ON_TILE(AUDIO_PIPELINE_OUTPUT_TILE_NO)
    size_t len = rtos_intertile_rx_len(intertile_ctx, appconfINTENT_ENGINE_READY_SYNC_PORT, RTOS_OSAL_WAIT_
↪FOREVER);
    xassert(len == sizeof(sync));
    rtos_intertile_rx_data(intertile_ctx, &sync, sizeof(sync));
#else
    rtos_intertile_tx(intertile_ctx, appconfINTENT_ENGINE_READY_SYNC_PORT, &sync, sizeof(sync));
#endif
```

intent_engine_sample_push This function has the role of sending the ASR output channel from the audio pipeline to the intent engine.

In Low Power FFD, the audio pipeline output is on tile 1 and the ASR engine on tile 0.

Listing 24: intent_engine_create snippet (intent_engine_io.c)

```
    intent_engine_samples_send_remote(
            intertile_ap_ctx,
            frames,
            buf);
```

The call to intent_engine_samples_send_remote() will send the audio samples to the previously configured intertile RX thread.

intent_engine_process_asr_result This function can be replaced by the application to handle the intent in a completely different manner.

Low Power Components The following APIs are the intent engine mechanisms needed by the power control task.

Listing 25: Low Power APIs (intent_engine.h)

```
void intent_engine_full_power_request(void);
void intent_engine_low_power_accept(void);
```

In this implementation, it is the responsibility of tile 0 (intent engine tile) to determine when to request a transition into low power mode; however, tile 1 may reject the request. When tile 1 accepts the request (via *LOW_POWER_ACK*), the power control task calls *intent_engine_low_power_accept*. When tile 1 rejects the request (via *LOW_POWER_NAK*), the power control task calls *intent_engine_full_power_request*.

Note: There is an additional *LOW_POWER_HALT* response where the power control task calls *intent_engine_halt*. This is primarily for end-of-evaluation handling logic for the underlying ASR engine and is not needed for a normal application.

After tile 1 accepts the low power request, tile 0 begins preparations for entering low power by locking various resources and waiting for any enqueued commands to finish up. The helper functions below are provided for this purpose.

Listing 26: Low Power Helper Functions (intent_engine.h)

```
int32_t intent_engine_keyword_queue_count(void);
void intent_engine_keyword_queue_complete(void);
uint8_t intent_engine_low_power_ready(void);
```

Before tile 1 sends *LOW_POWER_ACK* it also stops pushing audio samples via *intent_engine_sample_push*. After receiving the low power response, the application may clear the stream buffer and keyword queue to avoid processing stale samples/commands when returning to full power mode. The functions below provide this functionality.

Listing 27: Low Power Helper Functions (intent_engine.h)

```
void intent_engine_keyword_queue_reset(void);
void intent_engine_stream_buf_reset(void);
```

Note: Since it is possible that a command is spoken/recognized between the time when tile 0 requests low power and when tile 1 responds to the request, the application should not reset these buffer entities until it has received *LOW_POWER_ACK*; otherwise, recognized commands may be lost.

Evaluation Specific Components The following functions are provided for the primary purpose of facilitating the evaluation of the ASR model. The provided ASR models have evaluation periods which will end due to various factors. When the evaluation period ends, the application logic halts the intent engine via *intent_engine_halt*. This is primarily to ensure the device remains in full-power mode to allow functionality that may be exclusive to tile 0 to function.

Listing 28: Evaluation-specific Helper Functions (intent_engine.h)

```
void intent_engine_halt(void);
```

src/intent_handler This folder contains ASR output handling modules for the Low Power FFD application.

Table 31: FFD Intent handler

Filename/Directory	Description
intent_handler.c	contains the implementation of default intent handling code
intent_handler.h	header for intent handler code

Major Components The intent handling module provides the application with one API function:

Listing 29: Intent Handler API (intent_handler.h)

```
int32_t intent_handler_create(uint32_t priority, void *args);
```

If replacing the existing handler code, this is the only function that is required to be populated.

intent_handler_create This function has the role of creating the keyword handling task for the ASR engine. In the case of the Sensory model, the application provides a FreeRTOS Queue object. This handler is on the same tile as the Sensory engine, tile 0.

The call to intent_handler_create() will create one thread on tile 0. This thread will receive ID packets from the ASR engine over a FreeRTOS Queue object and output over various IO interfaces based on configuration.

src/power This folder contains the low power control logic and supporting logic.

Table 32: Low Power FFD power

Filename/Directory	Description
low_power_audio_buffer.c	Implementation of an audio sample ring buffer. Aids in responsiveness to commands during a transition to full power mode.
low_power_audio_buffer.c	Header for the low power audio buffer.
power_control.c	Implementation of the power control logic.
power_control.h	Header for power control logic.
power_state.c	Implementation of Tile 1 power state logic.
power_state.h	Header for power state logic.

Major Components The power control module provides the application with the following primary API functions:

Listing 30: Power Control API (power_control.h)

```
void power_control_task_create(unsigned priority, void *args);
void power_control_exit_low_power(void);
power_state_t power_control_state_get(void);
void power_control_halt(void);
void power_control_req_low_power(void);
void power_control_ind_complete(void);
```

power_control_task_create Creates and starts the power control task. To be called by each tile.

power_control_exit_low_power Applicable only for Tile 1. Begins a transition to full power mode and is intended to be called by the power_state_set() routine.

power_control_state_get Applicable only for Tile 1. Gets the current power state.

power_control_halt Applicable only for Tile 1. Halts the power control task. This is provided primarily for end-of-evaluation logic, but severs to terminate the low power logic. When halted, the system remains in full power mode.

power_control_req_low_power Applicable only for Tile 0. Requests a transition to low power mode.

power_control_ind_complete Applicable only for Tile 0. Indication that the last step for preparing for a low power transition has completed and allows the power control task to continue with final steps. This is primarily to ensure the LED indications are up-to-date before driver locks are taken (which include GPIO/LED control).

Power State Components The power state module provides the application with the following primary API functions:

Listing 31: Power State API (power_state.h)

```
void power_state_init();
void power_state_set(power_state_t state);
uint8_t power_state_timer_expired_get(void);
```

This module is also responsible for providing the base power state datatype (*power_state_t*) used by other low power logic.

power_state_init Initializes the power state module. Responsible to initializing the underlying timer that effectively determines whether a low power request by Tile 0 is accepted or rejected.

power_state_set Used by Tile 1's application to signal full power events (such as wake word detection or other application-specific events). Used by Tile 1's power control logic to signal low power only after Tile 0 has requested low power mode and the local timer has expired.

power_state_timer_expired_get Used by the Tile 1's power control logic to determine whether to accept or reject a low power request by Tile 0.

src/wakeword This folder contains the wake word recognition functionality for the Low Power FFD application.

Table 33: Low Power FFD wakeword

Filename/Directory	Description
wakeword.c	The wake word engine source file. Responsible for the transfer of audio samples into the ASR and handling of wake word detection events.
wakeword.h	The wake word engine header file.

Major Components The wakeword module provides the application with two API functions:

Listing 32: Wake Word API (wakeword.h)

```
void wakeword_init(void);
wakeword_result_t wakeword_handler(asr_sample_t *buf, size_t num_frames);
```

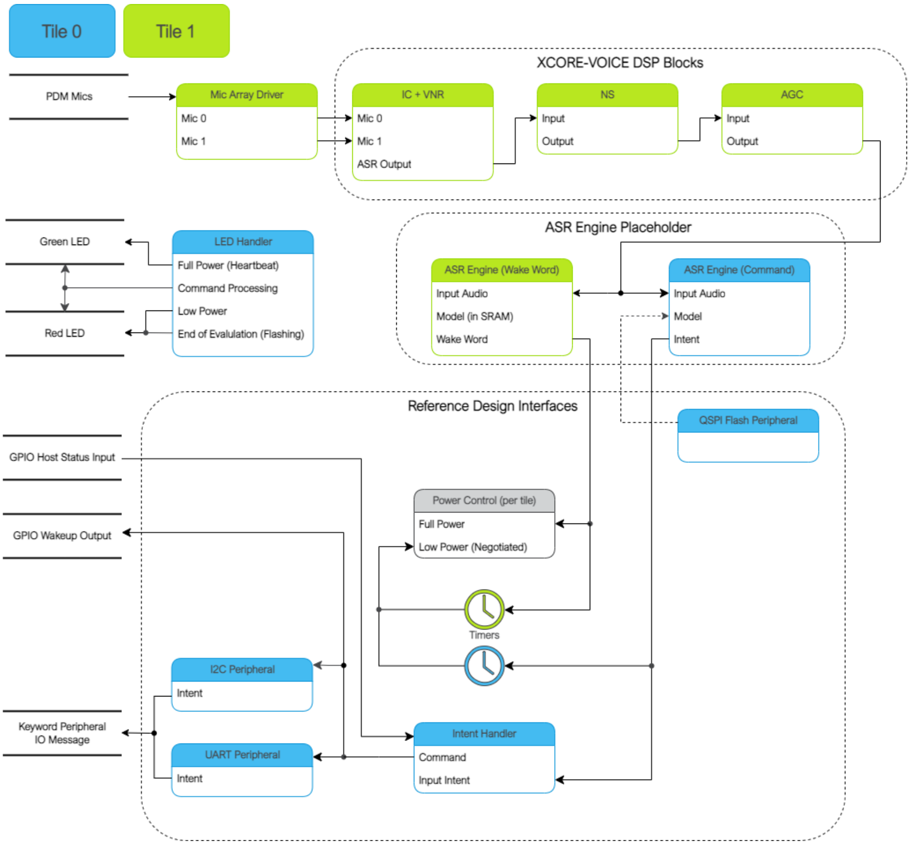
wakeword_init This function performs the required initialization for the wakeword_handler() function to operate. This involves initializing an instance of devmem_manager_t for use by the ASR abstraction layer and initialization of the ASR unit itself. It is to be called once during startup before any call to wakeword_handler() occurs.

wakeword_handler This function performs wake word detection logic and reports back to the caller a result, indicating whether a wake word was recognized. Note: this routine is called by audio_pipeline_output(), meaning this routine's logic should be kept to a minimum to ensure timing requirements are met.

In this implementation a single wake word ID of 1 is defined. Minimal adaptation is needed to support other models supporting other IDs or more than one valid wake word.

6.2.6.4 Software Modifications

The Low Power FFD example design consists of four major software blocks: the audio pipeline, ASR engine (wake word and intent engines), intent handler, and power control. This section will go into detail on how to replace each subsystem.



It is highly recommended to be familiar with the application as a whole before attempting replacing these functional units. This information can be found here: *Software Description*

See *Software Description* for more details on the memory footprint and CPU usage of the major software components.

Replacing XCORE-VOICE DSP Block

The audio pipeline can be replaced by making changes to the *audio_pipeline.c* file.

It is up to the user to ensure that the input and output frames of the audio pipeline remain the same, or the remainder of the application will not function properly.

This section will walk through an example of replacing the XMOS NS stage, with a custom stage foo.

Declaration and Definition of DSP Context

Replace:

Listing 33: XMOS NS (audio_pipeline.c)

```
typedef struct ns_stage_ctx {
    ns_state_t state;
} ns_stage_ctx_t;

static ns_stage_ctx_t ns_stage_state = {};
```

With:

Listing 34: Foo (audio_pipeline.c)

```
typedef struct foo_stage_ctx {
    /* Your required state context here */
} foo_stage_ctx_t;

static foo_stage_ctx_t foo_stage_state = {};
```

DSP Function

Replace:

Listing 35: XMOS NS (audio_pipeline.c)

```
static void stage_ns(frame_data_t *frame_data)
{
#if appconfAUDIO_PIPELINE_SKIP_NS
    (void) frame_data;
#else
    int32_t ns_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    configASSERT(NS_FRAME_ADVANCE == appconfAUDIO_PIPELINE_FRAME_ADVANCE);
    ns_process_frame(
                &ns_stage_state.state,
                ns_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, ns_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
#endif
}
```

With:

Listing 36: Foo (audio_pipeline.c)

```
static void stage_foo(frame_data_t *frame_data)
{
    int32_t foo_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    foo_process_frame(
                &foo_stage_state.state,
                foo_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, foo_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
}
```

Runtime Initialization

Replace:

Listing 37: XMOS NS (audio_pipeline.c)

```
ns_init(&ns_stage_state.state);
```

With:

Listing 38: Foo (audio_pipeline.c)

```
foo_init(&foo_stage_state.state);
```

Audio Pipeline Setup

Replace:

Listing 39: XMOS NS (audio_pipeline.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_ns,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_ns),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

With:

Listing 40: Foo (audio_pipeline.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_foo,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_foo),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

It is also possible to add or remove stages. Refer to the RTOS Framework documentation on the generic pipeline sw_service.

Replacing ASR Engine Block Replacing the keyword spotter engine has the potential to require significant changes due to various feature extraction input requirements and varied output logic.

The generic intent engine API only requires two functions be declared:

Listing 41: Intent API (intent_engine.h)

```
/* Generic interface for intent engines */
int32_t intent_engine_create(uint32_t priority, void *args);
int32_t intent_engine_sample_push(asr_sample_t *buf, size_t frames);
```

Refer to the existing Sensory model implementation for details on how the output handler is set up, how the audio is conditioned to the expected model format, and how it receives frames from the audio pipeline.

Replacing Example Design Interfaces It may be desired to have a different output interface to talk to a host, or not have a host at all and handle the intent local to the XCORE device.

Different Peripheral IO To add or remove a peripheral IO, modify the bsp_config accordingly. Refer to documentation inside the RTOS Framework on how to instantiate different RTOS peripheral drivers.

Direct Control In a single controller system, the XCORE can be used to control peripherals directly.

The proc_keyword_res task can be modified as follows:

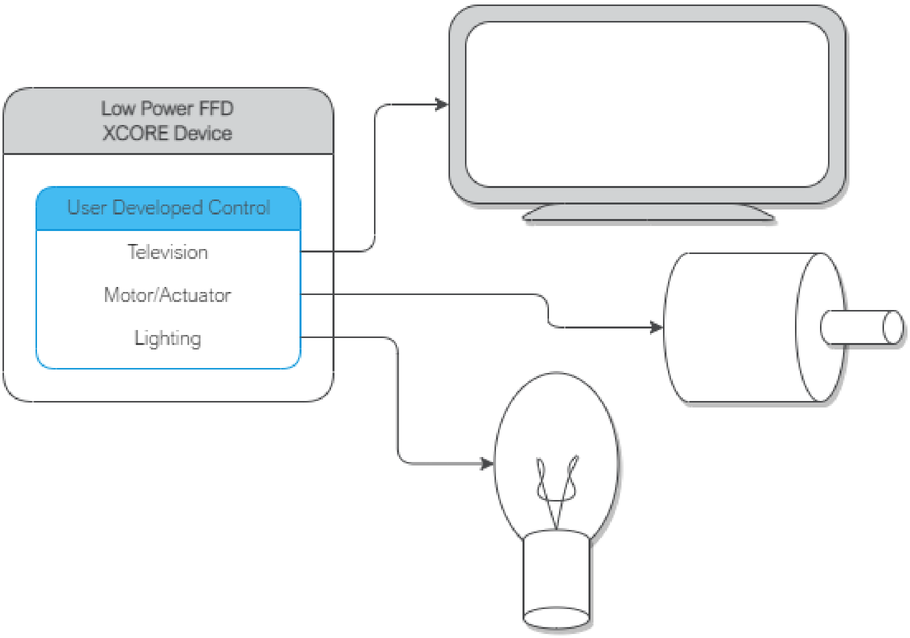
Listing 42: Intent Handler (intent_handler.c)

```
static void proc_keyword_res(void *args) {
    QueueHandle_t q_intent = (QueueHandle_t) args;
    int32_t id = 0;

    while(1) {
        xQueueReceive(q_intent, &id, portMAX_DELAY);

        /* User logic here */
    }
}
```

This code example will receive the ID of each intent, and can be populated by any user application logic. User logic can use other RTOS drivers to control various peripherals, such as screens, motors, lights, etc, based on the intent engine outputs.



Replacing Example Power Control Logic Depending on the peripherals used in the end application, the requirements and handling of the power control/state logic may need adaptation. The power control logic operates in a task where a state machine that is common to both tiles is used. During steady state, each tile is expected to remain is the same state. During transitions each tile executes its own state transition logic. Below outlines the various functions that may need adaptation for a given application.

Listing 43: Locking drivers (power_control.c)

```
static void driver_control_lock(void)
{
#if ON_TILE(POWER_CONTROL_TILE_NO)
    rtos_osal_mutex_get(&gpio_ctx_t0->lock, RTOS_OSAL_WAIT_FOREVER);
#else
    rtos_osal_mutex_get(&qspi_flash_ctx->mutex, RTOS_OSAL_WAIT_FOREVER);
    /* User logic here */
#endif
}
```

Listing 44: Unlocking drivers (power_control.c)

```
static void driver_control_unlock(void)
{
#if ON_TILE(POWER_CONTROL_TILE_NO)
    rtos_osal_mutex_put(&gpio_ctx_t0->lock);
#else
    /* User logic here */
    rtos_osal_mutex_put(&qspi_flash_ctx->mutex);
#endif
}
```

This implementation also includes function calls that are for evaluation/diagnosis purposes and may be removed for end applications. This includes calls to:

- led_indicate_awake
- led_indicate_asleep

When removing these calls, the associated call to *power_control_ind_complete* must either be moved to another location in the application (this is currently handled in led.c's *led_task*) or logic associated with *TASK_NOTIF_MASK_LP_IND_COMPLETE* should be removed/disabled. The *power_control_ind_complete* routine provides a basic means for the power control task to wait for another asynchronous process to complete before proceeding with the state transition logic.

6.2.6.5 Speech Recognition

License The Sensory TrulyHandsFree™ (THF) speech recognition library is *Copyright (C) 1995-2022 Sensory Inc., All Rights Reserved*.

Sensory THF software requires a commercial license granted by Sensory Inc. This software ships with an expiring development license. It will suspend recognition after 11.4 hours or 107 recognition events.

Overview The Sensory THF speech recognition engine runs proprietary models to identify keywords in an audio stream. Models can be generated using VoiceHub.

Two models are provided for the purpose of Low Power FFD. The small wake word model running on tile 1 is approximately 67KB. The command model running on tile 0 is approximately 289KB. On tile 1, the Sensory runtime and application supporting code consumes approximately 239KB of SRAM. On tile 0, the Sensory runtime and application supporting code consumes approximately 210KB of SRAM.

With the command model in flash, the Sensory engine requires a core frequency of at least 450 MHz to keep up with real time. Additionally, the intent engine that is responsible for processing the commands must be on the same tile as the flash.

To run with a different model, see the `Set Sensory model variables` section of the `low_power_ffd.cmake` file. There several variables are set pointing to files that are part of the VoiceHub generated model download. Change these variables to point to the files you downloaded. This can be done for both the wakeword and command models. The command model "net.bin" file, because it is placed in flash memory, must first be nibble swapped. A utility is provided that is part of the host applications built during install. Run that application with the following command:

```
nibble_swap <your-model-prod-net.bin> <your-model-prod-net.bin.nibble_swapped>
```

Make sure run the following commands to rebuild and re-flash the data partition:

```
make clean
make flash_app_example_low_power_ffd -j
```

You may also wish to modify the command ID-to-string lookup table which is located in the `src/intent_engine/intent_engine_io.c` source file.

Wake Word Dictionary

Table 34: English Language Wake Words

Return code (decimal)	Utterance
1	Hello XMOS

Command Dictionary

Table 35: English Language Commands

Return code (decimal)	Utterance
1	Switch on the TV
2	Channel up
3	Channel down
4	Volume up
5	Volume down
6	Switch off the TV
7	Switch on the lights
8	Brightness up
9	Brightness down
10	Switch off the lights
11	Switch on the fan
12	Speed up the fan
13	Slow down the fan
14	Set higher temperature
15	Set lower temperature
16	Switch off the fan

Application Integration In depth information on out of the box integration can be found here: *Host Integration*

6.3 Far-field Voice Assistant

6.3.1 Overview

This is the XCORE-VOICE far-field voice assistant example design.

This application can be used out of the box as a voice processor solution, or expanded to run local wakeword engines.

This application features a full duplex acoustic echo cancellation stage, which can be provided reference audio via I^2S or USB audio. An audio output ASR stream is also available via I^2S or USB audio.

By default, there are two audio integration options. The INT (Integrated) configuration uses I^2S for reference and output audio streams. The UA (USB Accessory) configuration uses USB UAC 2.0 for reference and output audio streams.

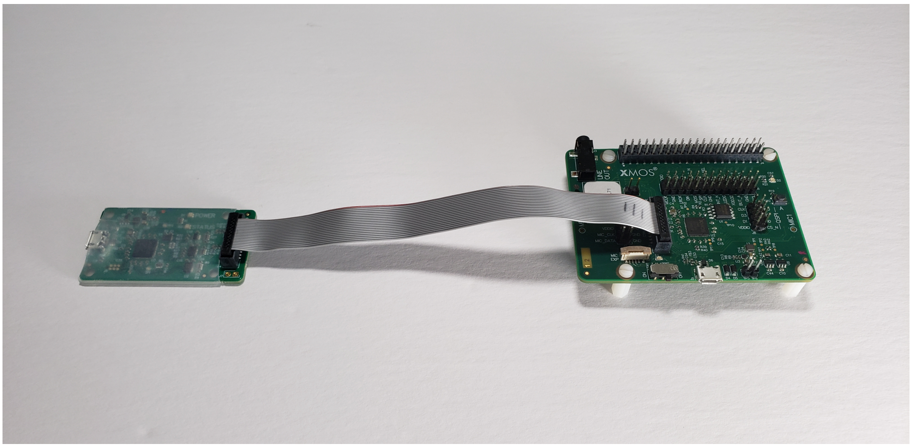
6.3.2 Supported Hardware

This example application is supported on the XK-VOICE-L71 board.

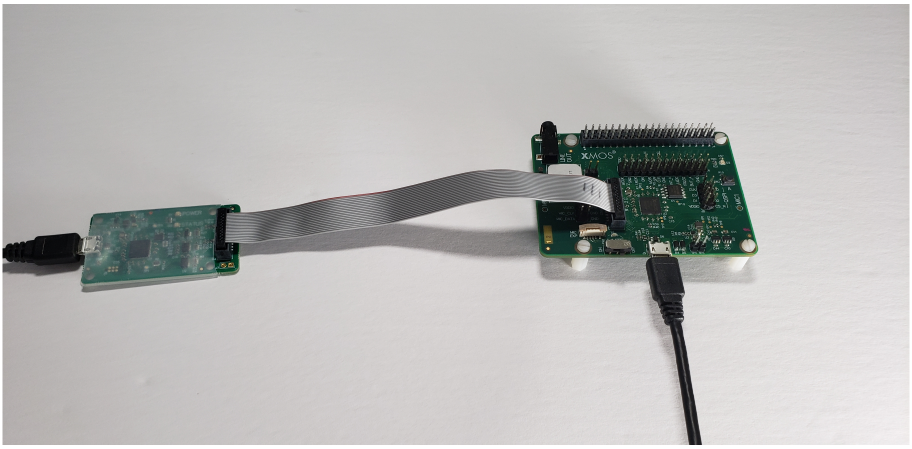
6.3.2.1 Setting up the Hardware This example design requires an XTAG4 and XK-VOICE-L71 board.

xTAG The xTAG is used to program and debug the device

Connect the xTAG to the debug header, as shown below.



Connect the micro USB XTAG4 and micro USB XK-VOICE-L71 to the programming host.


XMOS

6.3.3 Deploying the Firmware with Linux or macOS

This document explains how to deploy the software using *CMake* and *Make*.

6.3.3.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native Toolchain:

Note: Permissions may be required to install the host applications.

```
cmake -B build_host
cd build_host
make install
```

The host applications will be installed at `/opt/xmos/bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.3.3.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the I^{2}S firmware:

```
pip install -r requirements.txt
cmake -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffva_int_fixed_delay
```

After having your python environment activated, run the following commands in the root folder to build the I^{2}S firmware with the Cyberon ASR engine:

```
pip install -r requirements.txt
cmake -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffva_int_cyberon_fixed_delay
```

After having your python environment activated, run the following commands in the root folder to build the USB firmware:

```
pip install -r requirements.txt
cmake -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffva_ua_adec_altarch
```

6.3.3.3 Running the Firmware

Before the firmware is run, the filesystem must be loaded.

Inside of the build folder root, after building the firmware, run one of:

```
make flash_app_example_ffva_int_fixed_delay
make flash_app_example_ffva_int_cyberon_fixed_delay
make flash_app_example_ffva_ua_adec_altarch
```

Once flashed, the application will run.

After the filesystem has been flashed once, the application can be run without flashing. If changes are made to the filesystem image, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_ffva_int_fixed_delay.xe
xrun --xscope example_ffva_int_cyberon_fixed_delay.xe
xrun --xscope example_ffva_ua_adec_altarch.xe
```

6.3.3.4 Upgrading the Firmware

UA variant The UA variants of this application contain DFU over the USB DFU Class V1.1 transport method.

To create an upgrade image from the build folder run:

```
make create_upgrade_img_example_ffva_ua_adec_altarch
```

Once the application is running, a USB DFU v1.1 tool can be used to perform various actions. This example will demonstrate with dfu-util commands. Installation instructions for the respective operating systems can be found here.

To verify the device is running run:

```
dfu-util -l
```

This should result in an output containing:

```
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=2, name="DFU DATAPARTITION",
↪serial="123456"
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=1, name="DFU UPGRADE", serial=
↪"123456"
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=0, name="DFU FACTORY", serial=
↪"123456"
```

The DFU interprets the flash as 3 separate partitions, the read only factory image, the read/write upgrade image, and the read/write data partition containing the filesystem.

The factory image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 0 -U readback_factory_img.bin
```

The factory image can not be written to.

From the build folder, the upgrade image can be written by running:

```
dfu-util -e -d ,20b1:4001 -a 1 -D example_ffva_ua_adec_altarch_upgrade.bin
```

The upgrade image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 1 -U readback_upgrade_img.bin
```

On system reboot, the upgrade image will always be loaded if valid. If the upgrade image is invalid, the factory image will be loaded. To revert back to the factory image, you can upload a file containing the word 0xFFFFFFFF.

The data partition image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 2 -U readback_data_partition_img.bin
```

The data partition image can be written by running:

```
dfu-util -e -d ,20b1:4001 -a 2 -D readback_data_partition_img.bin
```

Note that the data partition will always be at the address specified in the initial flashing call.

INT variant The INT variants of this application contain DFU over I^2C.

To create an upgrade image from the build folder run:

```
make create_upgrade_img_example_ffva_int_fixed_delay
```

Once the application is running, the *xvf_dfu* tool can be used to perform various actions. Installation instructions for Raspbian OS can be found here.

Before running the *xvf_dfu* host application, the `I2C_ADDRESS` value in the file `transport_config.yaml` located in the same folder as the binary file `xvf_dfu` must be updated. This value must match the one set for `appconf_CONTROL_I2C_DEVICE_ADDR` in the `platform_conf.h` file.

The DFU interprets the flash as 3 separate partitions, the read only factory image, the read/write upgrade image, and the read/write data partition containing the filesystem.

The factory image can be read back by running:

```
xvf_dfu --upload-factory readback_factory_img.bin
```

The factory image can not be written to.

From the build folder, the upgrade image can be written by running:

```
xvf_dfu -d example_ffva_int_fixed_delay_upgrade.bin
```

The upgrade image can be read back by running:

```
xvf_dfu --upload-upgrade readback_upgrade_img.bin
```

The device can be rebooted remotely by running

```
xvf_dfu --reboot
```

On system reboot, the upgrade image will always be loaded if valid. If the upgrade image is invalid, the factory image will be loaded. To revert back to the factory image, you can upload a file containing the word 0xFFFFFFFF.

The FFVA-INT variants include some version numbers:

- *APP_VERSION_MAJOR*
- *APP_VERSION_MINOR*
- *APP_VERSION_PATCH*

These values are defined in the `app_conf.h` file, and they can read by running:

```
xvf_dfu --version
```

The data partition image cannot be read or write using the *xvf_dfu* host application.

6.3.3.5 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_ffva_int_fixed_delay.xe
xgdb -ex "connect --xscope" -ex "run" example_ffva_ua_adec_altarch.xe
```

6.3.4 Deploying the Firmware with Native Windows

This document explains how to deploy the software using *CMake* and *Ninja*. If you are not using native Windows MSVC build tools and instead using a Linux emulation tool, refer to *Deploying the Firmware with Linux or macOS*.

To install *Ninja* follow install instructions at https://ninja-build.org/ or on Windows install with `winget` by running the following commands in *PowerShell*:

```
# Install
winget install Ninja-build.ninja
# Reload user Path
$env:Path=[System.Environment]::GetEnvironmentVariable("Path","User")
```

6.3.4.1 Building the Host Applications

This application requires a host application to create the flash data partition. Run the following commands in the root folder to build the host application using your native Toolchain:

Note: Permissions may be required to install the host applications.

Note: A C/C++ compiler, such as Visual Studio or MinGW, must be included in the path.

Before building the host application, you will need to add the path to the XTC Tools to your environment.

```
set "XMOS_TOOL_PATH=<path-to-xtc-tools>"
```

Then build the host application:

```
cmake -G Ninja -B build_host
cd build_host
ninja install
```

The host applications will be installed at `%USERPROFILE%\.xmos\bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

6.3.4.2 Building the Firmware

After having your python environment activated, run the following commands in the root folder to build the I^{2}S firmware:

```
pip install -r requirements.txt
cmake -G Ninja -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
ninja example_ffva_int_fixed_delay
```

After having your python environment activated, run the following commands in the root folder to build the I^{2}S firmware with the Cyberon ASR engine:

```
pip install -r requirements.txt
cmake -G Ninja -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
ninja example_ffva_int_cyberon_fixed_delay
```

After having your python environment activated, run the following commands in the root folder to build the USB firmware:

```
pip install -r requirements.txt
cmake -G Ninja -B build --toolchain=xmos_cmake_toolchain/xs3a.cmake
cd build
ninja example_ffva_ua_adec_altarch
```

6.3.4.3 Running the Firmware

Before the firmware is run, the filesystem must be loaded.

Inside of the build folder root, after building the firmware, run one of:

```
ninja flash_app_example_ffva_int_fixed_delay
ninja flash_app_example_ffva_int_cyberon_fixed_delay
ninja flash_app_example_ffva_ua_adec_altarch
```

Once flashed, the application will run.

After the filesystem has been flashed once, the application can be run without flashing. If changes are made to the filesystem image, the application must be reflashed.

From the build folder run:

```
xrun --xscope example_ffva_int_fixed_delay.xe
xrun --xscope example_ffva_int_cyberon_fixed_delay.xe
xrun --xscope example_ffva_ua_adec_altarch.xe
```

6.3.4.4 Upgrading the Firmware

The UA variants of this application contain DFU over the USB DFU Class V1.1 transport method. In this section DFU over I^2C for the INT variants is not covered. The INT variants require an I^2C connection to the host, and Windows doesn't support this feature.

To create an upgrade image from the build folder run:

```
ninja create_upgrade_img_example_ffva_ua_adec_altarch
```

Once the application is running, a USB DFU v1.1 tool can be used to perform various actions. This example will demonstrate with dfu-util commands. Installation instructions for respective operating system can be found here

To verify the device is running run:

```
dfu-util -l
```

This should result in an output containing:

```
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=2, name="DFU DATAPARTITION",
↪serial="123456"
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=1, name="DFU UPGRADE", serial=
↪"123456"
Found DFU: [20b1:4001] ver=0001, devnum=100, cfg=1, intf=3, path="3-4.3", alt=0, name="DFU FACTORY", serial=
↪"123456"
```

The DFU interprets the flash as 3 separate partitions, the read only factory image, the read/write upgrade image, and the read/write data partition containing the filesystem.

The factory image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 0 -U readback_factory_img.bin
```

The factory image can not be written to.

From the build folder, the upgrade image can be written by running:

```
dfu-util -e -d ,20b1:4001 -a 1 -D example_ffva_ua_adec_altarch_upgrade.bin
```

The upgrade image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 1 -U readback_upgrade_img.bin
```

On system reboot, the upgrade image will always be loaded if valid. If the upgrade image is invalid, the factory image will be loaded. To revert back to the factory image, you can upload an file containing the word 0xFFFFFFFF.

The data partition image can be read back by running:

```
dfu-util -e -d ,20b1:4001 -a 2 -U readback_data_partition_img.bin
```

The data partition image can be written by running:

```
dfu-util -e -d ,20b1:4001 -a 2 -D readback_data_partition_img.bin
```

Note that the data partition will always be at the address specified in the initial flashing call.

6.3.4.5 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
xgdb -ex "connect --xscope" -ex "run" example_ffva_int_fixed_delay.xe
xgdb -ex "connect --xscope" -ex "run" example_ffva_ua_adec_altarch.xe
```

6.3.5 Modifying the Software

The FFVA example design is highly customizable. This section describes how to modify the application.

6.3.5.1 Host Integration

This example design can be integrated with existing solutions or modified to be a single controller solution.

Out of the Box Integration

Out of the box integration varies based on configuration.

INT requires I^2S connections to the host. Refer to the schematic, connecting the host reference audio playback to the ADC I^2S and the host input audio to the DAC I^2S. Out of the box, the INT configuration requires an externally generated MCLK of 12.288 MHz. 24.576 MHz is also supported and can be changed via the compile option MIC_ARRAY_CONFIG_MCLK_FREQ, found in ffva_int.cmake.

UA requires a USB connection to the host.

Support for ASR engine

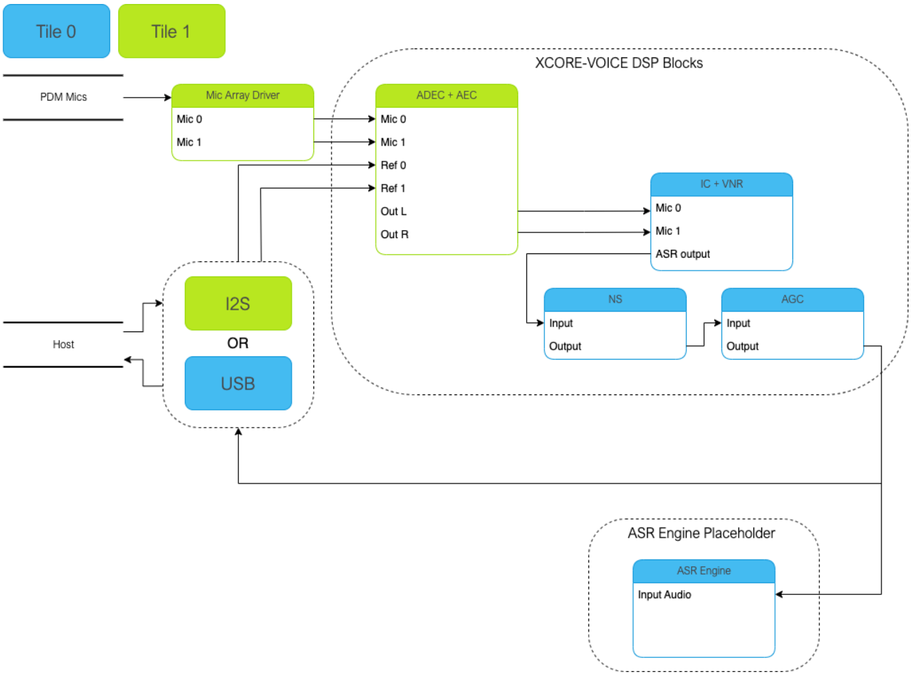
The `example_ffva_int_cyberon_fixed_delay` provides an example about how to include an ASR engine, the Cyberon DSPotter™.

Most of the considerations made in the *section about the FFD devices* are still valid for the FFVA example. The only notable difference is that the pipeline output in the FFVA example is on the same tile as the ASR engine, i.e. tile 0.

Note: Both the audio pipeline and the ASR engine process use the same sample block length. `appconfINTENT_SAMPLE_BLOCK_LENGTH` and `appconfAUDIO_PIPELINE_FRAME_ADVANCE` are both 240.

More information about the Cyberon engine can be found in *Speech Recognition - Cyberon* section.

6.3.5.2 Design Architecture The application consists of a PDM microphone input which is fed through the XMOS-VOICE DSP blocks. The output ASR channel is then output over I^2S or USB.



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

The DFU process is internally managed by the DFU controller module within the firmware. This module is tasked with overseeing the DFU state machine and executing DFU operations. The list of states and transactions are represented in the diagram in Fig. 1.

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

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

The DFU allows the following operations:

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

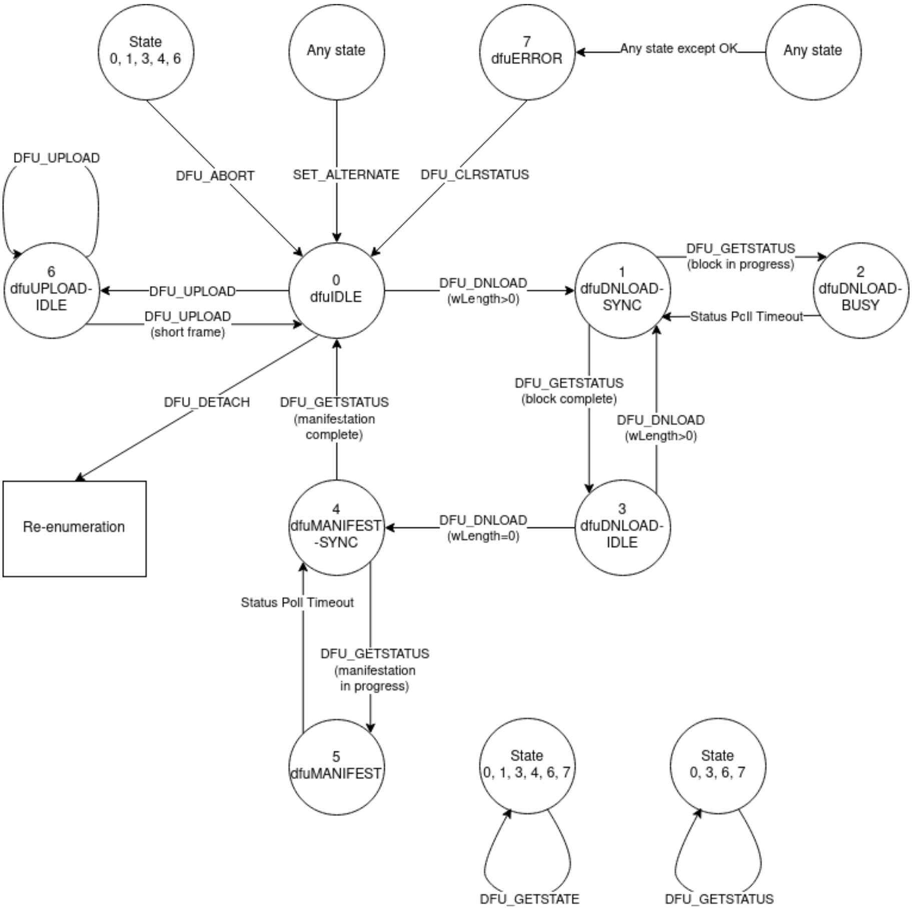


Fig. 1: State diagram of the DFU operations

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

A message sequence chart of the download operation is below:

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

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

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

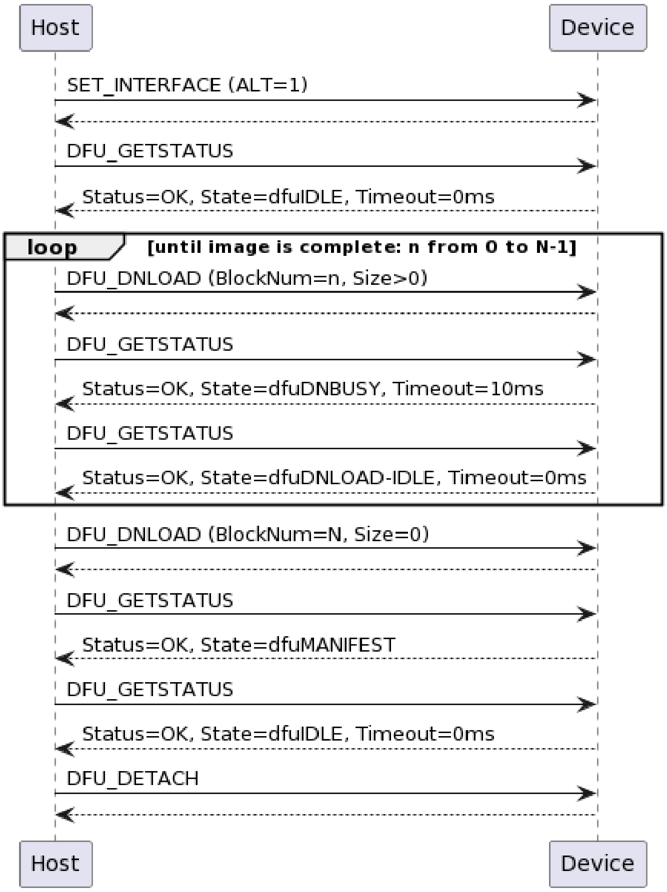


Fig. 2: Message sequence chart of the download operation

A message sequence chart of the reboot operation is below:

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

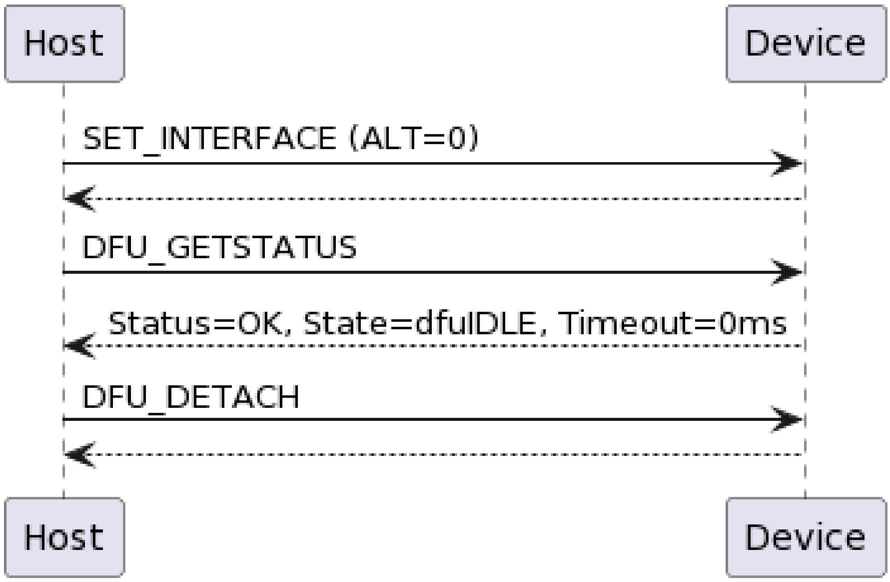


Fig. 3: Message sequence chart of the reboot operation

A message sequence chart of the upload operation is below:

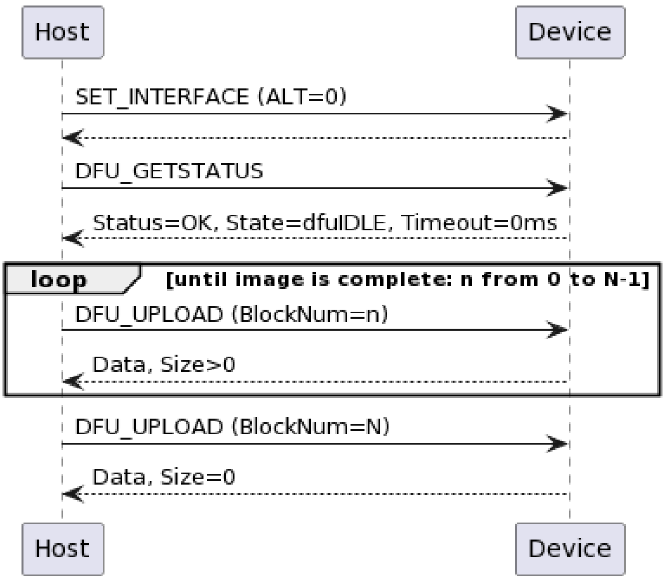


Fig. 4: Message sequence chart of the upload operation

Note: The end of the image transfer is indicated by a `DFU_UPLOAD` message of size less than the transport medium maximum; this is 4096 bytes in UA and 128 bytes in INT.

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

DFU over I²C implementation The INT variant of the device presents a DFU interface that may be controlled over I²C.

Fig. 5 shows the modules involved in processing the DFU commands. The *I2C* task has a dedicated logical core so that it is always ready to receive and send control messages. The DFU state machine is driven by the control commands. The DFU state machine interacts with a separate RTOS task in order to asynchronously perform flash read/write operations.

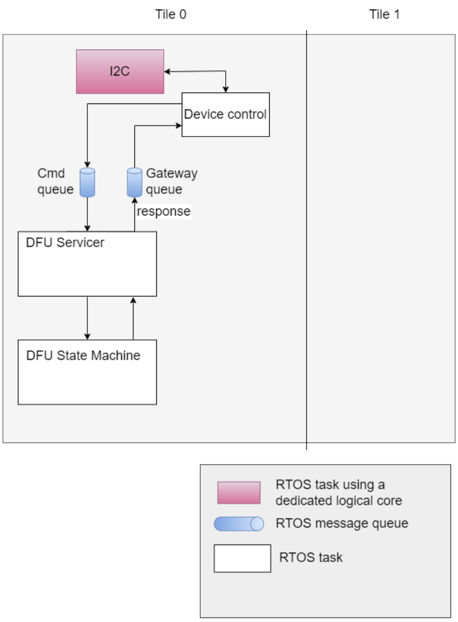


Fig. 5: sln_voice Control Plane Components Diagram

Fig. 6 shows the interaction between the Device Control module and the DFU Servicer. In this diagram, boxes with the same colour reside in the same RTOS task.

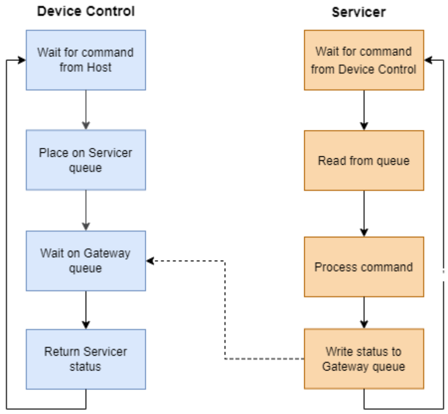


Fig. 6: sln_voice Device Control – Servicer Flow Chart

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

The FFVA-INT uses a packet protocol to receive control commands and send each corresponding response. Because packet transmission occurs over a very short-haul transport, as in I^2C, 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. 7 depicts the structure of each packet.

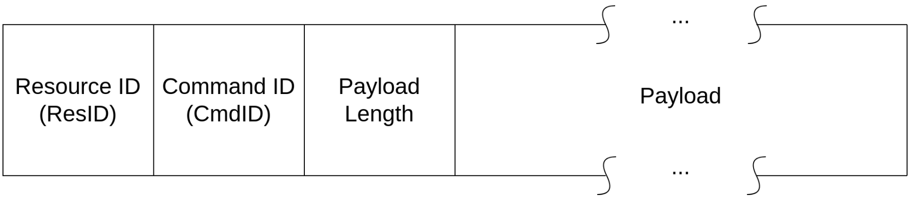


Fig. 7: sln_voice Control Plane Packet Diagram

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

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

Table 36: DFU commands

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

continues on next page

Table 36 – continued from previous page

Name	ID	Length	Payload Structure	Purpose
DFU_UPLOAD	2	130	2 bytes length marker, followed by 128 bytes of data buffer	Read-only command. The first two bytes indicate how many bytes of data are being transmitted in this packet. These bytes are little-endian, so byte 0 represents the low byte and byte 1 represents the high byte of an unsigned 16b integer. The remaining 128 bytes are a data buffer of data received from the device. All control command packets are a fixed length, and therefore this buffer will be padded to length 128 by the device before transmission. The device will, as per the USB DFU specification, mark the end of the upload process by sending a "short frame" - a packet with a length marker less than 128 bytes.
DFU_GETSTATUS	3	5	1 byte representing device status, 3 bytes representing the requested timeout, 1 byte representing the next device state.	Read-only command. The first byte returns the device status code, as described in the USB DFU specification in the table in section 6.1.2. The next 3 bytes represent the amount of time the host should wait, in ms, before issuing any other commands. This timeout is used in the DNLOAD process to allow the device time to write to flash. This value is little-endian, so bytes 1, 2, and 3 represent the low, middle, and high bytes respectively of an unsigned 24b integer. The final byte returns the number of the state that the device will move into immediately following the return of this request, as described in the USB DFU specification in the table in section 6.1.2.
DFU_CLRSTATUS	4	1	Payload unused	Write-only command. Moves the device out of state 10, dfuERROR. Payload is required for protocol, but is discarded within the device.

continues on next page

Table 36 – continued from previous page

Name	ID	Length	Payload Structure	Purpose
DFU_GETSTATE	5	1	1 byte representing current device state.	Read-only command. The first (and only) byte represents the number of the state that the device is currently in, as described in the USB DFU specification in the table in section 6.1.2.
DFU_ABORT	6	1	Payload unused	Write-only command. Aborts an ongoing upload or download process. Payload is required for protocol, but is discarded within the device.
DFU_SETALTERNATE	64	1	1 byte representing either factory (0) or upgrade (1) DFU target images	Write-only command. Sets which of the factory or upgrade images should be targeted by any subsequent upload or download commands. Use of this command entirely resets the DFU state machine to initial conditions: the device will move to dfuIDLE, clear all error conditions, wipe all internal DFU data buffers, and reset all other DFU state apart from the DFU_TRANSFERBLOCK value. This command is included to emulate the SET_ALTERNATE request available in USB.

continues on next page

Table 36 – continued from previous page

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

continues on next page

Table 36 – continued from previous page

Name	ID	Length	Payload Structure	Purpose
DFU_REBOOT	89	1	Payload unused	Write-only command. Restarts the device. Payload is required for protocol, but is discarded within the device. This is a utility command intended to provide a clear and unambiguous interface for restarting the device. Use of this command should be preferred over DFU_DETACH for this purpose.

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

When writing a custom compliant host application, the use of XMOS' **fwk_rtos** library is advised; the **device_control** library provided there gives a host API that can communicate effectively with the FFVA-INT. A description of the I^2C bus activity during the execution of the above DFU commands is provided below, in the instance that usage of the **device_control** library is inconvenient or impossible.

The FFVA-INT I^2C address is set by default as 0x42. This may be confirmed by examination of the `appconf_CONTROL_I2C_DEVICE_ADDR` define in the `platform_conf.h` file. The I^2C address may also be altered by editing this file. The DFU resource has an internal "resource ID" of 0xF0. This maps to the register that read/write operations on the DFU resource should target - therefore, the register to write to will always be 0xF0.

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

- First, set up a write to the device address. For a default device configuration, a write operation will always start by a write token to 0x42 (START, 7 bits of address [0x42], R/W bit [0 to specify write]), wait for ACK, followed by specifying the register to write [Resource ID 0xF0] (and again wait for ACK).
- Then, write the command ID (in this example, 64 [0x40]) from the above table.
- Then, write the total transfer size, *including the register byte*. In this example, that will be 4 bytes (register byte, command ID, length byte, and 1 byte of payload), so write 0x04.
- Finally, send the payload - e.g. 1 to set the alternate setting to "upgrade".
- The full sequence for this write command will therefore be START, 7 bits of address [0x42], 0 (to specify write), hold for ACK, 0xF0, hold for ACK, 0x40, hold for ACK, 0x04, hold for ACK, 0x01, hold for ACK, STOP.
- To complete the transaction, the device must then be queried; set up a read to 0x42 (START, 7 bits of address [0x42], R/W bit [1 to specify read], wait for ACK). The device will clock-stretch until it is ready, at which point it will release the clock and transmit one byte of status information. This will be a value from the enum `control_ret_t` from `device_control_shared.h`, found in `modules\rtos\modules\sw_services\device_control\api`.

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

- Set up a write to the device; as above, this will mean sending START, 7 bits of device address [0x42], 0 (to specify write), hold for ACK. Send the DFU resource ID [0xF0], hold for ACK.
- Then, write the command ID (in this example, 3), bitwise ANDed with 0x80 (to specify this as a read command) - in this example therefore 0x83 should be sent, and hold for ACK.
- Then, write the total length of the expected reply. In this example, the command has a payload of 5 bytes. The device will also prepend the payload with a status byte. Therefore, the expected reply length will be 6 bytes [0x06]. Hold for ACK.
- Then, issue a repeated START. Follow this with a read from the device: the repeated START, 7 bits of device address [0x42], 1 (to specify read), hold for ACK. The device will clock-stretch until it is ready. It will then send a status byte (from the enum `control_ret_t` as described above), followed by a payload of requested data - in this example, the device will send 5 bytes. ACK each received byte. After the last expected byte, issue a STOP.

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

6.3.5.4 Audio Pipeline The audio pipeline in FFVA processes two channel PDM microphone input into a single output channel, intended for use by an ASR engine.

The audio pipeline consists of 4 stages.

Table 37: FFVA Audio Pipeline

Stage	Description	Input Channel Count	Output Channel Count
1	Acoustic Echo Cancellation	2	2
2	Interference Canceller and Voice Noise Ratio	2	1
3	Noise Suppression	1	1
4	Automatic Gain Control	1	1

See the Voice Framework User Guide for more information.

6.3.5.5 Software Description

Overview There are three main build configurations for this application.

Table 38: FFVA INT Fixed Delay Resources

Resource	Tile 0	Tile 1
Total Memory Free	141k	80k
Runtime Heap Memory Free	75k	76k

Table 39: FFVA INT Cyberon Fixed Delay Resources

Resource	Tile 0	Tile 1
Total Memory Free	21k	79k
Runtime Heap Memory Free	19k	81k

Table 40: FFVA UA ADEC Resources

Resource	Tile 0	Tile 1
Total Memory Free	94k	59k
Runtime Heap Memory Free	54k	83k

The description of the software is split up by folder:

Table 41: FFVA Software Description

Folder	Description
Audio Pipelines	Preconfigured audio pipelines
examples/ffva/bsp_config	Board support configuration setting up software based IO peripherals
examples/ffva/filesystem_support	Filesystem contents for application
examples/ffva/src	Main application
modules/asr/intent_engine	Intent engine integration (FFVA INT Cyberon only)
modules/asr/intent_handler	Intent engine output integration (FFVA INT Cyberon only)

examples/ffva/bsp_config This folder contains bsp_configs for the FFVA application. More information on bsp_configs can be found in the RTOS Framework documentation.

Table 42: FFVA bsp_config

Filename/Directory	Description
dac directory	DAC ports for supported bsp_configs
XCORE-AI-EXPLORER directory	experimental bsp_config, not recommended for general use
XK_VOICE_L71 directory	default FFVA application bsp_config
bsp_config.cmake	cmake for adding FFVA bsp_configs

examples/ffva/filesystem_support This folder contains filesystem contents for the FFVA application.

Table 43: FFVA filesystem_support

Filename/Directory	Description
demo.txt	Example file

Audio Pipelines This folder contains preconfigured audio pipelines for the FFVA application.

Table 44: FFVA Audio Pipelines

Filename/Directory	Description
api directory	include folder for audio pipeline modules
src directory	contains preconfigured XMOS DSP audio pipelines
audio_pipeline.cmake	cmake for adding audio pipeline targets

Major Components The audio pipeline module provides the application with three API functions:

Listing 45: Audio Pipeline API (audio_pipeline.h)

```
void audio_pipeline_init(
        void *input_app_data,
        void *output_app_data);

void audio_pipeline_input(
        void *input_app_data,
        int32_t **input_audio_frames,
        size_t ch_count,
        size_t frame_count);

int audio_pipeline_output(
        void *output_app_data,
        int32_t **output_audio_frames,
        size_t ch_count,
        size_t frame_count);
```

audio_pipeline_init This function has the role of creating the audio pipeline task(s) and initializing DSP stages.

audio_pipeline_input This function is application defined and populates input audio frames used by the audio pipeline. In FFVA, this function is defined in *main.c*.

audio_pipeline_output This function is application defined and populates input audio frames used by the audio pipeline. In FFVA, this function is defined in *main.c*.

examples/ffva/src This folder contains the core application source.

Table 45: FFVA src

Filename/Directory	Description
gpio_test directory	contains general purpose input handling task
usb directory	contains intent handling code
ww_model_runner directory	contains placeholder wakeword model runner task
app_conf_check.h	header to validate app_conf.h
app_conf.h	header to describe app configuration
config.xscope	xscope configuration file
ff_appconf.h	default fatfs configuration header
FreeRTOSConfig.h	header to describe FreeRTOS configuration
main.c	main application source file

Main The major components of main are:

Listing 46: Main components (main.c)

```
void startup_task(void *arg)
void tile_common_init(chanend_t c)
void main_tile0(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
void main_tile1(chanend_t c0, chanend_t c1, chanend_t c2, chanend_t c3)
void i2s_rate_conversion_enable(void)
size_t i2s_send_upsample_cb(rtos_i2s_t *ctx, void *app_data, int32_t *i2s_frame, size_t i2s_frame_size, int32_t
↪*send_buf, size_t samples_available)

size_t i2s_send_downsample_cb(rtos_i2s_t *ctx, void *app_data, int32_t *i2s_frame, size_t i2s_frame_size, int32_
↪t *receive_buf, size_t sample_spaces_free)
```

startup_task This function has the role of launching tasks on each tile. For those familiar with XCORE, it is comparable to the main par loop in an XC main.

tile_common_init This function is the common tile initialization, which initializes the bsp_config, creates the startup task, and starts the FreeRTOS kernel.

main_tile0 This function is the application C entry point on tile 0, provided by the SDK.

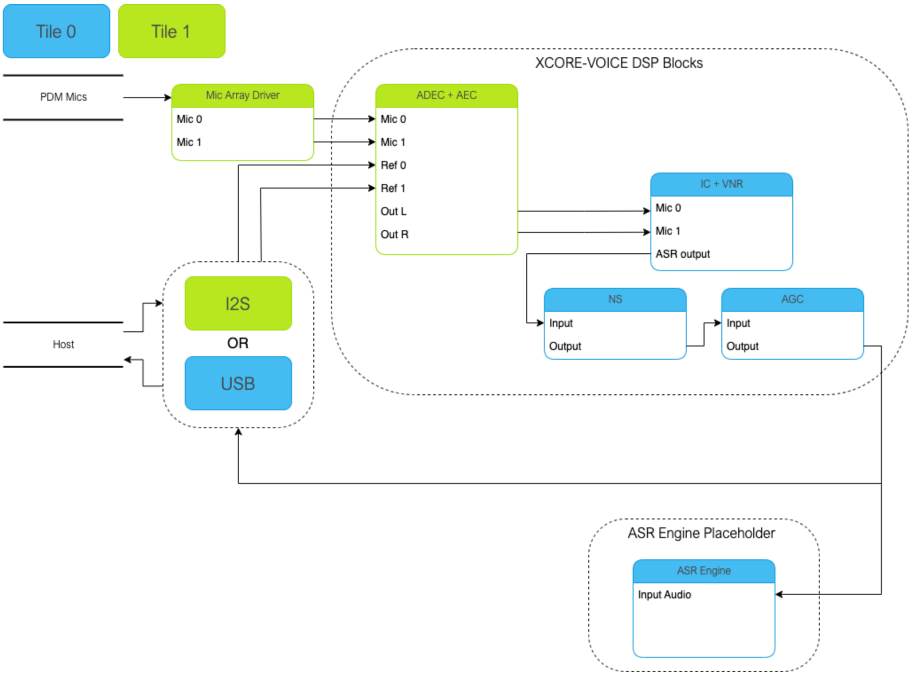
main_tile1 This function is the application C entry point on tile 1, provided by the SDK.

i2s_rate_conversion_enable This application features 16kHz and 48kHz audio input and output. The XMOS DPS blocks operate on 16kHz audio. Input streams are downsampled when needed. Output streams are upsampled when needed. When in I^2S modes This function is called by the bsp_config to enable the I^2S sample rate conversion.

i2s_send_upsample_cb This function is the I^2S upsampling callback.

i2s_send_downsample_cb This function is the I^2S downsampling callback.

6.3.5.6 Software Modifications The FFVA example design consists of three major software blocks, the audio interface, audio pipeline, and placeholder for a keyword handler. This section will go into detail on how to modify each/all of these subsystems.



It is highly recommended to be familiar with the application as a whole before attempting replacing these functional units.

See *Memory and CPU Requirements* for more details on the memory footprint and CPU usage of the major software components.

Replacing XCORE-VOICE DSP Block The audio pipeline can be replaced by making changes to the `audio_pipeline.c` file.

It is up to the user to ensure that the input and output frames of the audio pipeline remain the same, or the remainder of the application will not function properly.

This section will walk through an example of replacing the XMOS NS stage, with a custom stage foo.

Declaration and Definition of DSP Context Replace:

Listing 47: XMOS NS (audio_pipeline_t0.c)

```
static ns_stage_ctx_t DWORD_ALIGNED ns_stage_state = {};
```

With:

Listing 48: Foo (audio_pipeline_t0.c)

```
typedef struct foo_stage_ctx {
    /* Your required state context here */
} foo_stage_ctx_t;
```

(continues on next page)

(continued from previous page)

```
static foo_stage_ctx_t foo_stage_state = {};
```

DSP Function Replace:

Listing 49: XMOS NS (audio_pipeline_t0.c)

```
static void stage_ns(frame_data_t *frame_data)
{
#if appconfAUDIO_PIPELINE_SKIP_NS
#else
    int32_t DWORD_ALIGNED ns_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    configASSERT(NS_FRAME_ADVANCE == appconfAUDIO_PIPELINE_FRAME_ADVANCE);
    ns_process_frame(
                &ns_stage_state.state,
                ns_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, ns_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
#endif
}
```

With:

Listing 50: Foo (audio_pipeline_t0.c)

```
static void stage_foo(frame_data_t *frame_data)
{
    int32_t foo_output[appconfAUDIO_PIPELINE_FRAME_ADVANCE];
    foo_process_frame(
                &foo_stage_state.state,
                foo_output,
                frame_data->samples[0]);
    memcpy(frame_data->samples, foo_output, appconfAUDIO_PIPELINE_FRAME_ADVANCE * sizeof(int32_t));
}
```

Runtime Initialization Replace:

Listing 51: XMOS NS (audio_pipeline_t0.c)

```
ns_init(&ns_stage_state.state);
```

With:

Listing 52: Foo (audio_pipeline_t0.c)

```
foo_init(&foo_stage_state.state);
```

Audio Pipeline Setup Replace:

Listing 53: XMOS NS (audio_pipeline_t0.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_ns,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_ns),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

With:

Listing 54: Foo (audio_pipeline_t0.c)

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage_vnr_and_ic,
    (pipeline_stage_t)stage_foo,
    (pipeline_stage_t)stage_agc,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_vnr_and_ic) + RTOS_THREAD_STACK_SIZE(audio_pipeline_
↪input_i),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_foo),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage_agc) + RTOS_THREAD_STACK_SIZE(audio_pipeline_output_
↪i),
};
```

It is also possible to add or remove stages. Refer to the RTOS Framework documentation on the generic pipeline sw_service.

Changing the ASR engine THE FFVA provides an example with a specific ASR engine. A different ASR engine can be used by updating and adding the necessary files in `modules\asr`.

Replacing Example Design Interfaces It may be desired to have a different input or output interfaces to talk to a host.

Hybrid Audio Peripheral IO One example use case may be to create a hybrid audio solution where reference frames or output audio streams are used over an interface other than I^2S or USB.

Listing 55: Audio Pipeline Input (main.c)

```
void audio_pipeline_input(void *input_app_data,
                        int32_t **input_audio_frames,
                        size_t ch_count,
                        size_t frame_count)
{
    (void) input_app_data;
    int32_t **mic_ptr = (int32_t **)(input_audio_frames + (2 * frame_count));

    static int flushed;
    while (!flushed) {
        size_t received;
        received = rtos_mic_array_rx(mic_array_ctx,
                                    mic_ptr,
                                    frame_count,
                                    0);
        if (received == 0) {
            rtos_mic_array_rx(mic_array_ctx,
                                mic_ptr,
                                frame_count,
                                portMAX_DELAY);
            flushed = 1;
        }
    }

    rtos_mic_array_rx(mic_array_ctx,
                        mic_ptr,
                        frame_count,
                        portMAX_DELAY);

    /* Your ref input source here */
}
```

Refer to documentation inside the RTOS Framework on how to instantiate different RTOS peripheral drivers. Populate the above code snippet with your input frame source. Refer to the default application for an example of populating reference via I^2S or USB.

Listing 56: Audio Pipeline Output (main.c)

```
int audio_pipeline_output(void *output_app_data,
                        int32_t **output_audio_frames,
                        size_t ch_count,
```

(continues on next page)

(continued from previous page)

```
                        size_t frame_count)
{
    (void) output_app_data;

    /* Your output sink here */

#if appconfWW_ENABLED
    ww_audio_send(intertile_ctx,
                  frame_count,
                  (int32_t(*)[2])output_audio_frames);
#endif

    return AUDIO_PIPELINE_FREE_FRAME;
}
```

Refer to documentation inside the RTOS Framework on how to instantiate different RTOS peripheral drivers. Populate the above code snippet with your output frame sink. Refer to the default application for an example of outputting the ASR channel via I^{2}S or USB.

Different Peripheral IO To add or remove a peripheral IO, modify the bsp_config accordingly. Refer to documentation inside the RTOS Framework on how to instantiate different RTOS peripheral drivers.

Application Filesystem Usage This application is equipped with a FAT filesystem in flash for general use. To add files to the filesystem, simply place them in the *filesystem_support* directory before running the filesystem setup commands in *Deploying the Firmware with Linux or macOS* or *Deploying the Firmware with Native Windows*.

The application can access the filesystem via the *FatFS* API.

6.4 PDM Microphone Aggregator Example

> **Warning:** This example is deprecated and will be moved into a separate Application Note and may be removed in the next major release.

This example provides a bridge between 16 PDM microphones to either TDM16 slave or USB Audio and targets the xcore-ai explorer board.

This application is to support cases where many microphone inputs need to be sent to a host where signal processing will be performed. Please see the other examples in sln_voice where signal processing is performed within the xcore in firmware.

This example uses a modified mic_array with multiple decimator threads to support 16 DDR microphones on a single 8 bit input port. The example is written as 'bare-metal' and runs directly on the XCORE device without an RTOS.

6.4.1 Obtaining the app files

Download the main repo and submodules using:

```
$ git clone --recurse git@github.com:xmos/sln_voice.git
$ cd sln_voice/
```

6.4.2 Building the app

First make sure that your XTC tools environment is activated.

6.4.2.1 Linux or Mac

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
$ pip install -r requirements.txt
$ mkdir build
$ cd build
$ cmake --toolchain ../xmos_cmake_toolchain/xs3a.cmake  ..
$ make example_mic_aggregator_tdm -j
$ make example_mic_aggregator_usb -j
```

Following initial `cmake` build, as long as you don't add new source files, you may just type:

```
$ make example_mic_aggregator_tdm -j
$ make example_mic_aggregator_usb -j
```

If you add new source files you will need to run the `cmake` step again.

6.4.2.2 Windows

It is recommended to use *Ninja* or *xmake* as the make system under Windows. *Ninja* has been observed to be faster than *xmake*, however *xmake* comes natively with XTC tools. This firmware has been tested with *Ninja* version v1.11.1.

To install Ninja, follow these steps:

- Download `ninja.exe` from https://github.com/ninja-build/ninja/releases. This firmware has been tested with Ninja version v1.11.1.
- Ensure Ninja is on the command line path. You can add to the path permanently by following these steps https://www.computerhope.com/issues/ch000549.htm. Alternatively you may set the path in the current command line session using something like `set PATH=%PATH%;C:\Users\xmos\utils\ninja`

After having your python environment activated, run the following commands in the root folder to build the firmware:

```
$ pip install -r requirements.txt
$ md build
$ cd build
$ cmake -G "Ninja" --toolchain  ..\xmos_cmake_toolchain\xs3a.cmake ..
$ ninja example_mic_aggregator_tdm.xe -j
$ ninja example_mic_aggregator_usb.xe -j
```

Following initial **cmake** build, as long as you don't add new source files, you may just type:

```
$ ninja example_mic_aggregator_tdm.xe -j
$ ninja example_mic_aggregator_usb.xe -j
```

If you add new source files you will need to run the **cmake** step again.

6.4.3 Running the app

Connect the explorer board to the host and type:

```
$ xrun example_mic_aggregator_tdm.xe
$ xrun example_mic_aggregator_usb.xe
```

Optionally, you may use xrun **--xscope** to provide debug output.

6.4.4 Required Hardware

The application runs on the XCORE-AI Explorer board version 2 (with integrated XTAG debug adapter). You will require in addition:

- The dual DDR microphone board that attaches via the flat flex connector.
- Header pins soldered into:
  - J14, J10, SCL/SDA IOT, the I2S expansion header, MIC data and MIC clock.
- Six jumper wires. Please see the microphone aggregator main documentation for details on how these are connected.

An oscilloscope will also be handy in case of hardware debug being needed.

Note: You will only be able to inject PDM data to two channels at a time due to a single pair of microphones on the HW.

If you wish to see all 16 microphones running then an external microphone board with 16 microphones (DDR connected to 8 data lines) is required.

6.4.5 Operation

The design consists of a number of tasks connected via the xcore-ai silicon communication channels. The decimators in the microphone array are configured to produce a 48 kHz PCM output. The 16 output channels are loaded into a 16 slot TDM slave peripheral running at 24.576 MHz bit clock or a USB Audio Class 2 asynchronous interface and are optionally amplified. The TDM build also provides a simple I^2C slave interface to allow gains to be controlled at run-time. The USB build supports USB Audio Class 2 compliant volume controls.

For the TDM build, a simple TDM16 master peripheral is included as well as a local 24.576 MHz clock source so that mic_array and TDM16 slave operation may be tested standalone through the use of jumper cables. These may be removed when integrating into a system with TDM16 master supplied.

6.4.6 Software Architecture

The applications are written on bare metal and use logical cores (hardware threads) to implement the functional blocks. Each of the tasks are connected using channels provided in the xcore-ai architecture. The thread diagrams are shown in Fig. 8 and Fig. 9.

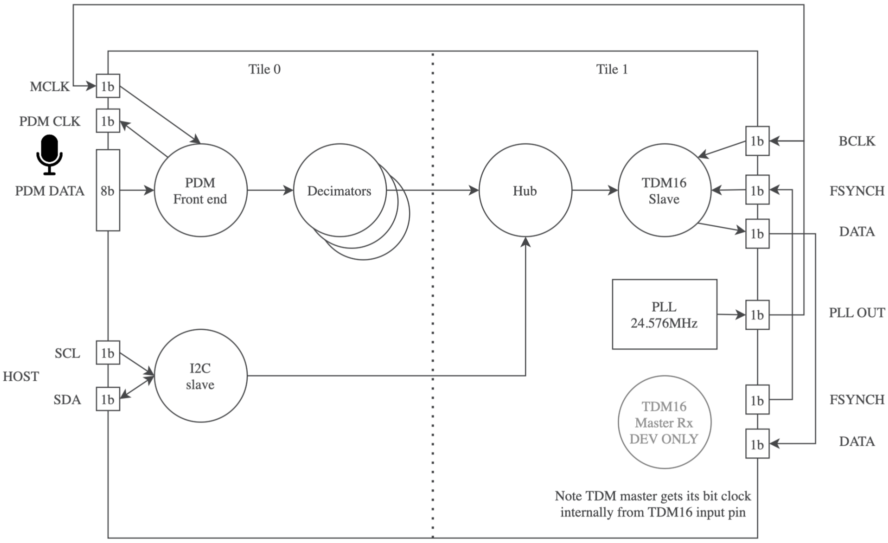


Fig. 8: Microphone Aggregator TDM Thread Diagram

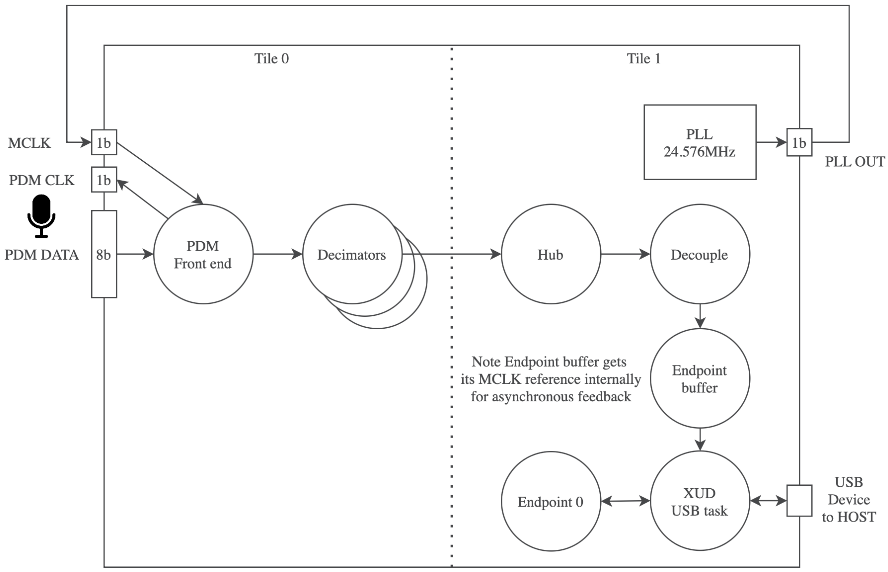


Fig. 9: Microphone Aggregator USB Thread Diagram

6.4.6.1 PDM Capture Both the TDM and USB aggregator examples share a common PDM front end. This consists of an 8 bit port with each data line connected to two PDM microphones each configured to provide data on a different clock edge. The 3.072 MHz clock for the PDM microphones is provided by the xcore-ai device on a 1 bit port and clocks all PDM microphones. The PDM clock is divided down from the 24.576 MHz local MCLK.

The data collected by the 8 bit port is sent to the lib_mic_array block which de-interleaves the PDM data streams and performs decimation of the PDM data down to 48 kHz 32 bit PCM samples. Due to the large number of microphones the PDM capture stage uses four hardware threads on tile[0]; one for the microphone capture and three for decimation. This is needed to divide the processing workload and meet timing comfortably.

Samples are forwarded to the next stage at a rate of 48 kHz resulting in a packet of 16 PCM samples per exchange.

6.4.6.2 Audio Hub The 16 channels of 48 kHz PCM streams are collected by *Hub* and are amplified using a saturated gain stage. The initial gain is set to 100, since a gain of 1 sounds very quiet due to the mic_array output being scaled to allow acoustic overload of the microphones without clipping within the decimators. This value can be overridden using the `MIC_GAIN_INIT` define in *app_conf.h*.

Additionally for the TDM configuration, the *Hub* task also checks for control packets from I^2C which may be used to dynamically update the individual gains at runtime.

A single hardware thread contains the task and a triple buffer scheme is used to ensure there is always a free buffer available to write into regardless of the relative phase between the production and consumption of microphone samples.

The *Hub* task has plenty of timing slack and is a suitable place for adding signal processing if needed.

6.4.6.3 TDM Host Connection The TDM build supports a 16-slot TDM slave Tx peripheral from the fwk_io sub-module. In this application it runs at 24.576 MHz bit clock which supports 16 channels of 32 bit, 48 kHz samples per frame.

The TDM component uses a single hardware thread.

For the purpose of debugging a simple TDM 16 Master Rx component is provided. This allows the transmitted TDM frames from the application to be received and checked without having to connect an external TDM Master. It may be deleted / disconnected without affecting the core application.

Note: The simple TDM 16 Master Rx component is not regression tested and is for evaluation of TDM 16 Slave Tx in this application only.

6.4.6.4 USB Host Connection As an alternative to TDM, a USB host connection is also supported. The USB connection uses the following specifications:

- USB High Speed (480 Mbps)
- USB Audio Class 2.0
- Asynchronous mode (audio clock is provided by the firmware)
- 24 bit Audio slots

- 48 kHz Sample Rate

The USB host connection functionality is provided by lib_xua which is the core library of XMOS's USB Audio solution.

The USB Audio subsection uses a total of four hardware threads in this application.

6.4.7 Resource Usage

The xcore-ai device has a total resource count of 2 x 524288 Bytes of memory and 2 x 8 hardware threads across two tiles. This application uses around half of the processing resources and a tiny fraction of the available memory meaning there is plenty of space inside the chip for additional functionality if needed.

6.4.7.1 TDM Build

Tile	Memory	Threads
0	25996	5
1	22812	2*
Total	48808	7

- An additional debug TDM Master thread is used on Tile[1] by default which is not needed in a practical deployment.

6.4.7.2 USB Build

Tile	Memory	Threads
0	24252	4
1	52116	5
Total	76368	9

6.4.8 Board Configuration

Make the following connections between headers using flying leads:

Host Connection	Board Connection	Note
MIC CLK	J14 '00'	This is the microphone clock which is to be sent to the PDM microphones from J14.
MIC DATA	J14 '14'	This is the data line for microphones 0 and 8. See below.
I2S LRCLK	J10 '36'	This is the FSYCNH input for TDM slave. J10 '36' is the TDM master FSYNCH output for the application.
I2S MCLK	I2S BCLK	MCLK is the 24.576MHz clock which directly drives the BCLK input for the TDM slave.
I2S DAC	J10 '38'	I2S DAC is the TDM Slave Tx out which is read by the TDM Master Rx input on J10.

To access other microphone inputs use the following:

Mic pair	J14 pin
0, 8	14
1, 9	15
2, 10	16
3, 11	17
4, 12	18
5, 13	19
6, 14	20
7, 15	21

For I^2C control, make the following connections:

Host Connection	Board Connection
SCL IOL	Your I2C host SCL.
SDA IOL	Your I2C host SDA.
GND	Your I2C host ground.

The I^2C slave is tested at 100 kHz SCL.

6.4.9 I2C Controlled Volume

For the TDM build, there are 32 registers which control the gain of each of the 16 output channels. The 8 bit registers contain the upper 8 bit and lower 8 bit of the microphone gain respectively. The initial gain is set to 100, since 1 is quiet due to the mic_array output being scaled to allow acoustic overload of the microphones without clipping. Typically a gain of a few hundred works for normal conditions. The gain is only applied after the lower byte is written.

The gain applied is saturating so no overflow will occur, only clipping.

Register	Value
0	Channel 0 upper gain byte
1	Channel 0 lower gain byte
2	Channel 1 upper gain byte
3	Channel 1 lower gain byte
4	Channel 2 upper gain byte
5	Channel 2 lower gain byte
6	Channel 3 upper gain byte
7	Channel 3 lower gain byte
8	Channel 4 upper gain byte
9	Channel 4 lower gain byte
10	Channel 5 upper gain byte
11	Channel 5 lower gain byte
12	Channel 6 upper gain byte

continues on next page

Table 46 – continued from previous page

Register	Value
13	Channel 6 lower gain byte
14	Channel 7 upper gain byte
15	Channel 7 lower gain byte
16	Channel 8 upper gain byte
17	Channel 8 lower gain byte
18	Channel 9 upper gain byte
19	Channel 9 lower gain byte
20	Channel 10 upper gain byte
21	Channel 10 lower gain byte
22	Channel 11 upper gain byte
23	Channel 11 lower gain byte
24	Channel 12 upper gain byte
25	Channel 12 lower gain byte
26	Channel 13 upper gain byte
27	Channel 13 lower gain byte
28	Channel 14 upper gain byte
29	Channel 14 lower gain byte
30	Channel 15 upper gain byte
31	Channel 15 lower gain byte

If using a raspberry Pi as the I^2C host you may use the following commands:

```
$ i2cset -y 1 0x3c 0 0 #Set the gain on mic channel 0 to 50
$ i2cset -y 1 0x3c 1 50 #Set the gain on mic channel 0 to 50

$ i2cget -y 1 0x3c 0 #Get the upper byte of gain on mic channel 0
$ i2cget -y 1 0x3c 1 #Get the lower byte of gain on mic channel 0

$ i2cset -y 1 0x3c 16 1 #Set the gain on mic channel 8 to 256
$ i2cset -y 1 0x3c 15 0 #Set the gain on mic channel 8 to 256
```

6.5 ASRC Application

6.5.1 Overview

Warning: This example is based on the RTOS framework and drivers. This choice simplifies the example design, but it leads to high latency in the system. The main sources of latency are:

- Large block size used for ASRC processing: this is necessary to minimise latency associated with the intertile context and thread switching overhead.
- Large size of the buffer to which the ASRC output samples are written: a stable level (half full) must be reached before the start of streaming out over USB.
- RTOS task scheduling overhead between the tasks.
- bInterval of USB in the RTOS drivers is set to 4, i.e. one frame every 1 ms.
- Block based implementation of the USB and I^2S RTOS drivers.

The expected latencies for USB at 48 kHz are as follows:

- USB -> ASRC -> I^2S: from 8 ms at I^2S at 192 kHz to 22 ms at 44.1 kHz

- I^2S -> ASRC -> USB: from 13 ms at I^2S at 192 kHz to 19 ms at 44.1 kHz

For a proposed implementation with lower latency, please refer to the bare-metal examples below:

- AN02003: SPDIF/ADAT/I2S Slave Receive to I2S Slave Bridge with ASRC

This is the XCORE-VOICE Asynchronous Sampling Rate Converter (ASRC) example design.

The example system implements a stereo I^2S Slave and a stereo Adaptive UAC2.0 interface and exchanges data between the two interfaces. Since the two interfaces are operating in different clock domains, there is an ASRC block between them that converts from the input to the output sampling rate. There are two ASRC blocks, one each in the I^2S -> ASRC -> USB and USB -> ASRC -> I^2S path, as illustrated in the *ASRC example top level system diagram*. The diagram also shows the rate calculation path, which monitors and computes the instantaneous ratio between the ASRC input and output sampling rate. The rate ratio is used by the ASRC task to dynamically adapt filter coefficients using spline interpolation in its filtering stage.

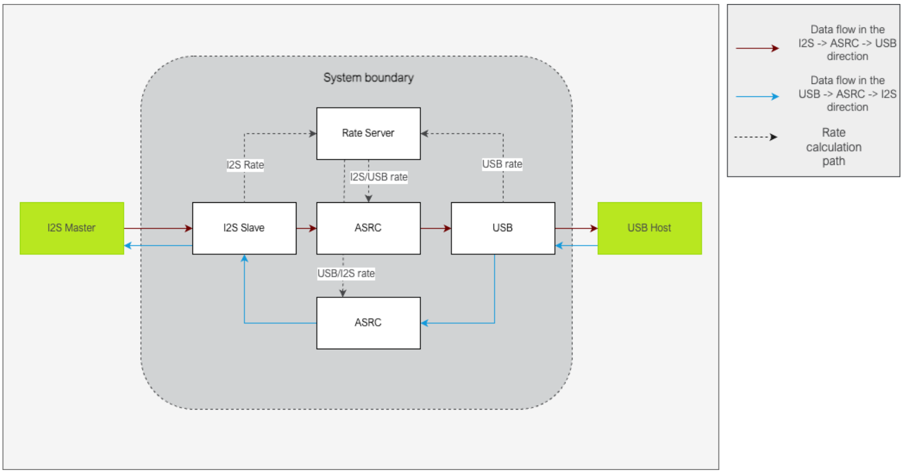


Fig. 10: ASRC example top level system diagram

The I^2S Slave interface is a stereo 32 bit interface supporting sampling rates between 44.1 kHz - 192 kHz.

The USB interface is a stereo, 32 bit, 48 kHz, High-Speed, USB Audio Class 2, Adaptive interface.

The ASRC algorithm implemented in the lib_src library is used for the ASRC processing. The ASRC processing is block based and works on a block size of 244 samples per channel in the I^2S -> ASRC -> USB path and 96 samples per channel in the USB -> ASRC -> I^2S path.

6.5.1.1 Supported Hardware

This example application is supported on the XK-VOICE-L71 board. In addition to the XK-VOICE-L71 board, it requires an XTAG4 to program and debug the device.

To demonstrate the audio exchange between the I^2S and USB interface, the XK-VOICE-L71 device needs to be connected to an I^2S Master device. To do this, connect the BCLK, MCLK, DOUT, DIN pins of the RASPBERRY PI HOST INTERFACE header (J4) on the XK-VOICE-L71 to the I^2S Master. The table *XK-VOICE-L71 RPI host interface header (J4) connections* lists the pins on the XK-VOICE-L71 RPI header and the signals on the I^2S Master that they need to be connected to.

Table 47: XK-VOICE-L71 RPI host interface header (J4) connections

XK-VOICE-L71 PI header pin	Signal to connect to on the I^2S Master board
12	BLCK output
35	LRCK output
38	I^2S Data input to the Master
40	I^2S Data output from the Master
One of the GND pins (6, 14, 20, 30, 34, 9, 25 or 39)	GND on the I^2S Master board

6.5.1.2 Obtaining the app files

Download the main repo and submodules using:

```
$ git clone --recurse git@github.com:xmos/sln_voice.git
$ cd sln_voice/
```

6.5.1.3 Building the app

First install and source the XTC version: 15.3.0 tools. For example with version 15.2.1, the output should be something like this:

```
$ xcc --version
xcc: Build 19-198606c, Oct-25-2022
XTC version: 15.2.1
Copyright (C) XMOS Limited 2008-2021. All Rights Reserved.
```

Linux or Mac To build for the first time, activate your python environment, run `cmake` to create the make files:

```
$ pip install -r requirements.txt
$ mkdir build
$ cd build
$ cmake --toolchain ../xmos_cmake_toolchain/xs3a.cmake  ..
$ make example_asrc_demo -j
```

Following initial `cmake` build, for subsequent builds, as long as new source files are not added, just type:

```
$ make example_asrc_demo -j
```

`cmake` needs to be rerun to discover any new source files added.

Windows It is recommended to use *Ninja* or *xmake* as the make system under Windows. *Ninja* has been observed to be faster than *xmake*, however *xmake* comes natively with XTC tools. This firmware has been tested with *Ninja* version v1.11.1.

To install Ninja, follow these steps:

- Download `ninja.exe` from here. This firmware has been tested with Ninja version v1.11.1.
- Ensure Ninja is on the command line path. It can be added to the path permanently by following the steps listed here. Alternatively, set the path in the current command

line session using something like `set PATH=%PATH%;C:\Users\xmos\utils\ninja`

To build for the first time, activate your python environment, run **cmake** to create the make files:

```
$ pip install -r requirements.txt
$ md build
$ cd build
$ cmake -G "Ninja" --toolchain  ..\xmos_cmake_toolchain\xs3a.cmake ..
$ ninja example_asrc_demo.xe
```

Following initial **cmake** build, for subsequent builds, as long as new source files are not added, just type:

```
$ ninja example_asrc_demo.xe
```

cmake needs to be rerun to discover any new source files added.

6.5.1.4 Running the app

To run the app, either xrun or xflash can be used. Connect the XK-VOICE-L71 board to the host and type the following to run with real-time debug output enabled:

```
$ xrun --xscope example_asrc_demo.xe
```

or to flash the application so that it always boots after a power cycle:

```
$ xflash example_asrc_demo.xe
```

6.5.1.5 Operation

When the example runs, the audio received by the device on the I^{2}S Slave interface at the I^{2}S interface sampling rate is sample rate converted using the ASRC to the USB sampling rate and streamed out from the device over the USB interface. Similarly, the audio streamed out by the USB host into the USB interface of the device is sample rate converted to the I^{2}S interface sampling rate and streamed out from the device over the I^{2}S Slave interface.

This example supports dynamic changes of the I^{2}S interface sampling frequency at run-time. It detects the I^{2}S sampling rate change and reconfigures the system for the new rate.

6.5.2 Software Architecture

The ASRC demo application is a two tile application developed to run on the XK-VOICE-L71 board running at a core frequency of 600 MHz.

It is a FreeRTOS based application where all the application blocks are implemented as FreeRTOS tasks.

Each tile has 5 bare metal cores dedicated to running RTOS tasks and since all processing is done within RTOS tasks, each core has 120 MHz of bandwidth available.

6.5.2.1 Task diagram The *ASRC example task diagram* shows the RTOS tasks and other components that make up the system.

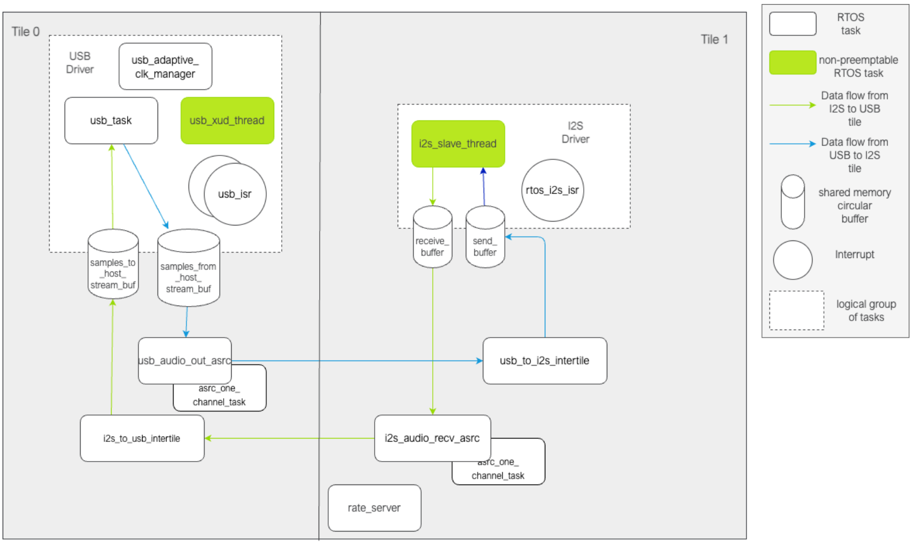


Fig. 11: ASRC example task diagram

The tasks can roughly be categorised as belonging to the USB driver, I²S driver or the application code categories. The actual ASRC processing happens in four tasks across the two tiles; the **usb_audio_out_asrc task**, **i2s_audio_recv_asrc task**, and two instances of **asrc_one_channel task**, one on each tile. This is described in more detail in the *Application components* section below.

Most of the tasks are involved in the ASRC processing data path, while a few are involved in monitoring the input and output data rates and computing the rate ratio, which is the ratio between the frequencies at the input and output of the ASRC tasks. The rate ratio is provided to the ASRC tasks every `asrc_process_frame()` call. Details about the rate ratio calculation are described in the *rate_server* section below.

6.5.2.2 USB Driver components This application presents a stereo, 48 kHz, 32 bit, high-speed, Adaptive UAC2.0 USB interface. It has two endpoints, Endpoint 0 for control and Endpoint 1 for bidirectional isochronous USB audio. The USB application level driver is TinyUSB based.

The **usb_xud_thread**, **usb_isr**, **usb_task** and **usb_adaptive_clk_manager** implement the USB driver. Together, these tasks handle the USB communication with the host and also monitor the average USB rate seen by the device. The average USB rate is used for calculating the rate ratios that are sent to the `asrc_process_frame()` function. This is described more in the *rate_server* section.

The **usb_xud_thread** runs `XUD_Main` which implements the USB HIL driver. It runs on a dedicated bare metal core so cannot be preempted by other RTOS tasks. It interfaces with the USB app level thread (**usb_task**) via shared memory and dedicated channels between the `XUD_Main` and each endpoint.

`XUD_Main` notifies the connected endpoint of a USB transfer completion through an interrupt on the respective channel. This interrupt is serviced by the **usb_isr** routine.

usb_task implements the app level USB driver functionality. The app level USB driver is based on TinyUSB which hooks into the application by means of callback functions. The **usb_isr** task is triggered by the interrupt and parses the data transferred from XUD and places it on a queue that the **usb_task** blocks on for further processing. For example, on completion of an EP1 OUT transfer, the transfer completion gets notified on the **usb_xud_thread -> usb_isr -> usb_task** path, and the **usb_task** calls the `tud_audio_rx_done_post_read_cb()` function to have the application process the data received from the host. On completion of an EP1 IN transfer, the transfer completion again follows the **usb_xud_thread -> usb_isr -> usb_task** path, and **usb_task** calls the `tud_audio_tx_done_pre_load_cb()` callback function to have the application load the EP1 IN data for the next transfer.

samples_to_host_stream_buf and **samples_from_host_stream_buf** are circular buffers shared between the application and the USB driver and allow for decoupling one from the other. The data frame received over USB from the host is written to the `samples_from_host_stream_buf` by the TinyUSB callback function `tud_audio_rx_done_post_read_cb()`, while the application reads `USB_TO_I2S_ASRC_BLOCK_LENGTH` samples of data out of it. Similarly, the application writes the ASRC output block of data to the `samples_to_host_stream_buf` while the TinyUSB callback function `tud_audio_tx_done_pre_load_cb()` reads from it to send one frame of data to the USB host.

usb_adaptive_clk_manager task is responsible for calculating the average USB rate as seen by the device. The average rate is calculated over a 16-second moving window. The averaging smooths out any jitter seen in the USB SOF timestamps that are used for calculating the rate.

6.5.2.3 I^{2}S Driver components

This application presents a stereo 32 bit, I^{2}S Slave interface that supports I^{2}S sampling rates of 44.1, 48, 88.2, 96, 176.4 and 192 kHz. The I^{2}S driver supports tracking dynamic sampling rate (SR) changes and recalculates the nominal sampling rate after detecting a SR change event. It also continuously monitors the timespan over which a fixed number of samples are received. This information is then used by the application for calculating the average I^{2}S rate seen by the device.

i2s_slave_thread, I^{2}S **send_buffer** and **receive_buffer** and **rtos_i2s_isr** make up the I^{2}S driver components.

i2s_slave_thread implements the I^{2}S HIL driver. The HIL level driver calls into the application callback functions for `i2s_init()`, `i2s_restart_check()`, `i2s_receive()` and `i2s_send()`. These functions, in addition to handling I^{2}S send and receive data, also detect sampling rate changes and gather information for tracking the average sampling rate.

I^{2}S **send_buffer** and **receive_buffer** are circular buffers shared between the driver and the application and contain data received over I^{2}S (`receive_buffer`) and data the application wants to send over I^{2}S (`send_buffer`). These buffers allow for decoupling the I^{2}S HIL driver from the ASRC application. The driver reads from and writes to these buffers at the I^{2}S sample rate while the application can read and write blocks of data to these buffers equal to the ASRC input or output block size.

The application calls `rtos_i2s_rx()` to read `I2S_TO_USB_ASRC_BLOCK_LENGTH` samples of data from the `receive_buffer`. The **i2s_slave_thread** independently calls `i2s_receive()` callback function to write a sample of data as it gets received over I^{2}S.

Similarly, the application calls `rtos_i2s_tx()` to write ASRC output size block of data into the `send_buffer`. Meanwhile, the driver independently calls the callback function `i2s_send()` to read a sample of data to send over the I^{2}S.

rtos_i2s_isr interrupt is used to ensure that the application calls to `rtos_i2s_rx()` and `rtos_i2s_tx()` block only on RTOS primitives when waiting for read data to be available or buffer space to be available when writing data.

6.5.2.4 Application components

usb_audio_out_asrc, **i2s_audio_recv_asrc**, **asrc_one_channel_task**, **usb_to_i2s_intertile**, **i2s_to_usb_intertile** and the **rate_server** tasks make up the non-driver components of the application.

usb_audio_out_asrc performs ASRC on data received from the USB host to the device. It waits to get notified by the TinyUSB callback function `tud_audio_rx_done_post_read_cb()` when there are one or more ASRC input blocks (96 USB samples) of data in the `samples_from_host_stream_buf`. It does ASRC processing of the first channel while coordinating with the **asrc_one_channel_task** for processing the second channel in parallel and sends the processed output to the other tile on the inter-tile context.

i2s_audio_recv_asrc performs ASRC on data received over the I^{2}S interface by the device. It blocks on the `rtos_i2s_rx()` function to receive one ASRC input block (244 I^{2}S samples) of data from I^{2}S and performs ASRC on one channel while coordinating with the **asrc_one_channel_task** for processing the second channel in parallel. It then sends the processed output to the other tile on the inter-tile context.

asrc_one_channel_task performs ASRC on a single channel of data. There is one of these on each tile. It waits on an RTOS message queue for an ASRC input block to be available, does ASRC processing on the block and posts the completion notification on another message queue.

usb_to_i2s_intertile task receives the ASRC output data generated by **usb_audio_out_asrc** over the inter-tile context onto the I^{2}S tile and writes it to the I^{2}S `send_buffer`. It has other rate-monitoring related responsibilities that are described in the *rate_server* section.

i2s_to_usb_intertile task receives the ASRC output data generated by **i2s_audio_recv_asrc** over the inter-tile context onto the USB tile and writes it to the USB `samples_to_host_stream_buf`. It has other rate-monitoring related responsibilities that are described in the *rate_server* section.

The *I2S -> ASRC -> USB data path* diagram shows the application tasks involved in the I^{2}S -> ASRC -> USB path processing and their interaction with each other.

The *USB -> ASRC -> I2S data path* diagram shows the application tasks involved in the USB -> ASRC -> I^{2}S path processing and their interaction with each other.

rate_server The ASRC `process_frame` API requires the caller to calculate and send the instantaneous ratio between the ASRC input and output rate. The **rate_server** is responsible for calculating these rate ratios for both USB -> ASRC -> I^{2}S and I^{2}S -> ASRC -> USB directions.

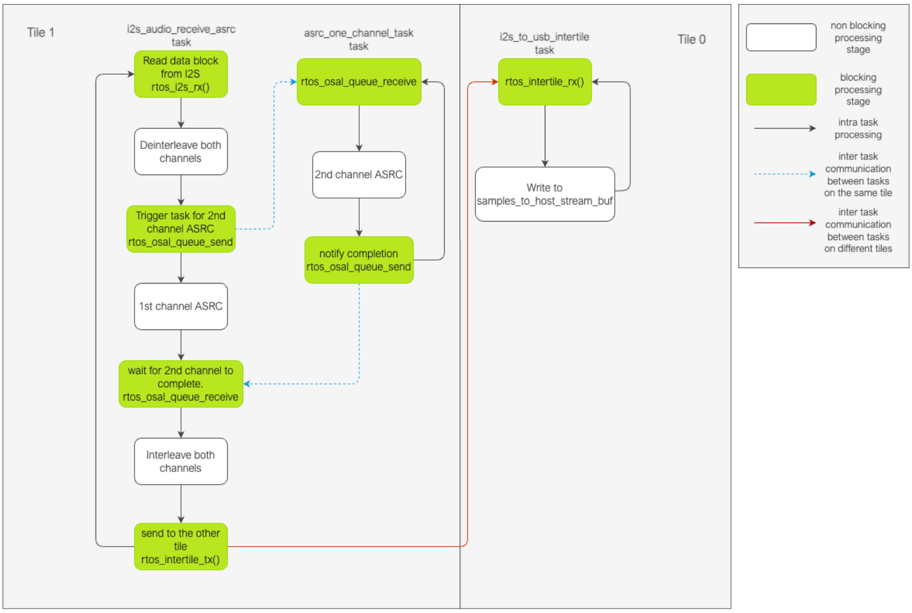


Fig. 12: I^2S -> ASRC -> USB data path

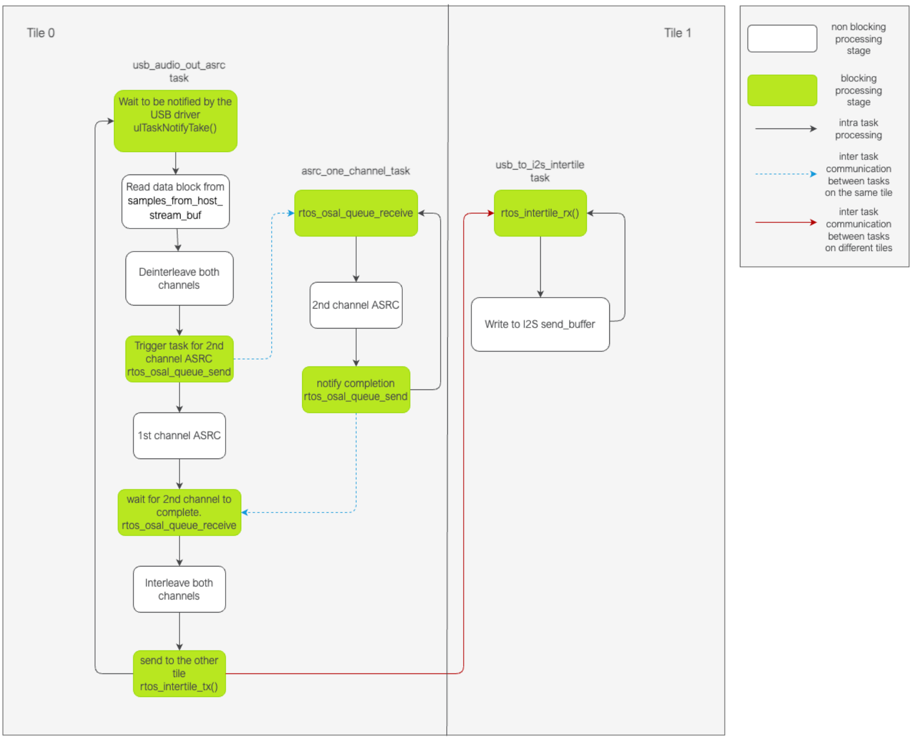


Fig. 13: USB -> ASRC -> I^2S data path

Additionally, the application also monitors the average buffer fill levels of the buffers holding ASRC output to prevent any overflows or underflows of the respective buffer. A gradual drift in the buffer fill level indicates that the rate ratio is being under or over calculated by the **rate_server**. This could happen either due to jitter in the actual rates or precision limitations when calculating the rates.

The average fill level of the buffer is monitored and a closed-loop error correction factor is calculated to keep the buffer level at an expected stable level. The error estimated based on the buffer fill level is used to compute the estimated rate ratio from the initial rate ratio. This estimated rate ratio is then sent to the ASRC `process_frame()` API.

```
estimated_rate_ratio = initial_rate_ratio + buffer_based_correction_factor
```

The **rate_server** runs on the I^{2}S tile (tile 1) and is periodically triggered from the USB tile (tile 0) by the **usb_to_i2s_intertile** task. The **rate_server** is triggered once after every 16 frames are written to the `samples_to_host_stream_buf`.

The following information is needed for calculating the rate ratios:

1. The average I^{2}S rate
2. The average USB rate
3. An error factor computed based on the USB `samples_to_host_stream_buf` fill level
4. An error factor computed based on the I^{2}S `send buffer` fill level
5. A USB `mic_interface_open` flag indicating if the USB host is streaming out from the device, since the rate ratio in the I^{2}S -> ASRC -> USB direction is calculated only when the host is reading data from the device
6. A USB `spkr_interface_open` flag indicating if the USB host is streaming into the device, since the rate ratio in the USB -> ASRC -> I^{2}S direction is calculated only when the host is sending data to the device

Of the above, the USB related information (2, 3, 5 and 6 above) is available on the USB tile. When triggering the **rate_server**, the **i2s_to_usb_intertile** task gets this information, either calculating it or getting it through shared memory from other USB tasks on the same tile, and sends it to the **rate_server** over the inter-tile context using the structure below.

```
typedef struct
{
    int64_t buffer_based_correction;
    float_s32_t usb_data_rate;
    bool mic_itf_open;
    bool spkr_itf_open;
}usb_rate_info_t;
```

The I^{2}S related information (1 and 4 above) is calculated in the **rate_server** itself with information available for calculating these available through shared memory from other tasks on this tile.

After calculating the rates, the **rate_server** sends the rate ratio for the USB -> ASRC -> I^{2}S side to the **usb_to_i2s_intertile** task over the inter-tile context and it is made available to the **usb_audio_out_asrc** task through shared memory. The I^{2}S -> ASRC -> USB side rate ratio is also made available to the **i2s_audio_recv_asrc** task through shared memory since it runs on the same tile as the rate server.

The *Rate calculation code flow* diagram shows the code flow during the rate ratio calculation process, focussing on the **usb_to_intertile** task that triggers the **rate_server** and the **rate_server** task where the rate ratios are calculated.

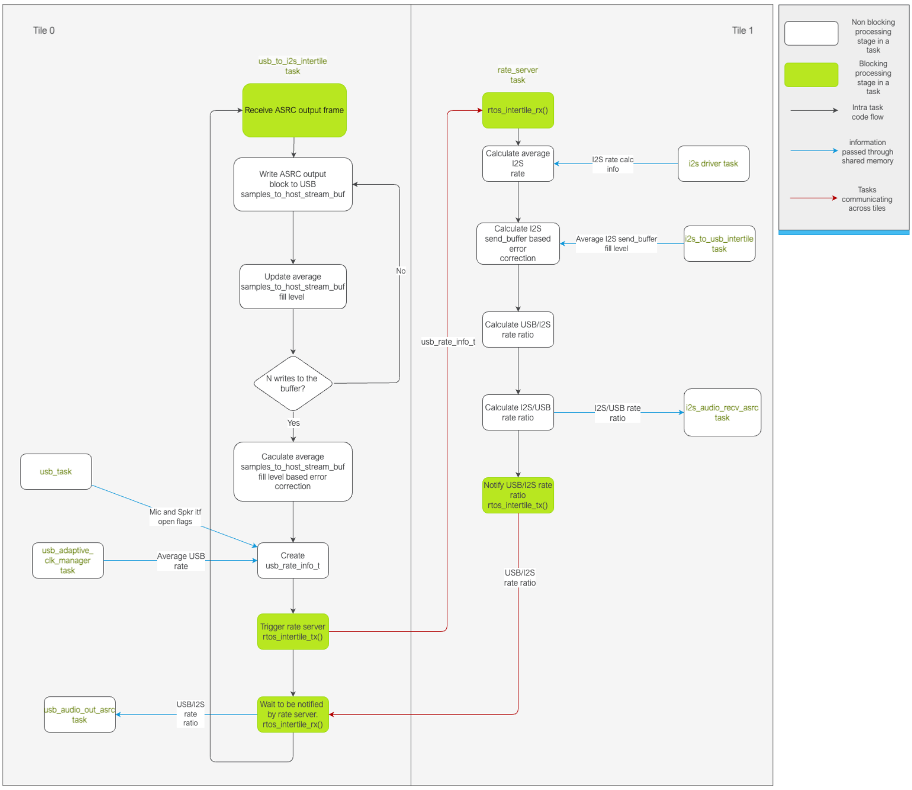


Fig. 14: Rate calculation code flow

6.5.2.5 Handling I^{2}S sampling rate change events

The I^{2}S driver monitors the I^{2}S nominal rate and provides this information to the application. When an I^{2}S sampling rate change happens:

- The ASRC instances on both tiles are re-initialised with the new sampling rate.
- The buffers that are used for buffer-fill-level based correction are reset. Streaming out of them is paused while zeroes are sent out over both USB and I^{2}S. Once the buffers fill to a stable level, streaming out from them resumes.
- The average buffer level calculation state is reset and the average buffer level calculation starts afresh. New stable buffer levels are also calculated and the buffer levels are now corrected against these new stable averages.

Note that the device starts with the nominal I^{2}S sampling rate set to zero. Device startup therefore follows the same path as an I^{2}S sampling rate change where the sampling rate goes from zero to first detected nominal sampling rate. Everything described above therefore also applies to the device startup behaviour.

6.5.2.6 Handling USB speaker interface close -> open events

When the USB host stops streaming to the device and then starts again, this event is detected through calls to the `tud_audio_set_itf_close_EP_cb` and `tud_audio_set_itf_cb` functions. The ASRC output buffer in the USB -> ASRC -> I^2S path (I^2S `send_buffer`) is reset. Zeroes are then sent over I^2S until the buffer fills to a stable level, when we resume streaming out of this buffer to send samples over I^2S. The average buffer calculation state for the I^2S `send_buffer` is also reset and a new stable average is calculated against which the average buffer levels are corrected.

6.5.2.7 Handling USB mic interface close -> open events

If the USB host stops streaming from the device and then starts again, this event is detected through calls to the `tud_audio_set_itf_close_EP_cb` and `tud_audio_set_itf_cb` functions. The ASRC output buffer in the I^2S -> ASRC -> USB is reset (USB `samples_to_host_stream_buf`). Zeroes are streamed to the host until the buffer fills to a stable level, when we resume streaming out of this buffer to send samples over USB. The average buffer calculation state for the USB `samples_to_host_stream_buf` is also reset and a new stable average is calculated against which the average buffer levels are corrected.

6.5.3 Resource Usage

6.5.3.1 Memory

Out of the 524288 bytes of memory available per tile, this application uses approximately 262000 bytes of memory on Tile 0 and 208000 bytes of memory on Tile 1.

6.5.3.2 Chanends

This application uses 19 chanends on the USB tile (tile 0) and 11 chanends on the I^2S tile (tile 1)

The chanend use for both tiles is described in the *Tile 0 chanend usage* and *Tile 1 chanend usage* tables.

Tile 0

Table 48: Tile 0 chanend usage

Resource	Chanends used
RTOS scheduler	5 (one per bare-metal core dedicated to RTOS)
RTOS USB driver	10 (2 per endpoint, per direction. 2 for SOF input)
Intertile contexts	3
xscope	1

Tile 1

Table 49: Tile 1 chanend usage

Resource	Chanends used
RTOS scheduler	5 (one per bare-metal core dedicated to RTOS)
RTOS I^2S driver	2
Intertile contexts	3
xscope	1

Intertile contexts

The application uses 3 intertile contexts for cross tile communication.

- A dedicated intertile context for sending ASRC output data from the I²S tile to the USB tile.
- A dedicated intertile context for sending ASRC output data from the USB tile to the I²S tile.
- The intertile context for all other cross tile communication.

6.5.3.3 CPU Profiling the CPU usage for this application using an RTOS friendly profiling tool is still TBD. However, profiling some application tasks has taken place. These numbers along with some already existing profiling numbers for the drivers are listed in the *Tile 0 tasks MIPS* and *Tile 1 tasks MIPS* tables. Each tile has 5 bare-metal cores being used for running RTOS tasks so each core has a fixed bandwidth of 120 MHz available.

Tile 0

Table 50: Tile 0 tasks MIPS

RTOS Task	MIPS
XUD	120 (from *CPU Requirements (@ 600 MHz)*)
ASRC in the USB -> ASRC -> I²S path for the worst case of 48 kHz to 192 kHz upsampling	85
usb_task	24
i2s_to_usb_intertile	14

Tile 1

Table 51: Tile 1 tasks MIPS

RTOS Task	MIPS
I²S Slave	96 (from *CPU Requirements (@ 600 MHz)*)
ASRC in the I²S -> ASRC -> USB path for the worst case of 192 kHz to 48 kHz downsampling	75
usb_to_i2s_intertile	0.7
rate_server	19

7 Speech Recognition Ports

Ports of the Sensory and Cyberon speech recognition libraries are provided.

Table 52: Speech Recognition Ports

Filename/Directory	Description
modules/asr directory	include folder for ASR modules and ports
module/asr/sensory directory	contains the Sensory library and associated port code
module/asr/Cyberon directory	contains the Cyberon library and associated port code
modules/asr/CmakeLists.txt	CMakeLists file for adding ASR port targets

8 Memory and CPU Requirements

8.1 Memory

The table below lists the approximate memory requirements for the larger software components. All memory use estimates in the table below are based on the default configuration for the feature. Alternate configurations will require more or less memory. The estimates are provided as guideline to assist application developers judge the memory cost of extending the application or benefit of removing an existing feature. It can be assumed that the memory requirement of components not listed in the table below are under 5 kB.

Table 53: Memory Requirements

Component	Memory Use (kB)
Stereo Adaptive Echo Canceler (AEC)	275
Sensory Speech Recognition Engine	180
Cyberon Speech Recognition Engine	125
Interference Canceler (IC) + Voice To Noise Ratio Estimator (VNR)	130
USB	20
Noise Suppressor (NS)	15
Adaptive Gain Control (AGC)	11

8.2 CPU

The table below lists the approximate CPU requirements in MIPS for the larger software components. All CPU use estimates in the table below are based on the default configuration for the feature. Alternate configurations will require more or less MIPS. The estimates are provided as guideline to assist application developers judge the MIP cost of extending the application or benefits of removing an existing feature. It can be assumed that the memory requirement of components not listed in the table below are under 1%.

The following formula was used to convert CPU% to MIPS:

MIPS = (CPU% / 100%) * (600 MHz / 5 cores)

Table 54: CPU Requirements (@ 600 MHz)

Component	CPU Use (%)	MIPS Use
USB XUD	100	120
I^2S (slave mode)	80	96
Stereo Adaptive Echo Canceler (AEC)	80	96
Sensory Speech Recognition Engine	80	96
Cyberon Speech Recognition Engine	72	87
Interference Canceler (IC) + Voice To Noise Ratio Estimator (VNR)	25	30
Noise Suppressor (NS)	10	12
Adaptive Gain Control (AGC)	5	6

9 How-Tos

This section includes instructions on anticipated or common software modifications.

9.1 Changing the input and output sample rate

In the example design `app_conf.h` file, change `appconfAUDIO_PIPELINE_SAMPLE_RATE` to either 16000 or 48000.

9.2 I^2S AEC reference input audio & USB processed audio output

The FFVA example design includes 2 basic configurations; INT and UA. The INT configuration is setup with I^2S for input and output audio. The UA configuration is setup with USB for input and output audio. This HOWTO explains how to modify the FFVA example design for I^2S input audio and USB output audio.

In the `ffva_ua.cmake` file, changing the `appconfAEC_REF_DEFAULT` to `appconfAEC_REF_I2S` will result in the expected input frames.

```
set(FFVA_UA_COMPILE_DEFINITIONS
    ${APP_COMPILE_DEFINITIONS}
    appconfI2S_ENABLED=1
    appconfUSB_ENABLED=1
    appconfAEC_REF_DEFAULT=appconfAEC_REF_I2S

    appconfI2S_MODE=appconfI2S_MODE_MASTER
    MIC_ARRAY_CONFIG_MCLK_FREQ=24576000
)
```

For integrating with I^2S there are a few other differences from the default UA configuration. When integrating with an external Raspberry Pi **BCLK** and **LRCLK**, you will want the following `FFVA_UA_COMPILE_DEFINITIONS`:

```
set(FFVA_UA_COMPILE_DEFINITIONS
    ${APP_COMPILE_DEFINITIONS}
    appconfI2S_ENABLED=1
    appconfUSB_ENABLED=1
    appconfAEC_REF_DEFAULT=appconfAEC_REF_I2S

    appconfI2S_MODE=appconfI2S_MODE_SLAVE
    appconfEXTERNAL_MCLK=0
    appconfI2S_AUDIO_SAMPLE_RATE=48000
```

(continues on next page)

(continued from previous page)

```
    MIC_ARRAY_CONFIG_MCLK_FREQ=12288000
)
```

appconfI2S_AUDIO_SAMPLE_RATE can also be 16000. Only 48k and 16k conversions is supported in FFVA.

The default FFVA INT device doesn't require an external MCLK, but this setting can be changed by setting appconfEXTERNAL_MCLK=1. In this case the FFVA example application will sit at initialization until it can lock on to that clock source, so it MUST be active during boot.

Since the FFVA example application is not receiving reference audio through USB in this configuration, USB adaptive mode will not adapt to the input. By default, FFVA will output the configured nominal rate.

If you enable appconfAEC_REF_DEFAULT=appconfAEC_REF_I2S and appconfI2S_MODE=appconfI2S_MODE_MASTER. You need to invert I2S_DATA_IN and I2S_MIC_DATA in the bsp_config/XK_VOICE_L71/XK_VOICE_L71.xn file to have the reference audio play properly.

Lastly, with I^2S enabled the DAC is always initialized by the FFVA example application. If FFVA cannot be the I^2C host then it is up to the host to initialize the DAC, like in the AVS demo.

10 Frequently Asked Questions

10.1 CMake hides XTC Tools commands

If you want to customize the XTC Tools commands like xflash and xrun, you can see what commands CMake is running by adding VERBOSE=1 to your build command line. For example:

```
make run_my_target VERBOSE=1
```

10.2 fatfs_mkimage: not found

This issue occurs when the fatfs_mkimage host utility cannot be found. The most common cause for these issues are an incomplete installation of XCORE-VOICE.

Ensure that the host applications build and install has been completed. Verify that the fatfs_mkimage binary is installed to a location on PATH, or that the default application installation folder is added to PATH.

10.3 FFD pdm_rx_isr() Crash

One potential issue with the low power FFD application is a crash after adding new code:

```
xrun: Program received signal ET_ECALL, Application exception.
    [Switching to tile[1] core[1]]
    0x0008a182 in pdm_rx_isr ()
```

This generally occurs when there is not enough processing time available on tile 1, or when interrupts were disabled for too long, causing the mic array driver to fail to meet timing. To resolve reduce the processing time, minimize context switching and other actions that require kernel locks, and/or increase the tile 1 core clock frequency.

10.4 Debugging low-power

The clock dividers are set high to minimize core power consumption. This can make debugging a challenge or impossible. Even adding a simple `printf` can cause critical timing to be missed. In order to debug with the low-power features enabled, temporarily modify the clock dividers in `app_conf.h`.

```
#define appconfLOW_POWER_SWITCH_CLK_DIV        1   // Resulting clock freq 600MHz.
#define appconfLOW_POWER_OTHER_TILE_CLK_DIV    1   // Resulting clock freq 600MHz.
#define appconfLOW_POWER_CONTROL_TILE_CLK_DIV  1   // Resulting clock freq 600MHz.
```

10.5 xcc2clang.exe: error: no such file or directory

Those strange characters at the beginning of the path are known as a byte-order mark (BOM). CMake adds them to the beginning of the response files it generates during the configure step. Why does it add them? Because the MSVC compiler toolchain requires them. However, some compiler toolchains, like *gcc* and *xcc*, do not ignore the BOM. Why did CMake think the compiler toolchain was MSVC and not the XTC toolchain? Because of a bug in which certain versions of CMake and certain versions of Visual Studio do not play nice together. The good news is that this appears to have been addressed in CMake version 3.22.3. Update to CMake version 3.22.2 or newer.

11 Licenses

11.1 XMOS

All original source code is licensed under the XMOS License.

11.2 Third-Party

Additional third party code is included under the following copyrights and licenses:

Table 55: Third Party Module Copyrights & Licenses

Module	Copyright & License
dr_wav	Copyright (C) 2022 David Reid, licensed under a public domain license
FatFS	Copyright (C) 2017 ChaN, licensed under a BSD-style license
FreeRTOS	Copyright (c) 2017 Amazon.com, Inc., licensed under the MIT License
Sensory TrulyHandsfree™	The Sensory TrulyHandsfree™ speech recognition library is *Copyright (C) 1995-2022 Sensory Inc.* and is provided as an expiring development license. Commercial licensing is granted by Sensory Inc.
Cyberon DSpotter™	For any licensing questions about Cyberon DSpotter™ speech recognition library please contact Cyberon Corporation.
TinyUSB	Copyright (c) 2018 hathach (tinyusb.org), licensed under the MIT license