XCORE-VOICE SOLUTION - Programming Guide

Release: 1.0.0

Publication Date: 2023/03/20

XMOS

Table of Contents

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 design is currently based on FreeRTOS, leveraging the flexibility of the xcore.ai platform and providing designers with a familiar environment to customize and develop products.

XCORE-VOICE example designs provide 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.

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^2S 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 Wanson 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
- 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 other transport

Power Consumption

- Typical power consumption 300-350mW
- Low power modes down to 55mW

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

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 15.1.4 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
```

5 Prerequisites

It is recommended that you download and install the latest release of the XTC Tools. XTC Tools 15.1.4 or newer are required for building, running, flashing and debugging the example applications.

CMake 3.21 or newer is 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.

XCORE-VOICE host build 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 which requires `libusb v1.0`. `libusb` requires the installation of a driver for use on a Windows host. Driver installation should be done using a third-party installation tool like Zadig.

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 low-power far-field voice local command (FFD) example design with Wanson speech recognition and local dictionary.

While inactive, low-power mode uses a fraction of energy otherwise required by normal operations while awaiting and processing speech.

When a wake-up phrase is followed by an command phrase, the application will output an audio response and a discrete message over I^2C and UART.

This software is an evaluation version only. It includes a mechanism that limits the maximum number of recognitions to 50. You can reset the counter to 0 by restarting or rebooting the application. The application can be rebooted by power cycling or pressing the SW2 button.

Note: Due to the hardware design, SW2 is only functional when in full-power operation.

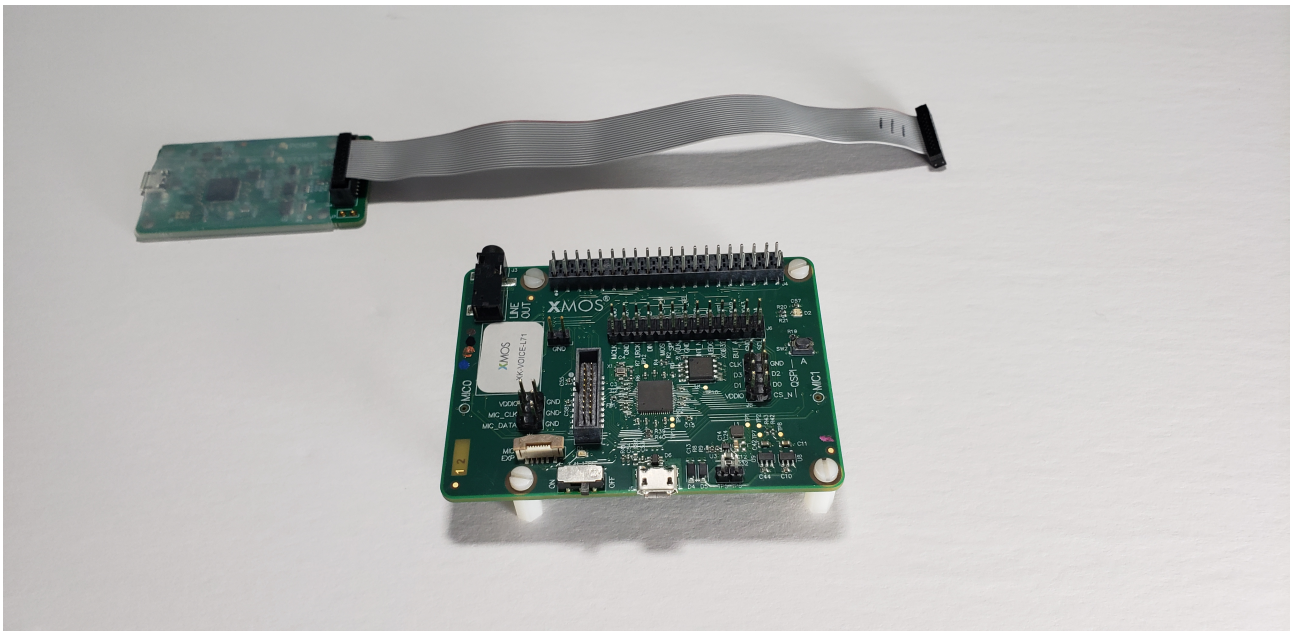
More information on the Wanson speech recognition library can be found here: *Wanson Speech Recognition*

6.1.2 Supported Hardware

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

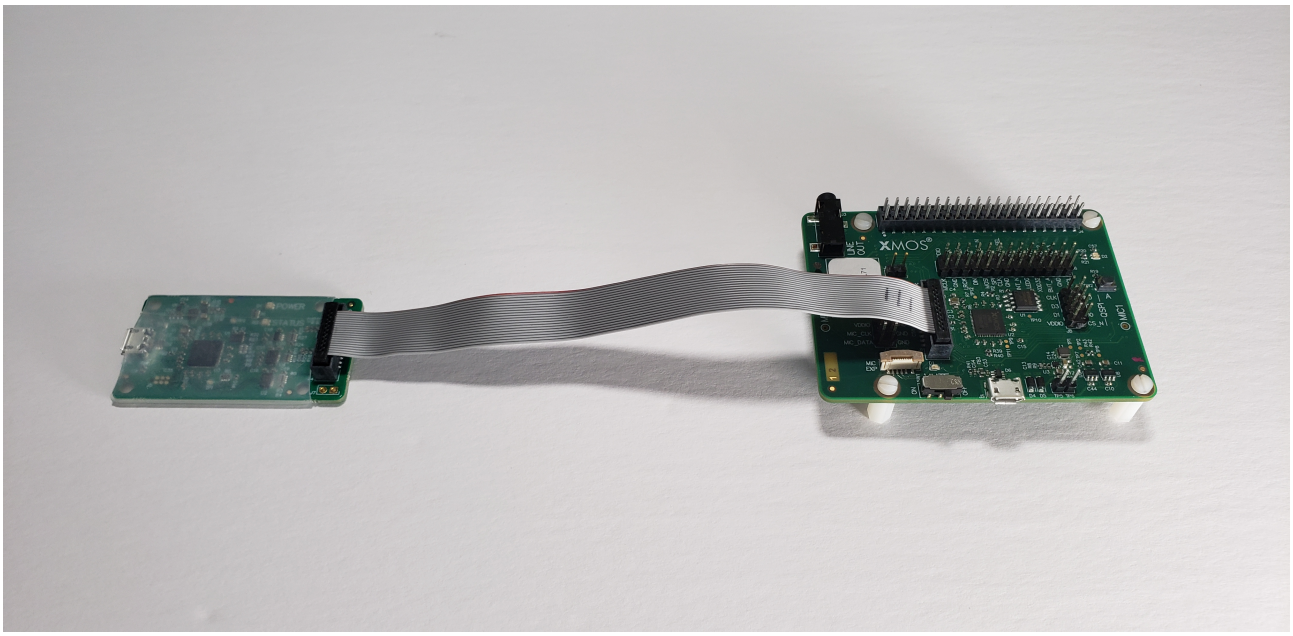
6.1.2.1 Setting up the Hardware

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

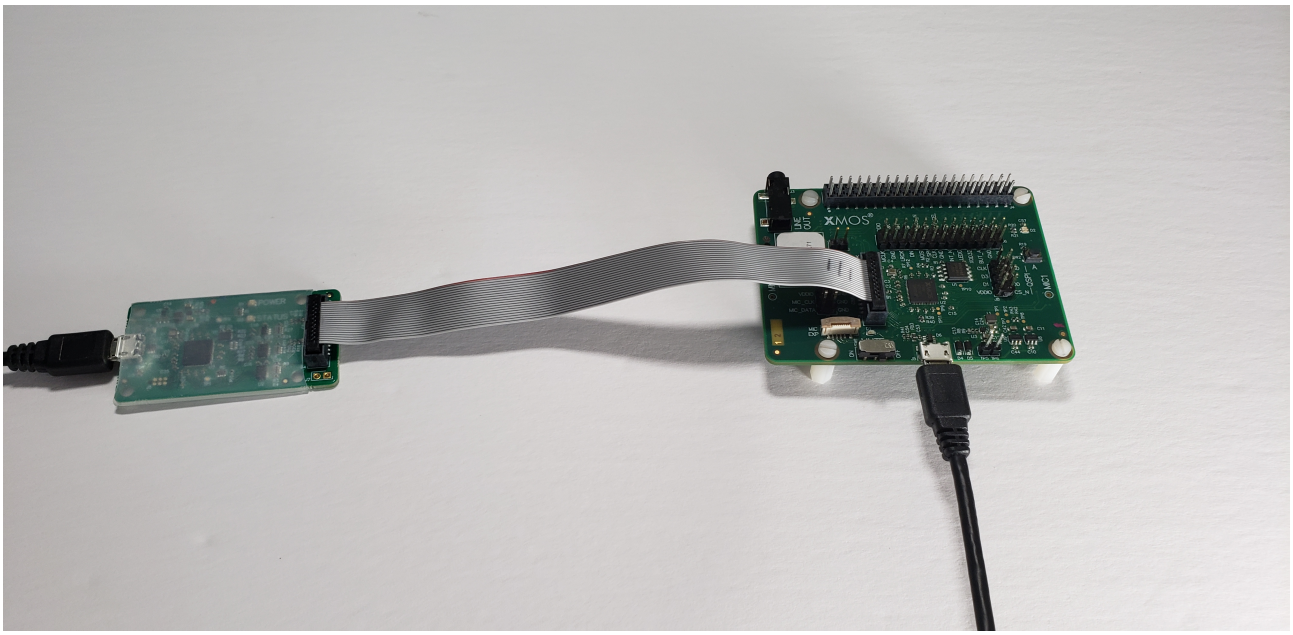


xTAG The xTAG is used to program and debug the device

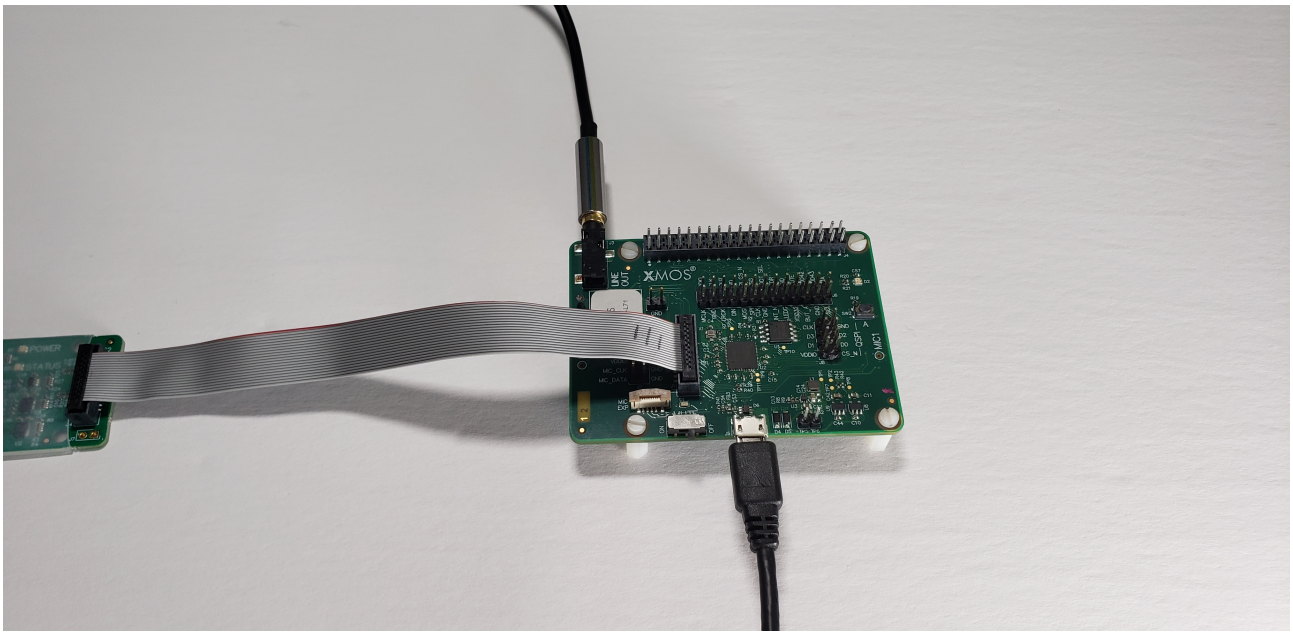
Connect the xTAG to the debug header, as shown below.



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 located here.

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

Table 6.1: FFD Compile Options

Compile Option	Description	Default Value
appconfINTENT_ENABLED	Enables/disables the intent engine, primarily for debug.	1
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_I2C_OUTPUT_ENABLED	Enables/disables the I^2C intent message	1
appconfUART_BAUD_RATE	Sets the baud rate for the UART tx intent interface	9600
appconfINTENT_I2C_OUTPUT_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
appconfLOW_POWER_ENABLED	Enables/disables low power feature	1
appconfLOW_POWER_SWITCH_CLK_DIV_ENABLE	Enables/disables low power feature adjusting the switch clock	1
appconfLOW_POWER_SWITCH_CLK_DIV	Sets the low power mode switch clock divider value	30
appconfLOW_POWER_OTHER_TILE_CLK_DIV	Sets the low power mode tile core clock divider value for the keyword engine tile	600
appconfLOW_POWER_CONTROL_TILE_CLK_DIV	Sets the low power mode tile core clock divider value for the audio pipeline and voice activity detection tile	2
appconfPOWER_FULL_HOLD_DURATION	Sets the minimum amount of time to expect a wakeword before requesting to be set back into low power	1000
appconfAUDIO_PIPELINE_BUFFER_ENABLED	Enables/disables a ring buffer to hold pre-trigger audio frames when in low power	1
appconfAUDIO_PIPELINE_BUFFER_NUM_FRAMES	Sets the number of audio frames held in the low power audio frame buffer	32
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

6.1.4 Deploying the Firmware with Linux or macOS

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

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

Run the following commands in the root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffd
```

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
```

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:

```
make run_example_ffd
```

6.1.4.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
make debug_example_ffd
```

6.1.5 Deploying the Firmware with Native Windows

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

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.

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 "NMake Makefiles" -B build_host
cd build_host
nmake install
```

The host applications will be install 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

Run the following commands in the root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -D CMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
nmake example_ffd
```

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:

```
nmake flash_app_example_ffd
```

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:

```
nmake run_example_ffd
```

6.1.5.4 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
nmake debug_example_ffd
```

6.1.6 Modifying the Software

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.

UART

Table 6.2: UART Connections

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

I^2C

Table 6.3: I^2C Connections

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

GPIO

Table 6.4: 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.

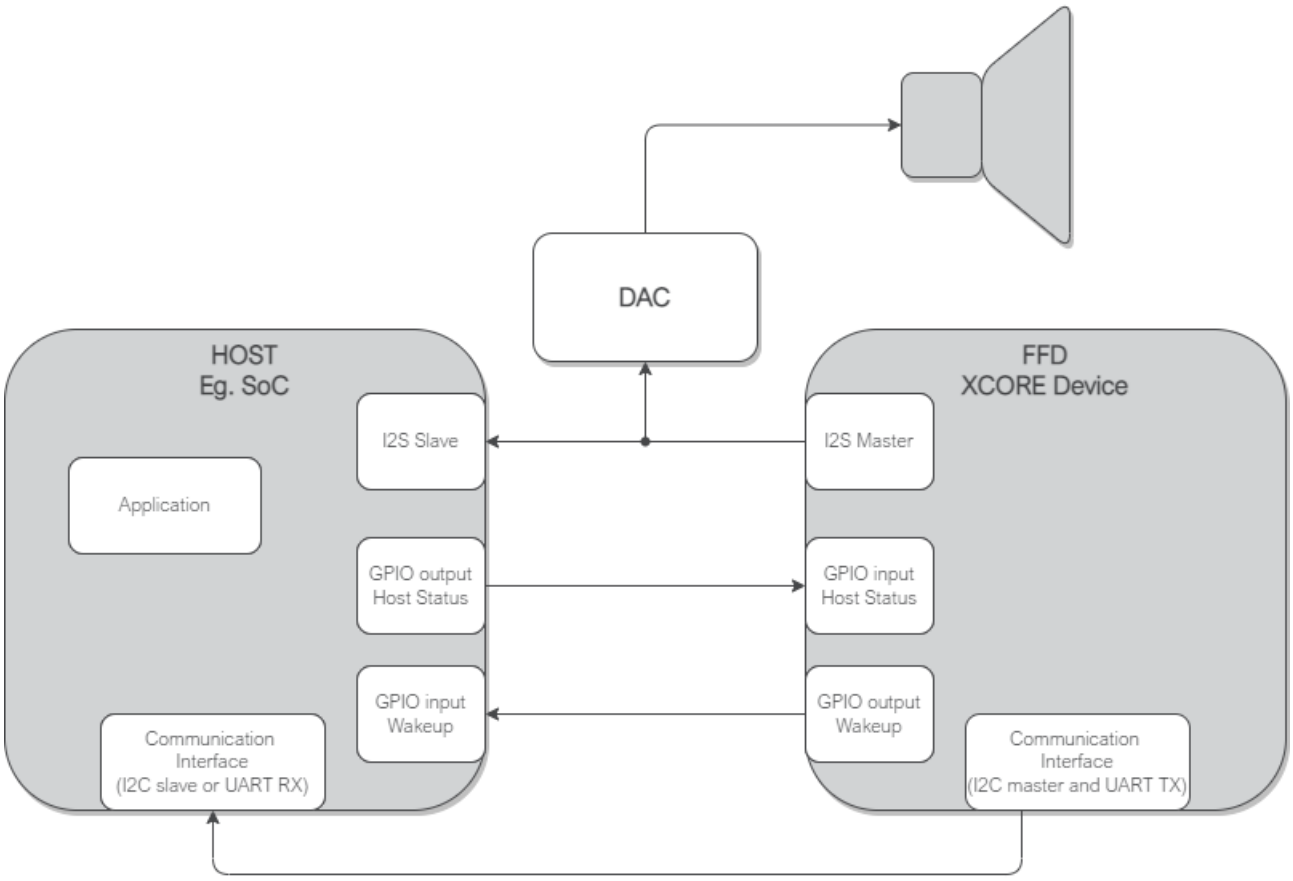

DAC
HOST
Eg. SoC
Application
I2S Slave
GPIO output
Host Status
GPIO input
Wakeup
Communication
Interface
(I2C slave or UART RX)
FFD
XCORE Device
I2S Master
GPIO input
Host Status
GPIO output
Wakeup
Communication
Interface
(I2C master and UART TX)

Table 6.5: 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, while in *POWER_STATE_FULL*, varies from 100-141 mW. This will vary based on component tolerances and any user added code and/or user added compile options.

By default, the application will startup using a system frequency of 600 MHz which will consume around 141 mW. After startup, *tile[1]* clock divider is enabled and set to 3 bringing the tile's frequency down to 300 MHz, where it will consume around 114 mW. Tile frequencies lower than this may lead to application instability. When the application enters *POWER_STATE_LOW*, the *tile[0]* clock frequency will be divided by 600 and the switch clock frequency by 30 bringing the frequencies to 1 MHz and 20 MHz, respectively. This low power state consumes around 55 mW.

Table 6.6: FFD Resources

Resource	Tile 0	Tile 1
Unused CPU Time (600 MHz)	83%	27%
Total Memory Free	192k	173k
Runtime Heap Memory Free	38k	42k

Table 6.7: FFD Power Usage

Power State	Power (mW)
Low Power	55
Full Power	114

The description of the software is split up by folder:

Table 6.8: FFD Software Description

Folder	Description
asr	ASR engine ports
bsp_config	Board support configuration setting up software based IO peripherals
ext	Application extensions
filesystem_support	Filesystem contents for application
src	Main application
src/intent_engine	Intent engine integration
src/intent_handler	Intent engine output integration
src/power	Low power state and control

asr This folder contains ASR ports.

Table 6.9: FFD ASR

Filename/Directory	Description
api directory	include folder for ASR modules
port directory	contains ports for supported ASR engines
port/wanson directory	contains the Wanson engine and associated port code
asr.cmake	cmake for adding ASR targets

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 6.10: 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

ext This folder contains FFD application debug and experimental extensions.

Table 6.11: FFD ext

Filename/Directory	Description
src directory	custom code for USB output and debug
ffd_dev.cmake	cmake for declaring FFD experimental configs
ffd_ext.cmake	cmake for declaring FFD extensions
ffd_usb_audio_testing.cmake	cmake for declaring FFD usb debug extension

filesystem_support This folder contains filesystem contents for the FFD application.

Table 6.12: 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

src This folder contains the core application source.

Table 6.13: FFD src

Filename/Directory	Description
audio_pipeline directory	contains example XMOS audio pipeline
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 state and control 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 6.1: 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.

This function is weak so the application can override it if desired.

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.

This function is weak so the application can override it if desired.

In FFD, the output is sent to the intent engine. If *appconfLOW_POWER_ENABLED* is set true, then the output will be dropped if the power state is not *POWER_STATE_FULL*. In certain conditions and environments, this behavior may cause the wake word to be missed. Further adjustments to the application configuration settings related to the VNR low power thresholds may mitigate such issues. See *src/power*.

Main The major components of main are:

Listing 6.2: 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 6.3: 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.

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

Table 6.14: 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 two API functions:

Listing 6.4: Intent Engine API (intent_engine.h)

```
int32_t intent_engine_create(uint32_t priority, void *args);
int32_t intent_engine_sample_push(int32_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.

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

Listing 6.5: intent_engine_create snippet (intent_engine_io.c)

```
#if ASR_TILE_NO == AUDIO_PIPELINE_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_sample_push This function has the role of sending the ASR output channel from the audio pipeline to the intent engine.

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

Listing 6.6: intent_engine_create snippet (intent_engine_io.c)

```
#if appconfINTENT_ENABLED && ON_TILE(AUDIO_PIPELINE_TILE_NO)
#if ASR_TILE_NO == AUDIO_PIPELINE_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 inter-tile 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 Several supporting helper functions to support the low power and audio playback features that are unique the the default FFD application. These include:

- intent_engine_keyword_queue_count
- intent_engine_keyword_queue_complete
- intent_engine_stream_buf_reset
- intent_engine_play_response
- intent_engine_low_power_ready
- intent_engine_low_power_reset
- intent_engine_full_power_request
- intent_engine_low_power_accept

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

Table 6.15: FFD 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 6.7: 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 Wanson model, the application provides a FreeRTOS Queue object. This handler is on the same tile as the Wanson 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 modules for lower power control and state reporting in the FFD application.

Configuration Notes The application monitors the VNR and produces low power events based on:

- *appconfPOWER_VNR_THRESHOLD*
- *appconfPOWER_LOW_ENERGY_THRESHOLD*
- *appconfPOWER_HIGH_ENERGY_THRESHOLD*
- *appconfPOWER_FULL_HOLD_DURATION*

The first three configuration options above determine when to begin transitioning, or continue to hold, the device in *POWER_STATE_FULL*. The last configuration option above determines the minimum period of time to device is allowed to wait for a wake word before requesting to transition into *POWER_STATE_LOW*. Each time *POWER_STATE_FULL* is set by the audio pipeline tile, the timer that is configured for a period of *appconf-POWER_FULL_HOLD_DURATION* milliseconds is reset, preventing any requests to *POWER_STATE_LOW* to be aborted.

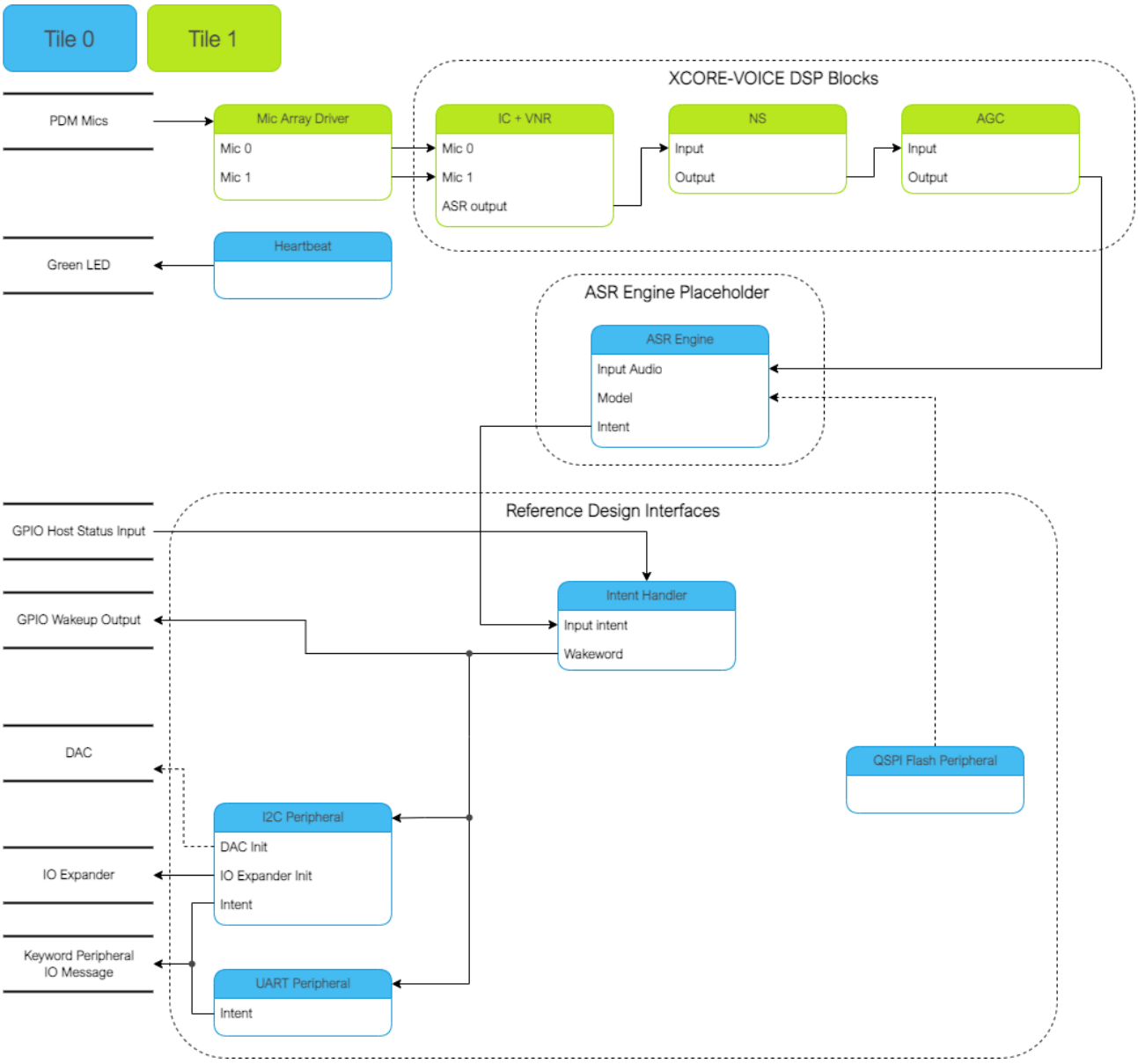
appconfLOW_POWER_ENABLED enables/disables use of this low power functionality.

When *appconfLOW_POWER_SWITCH_CLK_DIV_ENABLE* is enabled, *appconfLOW_POWER_SWITCH_CLK_DIV* should be set appropriately. Clock divider values that result in frequencies greater than or equal to 20MHz have been observed to work.

Values for *appconfLOW_POWER_CONTROL_TILE_CLK_DIV* that result in frequencies greater than or equal to 300MHz have been observed to work.

6.1.6.4 Software Modifications

Overview 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 6.8: 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 6.9: 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 6.10: 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 6.11: 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 *␣
```

(continues on next page)

(continued from previous page)

```
↪sizeof(int32_t));
}
```

Runtime Initialization Replace:

Listing 6.12: XMOS NS (audio_pipeline.c)

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

With:

Listing 6.13: Foo (audio_pipeline.c)

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

Audio Pipeline Setup Replace:

Listing 6.14: 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 6.15: 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 6.16: 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(int32_t *buf, size_t frames);
```

Refer to the existing Wanson 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:

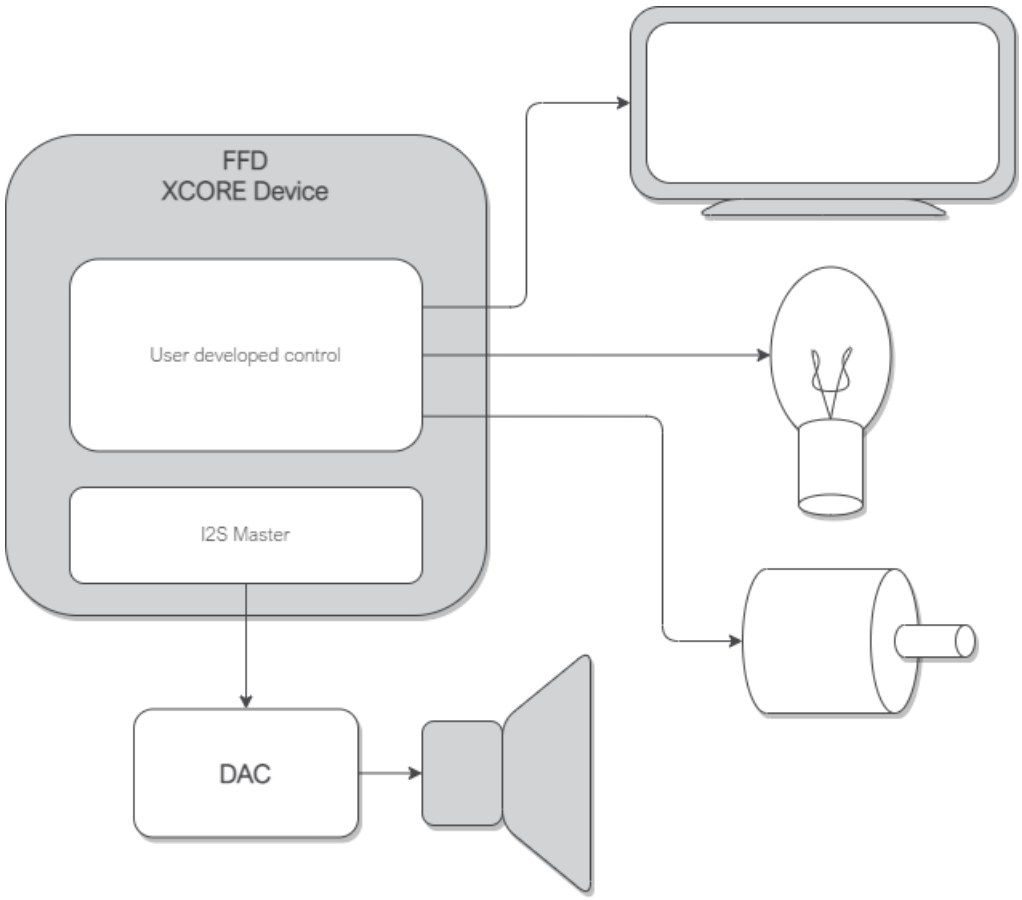
Listing 6.17: Intent Handler (intent_handler.c)

```
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 Wanson Speech Recognition

License This software is an evaluation version only. It includes a mechanism that limits the maximum number of recognitions to 50.

The Wanson speech recognition library is *Copyright 2022. Shanghai Wanson Electronic Technology Co.Ltd ("WANSON")* and is subject to the Wanson Restrictive License.

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

The model used in FFD is approximately 185k. The runtime and application supporting code consumes approximately 250k.

With the model in flash, the Wanson engine requires a core frequency of at least 400 MHz to keep up with real time. Additionally, the Wanson engine must be on the same tile as the flash.

To replace the Wanson engine with a different engine, refer to the FFD documentation on *Replacing ASR Engine Block*

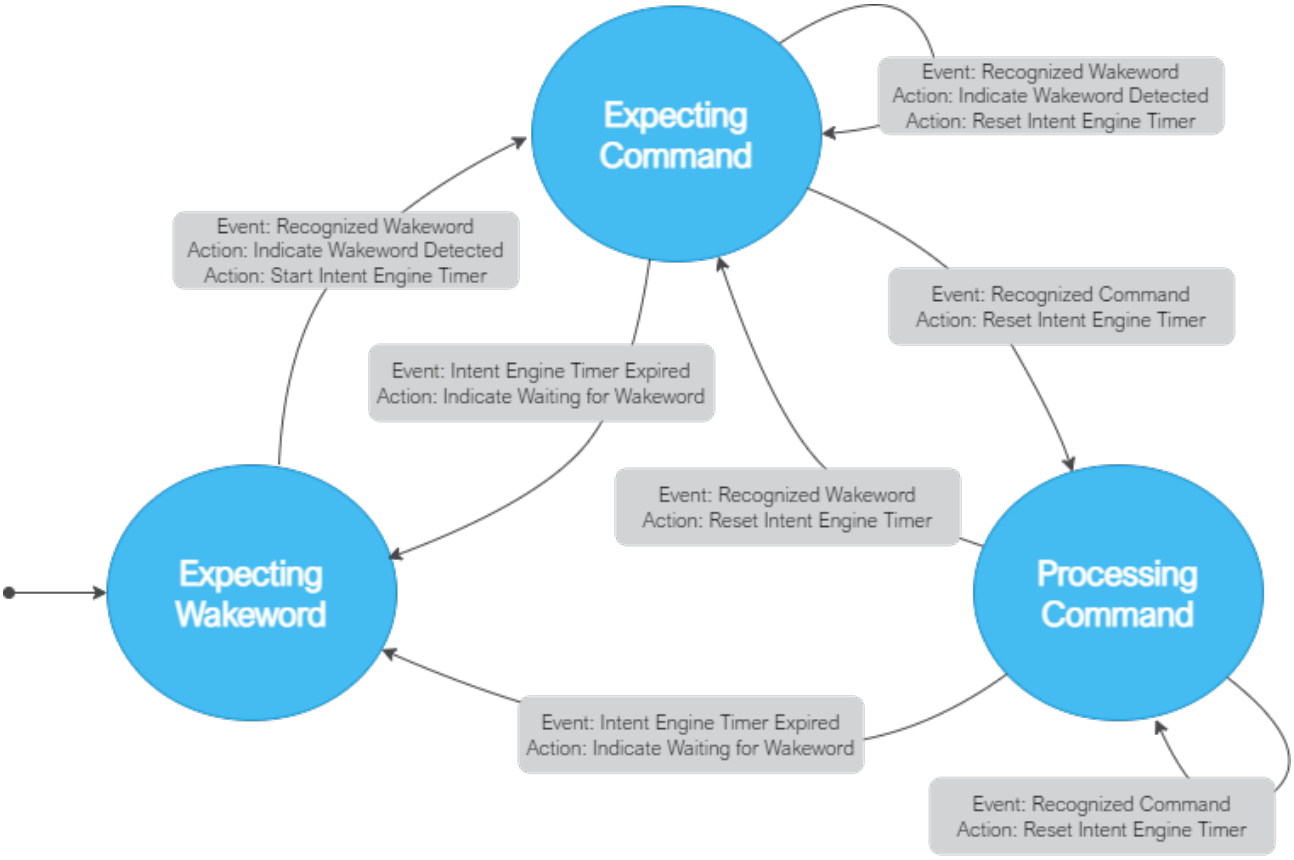
Dictionary command table

Table 6.16: 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

State Machine An optional state machine is used to condition the raw output of the Wanson speech engine.

When using the state machine, the application intent callback will only occur when a wake word and command have been detected within a time period.



The state machine logic can be disabled by setting the compile time option appconfINTENT_RAW_OUTPUT, to 1. The wake word to command timeout is compile time configurable via appconfINTENT_RESET_DELAY_MS.

More information on these options can be found in the FFD *Configuring the Firmware* section.

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

6.2 Far-field Voice Assistant

6.2.1 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.

6.2.2 Supported Hardware

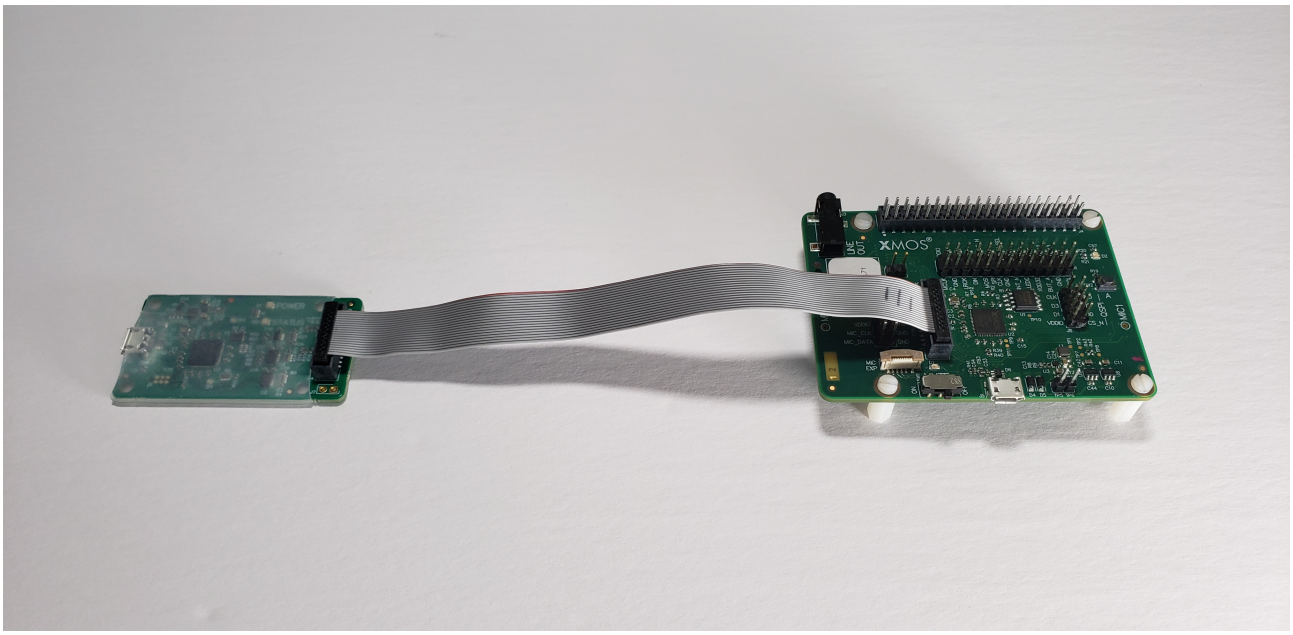
This example application is supported on the XK-VOICE-L71 board.

6.2.2.1 Setting up the Hardware

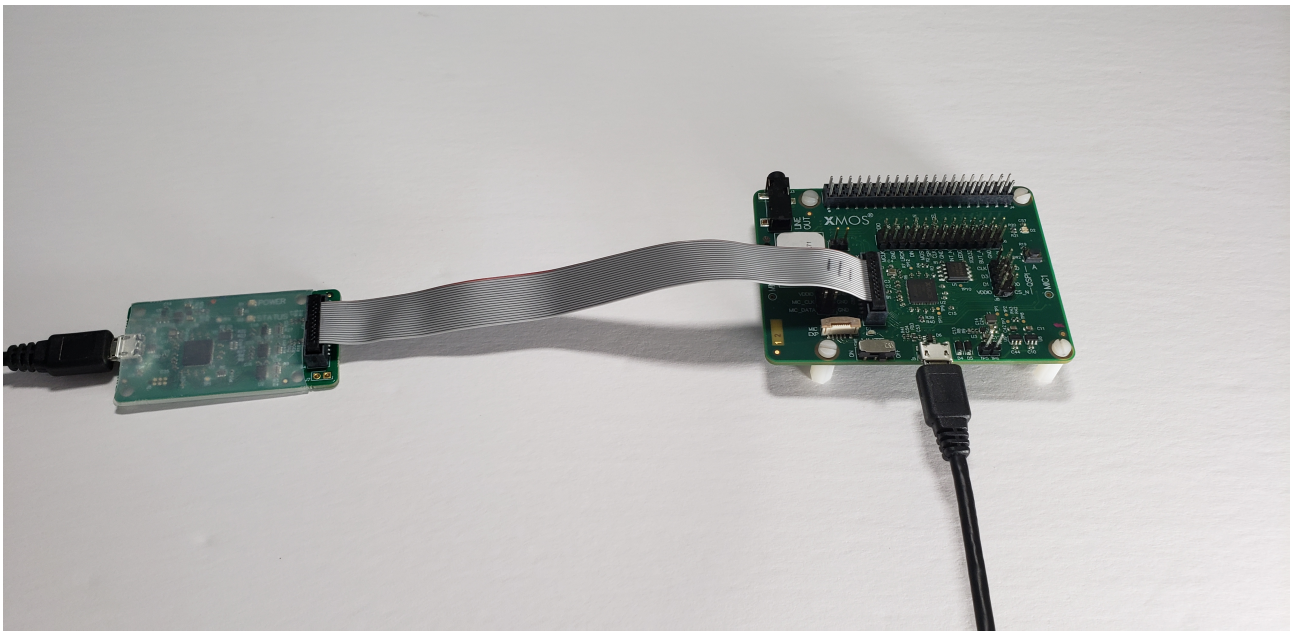
This example design requires an XTAG4 and XK-VOICE-L71 board.

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 Deploying the Firmware with Linux or macOS

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

6.2.3.1 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.

6.2.3.2 Building the Firmware

Run the following commands in the root folder to build the I^2S firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffva_int_fixed_delay
```

Run the following commands in the root folder to build the USB firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_ffva_ua_adec
```

6.2.3.3 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_ua_adec
```

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:

```
make run_example_ffva_int_fixed_delay
make run_example_ffva_ua_adec
```

6.2.3.4 Upgrading the Firmware

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
```

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_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.

6.2.3.5 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
make debug_example_int_adec
make debug_example_ua_adec
```

6.2.4 Deploying the Firmware with Native Windows

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

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.

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 "NMake Makefiles" -B build_host
cd build_host
nmake install
```

The host applications will be install at `<USERPROFILE>\.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

Run the following commands in the root folder to build the I^{2}S firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
nmake example_ffva_int_fixed_delay
```

Run the following commands in the root folder to build the USB firmware:

```
cmake -G "NMake Makefiles" -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
nmake example_ffva_ua_adec
```

6.2.4.3 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:

```
nmake flash_app_example_ffva_int_fixed_delay
nmake flash_app_example_ffva_ua_adec
```

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:

```
nmake run_example_ffva_int_fixed_delay
nmake run_example_ffva_ua_adec
```

6.2.4.4 Upgrading the Firmware

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:

```
nmake create_upgrade_img_example_ffva_ua_adec
```

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_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.

6.2.4.5 Debugging the Firmware

To debug with xgdb, from the build folder run:

```
nmake debug_example_int_adec
nmake debug_example_ua_adec
```

6.2.5 Modifying the Software

6.2.5.1 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.

Single Controller Solution In a single controller solution, a user can populate the model runner manager task with the application specific code.

This dummy thread receives only the ASR channel output, which has been downshifted to 16 bits.

The user must ensure the streambuffer is emptied at the rate of the audio pipeline at minimum, otherwise samples will be lost.

Populate:

Listing 6.18: Model Runner Dummy (model_runner.c)

```
void model_runner_manager(void *args)
{
    StreamBufferHandle_t input_queue = (StreamBufferHandle_t)args;

    int16_t buf[appconfWW_FRAMES_PER_INFERENCE];

    /* Perform any initialization here */

    while (1)
```

(continues on next page)

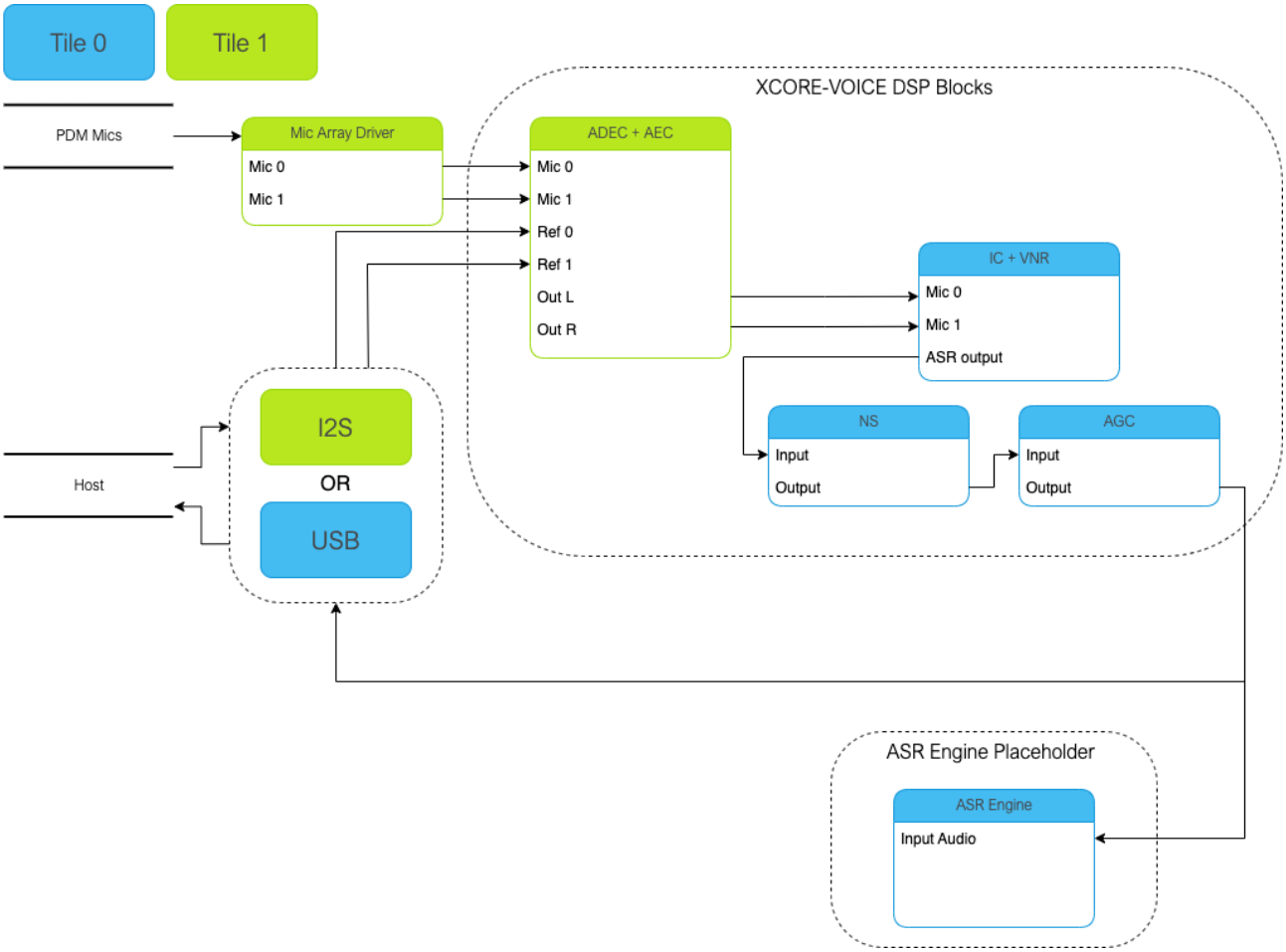
(continued from previous page)

```
    {
        /* Receive audio frames */
        uint8_t *buf_ptr = (uint8_t*)buf;
        size_t buf_len = appconfWW_FRAMES_PER_INFERENCE * sizeof(int16_t);
        do {
            size_t bytes_rxed = xStreamBufferReceive(input_queue,
                                                     buf_ptr,
                                                     buf_len,
                                                     portMAX_DELAY);
            buf_len -= bytes_rxed;
            buf_ptr += bytes_rxed;
        } while(buf_len > 0);

        /* Perform inference here */
        // rtos_printf("inference\n");
    }
}
```

6.2.5.2 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^{2}S or USB.



6.2.5.3 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 6.17: 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.

6.2.5.4 Software Description

Overview There are two main build configurations for this application.

Table 6.18: FFVA INT Fixed Delay Resources

Resource	Tile 0	Tile 1
Unused CPU Time (600 MHz)	98%	75%
Total Memory Free	166k	82k
Runtime Heap Memory Free	75k	82k

Table 6.19: FFVA UA ADEC Resources

Resource	Tile 0	Tile 1
Unused CPU Time (600 MHz)	83%	45%
Total Memory Free	123k	58k
Runtime Heap Memory Free	54k	83k

The description of the software is split up by folder:

Table 6.20: FFVA Software Description

Folder	Description
Audio Pipelines	Preconfigured audio pipelines
bsp_config	Board support configuration setting up software based IO peripherals
filesystem_support	Filesystem contents for application
src	Main application

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 6.21: 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

filesystem_support This folder contains filesystem contents for the FFVA application.

Table 6.22: FFVA filesystem_support

Filename/Directory	Description
demo.txt	Example file

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

Table 6.23: 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 6.19: 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*.

src This folder contains the core application source.

Table 6.24: 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 6.20: 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.

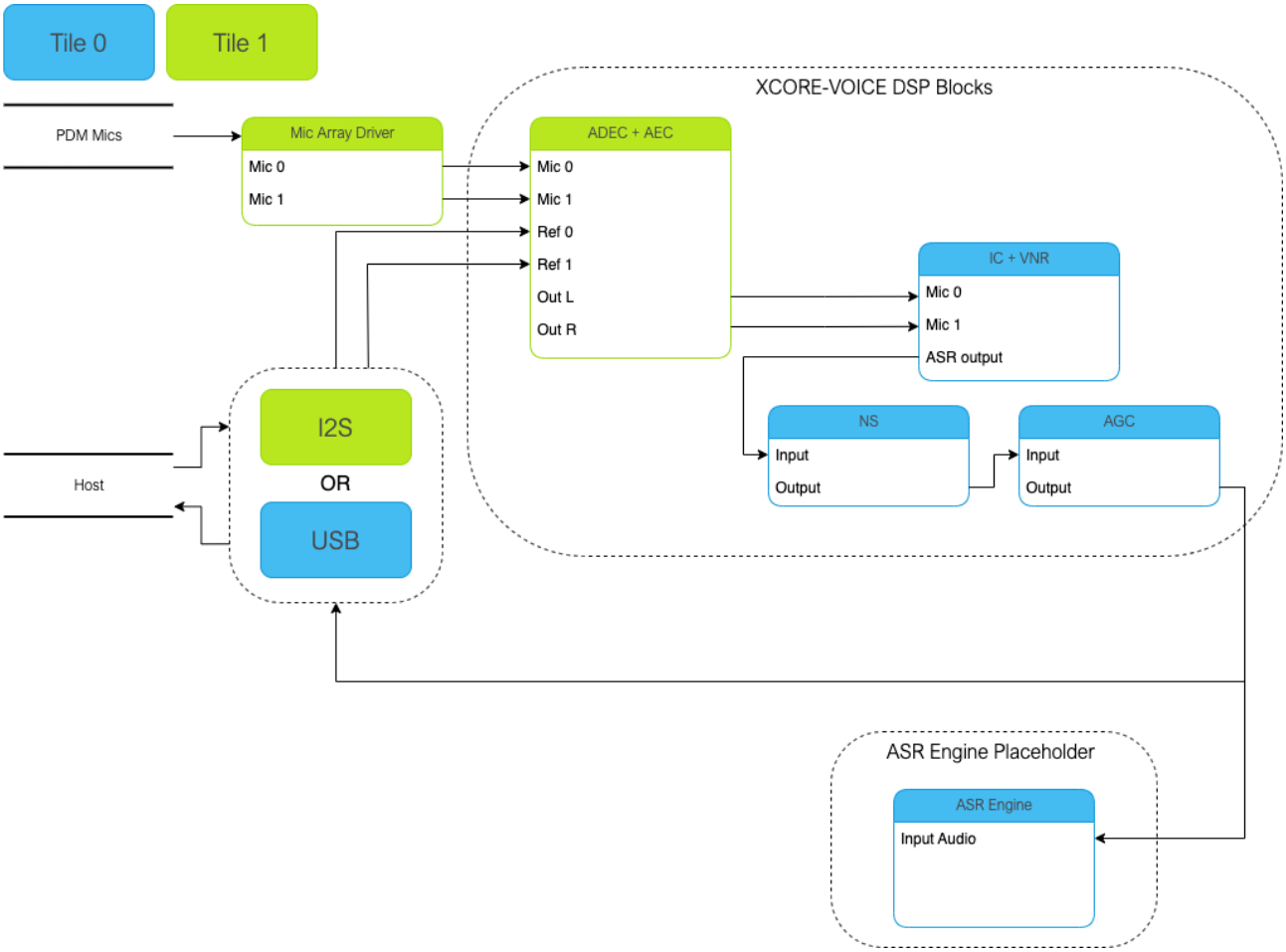
i2s_rate_conversion_enable This application features 16kHz and 48kHz audio input and output. The XMOS DPS blocks operate on 16kHz audio. Input streams are downsampled when needed. Output streams are upsampled when needed. When in I^2S modes This function is called by the bsp_config to enable the I^2S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

6.2.5.5 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 6.21: XMOS NS (audio_pipeline_t0.c)

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

With:

Listing 6.22: Foo (audio_pipeline_t0.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 6.23: 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 6.24: 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 6.25: XMOS NS (audio_pipeline_t0.c)

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

With:

Listing 6.26: Foo (audio_pipeline_t0.c)

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

Audio Pipeline Setup Replace:

Listing 6.27: 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 6.28: 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.

Populating a Keyword Engine Block To add a keyword engine block, a user may populate the existing `model_runner_manager()` function with their model:

Listing 6.29: Model Runner (model_runner.c)

```
configSTACK_DEPTH_TYPE model_runner_manager_stack_size = 287;

void model_runner_manager(void *args)
{
    StreamBufferHandle_t input_queue = (StreamBufferHandle_t)args;

    int16_t buf[appconfWW_FRAMES_PER_INFERENCE];

    /* Perform any initialization here */

    while (1)
    {
        /* Receive audio frames */
        uint8_t *buf_ptr = (uint8_t*)buf;
        size_t buf_len = appconfWW_FRAMES_PER_INFERENCE * sizeof(int16_t);
        do {
            size_t bytes_rxed = xStreamBufferReceive(input_queue,
                                                     buf_ptr,
                                                     buf_len,
                                                     portMAX_DELAY);
            buf_len -= bytes_rxed;
            buf_ptr += bytes_rxed;
        } while(buf_len > 0);

        /* Perform inference here */
        // rtos_printf("inference\n");
    }
}
```

Populate initialization and inference engine calls where commented. After adding user code, the stack size of the task will need to be adjusted accordingly based on the engine being used. The input streambuffer must be emptied at least at the rate of the audio pipeline otherwise frames will be lost.

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 6.30: 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;
```

(continues on next page)

(continued from previous page)

```
    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^{2}S or USB.

Listing 6.31: Audio Pipeline Output (main.c)

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

6.3 Automated Speech Recognition Porting

6.3.1 Overview

This is the XCORE-VOICE automated speech recognition (ASR) porting example design. This example can be used by 3rd-party ASR developers and ISVs to port their ASR library to xcore.ai.

The example reads a 1 channel, 16-bit, 16kHz wav file, slices it up into bricks, and calls the ASR library with each brick. The default brick length is 240 samples but this is configurable. ASR ports that implement the public API defined in `asr/api/asr.h` can easily be added to current and future XCORE-VOICE example designs that support speech recognition.

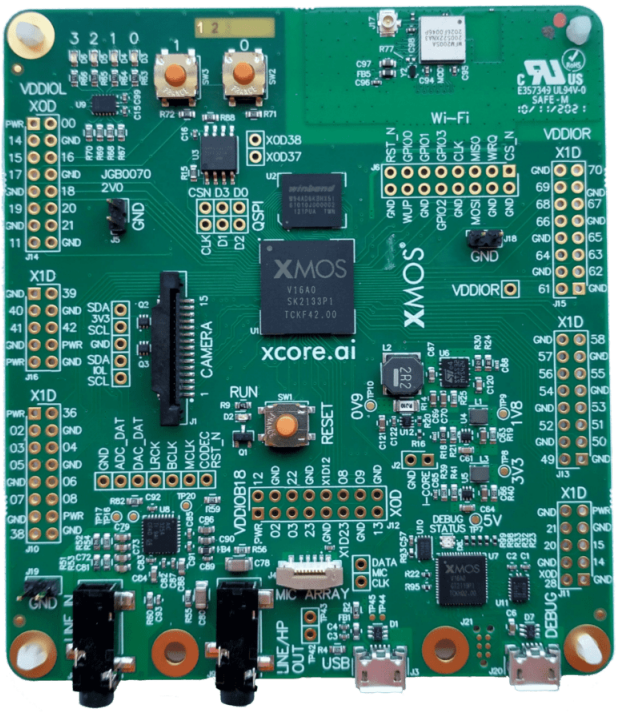
An oversimplified ASR port example is provided. This ASR port recognizes the “Hello XMOS” keyword if any acoustic activity is observed in 75 consecutive bricks.

6.3.2 Supported Hardware

This example is supported on the XCORE-AI-EXPLORER board. However, the XK-VOICE-L71 board can be supported with some minor modifications.

6.3.2.1 Setting up the Hardware

This example design requires an XCORE.AI Evaulation Kit.



Connect the XCORE.AI Evaluation Kit as described in the xcore.ai Explorer Board Quick Start guide.

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 Server

This application requires a host application to serve files to the device. The served file must be named `test.wav`. This filename is defined in `src/app_conf.h`.

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 xscope_host_endpoint
make install
```

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

Before running the host application, you may need to add the location of `xscope_endpoint.so` to your `LD_LIBRARY_PATH` environment variable. This environment variable will be set if you run the host application in the

XTC Tools command-line environment. For more information see Configuring the command-line environment.

6.3.3.2 Building the Firmware

Run the following commands in the root folder to build the firmware:

```
cmake -B build -DCMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
make example_asr
```

6.3.3.3 Flashing the Model

Run the following commands in the build folder to flash the model:

```
xflash --force --quad-spi-clock 50MHz --factory example_asr.xe --boot-partition-size␣
↪0x100000 --target-file ../examples/speech_recognition/XCORE-AI-EXPLORER.xn --data ../
↪examples/speech_recognition/asr/port/example/asr_example_model.dat
```

6.3.3.4 Running the Firmware

From the build folder run:

```
make run_example_asr
```

In a second console, run the following command in the `examples/speech_recognition` folder to run the host server:

```
xscope_host_endpoint 12345
```

6.3.4 Deploying the Firmware with Native Windows

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

6.3.4.1 Building the Host Server

This application requires a host application to serve files to the device. The served file must be named `test.wav`. This filename is defined in `src/app_conf.h`.

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.

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 "NMake Makefiles" -B build_host
cd build_host
nmake xscope_host_endpoint
nmake install
```

The host application, `xscope_host_endpoint.exe`, will install at `<USERPROFILE>\.xmos\bin`, and may be moved if desired. You may wish to add this directory to your `PATH` variable.

Before running the host application, you may need to add the location of `xscope_endpoint.dll` to your `PATH`. This environment variable will be set if you run the host application in the XTC Tools command-line environment. For more information see Configuring the command-line environment.

6.3.4.2 Building the Firmware

Run the following commands in the root folder to build the firmware:

```
cmake -G "NMake Makefiles" -B build -D CMAKE_TOOLCHAIN_FILE=xmos_cmake_toolchain/xs3a.cmake
cd build
nmake example_asr
```

6.3.4.3 Flashing the Model

Run the following commands in the build folder to flash the model:

```
xflash --force --quad-spi-clock 50MHz --factory example_asr.xe --boot-partition-size␣
↪0x100000 --target-file ../examples/speech_recognition/XCORE-AI-EXPLORER.xn --data ../
↪examples/speech_recognition/asr/port/example/asr_example_model.dat
```

6.3.4.4 Running the Firmware

From the build folder run:

```
nmake run_example_asr
```

In a second console, run the following command in the `examples/speech_recognition` folder to run the host server:

```
xscope_host_endpoint.exe 12345
```

6.3.5 Modifying the Software

6.3.5.1 Implementing the ASR API

Begin your ASR port by creating a new folder under `example/speech_recognition/asr/port`. Be sure to include `asr/api/asr.h` in your port's main source file. The `asr.h` and `device_memory.h` files include comments detailing the public API methods and parameters. ASR ports that implement the public API defined can easily be added to current and future XCORE-VOICE example designs that support speech recognition.

Pay close attention to the functions: - `asr_printf` - `devmem_malloc` - `devmem_free` - `devmem_read_ext` - `devmem_read_ext_async` - `devmem_read_ext_wait`

ASR libraries should call `asr_printf` instead of `printf` or xcore's `debug_printf`.

ASR libraries must not call `malloc` directly to allocate dynamic memory. Instead call the `devmem_malloc` and `devmem_free` functions. This allows the application to provide alternative implementations of these functions - like `pvPortMalloc` and `vPortFree` in a FreeRTOS application.

The `devmem_read_ext` function is provided to load data directly from external memory (QSPI flash or LPDDR) into SRAM. This is the recommended way to load coefficients or blocks of data from a model. It is far more efficient to load the data into SRAM and perform any math on the data while it is in SRAM. The `devmem_read_ext` function a signature similar to `memcpy`. The caller is responsible for allocating the destination buffer.

Like `devmem_read_ext`, the `devmem_read_ext_async` function is provided to load data directly from external memory (QSPI flash or LPDDR) into SRAM. `devmem_read_ext_async` differs in that it does not block the caller's thread. Instead it loads the data in another thread. One must have a free core when calling `devmem_read_ext_async` or an exception will be raised. `devmem_read_ext_async` returns a handle that can later be used to wait for the load to complete. Call `devmem_read_ext_wait` to block the callers thread until the load is complete. Currently, each call to `devmem_read_ext_async` must be followed by a call to `devmem_read_ext_wait`. You can not have more than one read in flight at a time.

Note: XMOS provides an arithmetic and DSP library which leverages the XS3 Vector Processing Unit (VPU) to accelerate costly operations on vectors of 16- or 32-bit data. Included are functions for block floating-point arithmetic, fast Fourier transforms, discrete cosine transforms, linear filtering and more. See the XMath Programming Guide for more information.

Note: To minimize SRAM scratch space usage, some ASR ports load coefficients into SRAM in chunks. This is useful when performing a routine such as a vector matrix multiply as this operation can be performed on a portion of the matrix at a time.

Note: You may also need to modify `BRICK_SIZE_SAMPLES` in `app_conf.h` to match the number of audio samples expected per process for your ASR port. In other example designs, this is defined by `appconfINTENT_SAMPLE_BLOCK_LENGTH`. This is set to 240 in the existing example designs.

In the current source code, the model data (and optional grammar data) are set in `src/process_file.c`. Modify these variables to reflect your data. The remainder of the API should be familiar to ASR developers. The API can be extended if necessary.

6.3.5.2 Flashing Models

To flash your model, modify the `--data` argument passed to `xflash` command in the *Flashing the Model* section.

See `asr/port/example/asr_example_model.h` to see how the model's flash address is defined.

6.3.5.3 Placing Models in SRAM

Small models (near or under 100kB in size) may be placed in SRAM. See *asr/port/example/asr_example_model.h`* and `asr/port/example/asr_example_model.c` for more information on placing your model in SRAM.

6.3.6 ASR API

enum `asr_error_enum`

Enumerator type representing error return values.

Values:

enumerator `ASR_OK`

Ok.

enumerator `ASR_ERROR`

General error

enumerator `ASR_INSUFFICIENT_MEMORY`

Insufficient memory for given model.

enumerator `ASR_NOT_SUPPORTED`

Function not supported for given model.

enumerator `ASR_INVALID_PARAMETER`

Invalid Parameter.

enumerator `ASR_MODEL_INCOMPATIBLE`

Model type or version is not compatible with the ASR library.

enumerator `ASR_MODEL_CORRUPT`

Model malformed.

enumerator `ASR_NOT_INITIALIZED`

Not Initialized.

enumerator `ASR_EVALUATION_EXPIRED`

Evaluation period has expired.

enum `asr_keyword_enum`

Enumerator type representing each supported keyword.

Values:

enumerator `ASR_KEYWORD_UNKNOWN`

Keyword is unknown.

enumerator `ASR_KEYWORD_HELLO_XMOS`

enumerator `ASR_KEYWORD_ALEXA`

enumerator `ASR_NUMBER_OF_KEYWORDS`

enum `asr_command_enum`

Enumerator type representing each supported command.

Values:

enumerator `ASR_COMMAND_UNKNOWN`

Command is unknown.

enumerator `ASR_COMMAND_TV_ON`

enumerator `ASR_COMMAND_TV_OFF`

enumerator `ASR_COMMAND_VOLUME_UP`

enumerator `ASR_COMMAND_VOLUME_DOWN`

enumerator `ASR_COMMAND_CHANNEL_UP`

enumerator `ASR_COMMAND_CHANNEL_DOWN`

enumerator `ASR_COMMAND_LIGHTS_ON`

enumerator `ASR_COMMAND_LIGHTS_OFF`

enumerator `ASR_COMMAND_LIGHTS_UP`

enumerator `ASR_COMMAND_LIGHTS_DOWN`

enumerator `ASR_COMMAND_FAN_ON`

enumerator `ASR_COMMAND_FAN_OFF`

enumerator `ASR_COMMAND_FAN_UP`

enumerator `ASR_COMMAND_FAN_DOWN`

enumerator `ASR_COMMAND_TEMPERATURE_UP`

enumerator `ASR_COMMAND_TEMPERATURE_DOWN`

enumerator `ASR_NUMBER_OF_COMMANDS`

typedef void *`asr_port_t`

Typedef to the ASR port context struct.

An ASR port can store any data needed in the context. The context pointer is passed to all API methods and can be cast to any struct defined by the ASR port.

typedef struct *asr_attributes_struct* `asr_attributes_t`

Typedef to the ASR port and model attributes

typedef struct *asr_result_struct* `asr_result_t`

Typedef to the ASR result

typedef enum *asr_error_enum* `asr_error_t`

Enumerator type representing error return values.

typedef enum *asr_keyword_enum* `asr_keyword_t`

Enumerator type representing each supported keyword.

typedef enum *asr_command_enum* `asr_command_t`

Enumerator type representing each supported command.

void `asr_printf`(const char *format, ...)

String output function that allows the application

to provide an alternative implementation.

ASR ports should call asr_printf instead of printf

asr_port_t `asr_init`(int32_t *model, int32_t *grammar, devmem_manager_t *devmem_ctx)

Initialize an ASR port.

Parameters

- `model` – A pointer to the model data.
- `grammar` – A pointer to the grammar data (Optional).
- `devmem_ctx` – A pointer to the device manager (Optional). Save this pointer if calling any device manager API functions.

Returns

the ASR port context.

asr_error_t `asr_get_attributes`(*asr_port_t* *ctx, *asr_attributes_t* *attributes)

Get engine and model attributes.

Parameters

- `ctx` – A pointer to the ASR port context.
- `attributes` – The attributes result.

Returns

Success or error code.

asr_error_t `asr_process`(*asr_port_t* *ctx, int16_t *audio_buf, size_t buf_len)

Process an audio buffer.

Parameters

- `ctx` – A pointer to the ASR port context.
- `audio_buf` – A pointer to the 16-bit PCM samples.
- `buf_len` – The number of PCM samples.

Returns

Success or error code.

asr_error_t `asr_get_result`(*asr_port_t* *ctx, *asr_result_t* *result)

Get the most recent results.

Parameters

- `ctx` – A pointer to the ASR port context.
- `result` – The processed result.

Returns

Success or error code.

asr_error_t `asr_reset`(*asr_port_t* *ctx)

Reset ASR port (if necessary).

Called before the next call to asr_process.

Parameters

- `ctx` – A pointer to the ASR port context.

Returns

Success or error code.

asr_error_t `asr_release`(*asr_port_t* *ctx)

Release ASR port (if necessary).

The ASR port must deallocate any memory.

Parameters

- `ctx` – A pointer to the ASR port context.

Returns

Success or error code.

asr_keyword_t `asr_get_keyword`(*asr_port_t* *ctx, int16_t asr_id)

Return the XCORE-VOICE supported keyword type.

Parameters

- `ctx` – A pointer to the ASR port context.
- `asr_id` – The ASR port keyword identifier.

Returns

XCORE-VOICE supported keyword type

asr_command_t `asr_get_command`(*asr_port_t* *ctx, int16_t asr_id)

Return the XCORE-VOICE supported command type.

Parameters

- `ctx` – A pointer to the ASR port context.
- `asr_id` – The ASR port command identifier.

Returns
XCORE-VOICE supported command type

struct `asr_attributes_struct`

#include <asr.h> Typedef to the ASR port and model attributes

struct `asr_result_struct`

#include <asr.h> Typedef to the ASR result

6.3.7 Device Memory API

void *`devmem_malloc`(devmem_manager_t *ctx, size_t size)

Memory allocation function that allows the application to provide an alternative implementation.

Call devmem_malloc instead of malloc

Parameters

- `ctx` – A pointer to the device memory context.
- `size` – Number of bytes to allocate.

Returns
A pointer to the beginning of newly allocated memory, or NULL on failure.

void `devmem_free`(devmem_manager_t *ctx, void *ptr)

Memory deallocation function that allows the application to provide an alternative implementation.

Call devmem_free instead of free

Parameters

- `ctx` – A pointer to the device memory context.
- `ptr` – A pointer to the memory to deallocate.

void `devmem_read_ext`(devmem_manager_t *ctx, void *dest, const void *src, size_t n)

Synchronous extended memory read function that allows the application

to provide an alternative implementation. Blocks the callers thread until the read is completed.

Call devmem_read_ext instead of any other functions to read memory from flash, LPDDR or SDRAM. Modules are free to use memcpy if the dest and src are both SRAM addresses.

Parameters

- `ctx` – A pointer to the device memory context.
- `dest` – A Pointer to the destination array where the content is to be read.
- `src` – A Pointer to the source of data to be read.
- `n` – Number of bytes to read.

int `devmem_read_ext_async`(devmem_manager_t *ctx, void *dest, const void *src, size_t n)

Asynchronous extended memory read function that allows the application

to provide an alternative implementation.

Call asr_read_ext_async instead of any other functions to read memory from flash, LPDDR or SDRAM.

Parameters

- `ctx` – A pointer to the device memory context.
- `dest` – A Pointer to the destination array where the content is to be read.
- `src` – A Pointer to the source of data to be read.
- `n` – Number of bytes to read.

Returns

A handle that can be used in a call to devmem_read_ext_wait.

void `devmem_read_ext_wait`(devmem_manager_t *ctx, int handle)

Wait in the caller's thread for an asynchronous extended memory read to finish.

Parameters

- `ctx` – A pointer to the device memory context.
- `handle` – The devmem_read_ext_asyc handle to wait on.

`IS_SRAM`(a)

`IS_SWMEM`(a)

`IS_FLASH`(a)

7 Memory and CPU Requirements

7.1 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 7.1: Memory Requirements

Component	Memory Use (kB)
Stereo Adaptive Echo Canceler (AEC)	275
Wanson Speech Recognition Engine	194
Interference Canceler (IC) + Voice To Noise Ratio Estimator (VNR)	130
USB	20
Noise Suppressor (NS)	15
Adaptive Gain Control (AGC)	11

7.2 CPU

The table below lists the approximate CPU requirements 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 7.2: 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
Wanson Speech Recognition Engine	80	96
Interference Canceler (IC) + Voice To Noise Ratio Estimator (VNR)	25	30
Noise Suppressor (NS)	10	12
Adaptive Gain Control (AGC)	5	6

8 Frequently Asked Questions

8.1 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
```

8.2 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.

8.3 FFD Crash At Start

One potential issue with the FFD application is a crash when trying to run:

```
Wanson init
xrun: Program received signal ET_ECALL, Application exception.
    [Switching to tile[0] core[3]]
    0x0008d308 in __xcore_ecallf ()
```

This generally occurs when the model was not properly loaded into flash. To flash the model and filesystem, see *Deploying the Firmware with Linux or macOS* or *Deploying the Firmware with Native Windows* based on host platform.

8.4 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.

8.5 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.
```

8.6 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.

9 Copyright & Disclaimer

10 Licenses

10.1 XMOS

All original source code is licensed under the XMOS License.

10.2 Third-Party

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

Table 10.1: Third Party Module Copyrights & Licenses

Module	Copyright & License
dr_wav	Copyright (C) 2022 David Reid, licensed under a public domain license
Wanson Speech Recognition Library	The Wanson speech recognition library is Copyright 2022. Shanghai Wanson Electronic Technology Co. Ltd ("WANSON") and is subject to the Wanson Restrictive License

10 Index

A

D

I