# MPLAB Harmony Audio Help

MPLAB Harmony Integrated Software Framework

# Audio Overview

This topic provides a brief overview of audio and Harmony support for audio.

## Description

This distribution package contains a variety of audio-related firmware projects that demonstrate the capabilities of the MPLAB Harmony audio offerings. Each project describes its hardware setup and requirements.

This package also contains drivers for I2S and hardware codecs that can be connected to development boards, such as the AK4954 and WM8904 Codec Daughterboards, and peripheral libraries for various I2S-related peripherals on supported processors. And it contains libraries for encoding or decoding audio streams in either Wave or ADPCM formats.

Finally, it contains BSP Audio Templates that can be used to make configuring a new audio project a matter of just a few mouse clicks.

Prior to using this demonstration, it is recommended to review the MPLAB Harmony Release Notes (`release_notes.md`) for any known issues. It is located in the `audio` folder.

The sample demonstration projects are in the `audio/apps` folder. The I2S driver is in the `audio/driver/i2s` folder. The codec drivers are in the `audio/driver/codec` folder. The peripheral drivers are in the `audio/peripheral` folder. The audio templates are in the `audio/templates` folder. The encoder and decoder libraries are in the `audio/encoder` and `audio/decoder` folders respectively.

**Important!** This repository only contains the files for the audio applications, drivers, peripheral drivers and templates.

Although you can build the applications standalone from within this repository, you will *not* be able to run the MHC or regenerate the code without the other Harmony repositories.

Please refer to the MPLAB Harmony Release Notes in the `audio` for the other Harmony repositories required to work with this one.

### Legal

Please review the Software License Agreement (`mplab_harmony_license.md`) prior to using MPLAB Harmony. It is the responsibility of the end-user to know and understand the software license agreement terms regarding the Microchip and third-party software that is provided in this installation. A copy of the agreement is available in the `audio` folder of your MPLAB Harmony installation.

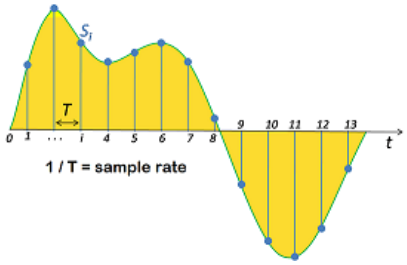
# Digital Audio Basics

This topic covers key concepts in digital audio.

## Description

**If you are new to digital audio, this section provides definitions of basic concepts found in most discussions of digital audio.**

Digital audio is sound that has been converted into digital form, by taking samples at a repeated rate (called the **sample rate** or **sampling rate**), at a particular resolution expressed as the number of bits per sample (called the **bit-depth**).




For example, audio stored on a compact disk (CD) is sampled at 44,100 samples/second, or 44.1 kHz, and saved as 16-bit signed samples. Other common sample rates are 8 kHz or 16 kHz for telephony-quality voice, 48 kHz for DVD audio, and 96 kHz for high-definition audio. The sample rate must be at least twice as fast as the highest frequency that is to be converted; therefore CD's have an upper frequency limit of 22.05 kHz. High-definition audio may also use 20, 24, or even 32-bits per sample. Low-quality voice may be sampled at only 8 bits per sample. Higher bit depths reduce the SNR (signal to noise ratio).

Changing from a lower sample rate to a higher one is called upsampling; changing from a higher sample rate to a lower one is called downsampling.

Sound is converted into digital using a analog to digital converter (ADC), connected to a microphone or other analog audio input,

and converted from digital back to analog using a digital to analog converter (DAC) connected to an amplifier and then a speaker or pair of headphones.

After being sampled, digital sound may be stored in several formats. Some formats compress the audio to save space. Compression can be either lossless, meaning the audio when uncompressed, will be exactly the same as the input; other formats may be lossy, meaning some of the original audio may be lost, but it will usually be sounds that are hard to hear. Lossy compression typically achieves much higher compression rates than lossless (lossy compressing down to 5% to 20% of the original size, compared to 50%-60% for lossless). Compression and decompression is done by software codecs.

**Common Audio Formats**

| Format | Lossy? | Proprietary? | Comments |
|---|---|---|---|
| AAC (Advanced Audio Coding) | Yes | No | Designed as successor to MP3. Audio codecs must be licensed. |
| FLAC (Free Lossless Audio Codec) | Yes | No, open-source | |
| MP3 | Yes | No | Was patented but patent has expired |
| Ogg Vorbis | Yes | No, open-source | |
| Opus | Yes | No, open-source | Low latency |
| PCM/WAV | No | No | Uncompressed linear PCM audio format used with CD's |
| WMA (Windows Media Audio) | No | Yes | |

## I2S Audio

There are various digital audio interfaces, designed to connect components together, either on the same board or via cables between boards. The one that is most relevant for us is $I^2S$ (Inter-IC sound) protocol which specifies a specific interface commonly used to connect hardware codecs, DACs, Bluetooth modules, and MCUs on the same board. It is not intended to be used over cables.

An $I^2S$ interface consists of the the following signals:

- Word clock line, which runs at the sampling rate and also indicates left/right channel, often abbreviated as LRCLK
- Bit clock line, often abbreviated as BCLK. The bit clock pulses once for each discrete bit of data on the data lines.

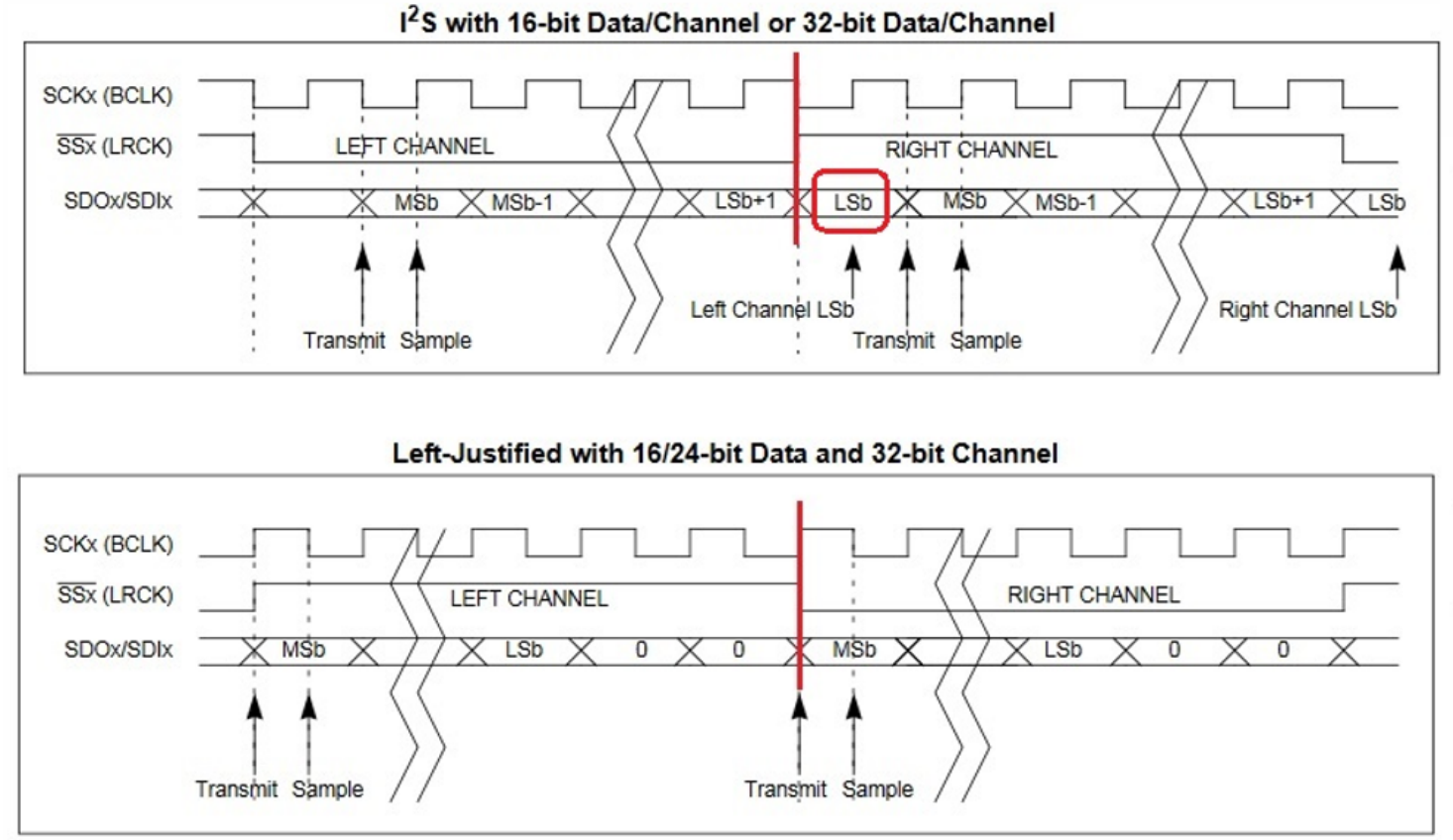
The bit clock rate = sample rate * # of channels * # of bits / channel

e.g. for CD audio, 44.1 kHz * 2 channels (stereo) * 16 bits/channel = 1.4112 MHz

- One or two serial data lines for input and output (ADCDAT and DACDAT)

Although not part of the standard, there is often a master clock (MCLK) typically running at 256 times the sample rate used for synchronization.

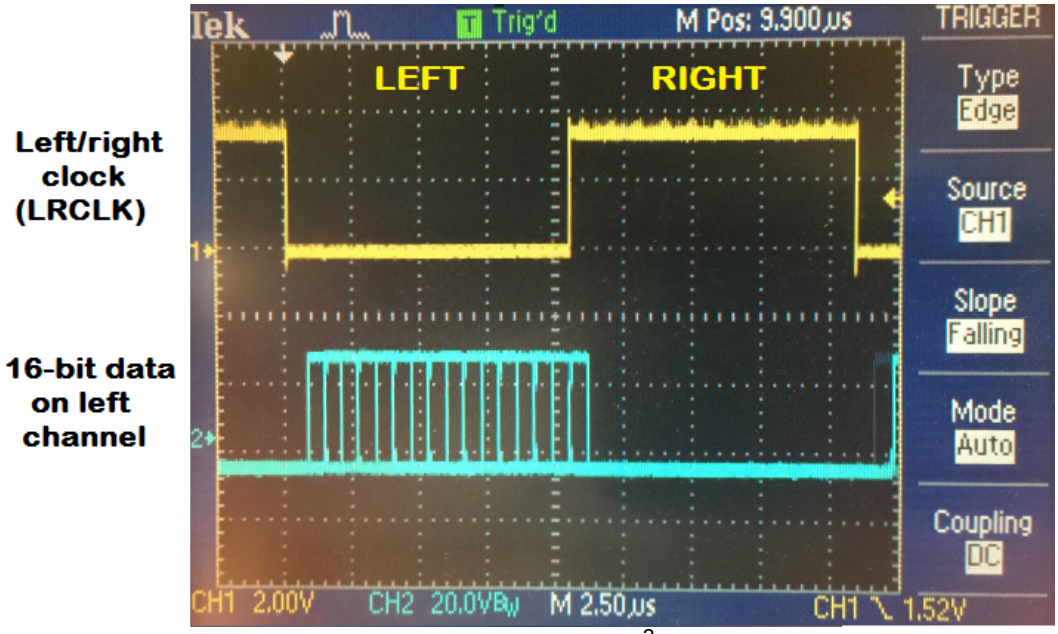
A typical codec may support both $I^2S$ format, and one or two variations (left-justified and/or right justified):



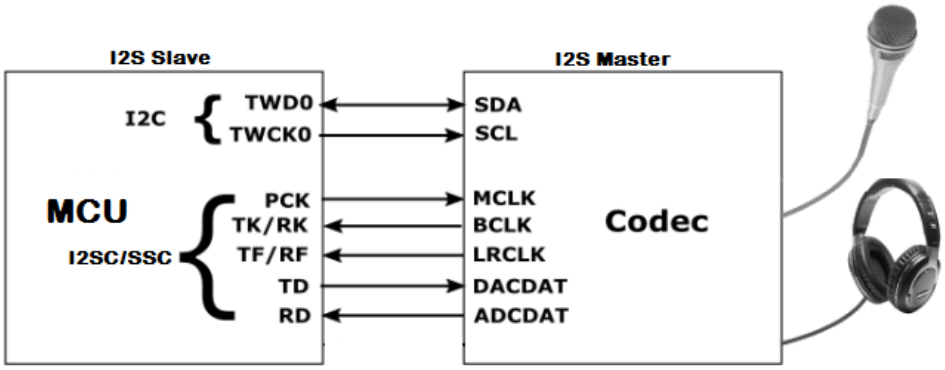

In I$^2$S format, as shown in the top half of the figure above, LRCLK is low for the left channel and high for the right channel. The most-significant bit (MSb) of the left channel data starts one bit clock (BCLK) late, such that the least significant bit (LSb) is actually in the right channel side.

In Left-Justified format, as shown in bottom half of the figure above, LRCLK is high for the left channel, and low for the right channel, and the most-significant bit (MSb) is lined up with the LRCLK and does not spill over into the opposite channel.

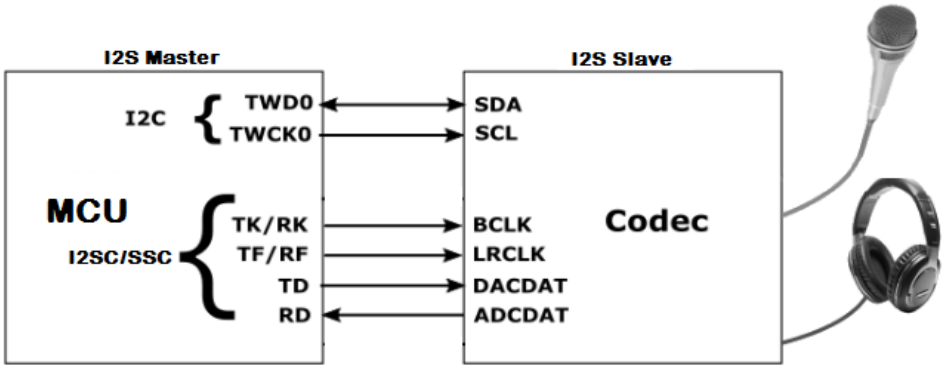
Below is a oscilloscope photo showing the LRCLK in yellow, and data in blue. Only the left channel has audio. The overlap in the I$^2$S format is clearly visible, as well as the 16 individual data bits.




A codec or Bluetooth module can act as a Master, in which it generates the I$^2$S clocks BCLK and LRCLK and sends them to the MCU (with the MCU providing a Master Clock):




If the MCU generates the $I^2S$ clocks BCLK and LRCLK, the codec or Bluetooth module is the Slave::

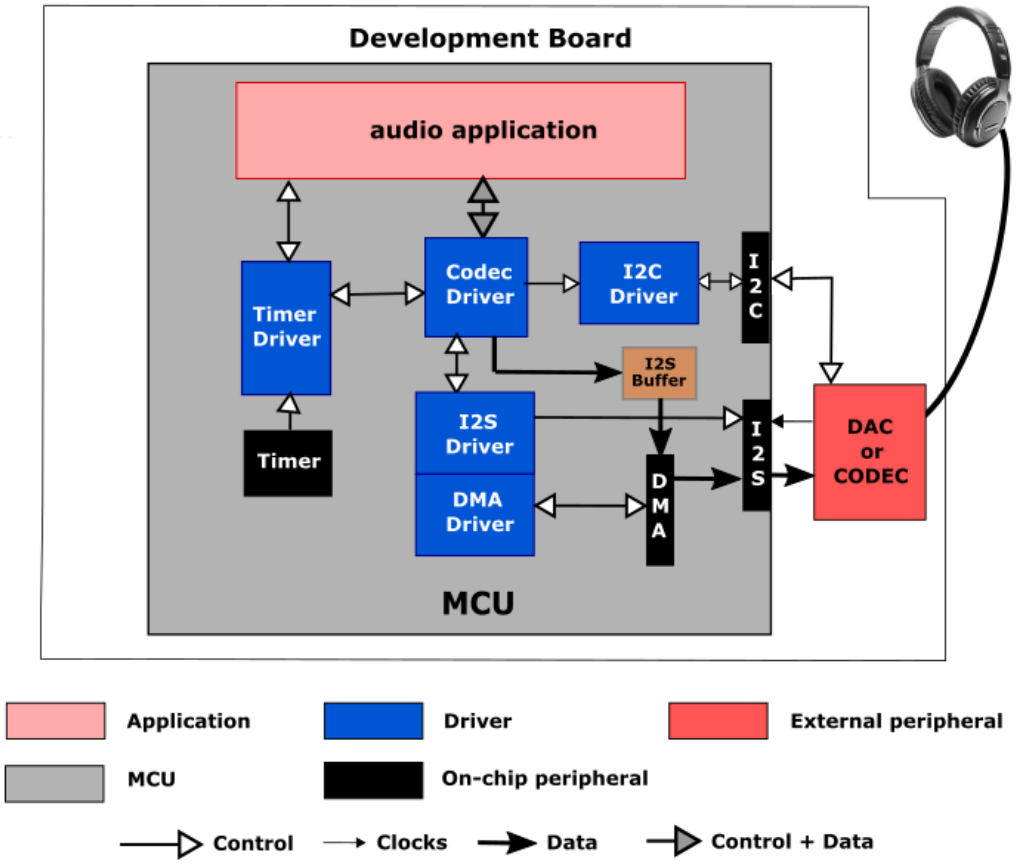



# Digital Audio in Harmony

This topic describes how digital audio is implemented in Harmony.

## Description

Audio projects in Harmony consist of an application, plus associated drivers and peripheral libraries (PLIBs). Projects using Harmony audio typically connect to either a hardware codec or DAC external to the MCU, using a $I^2S$ interface for audio. Some projects may also make use of USB.




The block diagram above depicts a generic audio application that sends audio to a hardware codec or DAC connected to a pair of headphones.

The application interfaces directly with a Codec Driver and a Timer. The Codec Driver in turn interfaces with an I2S Driver, and in this case, an I2C Driver which sends commands to and receives status from the DAC or codec via the I2C/TWIHS PLIB. In other instances an SPI interface might be used.

The I2S Driver interfaces with an $I^2S$-capable PLIB, which may be implemented using an I2S, SSC, or I2SC hardware module depending on the MCU.

The I2S Driver also uses DMA to transfer audio from the application's audio buffer to the $I^2S$-capable peripheral.

# Example Audio Projects

Example projects showing Harmony audio in action.

## Description

***The audio_enc Demonstration***

The audio_enc (encoder) application records and encodes PCM audio as wave (.wav) files on a mass storage device (e.g. thumb drive) connected to the USB Host port. Playback is an optional feature. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC), DMA, Timer and USB library. A FreeRTOS version is also available.

***The audio_player_basic Demonstration***

The audio_player_basic application scans a mass storage device (e.g. thumb drive) connected to the USB Host port for wave (.wav) and ADPCM files, and then plays them out one by one through a codec. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA, Timer and USB library. A FreeRTOS version as well as versions using an SD card instead of the USB thumb driver are also available.

***The audio_signal_generator Demonstration***

The audio_tone application sends out generated audio (sine, square, sawtooth and triangle waveforms) with volume and frequency modifiable through a touch screen display. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC), DMA and Timer.

***The audio_tone Demonstration***

The audio_tone application sends out generated audio waveforms (sine tone) with volume and frequency modifiable through the on-board push button. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA and Timer. A FreeRTOS version is also available.

***The audio_tone_linkeddma Demonstration***

The audio_tone_linkeddma application sends out generated audio waveforms (sine tone) with volume and frequency modifiable through the on-board push button, using the linked DMA feature of the MCU (where available). It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA and Timer. Linked DMA allows the DMA to process multiple buffers sequentially, without MCU intervention.

***The microphone_loopback Demonstration***

The microphone_loopback application receives audio data from the microphone input on a codec, and then sends this same audio data back out through the codec to a pair of headphones after a delay. The volume and delay are configurable using the on-board pushbutton or touch screen display. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA and Timer. A FreeRTOS version is also available.

***The usb_microphone Demonstration***

The usb_speaker application acts as a USB device, sending audio from a microphone to a USB host such as a PC. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. I2SC), DMA and Timer. A FreeRTOS version is also available.

***The usb_speaker Demonstration***

The usb_speaker applicationaacts as a USB device, playing streaming audio from a USB host such as a PC. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA and Timer. A FreeRTOS version is also available.

***The usb_speaker_hi_res Demonstration***

The usb_speaker application acts as a USB device, playing streaming audio from a USB host such as a PC at 96K samples/second. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. I2SC), DMA and Timer.

# Audio Demonstrations

This section provides information on the Audio Demonstrations provided in your installation of MPLAB Harmony.

## Introduction

MPLAB Harmony Audio Demonstrations Help.

### Description

This help file contains instructions and associated information about MPLAB Harmony Audio demonstration applications, which are contained in the MPLAB Harmony Library distribution.

## Demonstrations

This topic provides instructions about how to run the demonstration applications.

### audio_enc

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

### Description

The audio encoder (audio_enc) application configures the SAM E70 Xplained Ultra development board to be in USB Host Mass Storage Device (MSD) mode. The application supports read/write from/to the FAT file system. When a mass storage device is connected to the development board via its target USB port, the device is mounted. After the MSD is mounted, the application waits for a short button press (< 1 sec) to start recording data from the mic input on the WM8904 Audio Codec Module. This recording will continue until another short press is detected signaling to stop recording. The recorded data is then PCM encoded and packed into a .wav container file and saved on the MSD. The app then waits to playback the last recorded file or encode another file.

All encodings are at 16K sample rate, stereo, and 16 bit depth. The name of the encoded file will be created in according to the following (enc16K16b<000-999>.wav). A file will not be written over if it already exists. If a file is found to be already on the MSD, the numeric part of the file name will increment once until an unused file name is found. If trying to playback the last encoded file but there has been no file encoded yet, both LEDs will flash and the app will go back to waiting to start encoding a file. If there is a file available to be played, the app will read the .wav file data and write it to the codec for playback.

Command and control to the codec is done through an I2C driver. Data to the codec driver is sent through I2SC via I2S Driver and the output will be audible through the headphone output jack of the WM8904 Audio Codec Module connected to the SAM E70 Xplained Ultra board. The below architecture diagram depicts a representation of the application.

The current development board, SAM E70 Xplained Ultra, supplies one button and two LEDs for control and status feedback. LED 1 is amber and LED 2 is green. The application currently only supports PCM encoding and playback to and from a .wav file container.

Supported audio files are as represented in the table below.

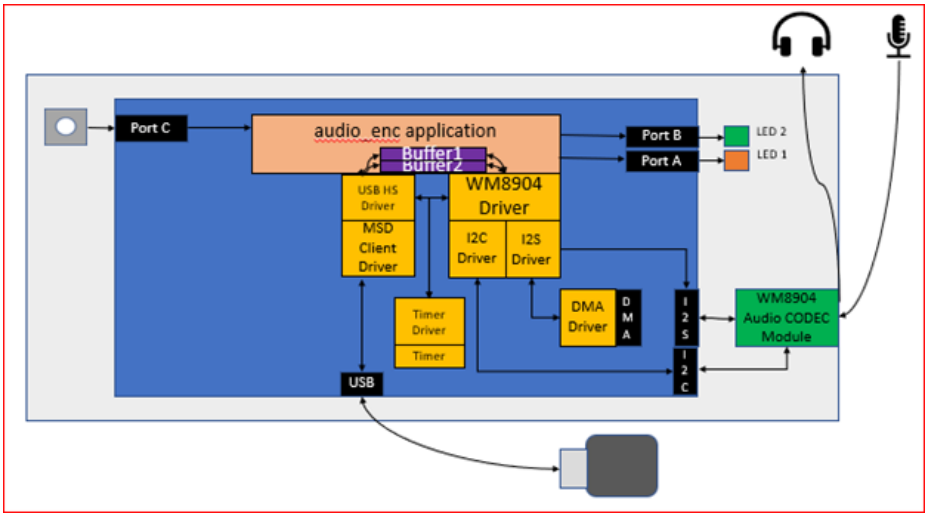
| Audio Format | Sample Rate (kHz) | Description |
|---|---|---|
| PCM | 16 | PCM (Pulse Code Modulation) is an uncompressed format. The digital data is a direct representation of the analog audio waveform. The container for the data will be a WAVE file (.wav) format. It is the native file format used by Microsoft Windows for storing digital audio data. |

#### Architecture

The audio encoder (audio_enc) application uses the MPLAB Harmony Configurator to setup the USB Host, FS, Codec, and other items in order to read the music files on a USB MSD drive and play it back on the WM8904 Codec Module. It scans WAV (PCM)

format files on mounted FAT USB thumb drive and streams audio through a WM8904 Audio Codec to a speaker. In the application, the number of audio output buffers can always set to be more than two to enhance the audio quality. And, the size of input buffer in this application is be chosen to be able to handle all data supported. The following figure shows the architecture for the demonstration.

**Architecture Block Diagram**



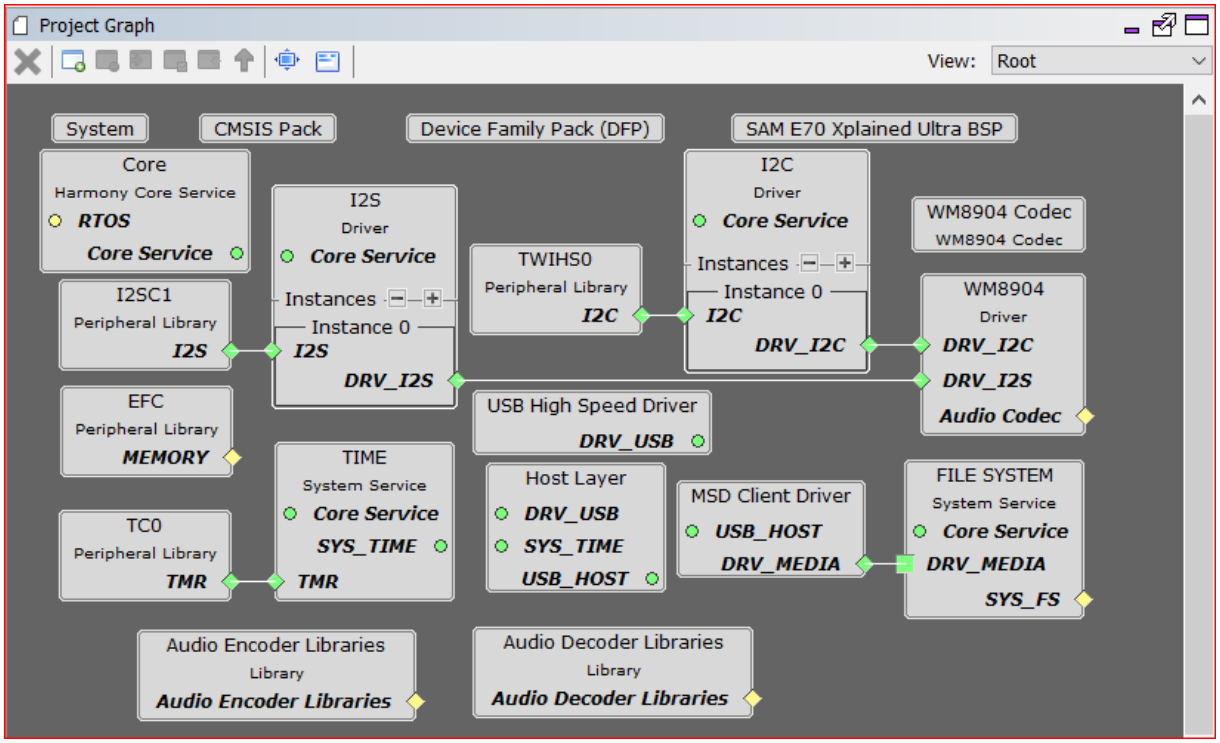

## Demonstration Features

- USB MSD Host Client Driver (see H3 USB MSD Host Client Driver Library)
- FAT File System (see H3 File System Service Library)
- Audio real-time buffer handling
- WM8904 Codec Driver (see H3 CODEC Driver Libraries)
- I2S usage in audio system (see H3 I2S Driver Library Help)
- DMA (see H3 DMA area)
- Timer (see H3 Timer area)
- GPIO Control (see H3 GPIO area)

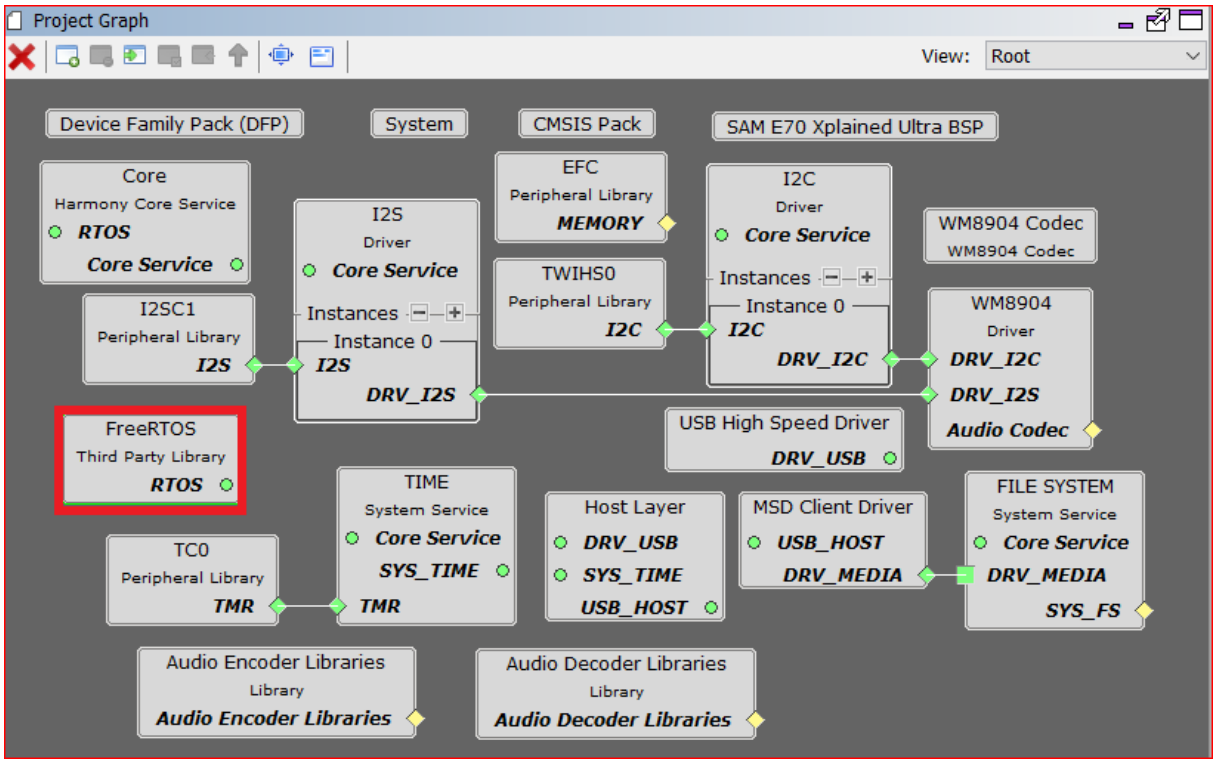
## Harmony Configuration

1. Add BSP->SAM E70 Xplained Ultra BSP
2. Add Audio->Templates->WM8904 Codec
   - Yes to all popups
   - Modify WM8904 Codec from SSC to I2SC
     - Yes to all popups
3. Add Libraries->USB->Host Stack->MSD Client Driver
   - a. Yes to all popups
4. Add Harmony->System Services->FILE SYSTEM
5. Add Harmony->Audio->Decoder->AudioDecoderLibraries
6. Add Harmony->Audio->Encoder->AudioEncoderLibraries
7. Remove FreeRTOS if Bare Metal version
8. Connect MSD Client Driver: DRV_MEDIA to FILE SYSTEM: DRV_MEDIA

After reorganization, your graph should look similar to the following:




If using FreeRTOS, your diagram will be slightly different. You will see an additional block for FreeRTOS as shown below.

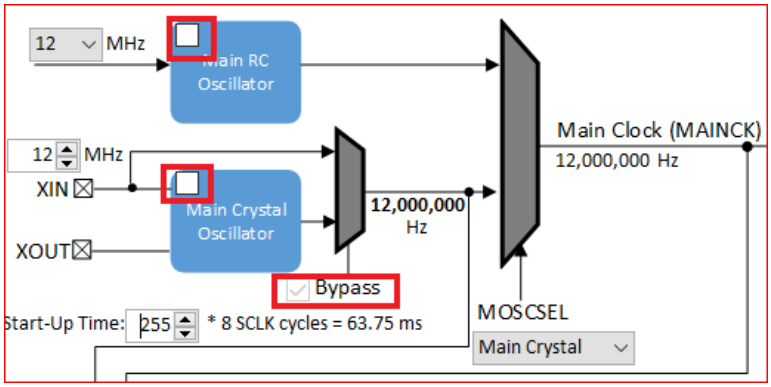



## Tools Setup Differences

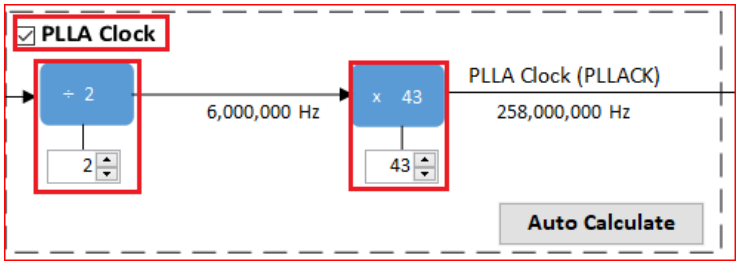
The default configuration should be correct for the majority of the app. There following configurations will need to be changed in order for proper operations.

MPLAB Harmony Configurator: Tools->Clock Configuration

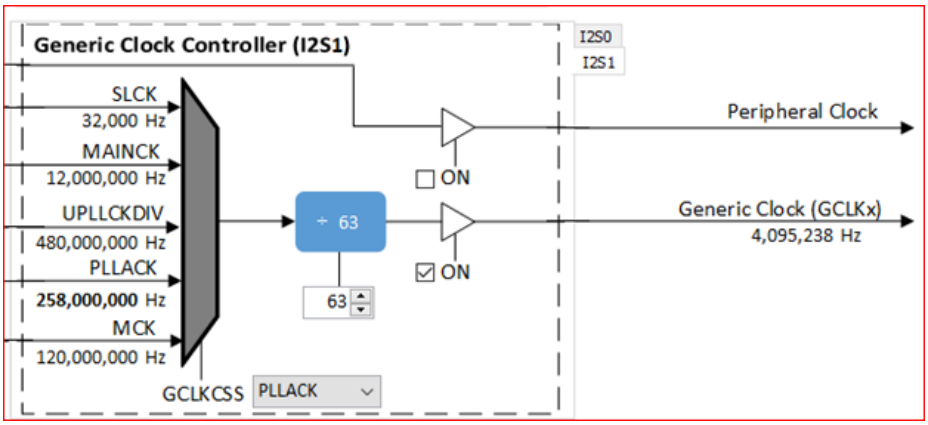
Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is check, it will cause the Main Crystal Oscillator to become unchecked.




Enable the PLLA Clock output.

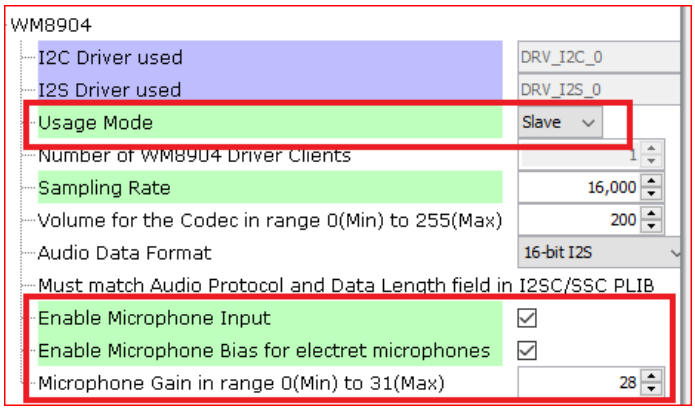



Enable clocking for the I2S1.



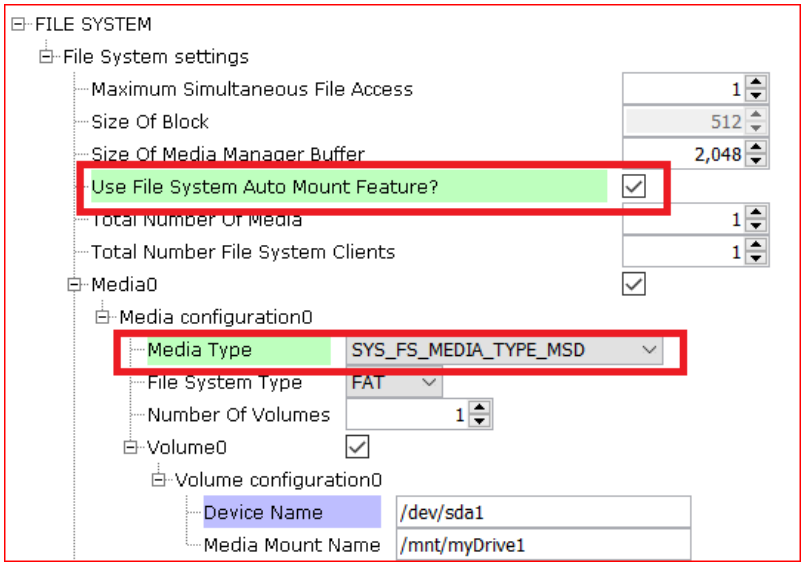

To set the sample rate to a fixed 16KHz, set the PLLA divisor to 2 and the multiplier to 43. Also, set the I2S1 divisor to 63. Please see the two images above.

MPLAB Harmony Configurator: WM8904



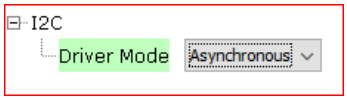

MPLAB Harmony Configurator: File System

The Auto Mount feature must be selected in order to expose the media type selection. The media type that is being used in this application is Mass Storage Device. This must be correctly configured, or the storage device will not mount.



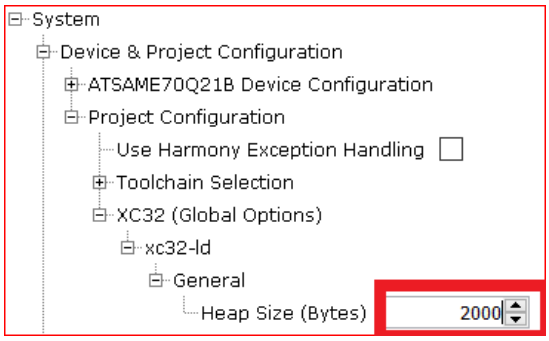

MPLAB Harmony Configurator: I2C Driver

If your are using FreeRTOS, set the driver mode back to Asynchronous.



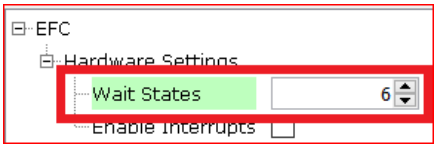

MPLAB Harmony Configurator: System

Set the heap size in Harmony if it is not already set for the linker.



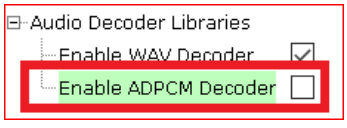

MPLAB Harmony Configurator: EFC

Set the memory wait states to 6, if not already set.




MPLAB Harmony Configurator: Audio Decoder Libraries

Disable the ADPCM Decoder for this particular app.




Save and generate code for the framework.

## *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

### Description

The parent folder for these files is `audio/apps/audio_enc`. To build this project, you must open the `audio/apps/audio_enc/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

#### MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

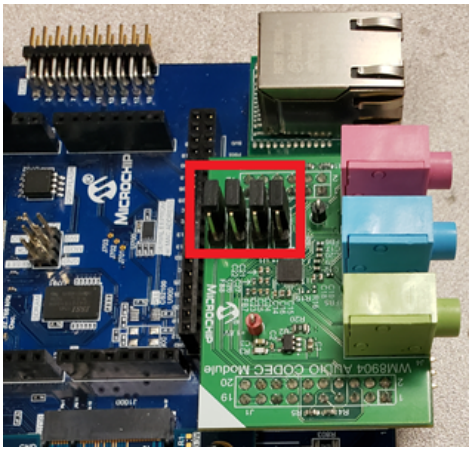
| Project Name | BSP Used | Description |
|---|---|---|
| audio_enc_sam_e70_xult_wm8904_i2sc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Module attached. The project configuration is for encode/playback of an audio file to/from a USB Mass Storage Device. Data to be tx/rx to/from the Codec via I2S audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. |
| audio_enc_sam_e70_xult_wm8904_i2sc_freertos | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Module attached. The project configuration is for encode/playback of an audio file to/from a USB Mass Storage Device. Data to be tx/rx to/from the Codec via I2S audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. This demonstration also uses FreeRTOS. |

## *Configuring the Hardware*

This section describes how to configure the supported hardware.

### Description

This application uses the I2SC PLIB to transfer data to the WM8904 Audio Codec Daughter Board. To connect to the I2SC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughter Board must be oriented towards the pink, mic in, connector. See the red outlined jumpers in the below image as reference.



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

## *Running the Demonstration*

This section demonstrates how to run the demonstration.

### Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based. Refer to Building the Application for details.

1. Connect headphones to the green HP OUT jack of the WM8904 Audio Codec Daughter Board, and a microphone to the pink MIC IN jack.
2. Connect power to the board. The system will be in a wait state for USB to be connected (amber LED1 blinking).
3. Connect a USB mass storage device (thumb drive) to the USB TARGET connector of the SAM E70 Xplained Ultra board. You will probably need a USB-A Female to Micro-B Male adapter cable to do so. The application currently can only record WAVE (.wav) format audio files.

### Control Description

1. Long presses of the push button cause the app to attempt to playback the last encoded file that is saved on the MSD. The two LEDs flash if there is no file that has been encoded.
2. Short presses of the push button cause the app to start or stop encoding.

Button control is shown in the table below.

**Button Operations**

| Operation | Function |
|---|---|
| Long Press (> 1 sec) | Playback last encoded |
| Short Press (< 1 sec) | Start/Stop Encoding |

### Status Indicator Description

1. When the application first starts running, it looks to find an attached storage device. If one is not found, LED1 will toggle on and off about every 100 ms indicating that a storage device is not attached.
2. If/When a storage device is attached, LED1 will turn off.
3. LED2 will turn on when the application is ready to start encoding or playback the last encoded file. If playback is chosen without first creating an encoded file, LED1 and LED2 will toggle briefly then the app will wait for encoding to start.

LED status indication is shown in the table below.

**LED Status**

| Operation | LED 1 Status | LED 2 Status |
|---|---|---|
| No Storage Device Connected | Toggling | Off |
| Storage Device Connected | Off | Don’t Care |
| Ready to Encode/Playback | Off | Don’t Care |
| Not Ready to Encode/Playback | Off | Off |
| Encoding | Off | Toggling |
| Playback before Encoded File Created | Toggling | Toggling |

# audio_player_basic

This topic provides instructions and information about the MPLAB Harmony 3 Audio Player Basic demonstration application, which is included in the MPLAB Harmony Library distribution.

## Description

The audio player (audio_player_basic) application configures the development board to use a microSD card or be in USB Host Mass Storage Device (MSD) mode. The application supports the FAT file system. When a mass storage device is connected to the development board via its Target USB port, the device is mounted, and the application begins to scan for files starting at the root directory. It will search for .wav files up to 10 directory levels deep. A list of files found, and their paths, will be created and stored.

Once the scan is complete, the first track in the list will be opened, validated and played. The application will read the .wav file header to validate. Configuration of the number of channels, sample size, and sample rate stated in the file will be handled by the application for proper playback. If a file that can't be played is found, it will be skipped, and the next sequential file will be tried. If the file can be played, it will then go on and read the .wav file data and write it to the codec for playback.

Command and control of the codec is done through an I2C driver. Data to the codec driver is sent through SSC via I2S Driver and the output will be audible through the headphone output jack of the WM8904 Audio Codec Module connected to the SAM E70 Xplained Ultra board.

Supported audio files are as represented in the table below.

**Supported Format**

| Audio Format | Sample Rate (kHz) | Description |
|---|---|---|
| ADPCM | 8 to 16 | Adaptive Delta Pulse Code Modulation (ADPCM) is a sub-class of the Microsoft waveform (`.wav`) file format. In this demonstration, it only decodes ADPCM audio, which has a WAV header. The extension name for this format is pcm or PCM. |
| PCM | 8 to 96 | PCM (Pulse Code Modulation) is an uncompressed format. The digital data is a direct representation of the analog audio waveform. The container for the data will be a WAVE file (.wav) format. It is the native file format used by Microsoft Windows for storing digital audio data. |

The defines DISK_MAX_DIRS and DISK_MAX_FILES in the `app.h` file, determines the maximum number of directories that should be scanned at each level of the directory tree (to prevent stack overflow, the traversing level is limited to 10), and the maximum number of songs in total the demonstration should scan (currently set to 4000 because of memory limitations).

### Architecture

The application runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- One push button (SW1)
- Two LEDs (amber LED1 and green LED2)
- WM8904 Codec Daughter Board mounted on a X32 socket

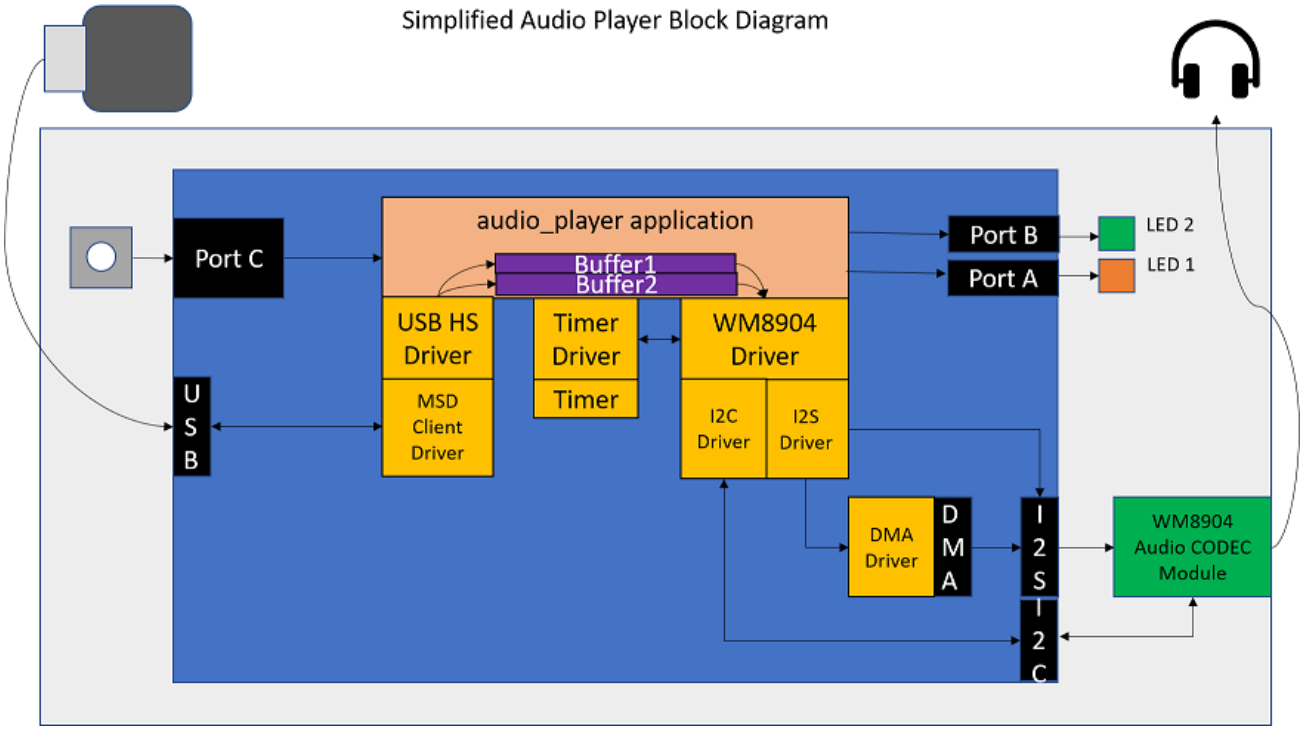
The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The application currently only supports WAVE (.wav) format files and ADPCM (.pcm) files.

The audio_player_basic application uses the MPLAB Harmony Configurator to setup the USB MSD Host, file system, codec, and other items in order to read the music files on a USB mass storage device and play it back through the WM8904 Codec Module. It scans WAV (PCM) format files from a mounted FAT drive and streams audio through a WM8904 Audio Codec to a pair of headphones. In the application, the number of audio output buffers can always set to be more than two to enhance the audio quality. The size of input buffer in this application is be chosen to be able to handle the data supported.

The following figure shows the architecture for the demonstration (USB Host version shown):

**Architecture Block Diagram**




## Demonstration Features

- USB MSD Host Client Driver (see USB MSD Host Client Driver Library)
- microSD card
- FAT File System (see File System Service Library)
- Audio real-time buffer handling
- WM8904 Codec Driver (see Audio Codec Driver Libraries)
- $I^2S$ usage in audio system (see I2S Driver Library)
- DMA (see DMA Peripheral Library)
- Timer (see Timer Peripheral Library)
- GPIO Control (see Port Peripheral Library)

**Harmony Configuration**

1. Add BSP->SAM E70 Xplained Ultra BSP
2. If creating Graphics version, add Graphics->Templates->Aria Graphics w/ PDA TM4301B Display

a. Yes to all popups.

3. Add Audio->Templates->WM8904 Codec

a. Yes to all popups.

4. Add Libraries->USB->Host Stack->MSD Client Driver

a. Yes to all popups.

5. Add Harmony->System Services->FILE SYSTEM
6. Remove FreeRTOS if Bare Metal or Graphics versions.
7. Add Harmony->Audio->Decoder->Audio Decoder Libraries
8. Connect MSD Client Driver: DRV_MEDIA to FILE SYSTEM: DRV_MEDIA
9. Connect WM8904 Driver:DRV_I2S to I2S Driver:DRV_I2S
10. If creating Graphics version, connect MaxTouch Controller:DRV_I2C to I2C Driver:DRV_I2C

After reorganization, your graph would look similar to one of the following project graphs. They specify the drivers, services, and libraries being brought into the project to further extend the applications abilities.

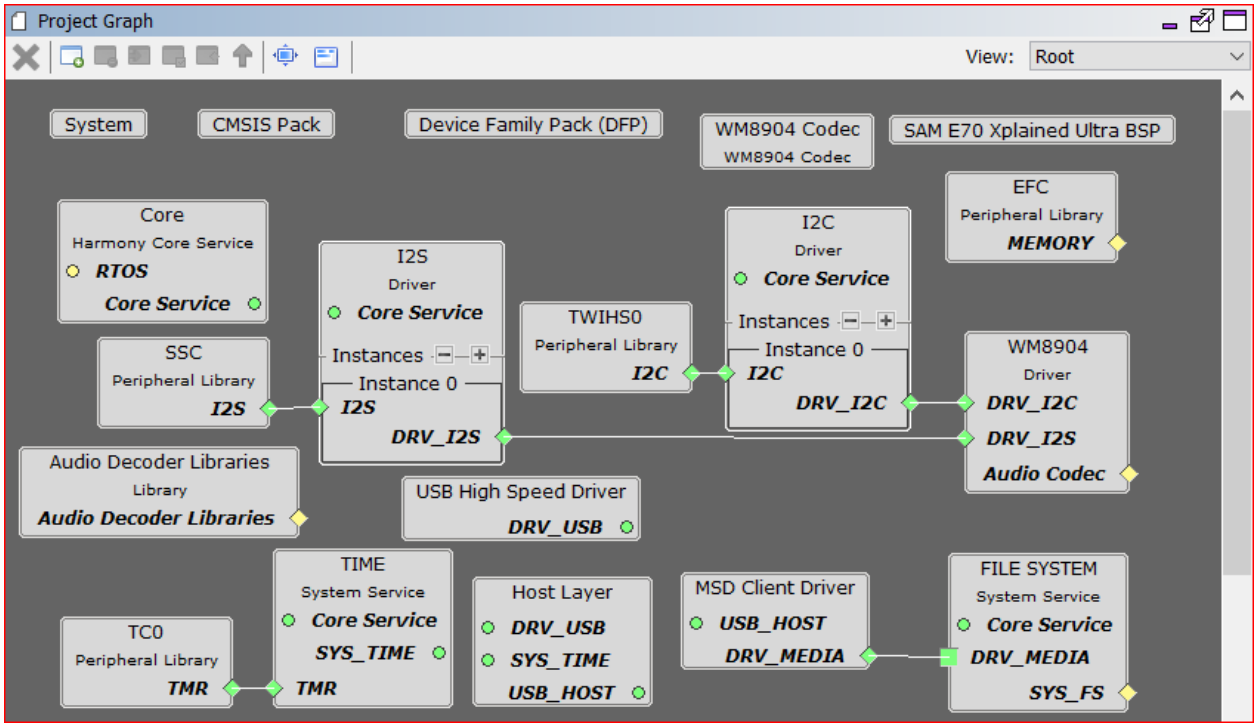



Fig 5 (Project graph of the default E70 bare metal configuration using the SSC)

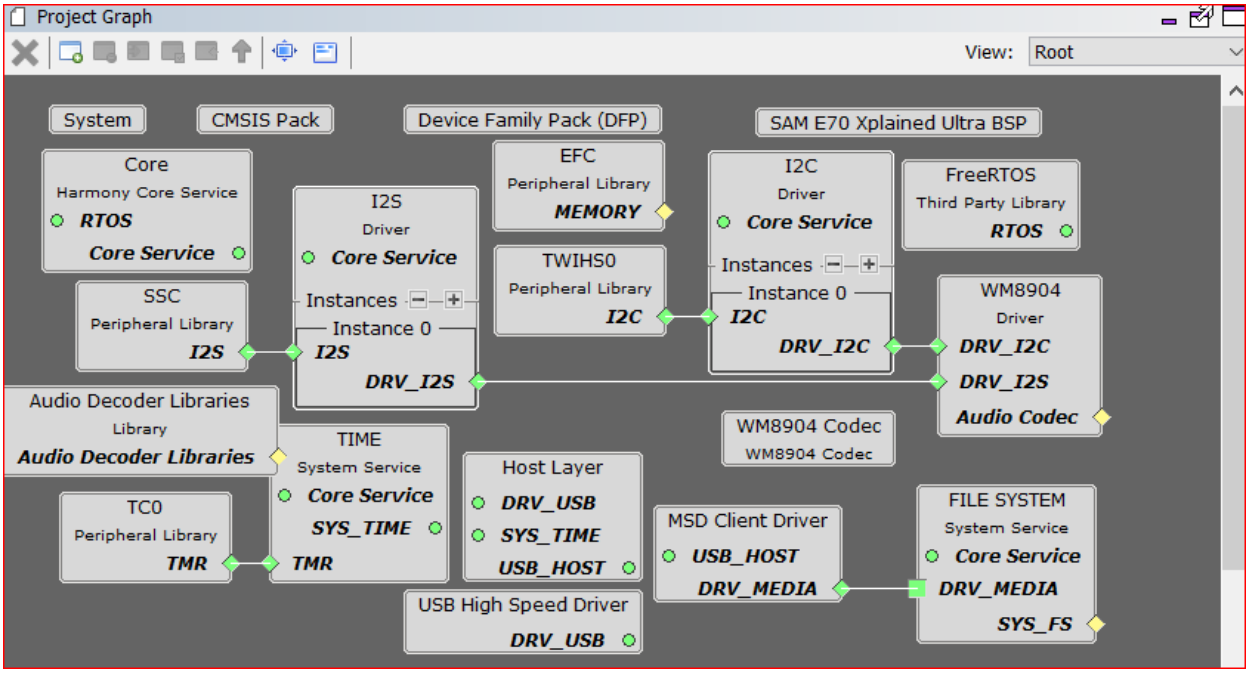



Fig 6 (Project graph of the E70 FreeRTOS configuration using the SSC)

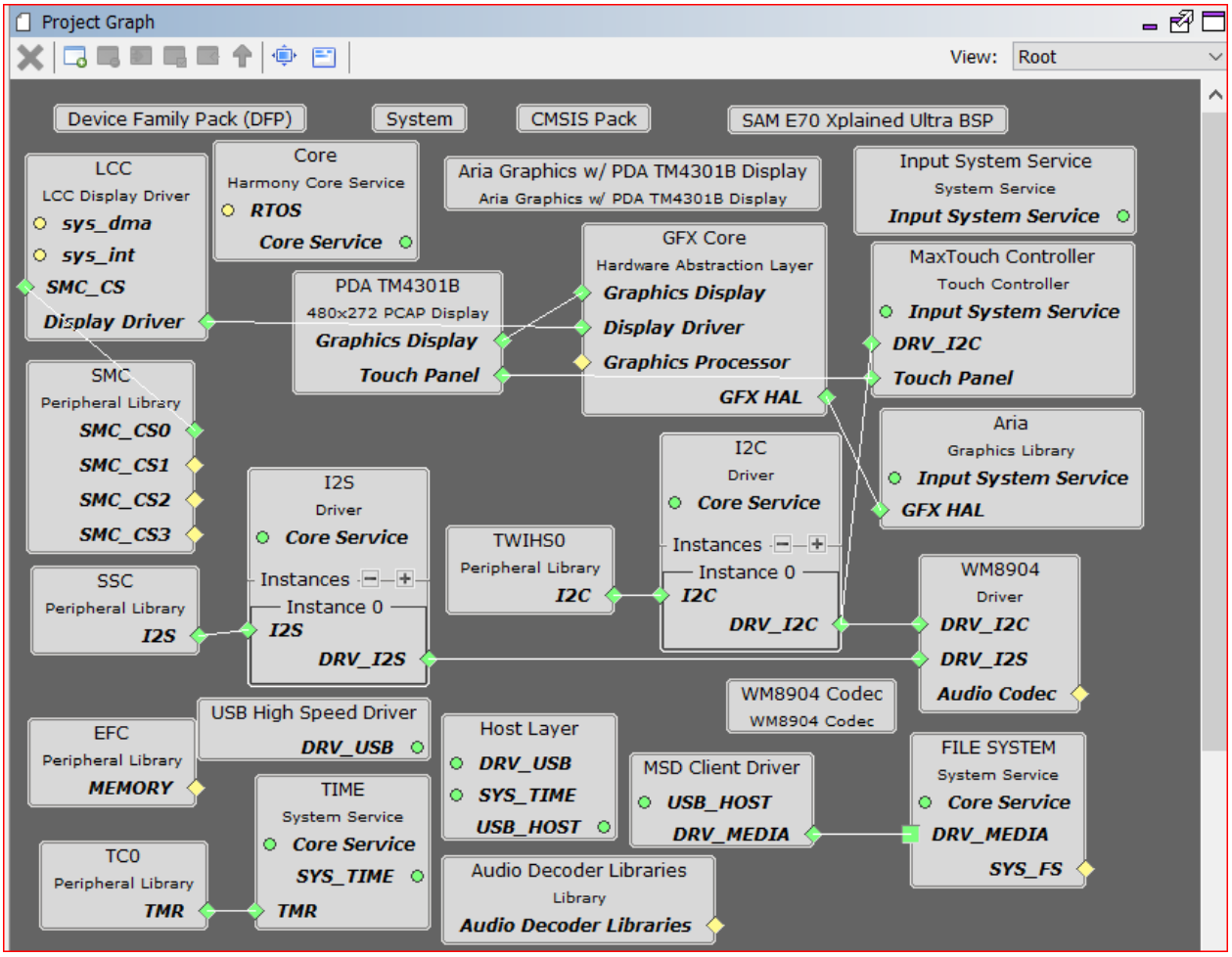



Fig 7 (Project graph of the E70 bare metal configuration using the SSC with the TM4301B graphics display)

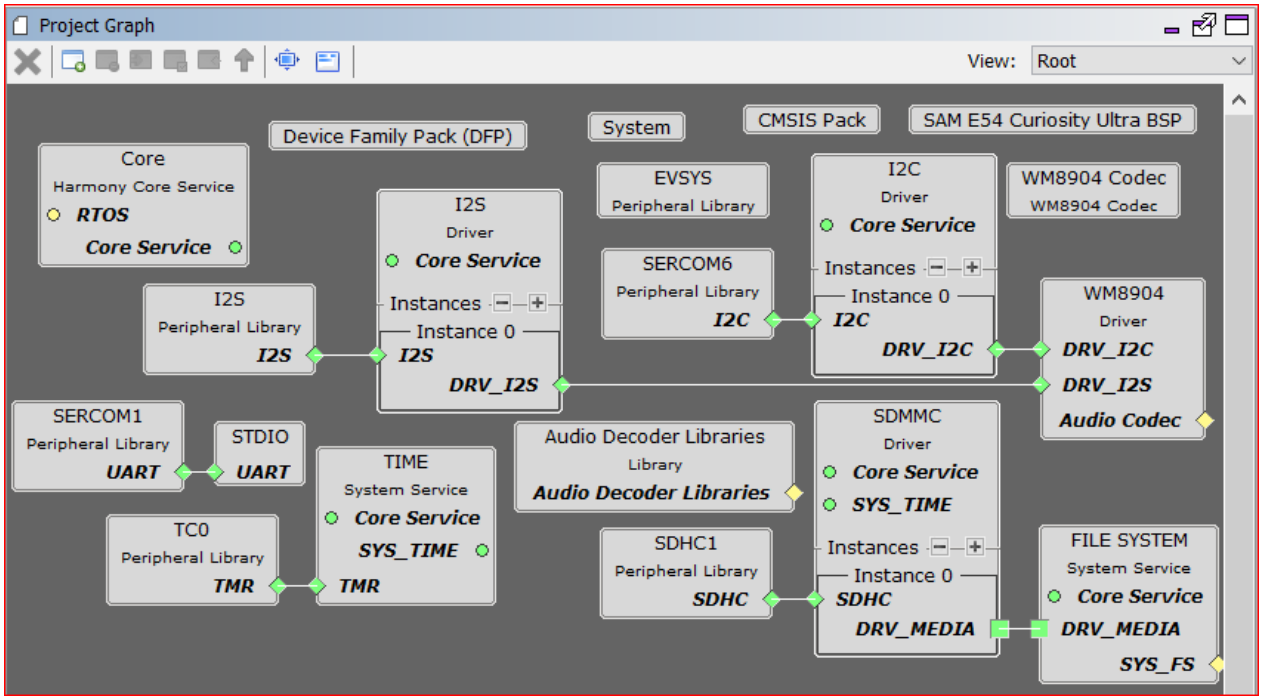



Fig 8 (Project graph of the E54 bare metal configuration using the I2S)

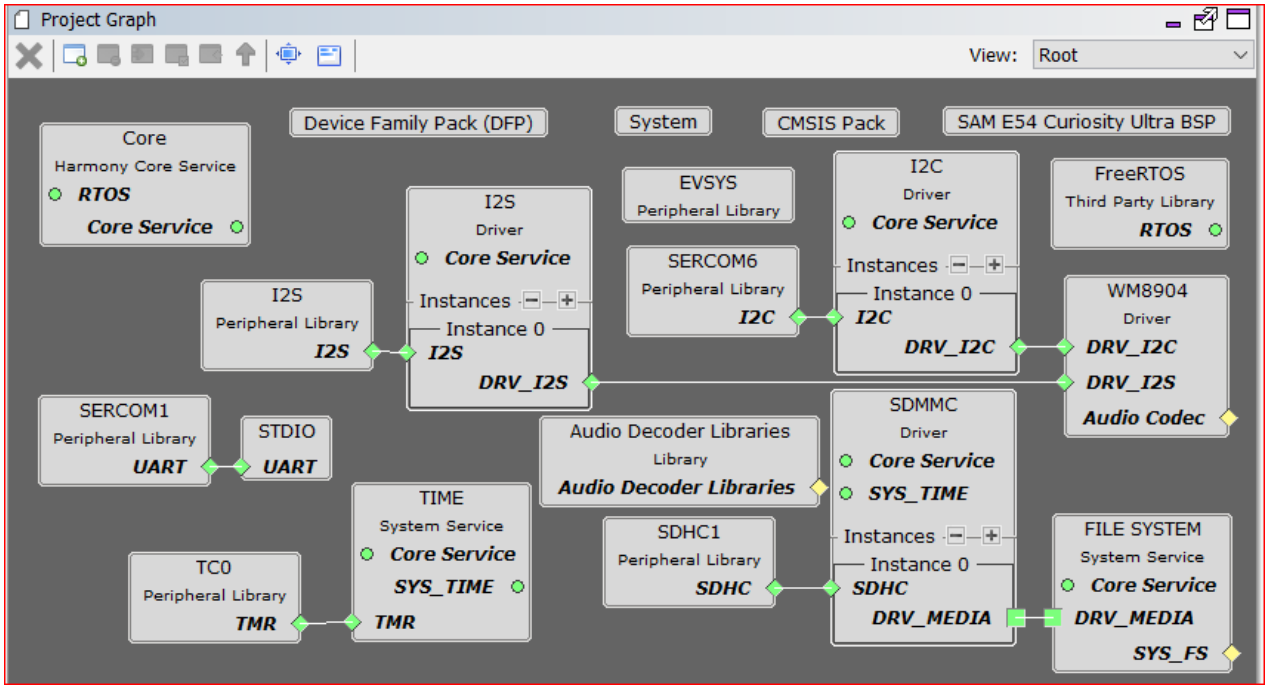



Fig 9 (Project graph of the E54 FreeRTOS configuration using the I2C)

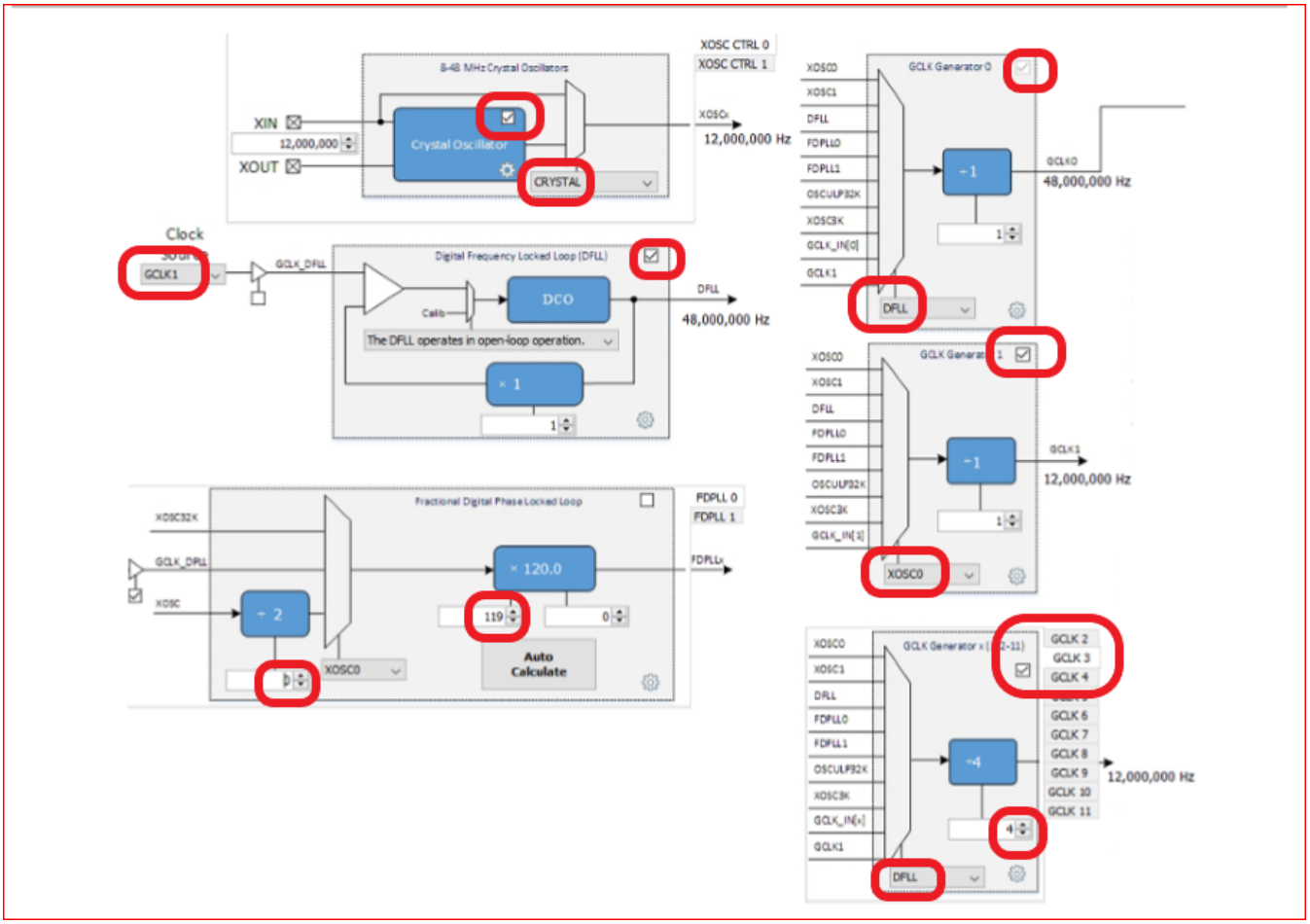
## Tools Setup Differences

The default configuration should be correct for the majority of the application. The following configurations will need to be changed in order for proper operations.

For projects using the E54 Curiosity Ultra, the I2S interface and the WM8904 as a Slave (the E54 generates the I2S clocks):

In the MPLAB Harmony Configurator: *Tools>Clock Configuration* dialog:

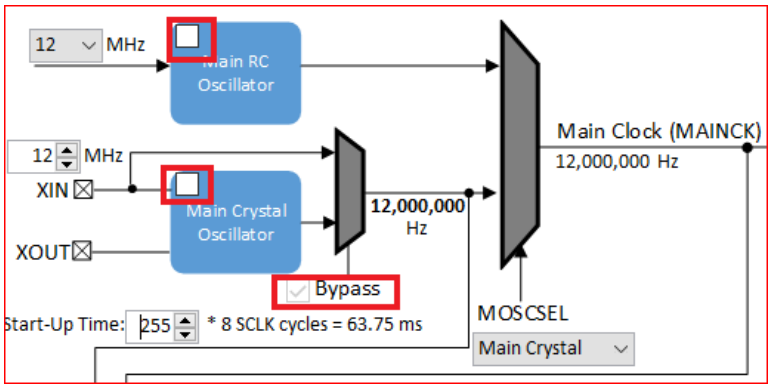
- Set enable the Crystal Oscillator, change it frequency to 12,000,000 Hz, and select CRYSTAL.
- Uncheck the Fractional Digital Phase Locked Loop enable (FDPLL 0).
- In the CLK Generator 0 box, change the input to DFLL for an output of 48 MHz.
- In the CLK Generator 1 box, change the input to XOSC0 with a divider of 1 for an output of 12 MHz.
- In the GCLK Generator, uncheck the selection for GCLK 2, and then select the GCLK 3 tab. Choose the DFLL as the input, with a divide by 4 for an output of 12 MHz. You should end up with a clock diagram like this:



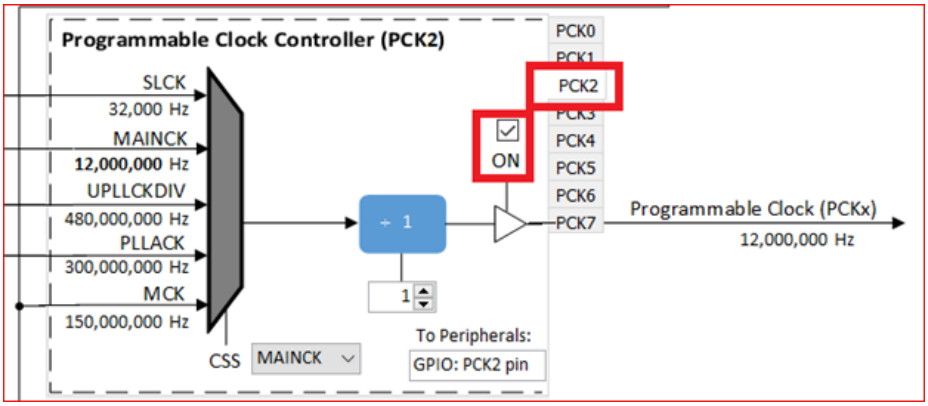

For projects using the E70 Xplained Ultra, the SSC interface and the WM8904 as a Master (the WM8904 codec generates the I2S clocks):

In the MPLAB Harmony Configurator: *Tools>Clock Configuration* dialog:

Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is checked, it will cause the Main Crystal Oscillator to become unchecked.

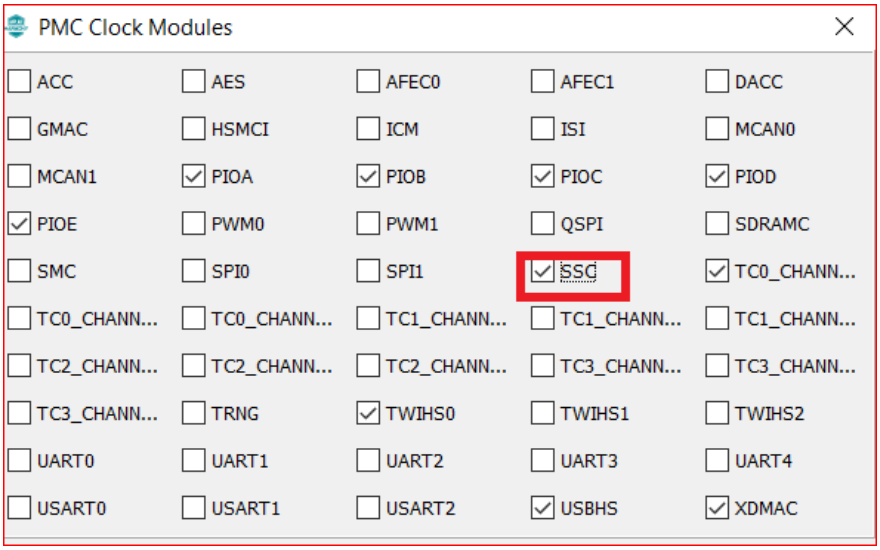



Enable the PCK2 output to enable the WM8904 master clock:



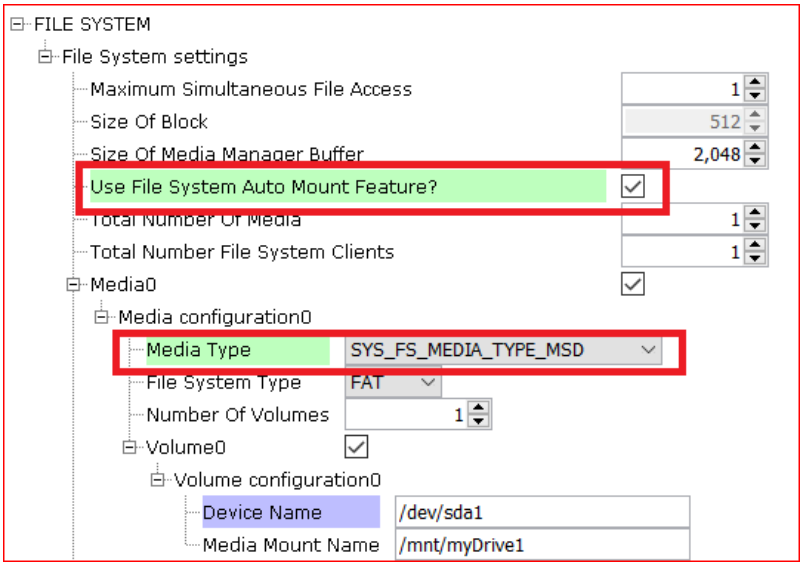

*Clock Diagram>Peripheral Clock Enable*

Enable clocking for the SSC.



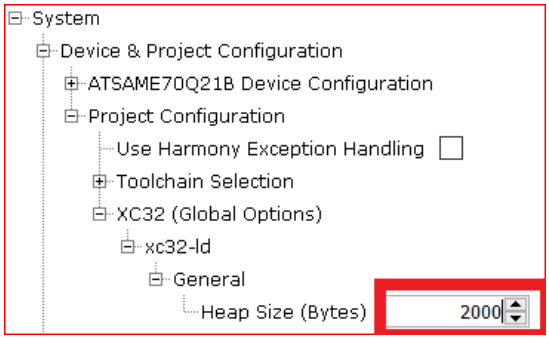

In the MPLAB Harmony Configurator: *File System* dialog*:*

The Auto Mount feature must be selected in order to expose the media type selection. The media type that is being used in this application is Mass Storage Device. This must be correctly configured, or the storage device will not mount.




In the MPLAB Harmony Configurator: *System* dialog:

Set the heap size in Harmony if it is not already set for the linker. Certain projects may set the heap to a larger size automatically.




### *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/audio_player_basic`. To build this project, you must open the `audio/apps/audio_player_basic/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

### MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

| Project Name | BSP Used | Description |
|---|---|---|
| apb_sam_e54_ult_wm8904_i2s_sdmmc | sam_e54_cult | This demonstration runs on the ATSAME54P20A processor on the SAM E54 Curiosity Ultra board with the WM8904 Audio Codec Daughter Board. The WM8904 codec is configured as the master, and the I2S peripheral as the slave. The project config is for reading data from a microSD card. The data is sent to the headphone output of the codec daughter card via I2S driver. |
| apb_sam_e54_ult_wm8904_i2s_sdmmc_freertos | sam_e54_cult | This demonstration runs on the ATSAME54P20A processor on the SAM E54 Curiosity Ultra board with the WM8904 Audio Codec Daughter Board. The WM8904 codec is configured as the master, and the I2S peripheral as the slave. The project config is for reading data from a microSD card. The data is sent to the headphone output of the codec daughter card via I2S driver. This demonstration uses FreeRTOS to schedule running tasks. |
| apb_sam_e70_xult_wm8904_ssc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Daughter Board. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the SSC PLib. |
| apb_sam_e70_xult_wm8904_ssc_freertos | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Daughter Board. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the SSC PLib. This demonstration uses FreeRTOS to schedule tasks to run. |

| | | |
|---|---|---|
| apb_sam_e70_xult_wm8904_ssc_wqvga | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Daughter Board. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the SSC PLib. This demonstration also uses graphics to display status and controls for the board. |

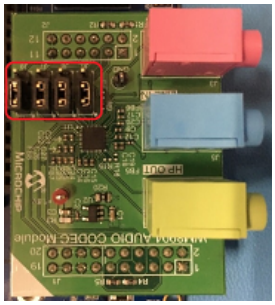
### *Configuring the Hardware*

This section describes how to configure the supported hardware.

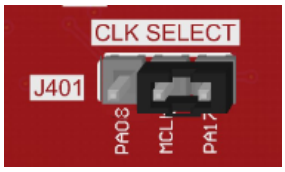
## Description

Using the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board, using the I2S PLIB:

To connect to the I2S, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented away from the pink, mic in, connector. See the red outlined jumpers in the below image as reference.



In addition, make sure the J401 jumper (CLK SELECT) is set for the PA17 pin:

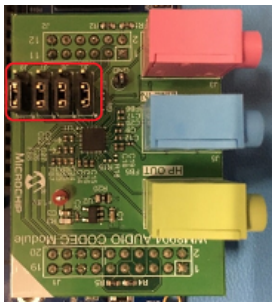



**Note:** The SAM E54 Curiosity Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the SSC PLIB:

Jumper J204, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for LED2.

To connect to the SSC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented away from the pink, mic in, connector. See the red outlined jumpers in the below image as reference.



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

### *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based. Refer to Building the Application for details.

1. Connect headphones to the HP OUT jack of the WM8904 Audio Codec Daughter Board (see **Figure 2 below**).
2. Connect power to the board. The system will be in a wait state for USB to be connected (amber LED1 blinking).
3. For the E54 board:

- Insert a micro SD card into the slot on the bottom side of the board as shown in Figure 1. The contacts of the micro SD card should be facing the bottom side of the board.

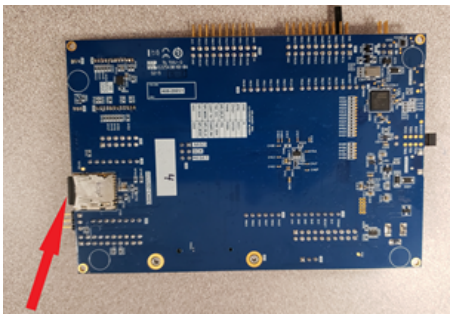


**Figure 1: SD Card slot on bottom of SAM E54 Curiosity Ultra Board**

3. For the E70 board:

- Connect a USB mass storage device (thumb drive) that contains songs of supported audio format to the USB TARGET connector of the SAM E70 Xplained Ultra board. You will probably need a USB-A Female to Micro-B Male adapter cable to do so. The application currently can only stream WAVE (.wav) format audio files.

4. When the device is connected the system will scan for audio files. Once the scanning is complete and at least one file is found (green LED2 on steady), listen to the audio output on headphones connected to the board. Use Switch SW1 as described under Control Description to change the volume or advance to the next track.

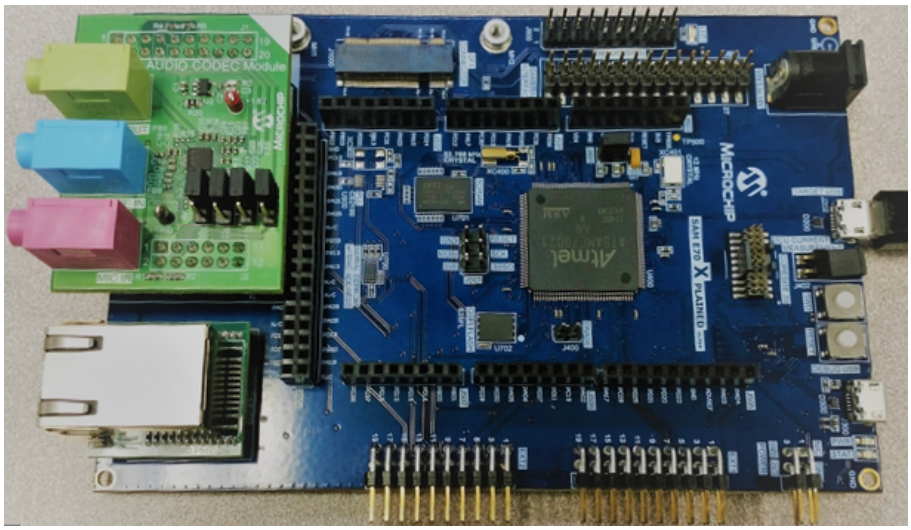



**Figure 2: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board**

### Control Description

Long presses of the push button cycle between volume control and the linear track change mode.

When in volume control mode, short presses of the push button cycle between Low, Medium, High, and Mute volume outputs. While in the Mute mode, a pause of the playback will also take place.

When in the linear track change mode, short presses of the push button will seek to the end of the currently playing track and start the next track that was found in sequence. After all tracks have been played, the first track will start again in the same sequential order.

Button control is shown in the table below.

**Button Operations**

| Long Press (> 1 sec) | Short Press (< 1 sec) |
|---|---|
| Volume Control | Low (-66 dB) |
| | Medium (-48 dB) |
| | High (0 dB) |
| | Mute/Pause |
| Linear Track Change | Next sequential track |

**Status Indicator Description**

When the application first starts running, it looks to find an attached storage device. If one is not found, LED1 will toggle on and off about every 100ms indicating that a storage device is not attached.

When a storage device is attached, LED1 will turn off. At this time, the file system will be scanned for WAVE files with a .wav extension.

If no WAVE files are found on the storage device, LED2 will remain off and scanning of the device will continue.

If any WAVE files are found, LED2 will turn on and playback of the first file found in sequence will start.

LED status indication is shown in the table below.

**LED Status**

| Operation | LED1 Status (Red) | LED2 Status (Green) |
|---|---|---|
| No Storage Device Connected | Toggle 100ms | Off |
| Storage Device Connected | Off | See Files Found Operation |
| Playback: Volume Control | Off | See Files Found Operation |
| Playback: Volume Mute | Toggle 500ms | See Files Found Operation |
| Playback: Linear Track Change | On | See Files Found Operation |
| Files Found (Yes/No) | See above operations | On/Off |

# audio_signal_generator

This topic provides instructions and information about the MPLAB Harmony 3 Audio Signal Generator demonstration application, which is included in the MPLAB Harmony Library distribution.

## Description

In this demonstration application the Codec Driver sets up the WM8904 Codec. The demonstration sends out generated audio waveforms (sine wave, square wave, sawtooth wave, and triangle wave) with volume, duration and frequency modifiable through buttons on a touch screen. Success is indicated by an audible output corresponding to displayed parameters.

The tones can be varied from 50 Hz to 6000 Hz, sent by default at 48,000 samples/second, which is modifiable in the MHC as described below.

To know more about the MPLAB Harmony Codec Drivers, configuring the Codec Drivers, and the APIs provided by the Codec Drivers, refer to Codec Driver Libraries.

### Architecture

SAM E70 Xplained Ultra Projects:

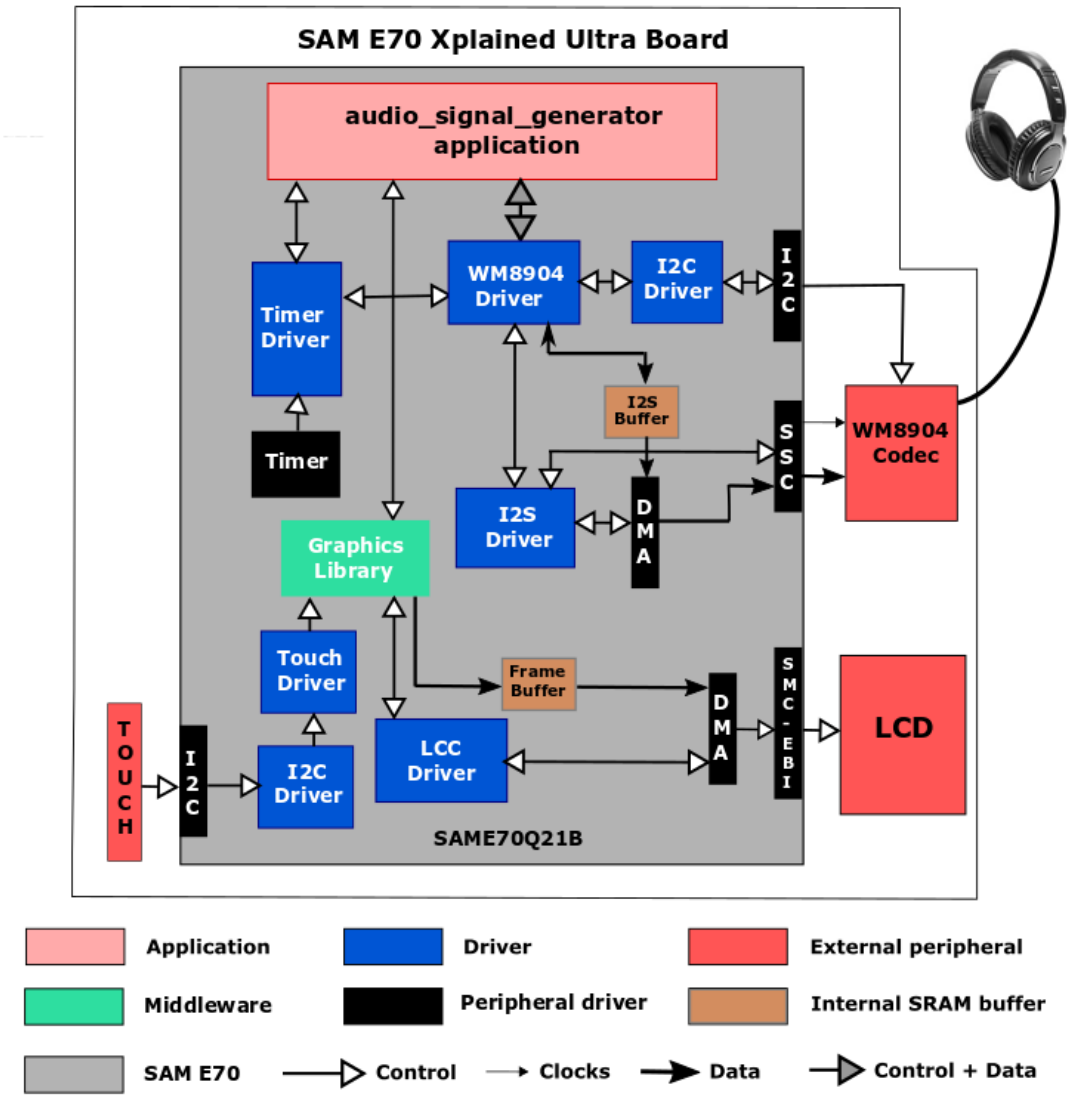
There is one project which runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- WM8904 Codec Daughter Board mounted on a X32 socket
- PDA TM4301B 480x272 (WQVGA) Display

The SAM E70 Xplained Ultra board does not include either the WM8904 Audio Codec daughterboard or the TM4301B graphics card, which are sold separately on microchipDIRECT as part numbers AC328904 and AC320005-4, respectively.

The program takes up to approximately 18% (350 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 87% (332 KB) of the RAM. A 32 KB heap is used.

The following figure illustrates the application architecture for the SAM E70 Xplained Ultra configuration:




The WM8904 codec is configured in master mode, meaning it generates the $I^2S$ clocks (LRCLK and BCLK), and the SSC peripheral is configured as a slave. The I2SC peripheral cannot be used with this project as its interface conflicts with the graphics interface.

The SAM E70 microcontroller (MCU) runs the application code, and communicates with the WM8904 codec via an $I^2C$ interface. The audio interface between the SAM E70 and the WM8904 codec use the $I^2S$ interface. Audio is configured as 16-bit, 48,000 samples/second, $I^2S$ format. (16-bit, 48 kHz is the standard rate used for DVD audio. An alternate that could be used is 44,100 samples/second. This is the same sample rate used for CD's. The sample rate is configurable in the MHC.

The Master Clock (MCLK) signal used by the codec is generated by the Peripheral Clock section of the SAM E70, and is fixed at 12 MHz.

As with any MPLAB Harmony application, the SYS_Initialize function, which is located in the `initialization.c` source file, makes calls to initialize various subsystems as needed, such as the clock, ports, board support package (BSP), display, WM8904 codec, I2S, I2C, DMA, timers, and interrupts.

The codec driver and the application state machines are all updated through calls located in the SYS_Tasks function in the

`tasks.c` file.

The application code is contained in the several source files. The application's state machine (`APP_Tasks`) is contained in `app.c`. It first gets a handle to a timer driver instance and sets up a periodic (alarm) callback. In the next state it gets a handle to the codec driver by calling the `DRV_CODEC_Open` function with a mode of `DRV_IO_INTENT_WRITE` and sets up the volume.

The application state machine then registers an event handler `APP_CODEC_BufferEventHandler` as a callback with the codec driver (which in turn is called by the DMA driver).

Two buffers are used for generating the various waveformse in a ping-pong fashion. Initially values for the first buffer are calculated, and then the buffer is handed off to the DMA using a `DRV_CODEC_BufferAddWrite`. While the DMA is transferring data to the SSC peripheral, causing audio to be sent to the codec over $I^2S$, the program calculates the values for the next cycle. (In the current version of the program, this is always the same unless the frequency is changed manually.) Then when the DMA callback `Audio_Codec_BufferEventHandler` is called, the second buffer is handed off and the first buffer re-initialized, back and forth.

A local routine called initSweep calculates the total number of cycles that are to be generated, based on the starting frequency, ending frequency and sample rate. This results in an average period, which is divided into the duration to yield the the total number of cycles. The cycles for each frequency step is then calculated, and based on this, the number of cycles per buffer is calculated based on the size of the buffer.

A second local routine fillInNumSamplesTable is then called to fill in the current buffer based on the calculated parameters. A table (`appData.numSamples1` or 2) is filled in with the number of samples for each cycle to be generated. This table with the number of samples per cycle to be generated is then passed to the function `APP_TONE_LOOKUP_TABLE_Initialize` along with which buffer to work with (1 or 2) and the sample rate.

For sine waves, the 16-bit value for each sample is calculated based on the relative distance (angle) from 0, based in turn on the current sample number and total number of samples for one cycle. First the angle is calculated in radians:

```
double radians = (M_PI*(double)(360.0/(double)currNumSamples)*(double)i)/180.0;
```

Then the sample value is calculated using the sine function:

```
lookupTable[i].leftData = (int16_t)(0x7FFF*sin(radians));
```

If the number of samples divides into the sample rate evenly, then only 1/4 (90°) of the samples are calculated, and the remainder is filled in by reflection. Otherwise each sample is calculated individually. Before returning, the size of the buffer is calculated based on the number of samples filled in.

For the other waveforms, the 16-bit value of each sample is computed algorithmically based on the shape of the waveform and the position of the sample within the cycle -- square waves, half positive and half negative; sawtooth a steady ramp, and triangle waves, a ramp up followed by a ramp down.

## Demonstration Features

- Calculation of a sine wave based on the number of samples and sample rate using the sin function, with reflection if possible
- Calculation of other waveforms based on the number of samples and sample rate and shape of the waveform.
- Uses the Codec Driver Library to write audio samples to the WM8904
- At a lower level, uses the I2S Driver Library between the codec library and the chosen peripheral (SSC or I2SC) to send the audio to the codec
- Use of ping-pong buffers and DMA
- Use of two timers: one as a periodic 1 ms timer for the application for button debouncing, and a second used by the Codec Driver (see Timer Driver Library)

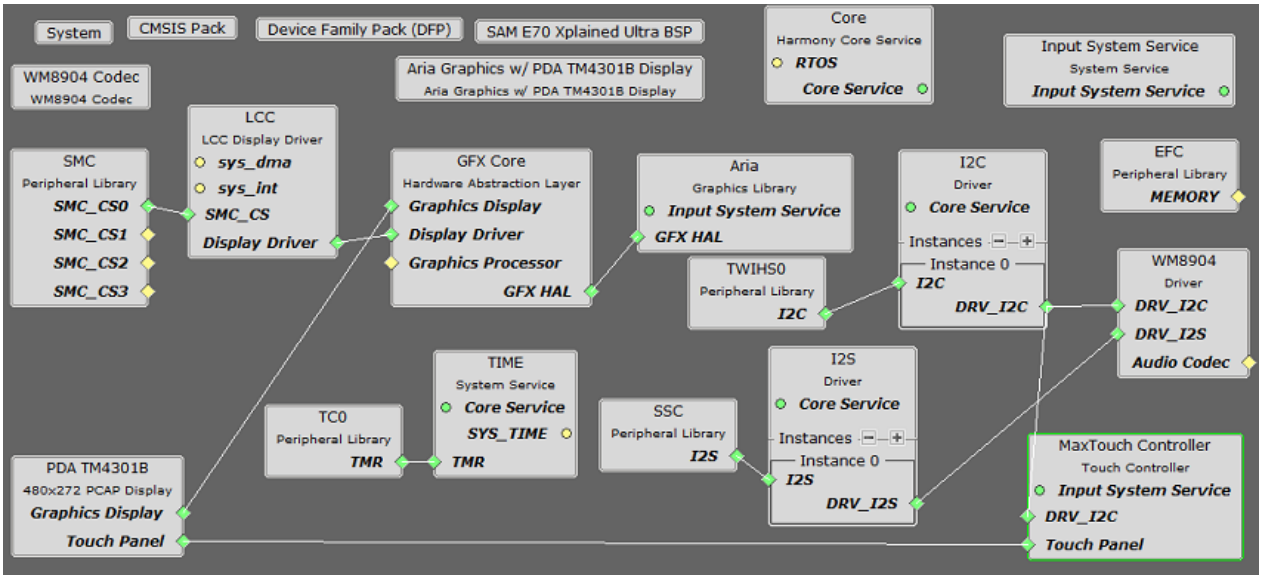
## Tools Setup Differences

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the same based on the BSP used (SAM E70 Xplained Ultra). Select the appropriate processor (ATSAME70Q21B). Click Finish.

In the MHC, under Available Components select the BSP (SAM E70 Xplained Ultra). Under *Graphics>Templates*, double-click on Aria Graphics w/ PDA TM4301B Display. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

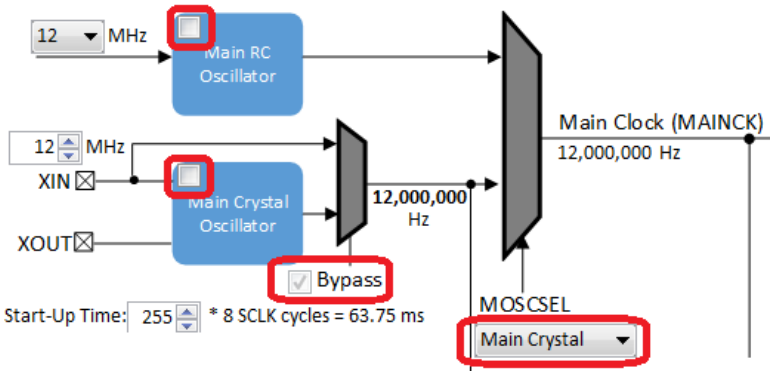
Then under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




Click on the WM8904 Driver. In the Configurations Options, set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz). Then check the SSC checkbox in the **Peripheral Clock Controller** section.

It is also possible to change the audio format from 16 to 32-bits, and from I2S to Left Justified (SSC only). These changes need to be done in the MHC in both the WM8904, and SSC/I2SC Peripherals. In the current application code (app.h), a #define is also set to the current width.

## *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/audio_signal_generator`. To build this project, you must open the `audio/apps/audio_signal_generator/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

**MPLAB X IDE Project Configurations**

The following table lists and describes supported project configurations.

| Project Name | BSP Used | Description |
|---|---|---|
| audio_sig_gen_sam_e70_xult_wm8904_ssc_wqvga | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |

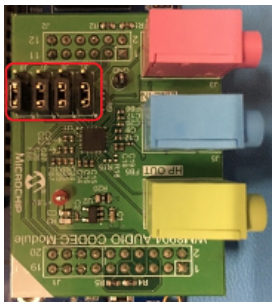
## *Configuring the Hardware*

This section describes how to configure the supported hardware.

### Description

Attach the flat cable of the PDA TM4301B 480x272 (WQVGA) display to the 565 daughterboard connected to the SAM E70 Xplained Ultra board GFX CONNECTOR.

The WM8904 Audio Codec Daughter Board will be using the SSC PLIB; all jumpers on the WM8904 should be toward the **front:**



**Note:** The SAM E70 Xplained Ultra board does not include the either the WM8904 Audio Codec daughterboard or the PDA TM4301B 480x272 (WQVGA) display. They are sold each sold separately on microchipDIRECT as part numbers AC328904 and AC320005-4 respectively.

## *Running the Demonstration*

This section demonstrates how to run the demonstration.

### Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based. Refer to Building the Application for details.

1. Connect headphones to the HP OUT jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).

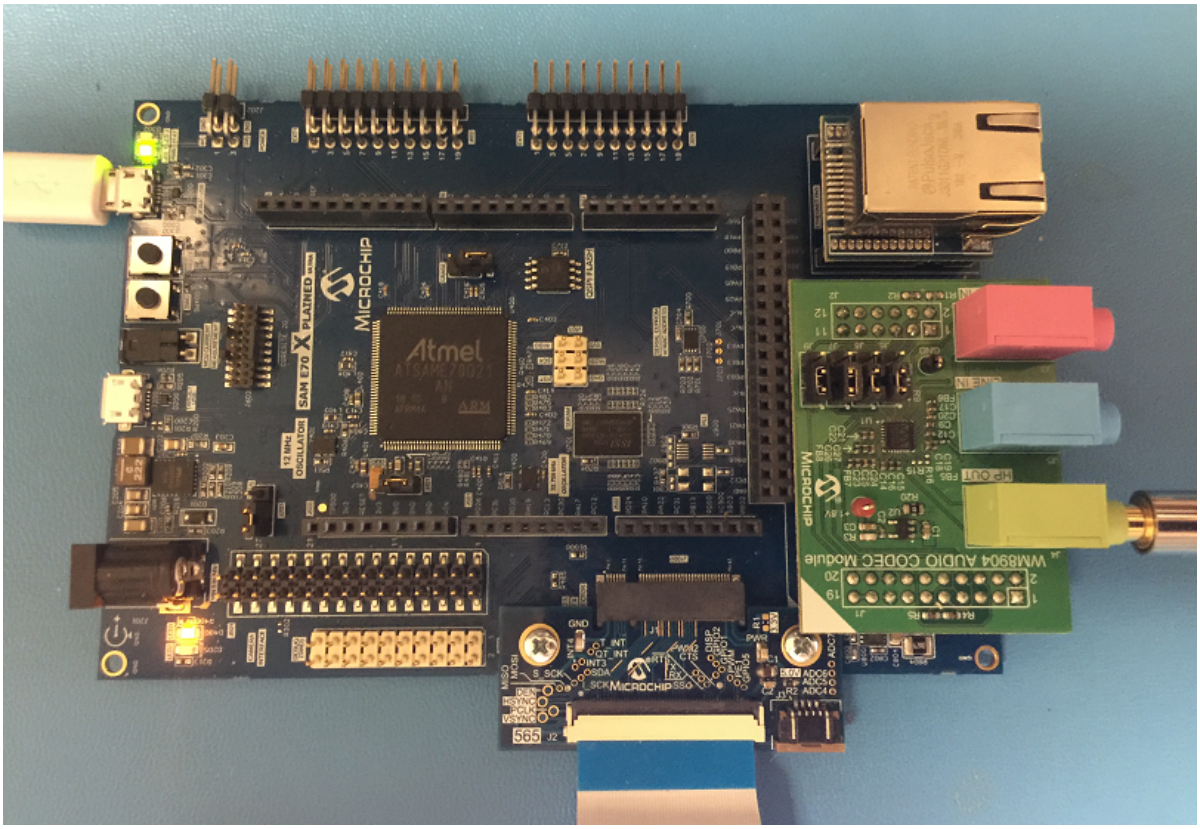
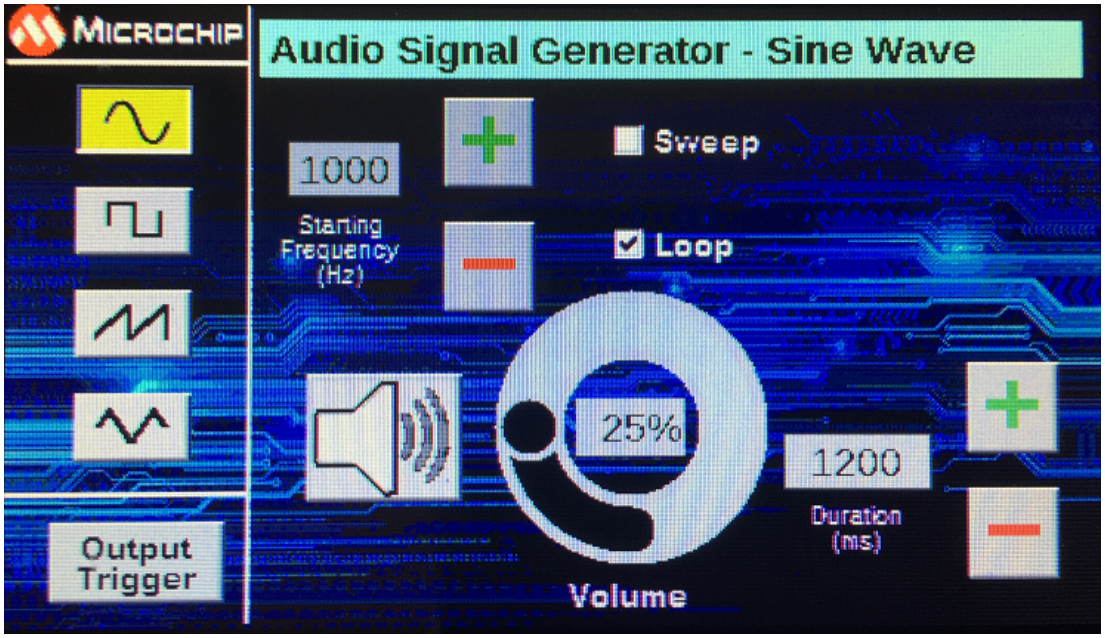
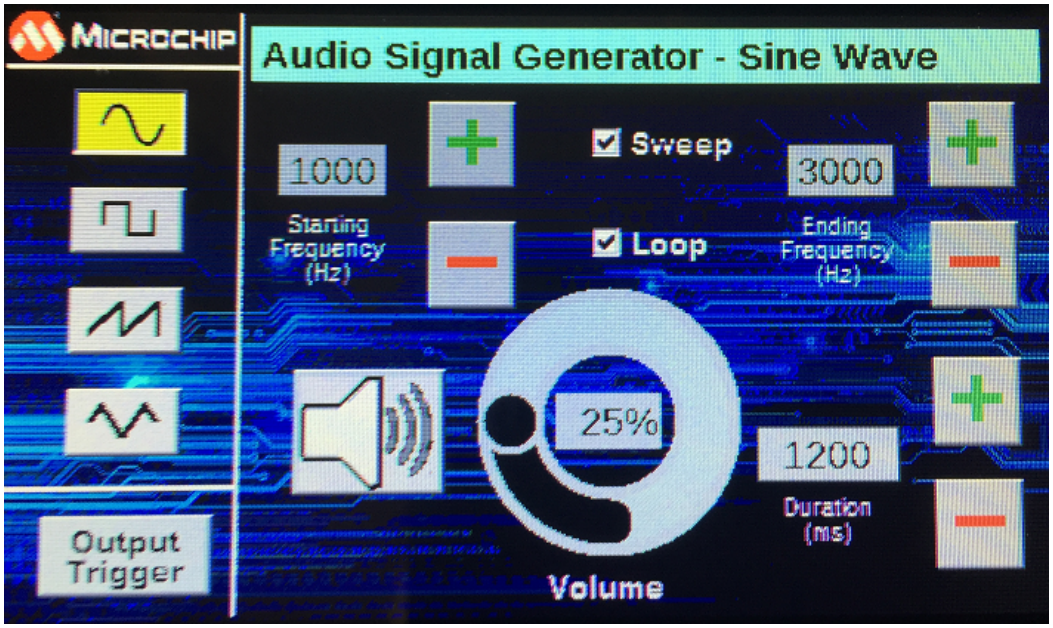



*Fig 1. SAM E70 Xplained Ultra Board with WM8904 Codec Daughterboard*

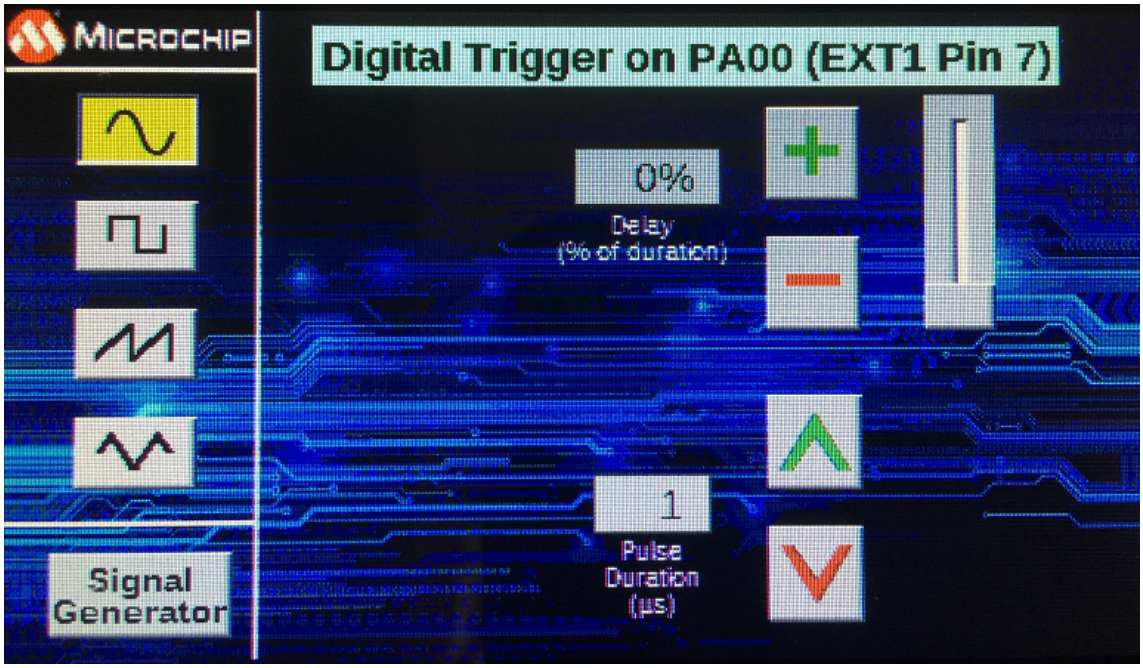
2. Initially the program will be a single-tone continuous sine wave mode, frequency set at 1000 Hz, and a 25% volume level.
3. Turn the tone on and off by tapping on the speaker icon in the center of the screen, to the left of the Volume control. The sound waves will change to a red **X** when enabled.
4. Change the waveform type by pressing one of the four buttons on the left of the screen: sine, square, sawtooth, and triangle wave.
5. Change the volume by rotating the circular control in the middle of the screen.
6. To swtich from a continuous tone to a timed one, tap the Loop checkbox. When unchecked, the tone duration will be determined by the Duration setting in the lower right of the screen. Durations can range from 10 ms to 20000 ms (20 seconds). Pressing the **+** button will increase the duration by 10 ms; pressing the **-** button will decrease by 10 ms. Holding either button down over one second will change it at a faster rate.
7. The frequency can be adjusted using the Starting Frequency control. Pressing the **+** button will increase the frequency by 10 Hz; pressing the **-** button will decrease by 10 Hz. Holding either button down over one second will change it at a faster rate. Some frequencies can not be output exactly.




8. If the Sweep checkbox is tapped so it becomes checked, an Ending Frequency control becomes visible. This used to sweep the frequency from a Starting Frequency to an Ending Frequency over a specified Duration. The frequency can be adjusted the same as for the Starting Frequency control.




9. Pressing the Output Trigger button on the lower left corner of the display switches to a second screen, which is used to control a digital trigger that is output on pin PA00 (pin 7 of the EXT1 connector on the SAM E70 Xplained Ultra board), which by default is 1 µs long. The Pulse Duration control can change the pulse width to either 1, 5, or 10 µs using up and down arrow buttons.
10. By default, the pulse is output at the beginning of a sweep cycle. But this can be delayed using the Delay control, as a per cent of the Duration. Either + / - buttons of the slider can be used to change the Delay.




11. Pressing the Signal Generator button on the lower left corner of the display switches back to the first screen.

## audio_tone

This topic provides instructions and information about the MPLAB Harmony 3 Audio Tone demonstration application, which is included in the MPLAB Harmony Library distribution.

### Description

In this demonstration application the Codec Driver sets up an AK4954 or WM8904 Codec. The demonstration sends out generated audio waveforms (sine tone) with volume and frequency modifiable through the on-board push button. Success is indicated by an audible output corresponding to displayed parameters.

The sine tone is one of four frequencies: 250 Hz, 500 Hz, 1 kHz, and 2 kHz, sent by default at 48,000 samples/second, which is modifiable in the MHC as described below.

To know more about the MPLAB Harmony Codec Drivers, configuring the Codec Drivers, and the APIs provided by the Codec Drivers, refer to Codec Driver Libraries.

#### Architecture

There are nine different projects packaged in this application.

PIC32_MX/MZ Bluetooth Audio Development Kit Projects:

Two projects run on the Bluetooth Audio Development Kit (BTADK). One uses the microcontroller installed on the board, namely the PIC32MX470F512L with 512 KB of Flash memory and 128 KB of RAM running at 96 MHz,and the other uses a PIC32MZ EF PIM, which contains a PIC32MZ2048EFH144 microcontroller with 2 MB of Flash memory and 512 KB of RAM running at 198 MHz. The BTADK includes the following features:

- Six push buttons (SW1-SW6, only SW1 is used)
- Five LEDs (D5-D9, only D5 is used)
- AK4954 Codec Daughter Board mounted on the rear X32 socket

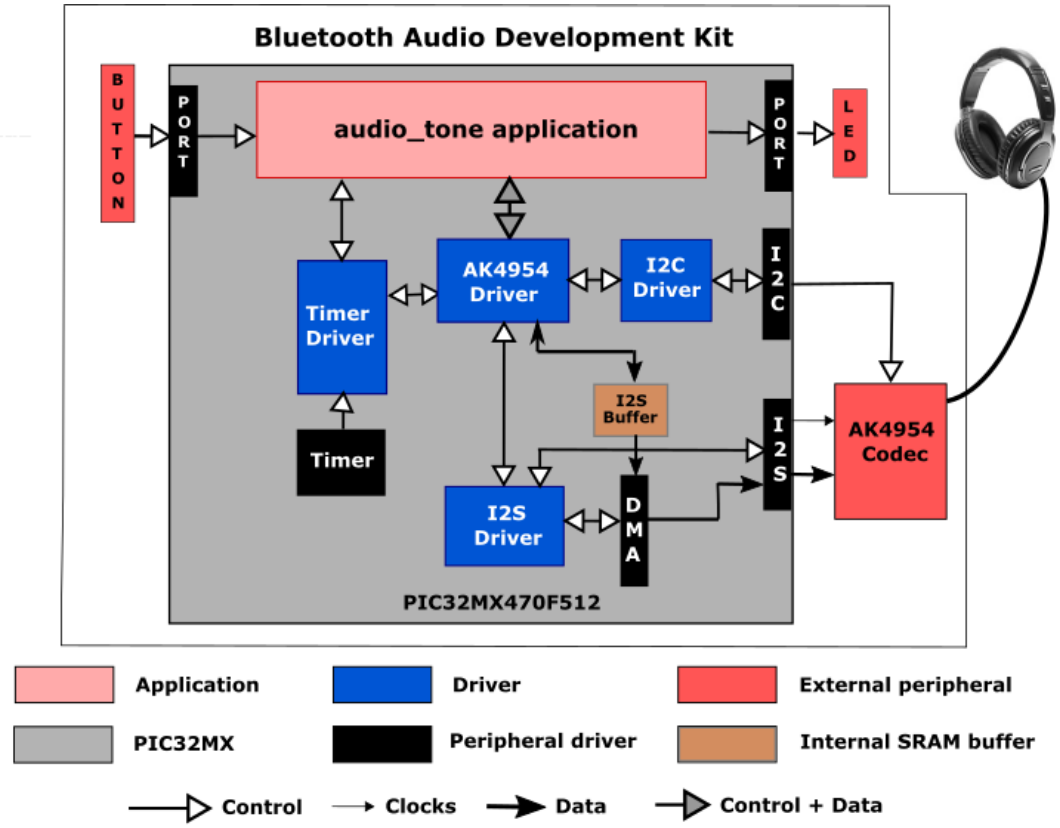
The BTADK also includes an LCD display, which is not used in these projects.

The BTADK does not include the AK4954 Audio Codec daughterboard or the PIC32MZ EF Audio PIM, which are sold separately on microchipDIRECT as part number AC324954 and MA320018 respectively.

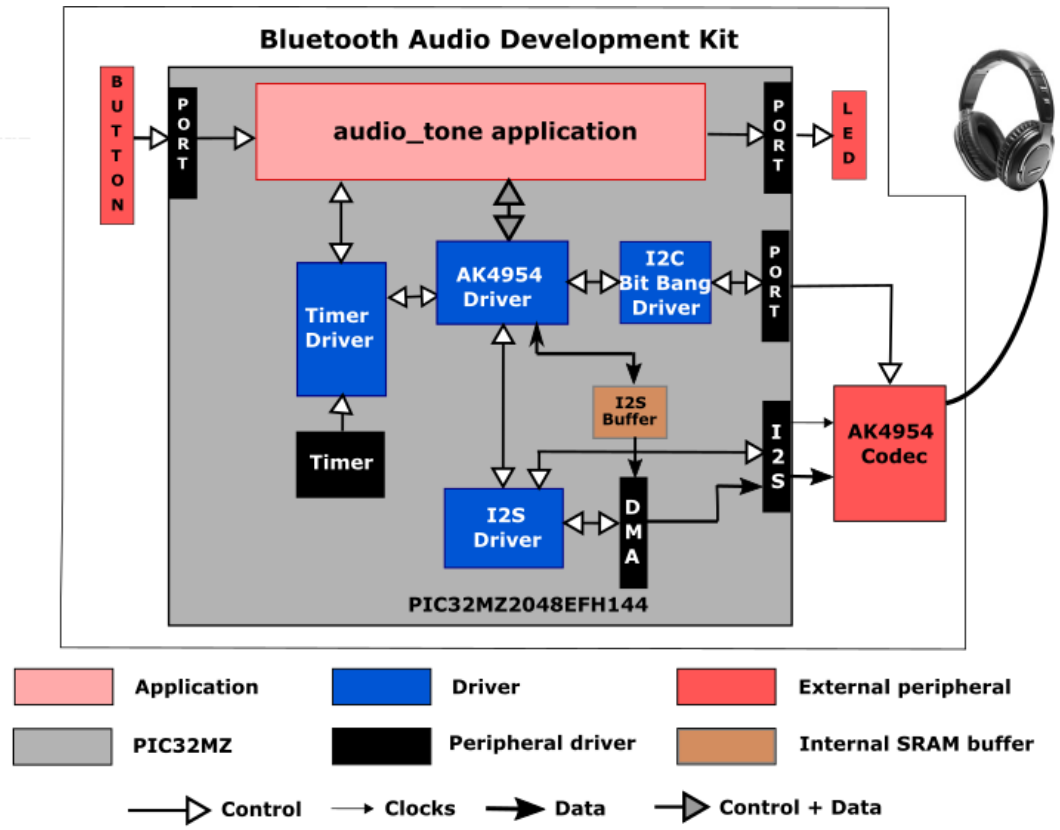
For the version using the PIC32MX470F512L microcontroller, the program takes up to approximately 9% (50 KB) of the microcontroller's program space, and 89% (114 KB) of the RAM. No heap is used. For the version using the PIC32MZ2048EFH144 microcontroller, the program takes up to approximately 3% (55 KB) of the microcontroller's program space,

and 23% (115 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the two PIC32 Bluetooth Audio Development Kit configurations:




The PIC32MX470F512L configuration (above) uses the hardware I2C driver, whereas the PIC32MZ2048EFH144 configuration (below) uses the I2C Bit-Banged Library.




The I2S (Inter-IC Sound Controller) is used with the AK4954 codec. The AK4954 is configured in slave mode, meaning it receives

$I^2S$ clocks (LRCLK and BCLK) from the PIC32, and the I2S peripheral is configured as a master.

PIC32 MZ EF Curiosity 2.0 Project:

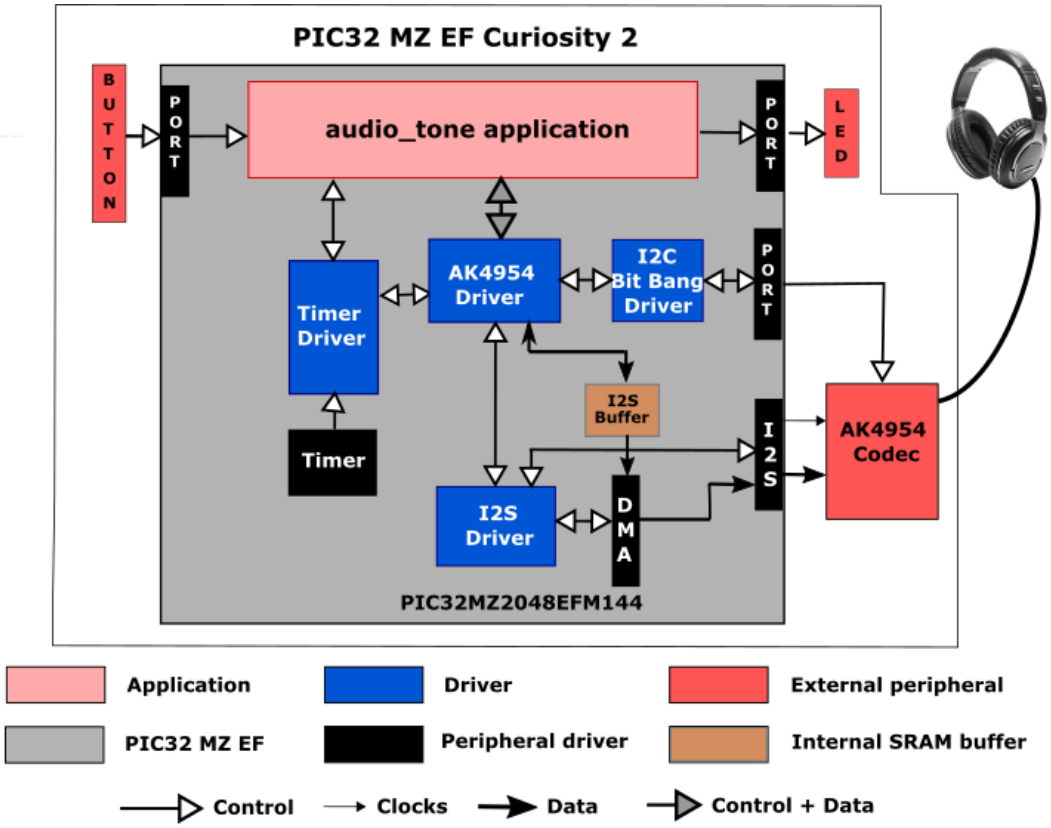
One project runs on the PIC32 MZ EF Curiosity 2.0 board, using the PIC32MZ2048EFM144 microcontroller with 2 MB of Flash memory and 512 KB of RAM running at 198 MHz. The PIC32 MZ EF Curiosity 2.0 board includes the following features:

- Four push buttons (SW1-SW4, only SW1 is used)
- Four LEDs (LED1-LED4, only LED1 is used)
- AK4954 Codec Daughter Board mounted on X32 HEADER 2 socket

The PIC32 MZ EF Curiosity 2.0 board does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

The program takes up to approximately 3% (55 KB) of the PIC32MZ2048EFM144 microcontroller's program space, and 23% (115 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the PIC32 MZ EF Curiosity 2.0 configuration:




For both configurations, the I2S (Inter-IC Sound Controller) is used with the AK4954 codec. The AK4954 is configured in slave mode, meaning it receives $I^2S$ clocks (LRCLK and BCLK) from the PIC32, and the I2S peripheral is configured as a master.

SAM E54 Curiosity Ultra Projects:

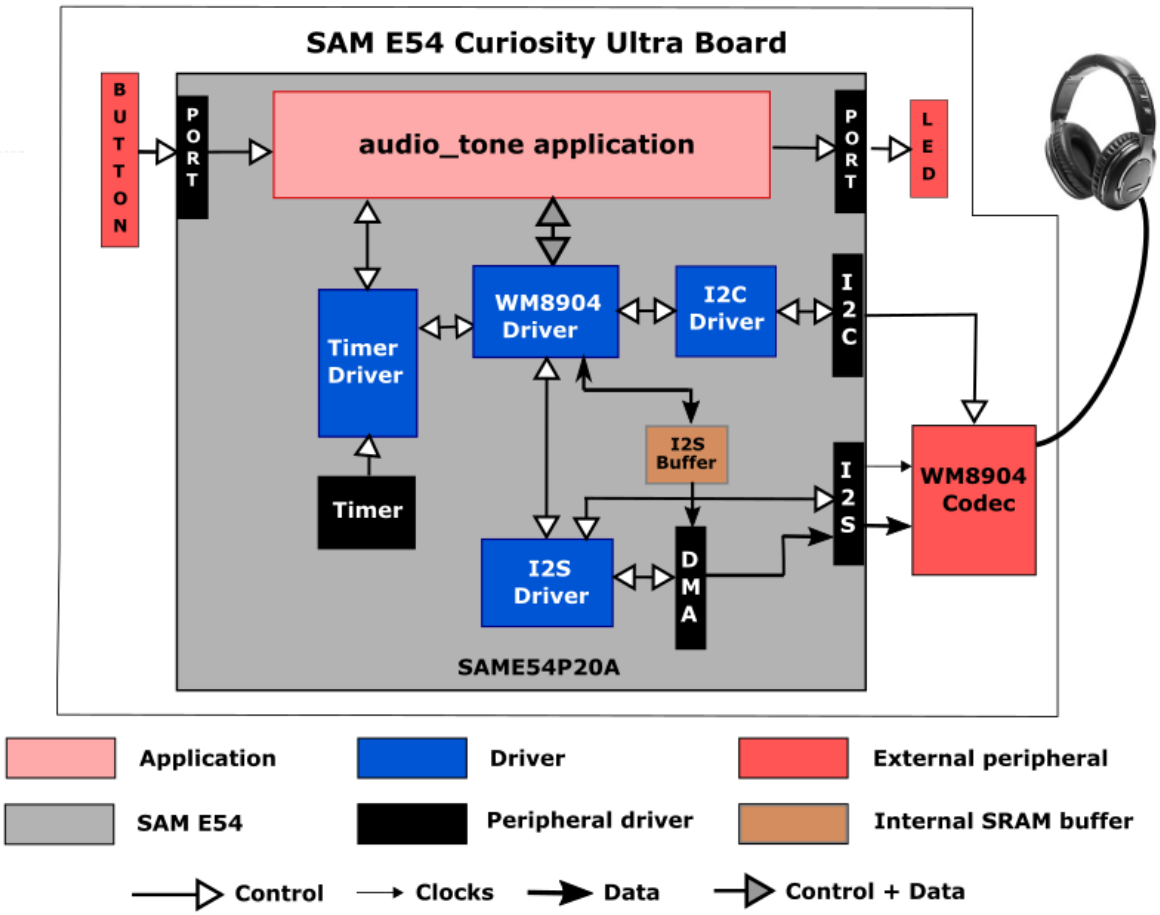
Two projects run on the SAM E54 Curiosity Ultra Board, which contains a ATSAME54P20A microcontroller with 1 MB of Flash memory and 256 KB of RAM running at 48 MHz using the following features:

- Two push buttons (SW1 and SW2, only SW1 is used)
- Two LEDs (LED1 and 2, only LED1 is used)
- WM8904 Codec Daughter Board mounted on a X32 socket

The SAM E54 Curiosity Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The non-RTOS version of the program takes up to approximately 3% (28 KB) of the ATSAME54P20A microcontroller's program space. The 16-bit configuration uses 45% (115 KB) of the RAM. No heap is used. For the FreeRTOS project, the program takes up to approximately 1% (22 KB) of the ATSAME54P20A microcontroller's program space, and the 16-bit configuration uses 60% (155 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the two SAM E54 Xplained Ultra configurations (RTOS not shown):




The I2S (Inter-IC Sound Controller) is used with the WM8904 codec. The WM8904 is configured in master mode, meaning it generates the $I^2S$ clocks (LRCLK and BCLK), and the I2S peripheral is configured as a slave.

SAM E70 Xplained Ultra Projects:

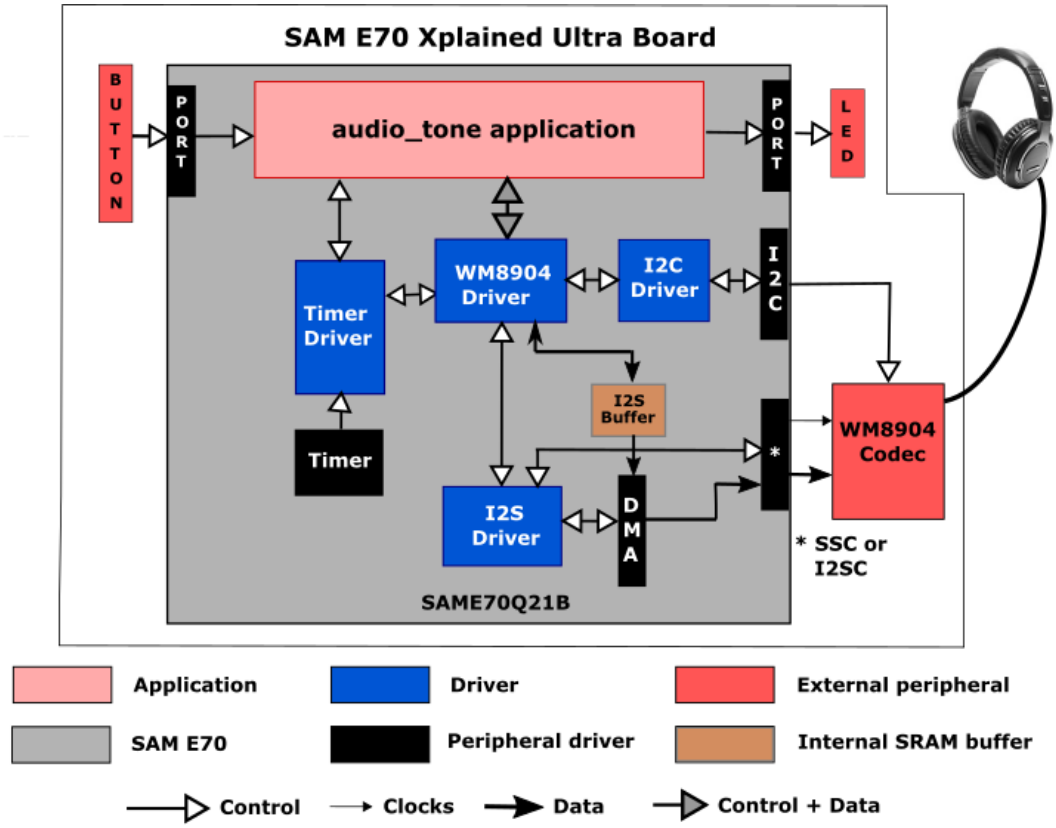
Four projects run on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- One push button (SW1, may be labeled as SW0 on some boards)
- Two LEDs (LED1 and 2, only LED1 is used)
- AK4954 or WM8904 Codec Daughter Board mounted on a X32 socket

The SAM E70 Xplained Ultra board does not include the AK4954 or WM8904 Audio Codec daughterboards, which are sold separately on microchipDIRECT as part numbers AC324954 and AC328904, respectively.

The two non-RTOS versions of the program take up to approximately 1% (18 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 30% (115 KB) of the RAM. No heap is used. For the FreeRTOS project, the program takes up to approximately 1% (22 KB) of the ATSAME70Q21B microcontroller's program space, and the 16-bit configuration uses 41% (155 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the three SAM E70 Xplained Ultra configurations using the WM8904 (RTOS not shown). The AK4954 version is the same with the substitution of the AK4954 Driver and Codec blocks.




The AK4954 codec is only used with the SSC (Synchronous Serial Controller). The WM8904 is configured in slave mode and the SSC peripheral is a master and generates the I2SC clocks.

Depending on the project, either the SSC (Synchronous Serial Controller) or I2SC (Inter-IC Sound Controller) is used with the WM8904 codec, selected by a strapping option on the WM8904 daughterboard. When using the SSC interface, the WM8904 is configured in master mode, meaning it generates the $I^2S$ clocks (LRCLK and BCLK), and the SSC peripheral is configured as a slave. When using the I2SC interface, the WM8904 is configured in slave mode and the SSC peripheral is a master and generates the I2SC clocks. The other two possibilities (SSC as master and WM8904 as slave, or I2SC as slave and WM8904 as master) are possible, but not discussed.

SAMV71 Xplained Ultra Project:

One project runs on the SAMV71 Xplained Ultra Board, which contains a ATSAMV71Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- Two push buttons (SW0 and SW1, only SW0 is used)
- Two LEDs (LED0 and 1, only LED0 is used)
- WM8904 codec (on board)

The program takes up to approximately 1% (18 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 30% (115 KB) of the RAM (with 15K of that used by the three audio buffers). No heap is used.

The architecture is the same as for the SAM E70 Ultra board configurations; however only the SSC is available.

The same application code is used without change between the seven projects.

The PIC32 or SAM microcontroller (MCU) runs the application code, and communicates with the AK4954 or WM8904 codec via an $I^2C$ interface. The audio interface between the microcontroller and the codec use the $I^2S$ interface. Audio is configured as 16-bit, 48,000 samples/second, $I^2S$ format. (16-bit, 48 kHz is the standard rate used for DVD audio. An alternate that could be used is 44,100 samples/second. This is the same sample rate used for CD's. The sample rate is configurable in the MHC.)

The Master Clock (MCLK) signal used by the codec is generated by the Peripheral Clock section of the microcontroller, and is fixed at 12 MHz.

The button and LED are interfaced using GPIO pins. There is no screen.

As with any MPLAB Harmony application, the SYS_Initialize function, which is located in the `initialization.c` source file, makes calls to initialize various subsystems as needed, such as the clock, ports, board support package (BSP), codec, I2S, I2C, DMA, timers, and interrupts.

The codec driver and the application state machines are all updated through calls located in the SYS_Tasks function in the `tasks.c` file.

The application code is contained in the several source files. The application's state machine (`APP_Tasks`) is contained in `app.c`. It first initializes the application, which includes `APP_Tasks` then periodically calls `APP_Button_Tasks` to process any pending button presses.

Then the application state machine inside `APP_Tasks` is given control, which first gets a handle to a timer driver instance and sets up a periodic (alarm) callback. In the next state it gets a handle to the codec driver by calling the `DRV_CODEC_Open` function with a mode of `DRV_IO_INTENT_WRITE` and sets up the volume.

The application state machine then registers an event handler `APP_CODEC_BufferEventHandler` as a callback with the codec driver (which in turn is called by the DMA driver).

Two buffers are used for generating a sine wave in a ping-pong fashion. Initially values for the first buffer are calculated, and then the buffer is handed off to the DMA using a `DRV_CODEC_BufferAddWrite`. While the DMA is transferring data to the SSC or I2SC/I2S peripheral, causing the tone to sent to the codec over $I^2S$, the program calculates the values for the next cycle. (In the current version of the program, this is always the same unless the frequency is changed manually.) Then when the DMA callback `Audio_Codec_BufferEventHandler` is called, the second buffer is handed off and the first buffer re-initialized, back and forth.

A table called `samples` contains the number of samples for each frequency that correspond to one cycle of audio (e.g. 48 for 48000 samples/sec, and 1 kHz tone). This is divided into the `MAX_AUDIO_NUM_SAMPLES` value (maximum number of elements in the tone) to provide the number of cycles of tone to be generated to fill the table. Another table (`appData.numSamples1` or 2) is then filled in with the number of samples for each cycle to be generated. **Note:** the `samples` table will need to be modified if changing the sample rate to something other than 48000 samples/second.

This table with the number of samples per cycle to be generated is then passed to the function `APP_TONE_LOOKUP_TABLE_Initialize` along with which buffer to work with (1 or 2) and the sample rate. The 16-bit value for each sample is calculated based on the relative distance (angle) from 0, based in turn on the current sample number and total number of samples for one cycle. First the angle is calculated in radians:

```
double radians = (M_PI*(double)(360.0/(double)currNumSamples)*(double)i)/180.0;
```

Then the sample value is calculated using the sine function:

```
lookupTable[i].leftData = (int16_t)(0x7FFF*sin(radians));
```

If the number of samples divides into the sample rate evenly, then only 1/4 (90°) of the samples are calculated, and the remainder is filled in by reflection. Otherwise each sample is calculated individually. Before returning, the size of the buffer is calculated based on the number of samples filled in.

## Demonstration Features

- Calculation of a sine wave based on the number of samples and sample rate using the sin function, with reflection if possible
- Uses the Codec Driver Library to write audio samples to the AK4954 or WM8904
- At a lower level, uses the I2S Driver Library between the codec library and the chosen peripheral (SSC or I2SC or I2S) to send the audio to the codec
- Use of ping-pong buffers and DMA
- Use of two timers: one as a periodic 1 ms timer for the application for button debouncing, and a second used by the Codec Driver (see Timer Driver Library)

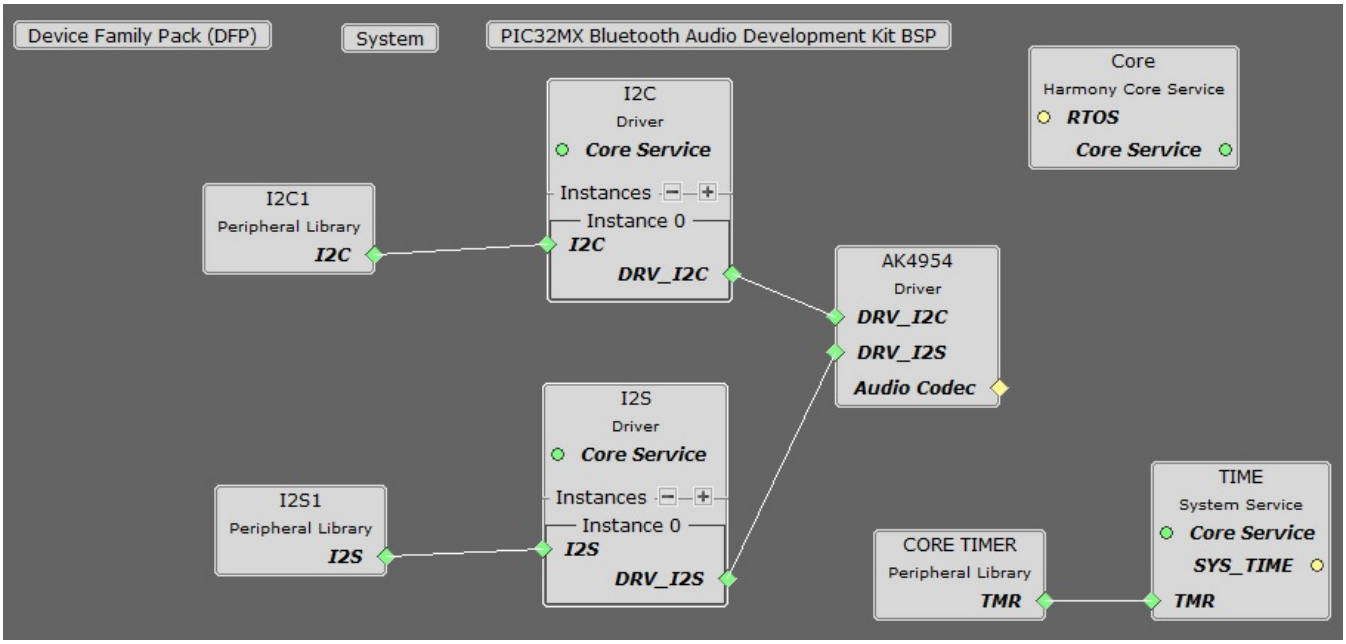
## Tools Setup Differences

For the projects using the PIC32MX and the I2S interface and the AK4954 as a Slave (the I2S peripheral generates the $I^2S$ clocks):

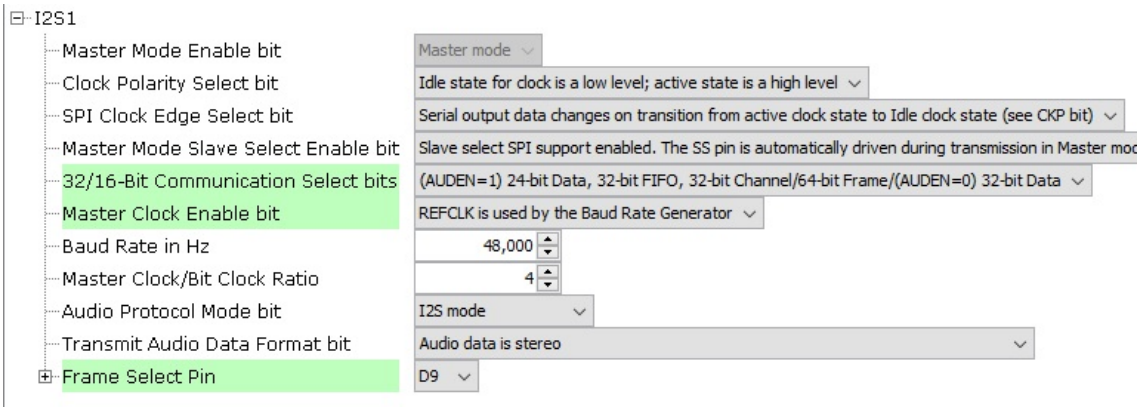
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File* > *New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (PIC32MX470F512L). Click Finish.

In the MHC, under Available Components select the BSP PIC32MX Bluetooth Audio Development Kit. Under *Audio*>*Templates*, double-click on AK4954 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

You should end up with a project graph that looks like this, after rearranging the boxes:



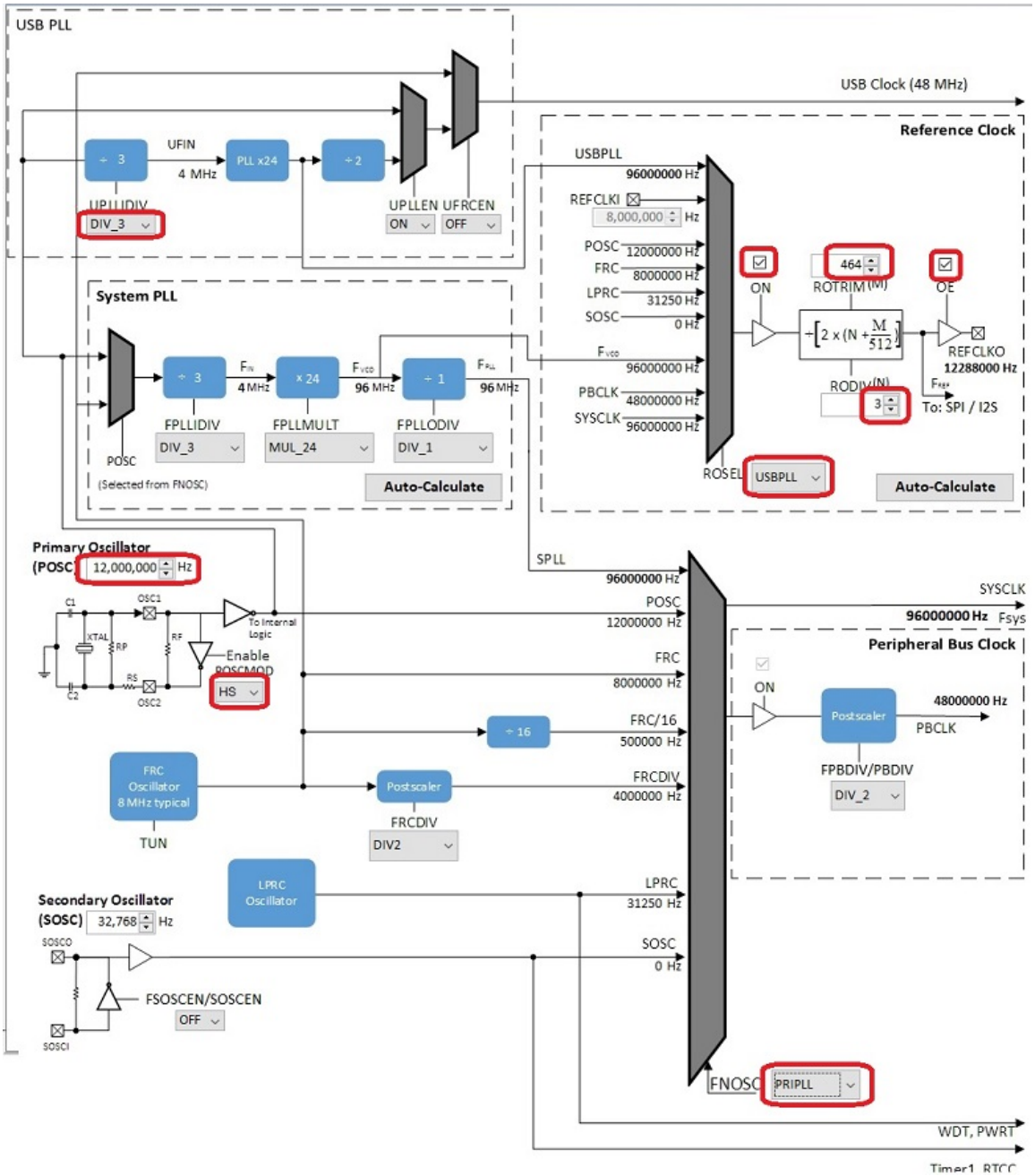

Click on the I2S1 Peripheral. In the Configurations Options, under 32/16-Bit Communication Select bits, select (AUDEN=1) 24-bit data, 32-bit FIFO, 32-bit Channel/64-bit Frame. Change Master Clock Enable bit to REFCLK. Set the Frame Select Pin to D9. The configuration should look like this:




Under Tools, click on Clock Configuration. In the Clock Diagram:

- Change the Primary Oscillator frequency to 12,000,000 Hz, and select HS under POSCMOD.
- Change the UPPLLDIV divider to DIV_3.
- Change the FNOSC input to PRIPLL.
- In the Reference CLock section, check the ON and OE checkboxes, change the input to USBPLL, and set ROTRIM to 464 and RODIV to 3. This should give a REFCLKO output of 12,288,000 Hz, which is 256 times the 48 KHz sample rate.

You should end up with a clock diagram like this:

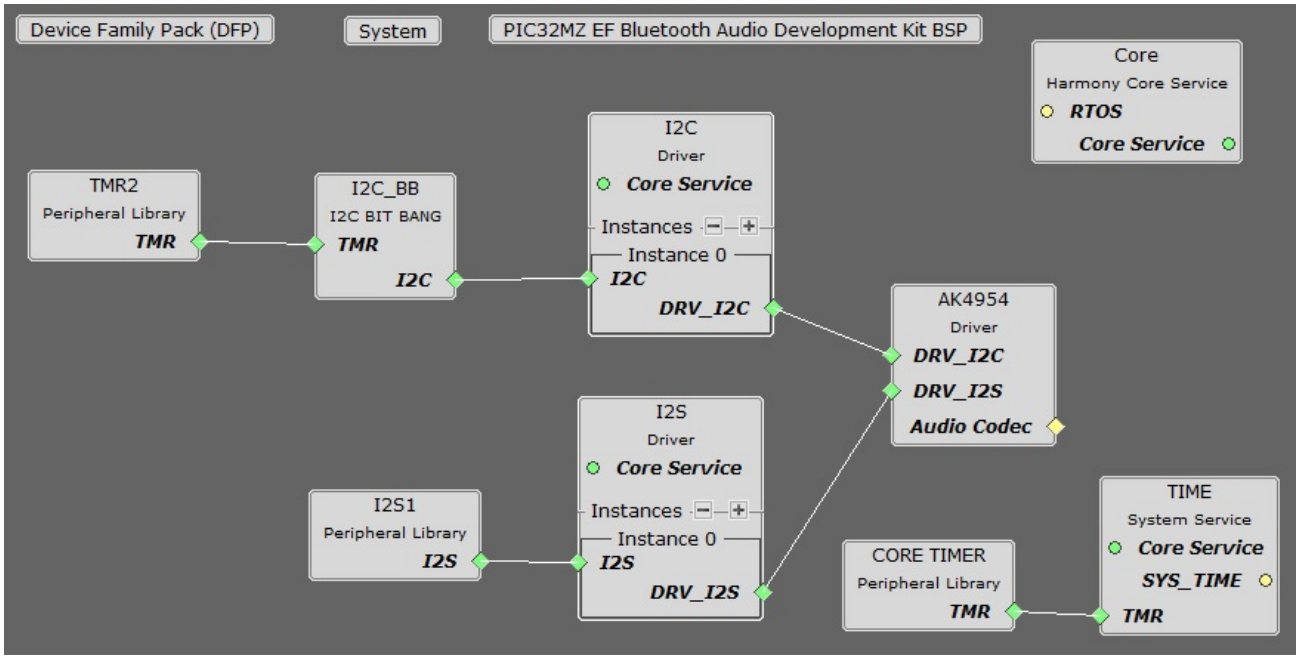



For the projects using the PIC32MZ, the I2S interface and the AK4954 as a Slave (the I2S peripheral generates the $I^2S$ clocks):

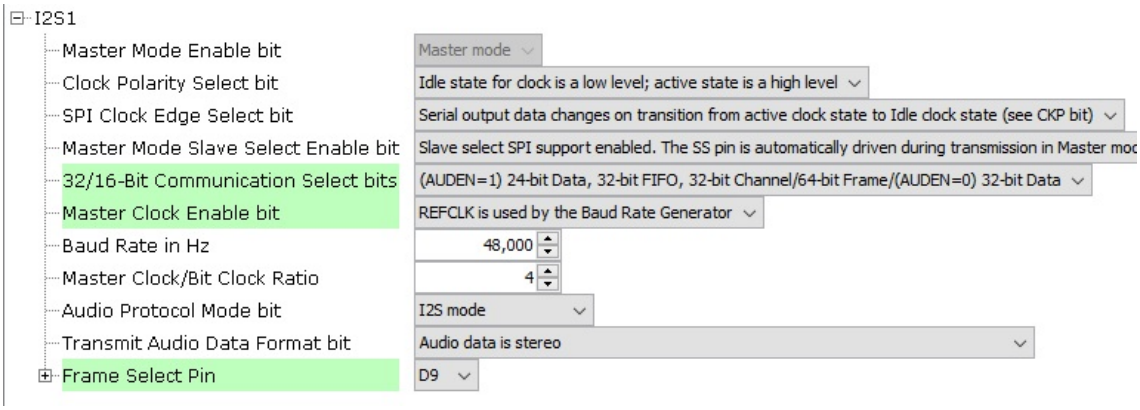
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (PIC32MZ2048EFH144 for the BTADK or PIC32MZ2048EFM144 for the PIC32 MZ EF Curiosity 2.0). Click Finish.

In the MHC, under Available Components select the BSP PIC32MZ Bluetooth Audio Development Kit or PIC32 MZ EF Curiosity 2.0. Under *Audio>Templates*, double-click on AK4954 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

You should end up with a project graph that looks like this, after rearranging the boxes:




The diagram above is for the PIC32 MZ EF PIM and BTADK; the one for the PIC32 MZ EF Curiosity 2.0 is almost the same except it uses I2S2 instead of I2S1. Click on the I2S Peripheral. In the Configurations Options, under 32/16-Bit Communication Select bits, select (AUDEN=1) 24-bit data, 32-bit FIFO, 32-bit Channel/64-bit Frame. Change Master Clock Enable bit to REFCLK. Set the Frame Select Pin to D9 for the BTADK or C2 for the Curiosity 2.0 board. The configuration should look like this (i2S1/BTADK version shown):

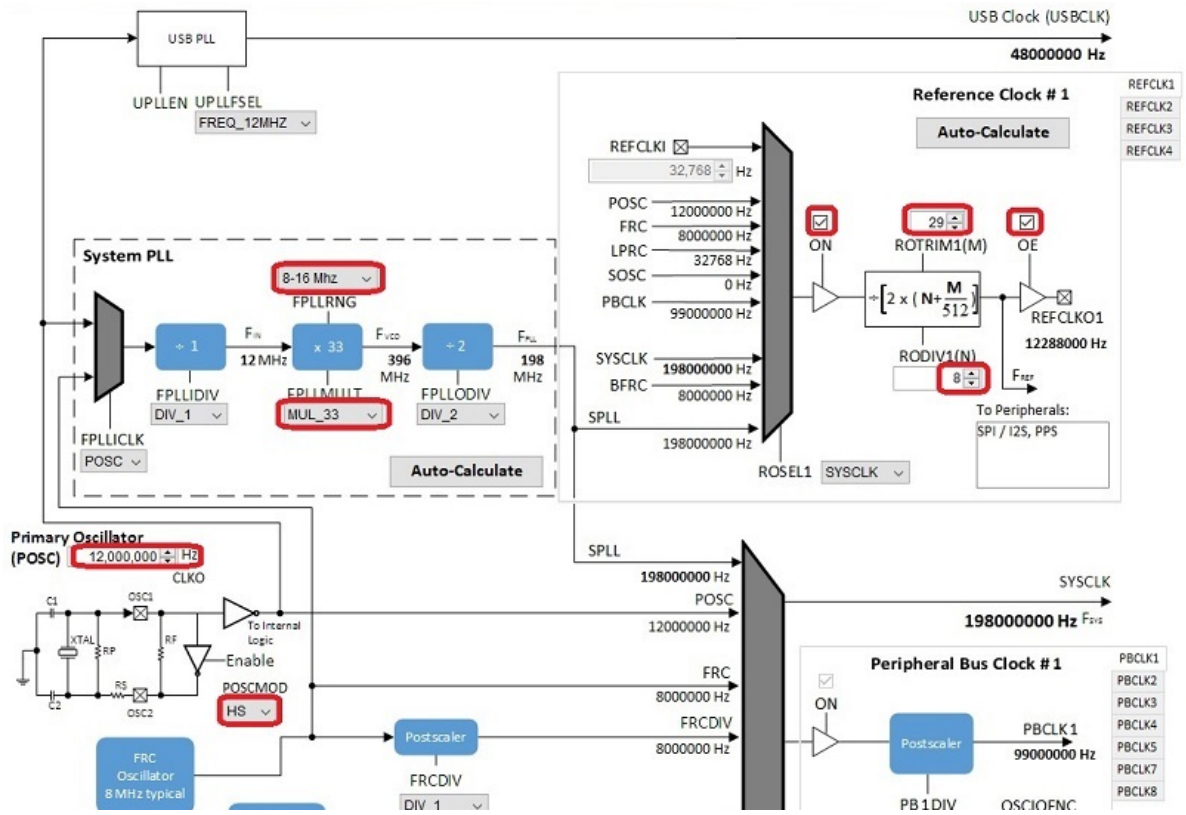



The Curiosity 2.0 looks the same except for the Frame Select Pin being C2.

Under Tools, click on Clock Configuration. In the Clock Diagram:

- Change the Primary Oscillator frequency to 12,000,000 Hz, and select HS under POSCMOD.
- Change the FPLLRNG to 8-16 MHz, and FPLLMULT to MUL_33.
- In the Reference CLock section, check the ON and OE checkboxes,, and set ROTRIM1 to 29 and RODIV1 to 8. This should give a REFCLKO1 output of 12,288,000 Hz, which is 256 times the 48 KHz sample rate.

You should end up with a clock diagram like this:

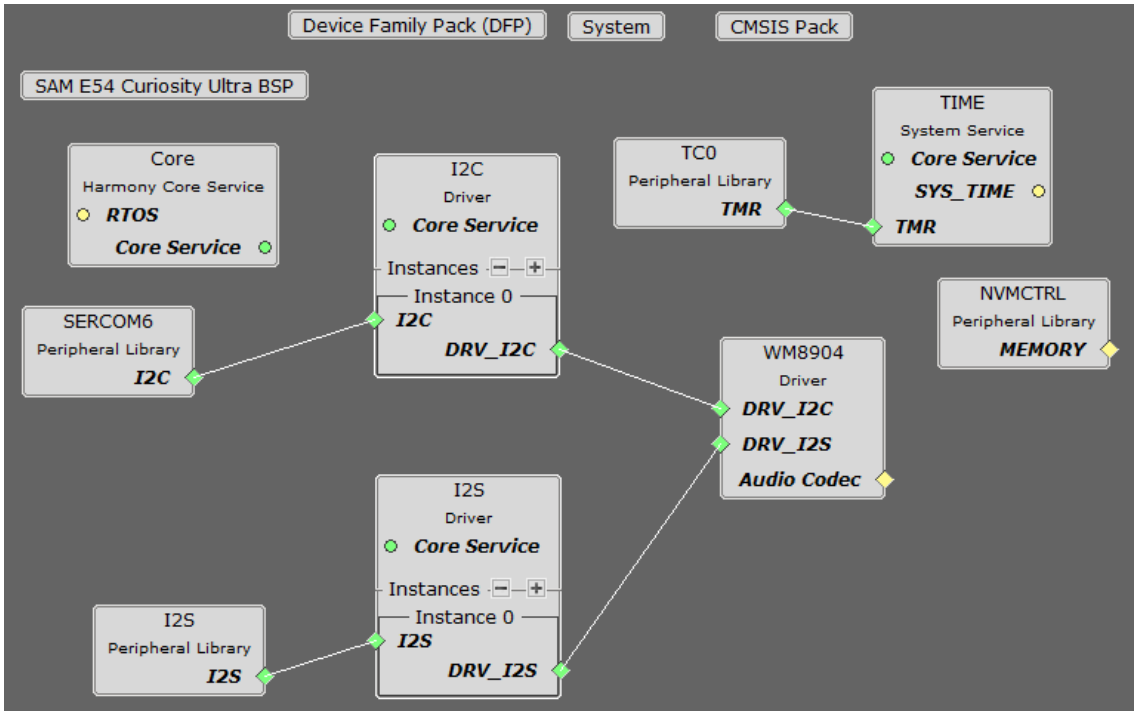



For projects using the E54, the I2S interface and the WM8904 as a Master (the WM8904 codec generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME54P20A). Click Finish.

In the MHC, under Available Components select the BSP SAM E54 Curiosity Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes:



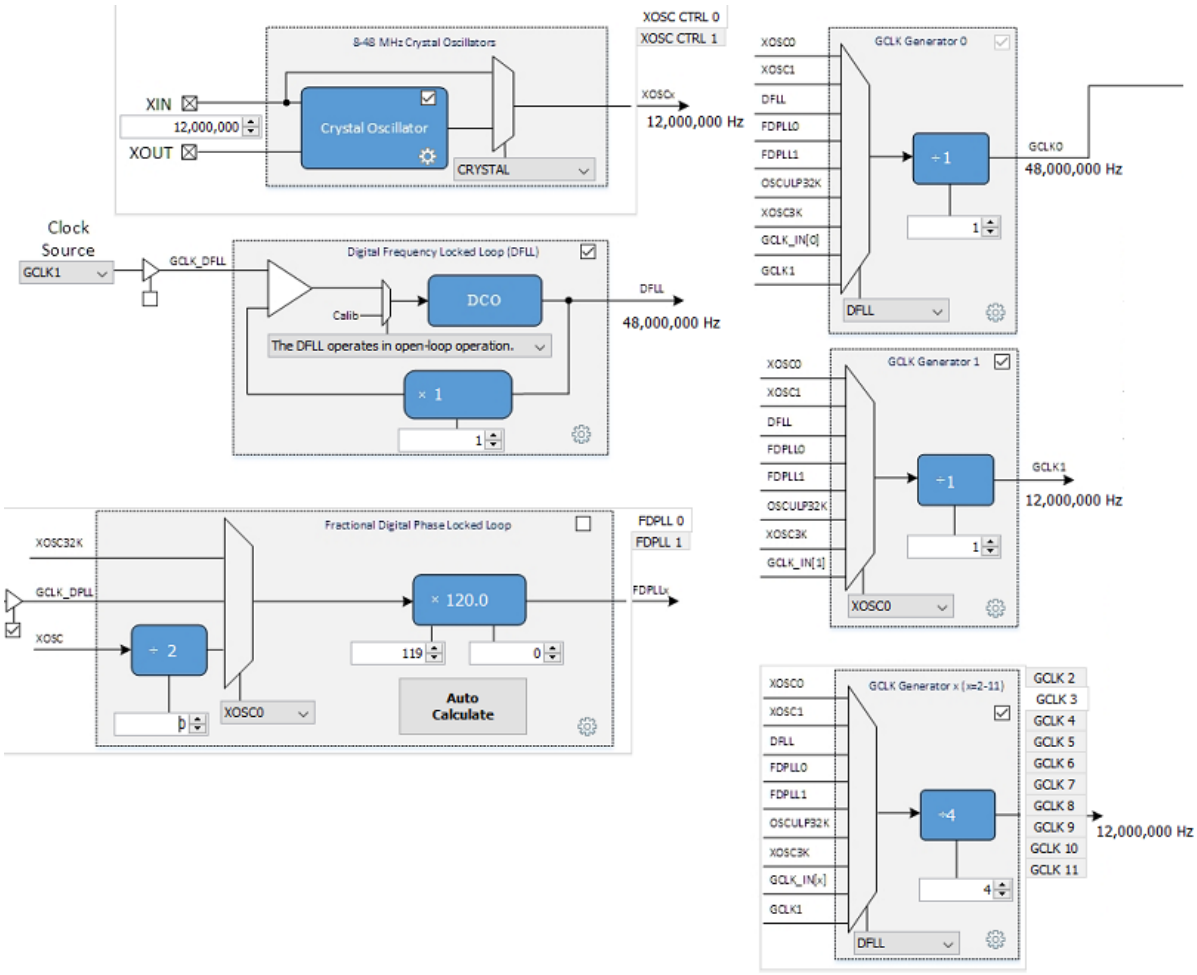

Click on the WM8904 Driver. In the Configurations Options, under Audio Data Format, change it to 32-bit I2S. Set the desired

Sample Rate if different from the default (48,000) under Sampling Rate.

In the Clock Diagram:

- Enable the Crystal Oscillator, change it frequency to 12,000,000 Hz, and select CRYSTAL.
- Uncheck the Fractional Digital Phase Locked Loop enable (FDPLL 0).
- In the CLK Generator 0 box, change the input to DFLL for an output of 48 MHz.
- In the CLK Generator 1 box, change the input to XOSC0 with a divider of 1 for an output of 12 MHz.
- In the GCLK Generator, uncheck the selection for GCLK 2, and then select the GCLK 3 tab. Choose the DFLL as the input, with a divide by 4 for an output of 12 MHz.

You should end up with a clock diagram like this:

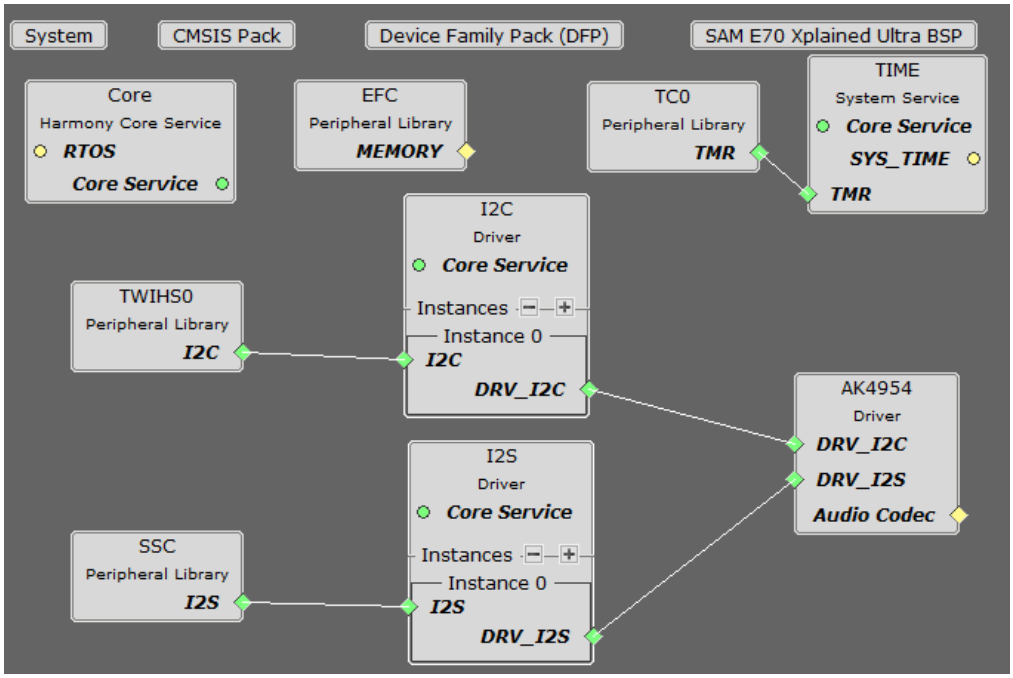



For projects using the E70, the SSC interface and the AK4954 as a Slave (the SAM E70 generates the I$^2$S clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME70Q21B). Click Finish.

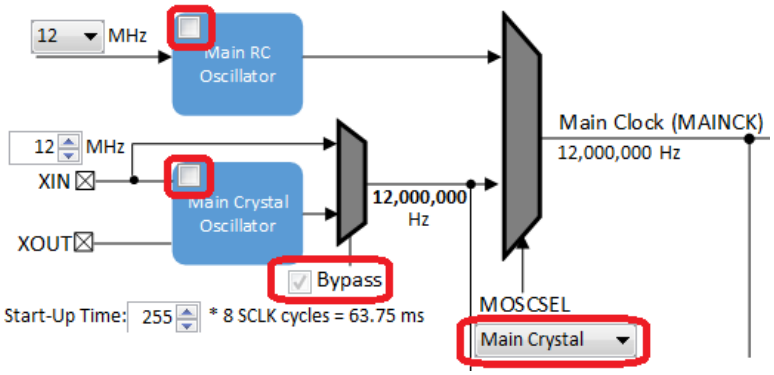
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on AK4954 Codec. Answer Yes to all questions. Click on the AK4954 Codec component (*not* the AK4954 Driver). In the Configuration Options, under AK4954 Interface, select I2SC instead of SSC. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes:

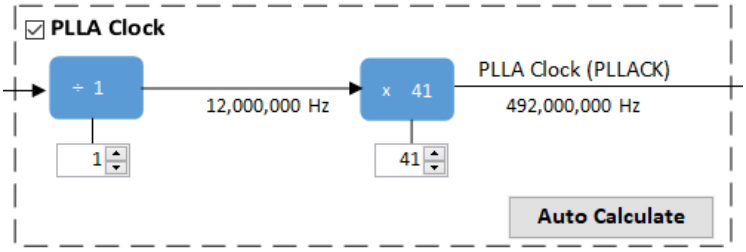



Click on the AK4954 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

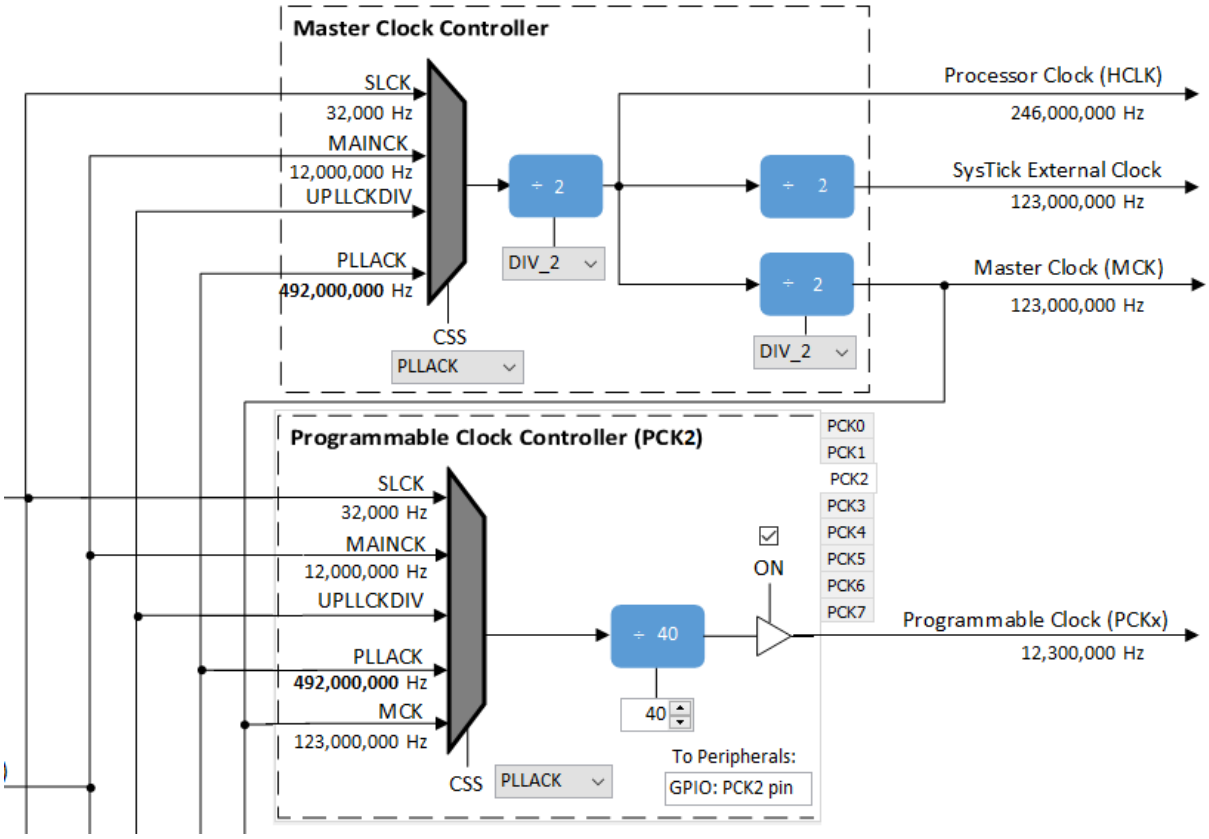
In the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




Still in the Clock Diagram, enable the PLLA Clock. Leave the divider at 1 (12 MHz) and change the multiplier to 41 for an output of 492 MHz.




In the Master Clock Controller, select the source (CSS) as PLLACK (492 MHz), all all three dividers as divide by 2 to generate a Processor Clock of 246 MHz and a Master Clock of 123 MHz. In the Programmable Clock Controller section, select the PCK2 tab, select the source (CSS) as PLLACK (492 MHz), the divider of 40 for a PCK2 output of 12,300,000 Hz.



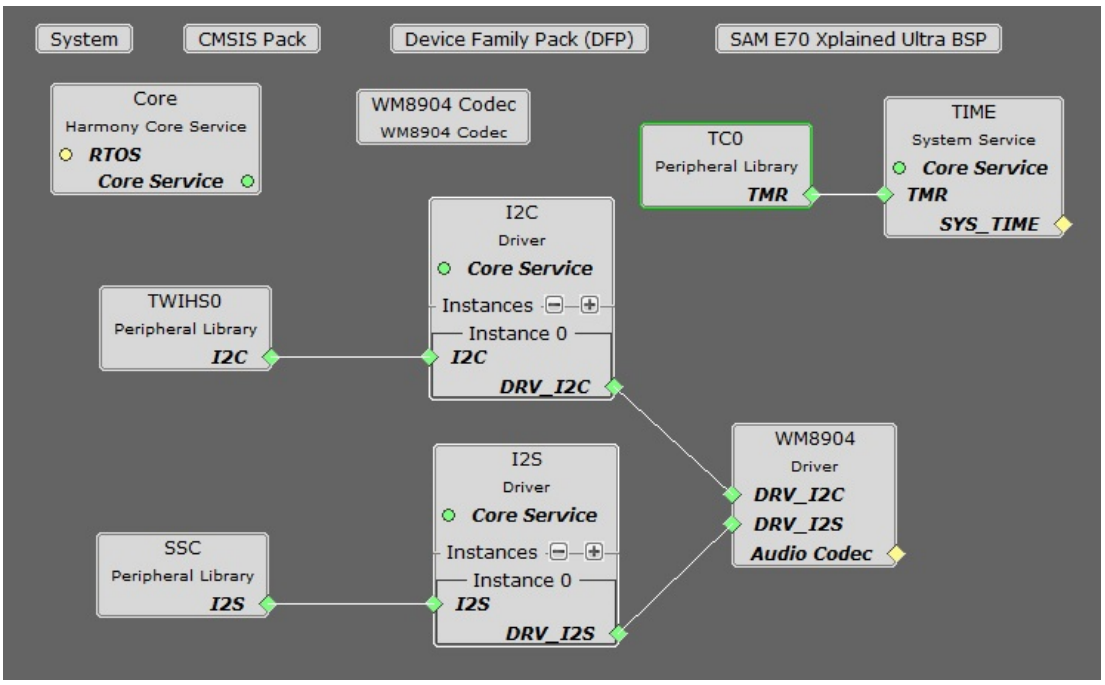

Then check the SSC checkbox in the **Peripheral Clock Controller** section.

For projects using the E70 or V71, the SSC interface and the WM8904 as a Master (the WM8904 codec generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP used. Select the appropriate processor (ATSAME70Q21B or ATSAMV71Q21B) depending on your board. Click Finish.

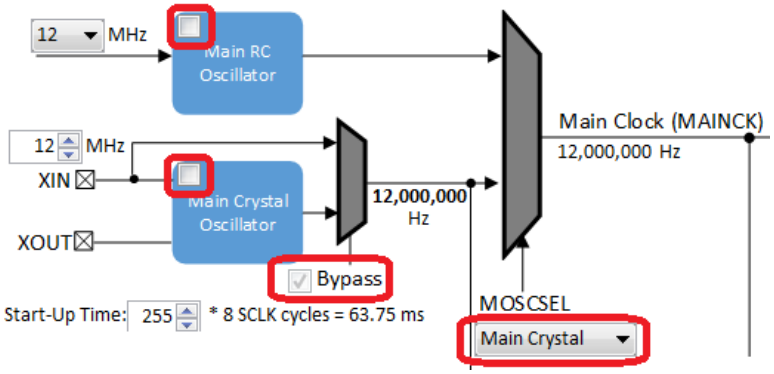
In the MHC, under Available Components select the appropriate BSP (SAM E70 Xplained Ultra or SAM V71 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:



Click on the WM8904 Driver. In the Configurations Options, set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




If using the ATSAMV71Q21B, in the Clock Diagram set MOSCEL to Main Crystal, uncheck the Bypass checkbox and RC Crystal Oscillator checkbox, and check the Main Crystal Oscillator box.

Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz). Then check the SSC checkbox in the **Peripheral Clock Controller** section.

It is also possible to change the audio format from 16 to 32-bits, and from I2S to Left Justified (SSC only). These changes need to be done in the MHC in both the WM8904, and SSC/I2SC Peripherals. In the current application code (app.h), a #define is also set to the current width.

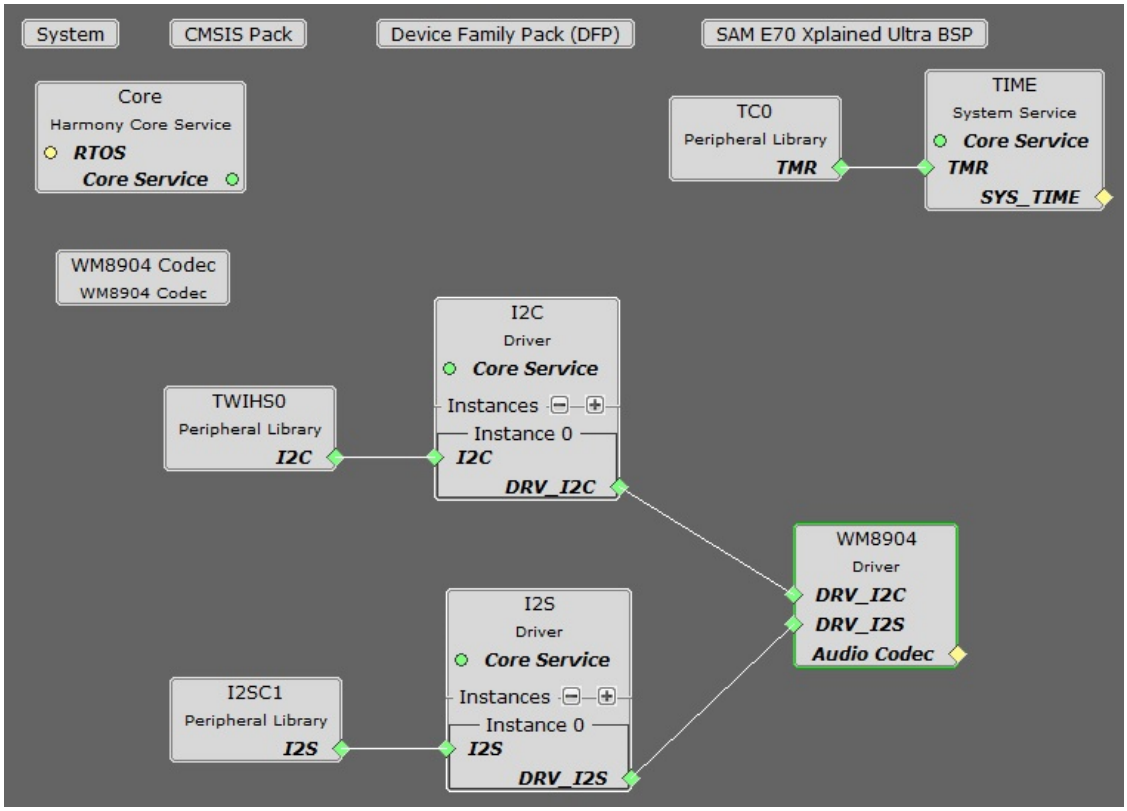
If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the SYS_Tasks function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of _APP_Tasks (just before the call to APP_Tasks).

For projects using the E70, the I2SC interface and the WM8904 as a Slave (the SAM E70 generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME70Q21B). (The WM8904 on the SAM V71 Xplained Ultra cannot be used with I2SC.) Click Finish.

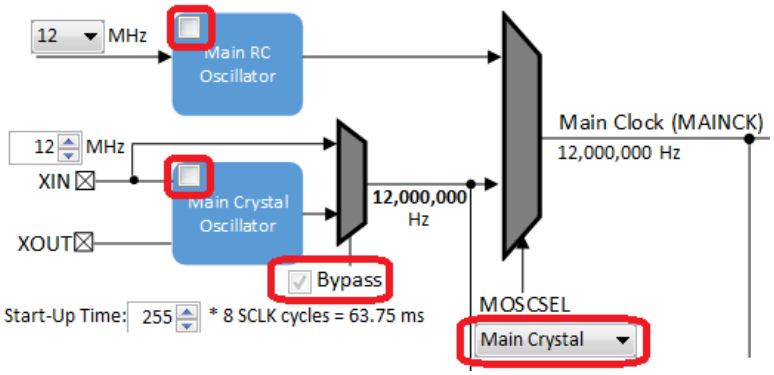
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead of SSC. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes:



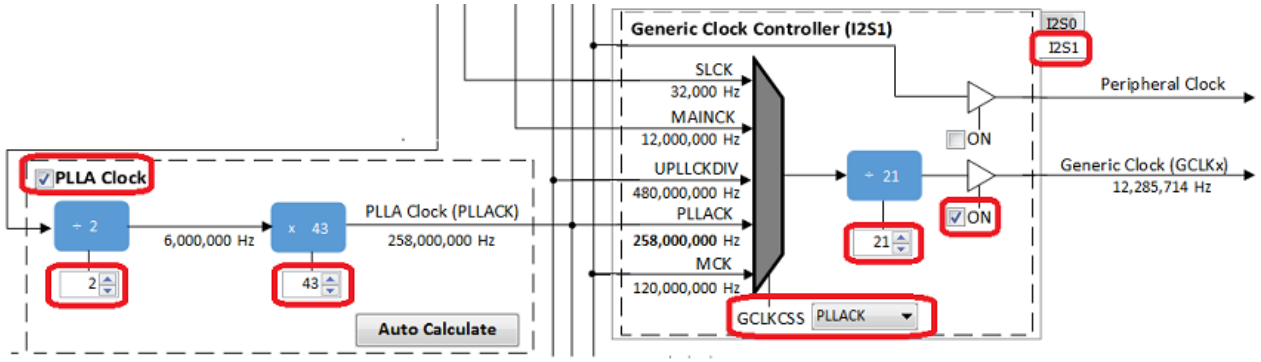

Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz).

The following tables show suggested settings for various sample rates in the Clock Diagram when using the I2SC Peripheral in Master mode. Make sure **PLLA Clock** checkbox is checked, and fill in the values for the PLLA Multiplier and Divider boxes. Select the I2S1 tab under **Generic Clock Controller**, set GCLKCSS to PLLACK, fill in the Divider value as shown, and check the checkbox next to it.




The values in the first table give the lowest error rate, but have varying PLLACK values so it is best to use the UPPCLKDIV selection for CSS under **Master Clock Controller**, for a Processor Clock of 240 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 43 | 258 MHz | 126 | 7998 | -0.025% |
| 16000 | 2 | 43 | 258 MHz | 63 | 15997 | -0.0187% |
| 44100 | 1 | 16 | 192 MHz | 17 | 41117 | 0.0385% |
| 48000 | 2 | 43 | 258 MHz | 21 | 47991 | -0.0187% |
| 96000 | 3 | 43 | 172 MHz | 7 | 95982 | -0.0187% |

The values in the second table have somewhat higher error rates, but use a PLLACK value of 294 MHz which is suitable to be used as a Processor Clock (using the PLLACK selection for CSS) which is closer to the maximum of 300 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 49 | 294 MHz | 144 | 7975 | -0.3125% |
| 16000 | 2 | 49 | 294 MHz | 72 | 15950 | -0.3125% |
| 44100 | 2 | 49 | 294 MHz | 26 | 41170 | 0.1587% |
| 48000 | 2 | 49 | 294 MHz | 24 | 47851 | -0.3104% |
| 96000 | 3 | 49 | 294 MHz | 12 | 95703 | -0.3094% |

It is also possible to change the audio format from 16 to 32-bits. This changes need to be done in the MHC in both the WM8904 Driver and I2SC Peripheral. In the current application code (app.h), a #define is also set to the current width.

If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the `SYS_Tasks` function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of `_APP_Tasks` (just before the call to `APP_Tasks`).

## *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

### Description

The parent folder for these files is `audio/apps/audio_tone`. To build this project, you must open the `audio/apps/audio_tone/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

### MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

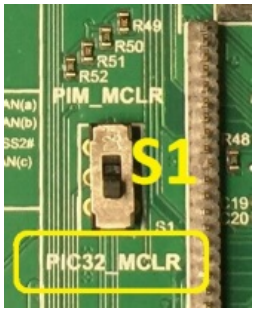
| Project Name | BSP Used | Description |
|---|---|---|
| audio_tone_ pic32mx_btadk_ ak4954 | pic32mx_btadk | This demonstration runs on the PIC32MX470F512L processor on the Bluetooth Audio Development Kit (BTADK) and the AK4954 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2S PLIB. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. |
| audio_tone_ pic32mz_ef_btadk_ ak4954 | pic32mz_ef_btadk | This demonstration runs on the PIC32MZ2048EFH144 processor on the Bluetooth Audio Development Kit (BTADK) and the AK4954 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2S PLIB. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. |
| audio_tone_ pic32mz_ef_c2_ ak4954 | pic32mz_ef_curiosity_v2 | This demonstration runs on the PIC32MZ2048EFM144 processor on the PIC32 MZ EF Curiosity 2.0 board and the AK4954 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2S PLIB. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. |
| audio_tone_ sam_e54_cult wm8904_ssc_freertos | sam_e54_cult | This demonstration runs on the ATSAME54P20A processor on the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. This demonstration also uses FreeRTOS. |
| audio_tone_ sam_e70_xult_ ak4954_ssc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the AK4954 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The AK4954 codec is configured as the slave, and the SSC peripheral as the master. |
| audio_tone_ sam_e70_xult_ wm8904_ssc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |
| audio_tone_ sam_e70_xult_ wm8904_ssc_freertos | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. This demonstration also uses FreeRTOS. |
| audio_tone_ sam_e70_xult _wm8904_i2sc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. |
| audio_tone_ sam_v71_xult | sam_v71_xult | This demonstration runs on the ATSAMV71Q21B processor on the SAM V71 Xplained Ultra board along with the on-board WM8904 codec. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |

### *Configuring the Hardware*

This section describes how to configure the supported hardware.

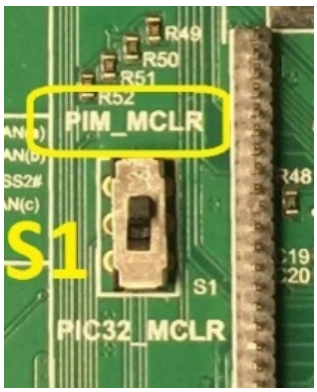
## Description

Using the Bluetooth Audio Development Kit and the AK4954 Audio Codec Daughter Board, and I2S PLIB. The AK4954 daughter board is plugged into the rear set of X32 connectors (J9/J10). Switch S1 on the PIC32 Bluetooth Audio Development Board (between the audio codec daughter board and the microcontroller PIM) should be set to PIC32_MCLR.




**Note:** The Bluetooth Audio Development Kit does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

Using the Bluetooth Audio Development Kit and the PIC32MZ EF Audio PIM with the AK4954 Audio Codec Daughter Board and I2S PLIB. Plug the PIM onto the center square connector P1. Double-check to make sure the pins are aligned correctly. The AK4954 daughter board is plugged into the rear set of X32 connectors (J9/J10). Switch S1 on the PIC32 Bluetooth Audio Development Board (between the audio codec daughter board and the microcontroller PIM) should be set to PIM_MCLR.

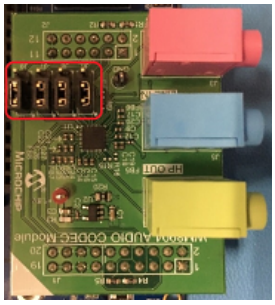



**Note:** The Bluetooth Audio Development Kit does not include the either the PIC32MZ EF Audio PIM or the AK4954 Audio Codec daughterboard, which are sold separately on microchipDIRECT as part numbers MA320018 and AC324954 respectively.

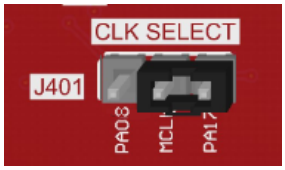
Using the PIC32 MZ EF Curiosity 2.0 board and the AK4954 Audio Codec Daughter Board, and I2S PLIB. The AK4954 daughter board is plugged into the set of X32 connectors on the right side of the board (X32 HEADER 2).

**Note:** The PIC32 MZ EF Curiosity 2.0 does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

Using the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board, and I2S PLIB. All jumpers on the WM8904 should be toward the **front:**



In addition, make sure the J401 jumper (CLK SELECT) is set for the PA17 pin:

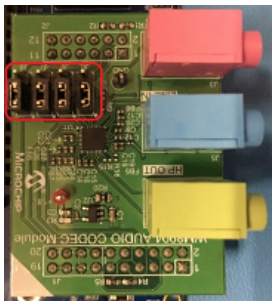



**Note:** The SAM E54 Curiosity Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the AK4954 Audio Codec Daughter Board, and SSC PLIB. No special configuration needed.

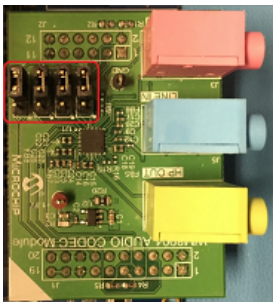
**Note:** The SAM E70 Xplained Ultra board does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, and SSC PLIB. All jumpers on the WM8904 should be toward the **front:**



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, and I2SC PLIB. All jumpers on the WM8904 should be toward the **back:**



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM V71 Xplained Ultra board with on-board WM8904, with SSC PLIB. No special configuration needed.

### *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

All configurations:

Continuous sine tones of four frequencies can be generated. **Tables 1-3** provides a summary of the button actions that can used

to control the volume and frequency.

Compile the program using MPLAB X, and program the target device using the EDBG interface (ICD4 for projects using the Bluetooth Audio Development Kit). While compiling, select the appropriate MPLAB X IDE project. Refer to Building the Application for details.

1. Connect headphones to the HP OUT jack of the AK4954 or WM8904 Audio Codec Daughter Board (see **Figures 1-3**), or the HEADPHONE jack of the SAM V71 Xplained Ultra board.
2. The tone can be quite loud, especially when using a pair of headphones.
3. The lowest-numbered pushbutton (SW0 or SW1) and LED are used as the user interface to control the program, as shown in Tables 1-3 below, depending on the board used. Initially the program will be in volume-setting mode (LED off) at a medium volume setting. Pressing the pushbutton with the LED off will cycle through four volume settings (including mute).
4. Pressing the pushbutton longer than one second will change to frequency-setting mode (LED on). Pressing the pushbutton with the LED on will cycle through four frequency settings -- 250 Hz, 500 Hz, 1 kHz, and 2 kHz.
5. Pressing the pushbutton longer than one second again will switch back to volume-setting mode again (LED off).

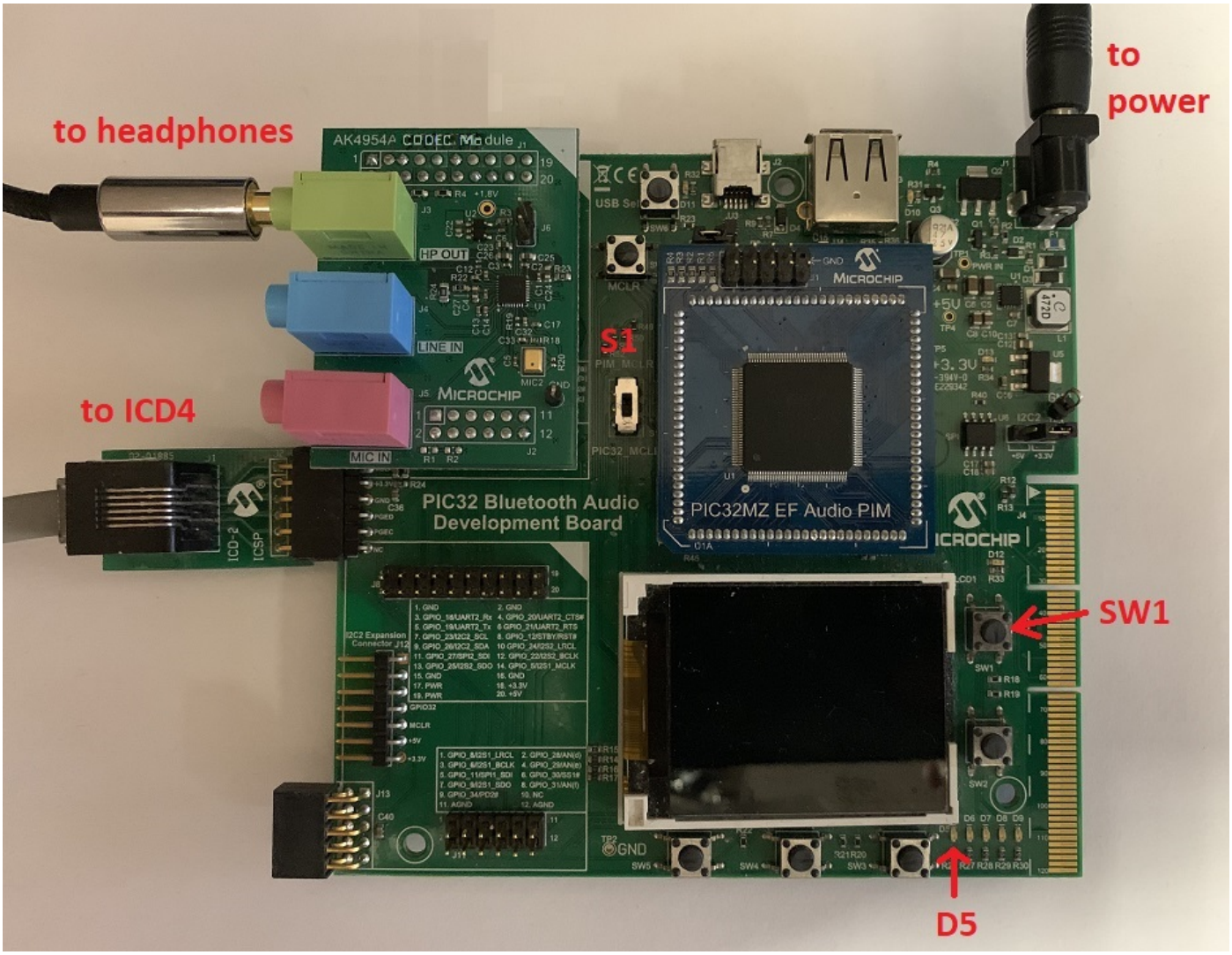



**Figure 1: AK4954 Audio Codec Daughter Board on Bluetooth Audio Development Kit with PIC32MZ EF Audio PIM**

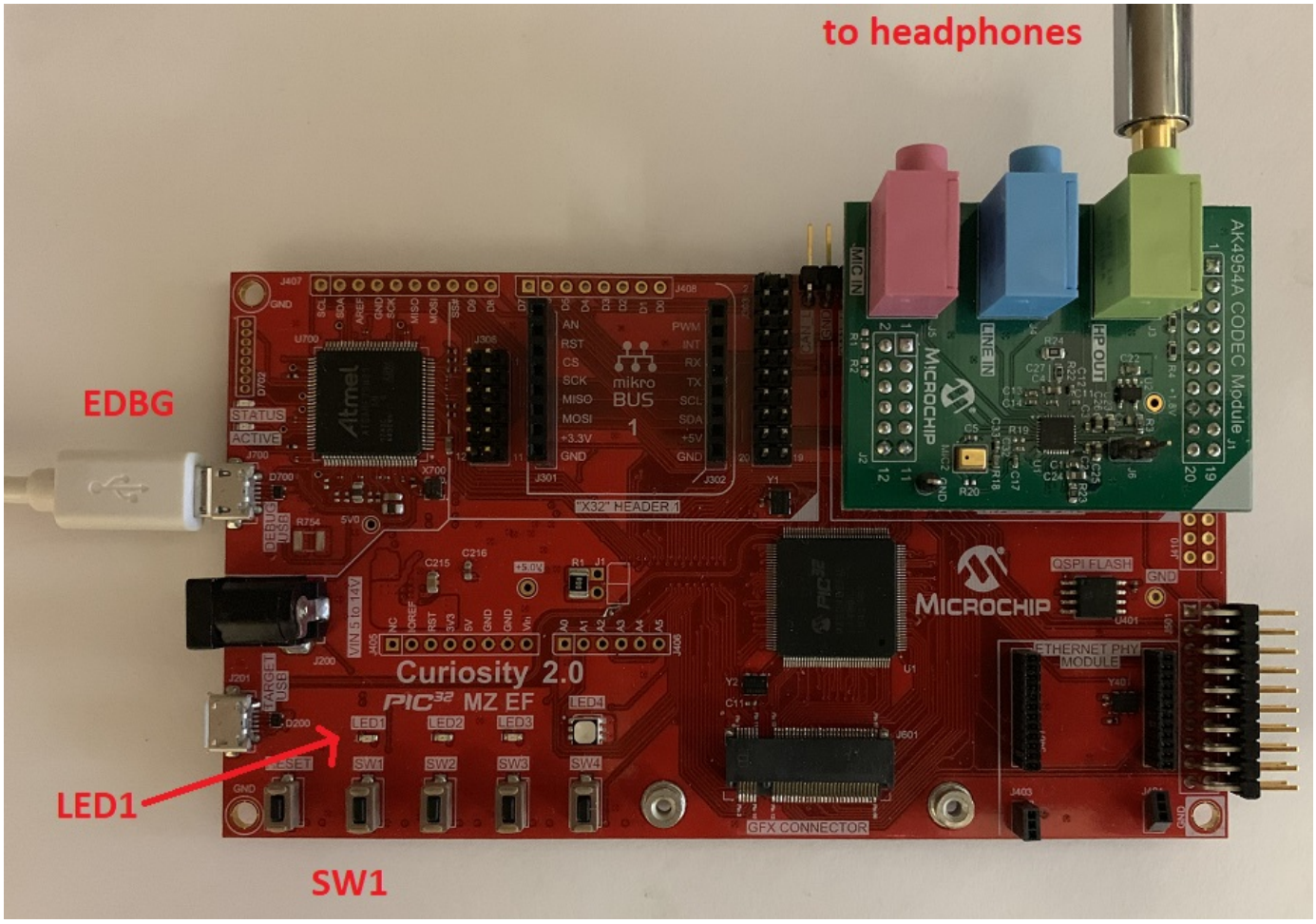



**Figure 2: AK4954 Audio Codec Daughter Board on PIC32 MZ EF Curiosity 2.0 Board**

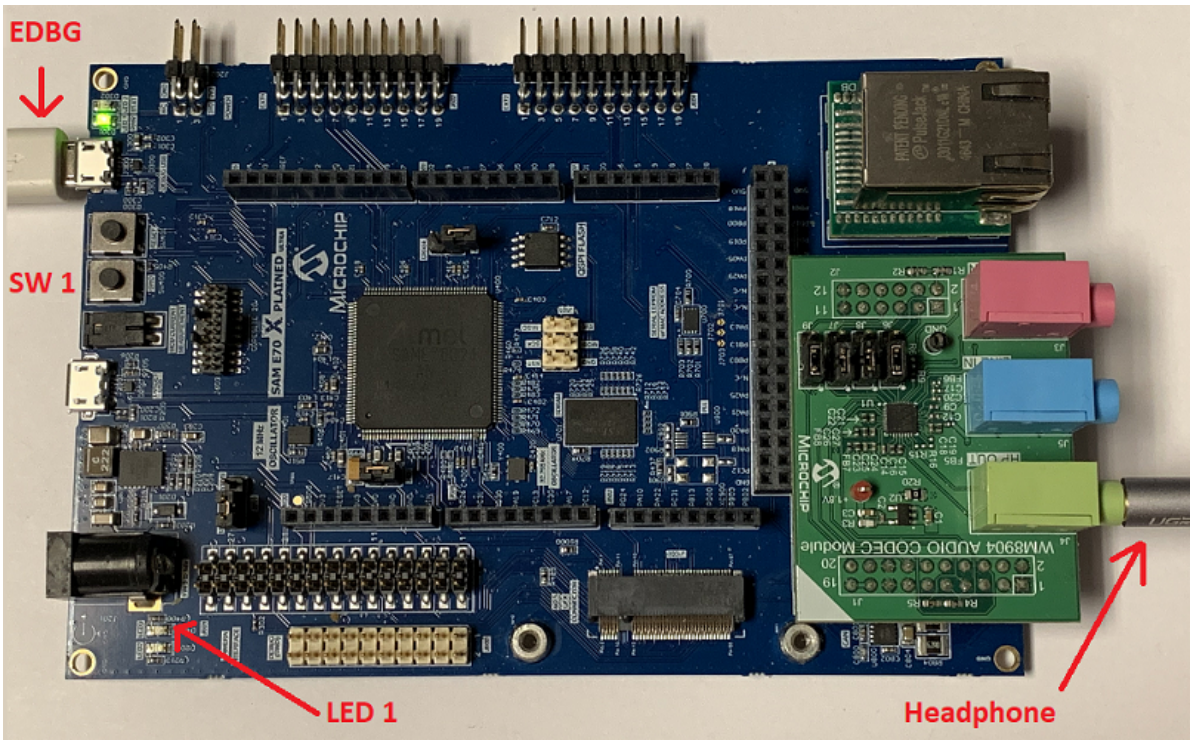



**Figure 3: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board**

### Control Descriptions

**Table 1: Button Controls for Bluetooth Audio Development Kit**

| Control | Description |
|---|---|
| SW1 short press | If LED D5 is off, SW1 cycles through four volume levels (one muted). If LED D5 is on, SW1 cycles through four frequencies of sine tone. |
| SW1 long press (> 1 second) | Alternates between modes (LED D5 on or off). |

**Table 2: Button Controls for PIC32 MZ EF Curiosity 2.0 board, SAM E54 Curiosity Ultra board, and SAM E70 Xplained Ultra board**

(On some E70 boards, SW0/LED0 are the lowest numbered pushbutton and LED, so use Table 3 instead.)

| Control | Description |
|---|---|
| SW1 short press | If LED1 is off, SW1 cycles through four volume levels (one muted). If LED1 is on, SW1 cycles through four frequencies of sine tone. |
| SW1 long press (> 1 second) | Alternates between modes (LED1 on or off). |

**Table 3: Button Controls for SAM V71 Xplained Ultra board**

| Control | Description |
|---|---|
| SW0 short press | If LED0 is off, SW0 cycles through four volume levels (one muted). If LED0 is on, SW0 cycles through four frequencies of sine tone. |
| SW0 long press (> 1 second) | Alternates between modes (LED0 on or off). |

## audio_tone_linkeddma

This topic provides instructions and information about the MPLAB Harmony 3 Audio Tone using Linked DMA demonstration application, which is included in the MPLAB Harmony Library distribution.

### Description

In this demonstration application the Codec Driver sets up the WM8904 Codec. The demonstration sends out generated audio waveforms (sine tone) using linked DMA channels with volume and frequency modifiable through the on-board push button. Success is indicated by an audible output corresponding to displayed parameters.

The sine tone is one of four frequencies: 250 Hz, 500 Hz, 1 kHz, and 2 kHz, sent by default at 48,000 samples/second, which is modifiable in the the MHC as described below.

To know more about the MPLAB Harmony Codec Drivers, configuring the Codec Drivers, and the APIs provided by the Codec Drivers, refer to Codec Driver Libraries.

### Architecture

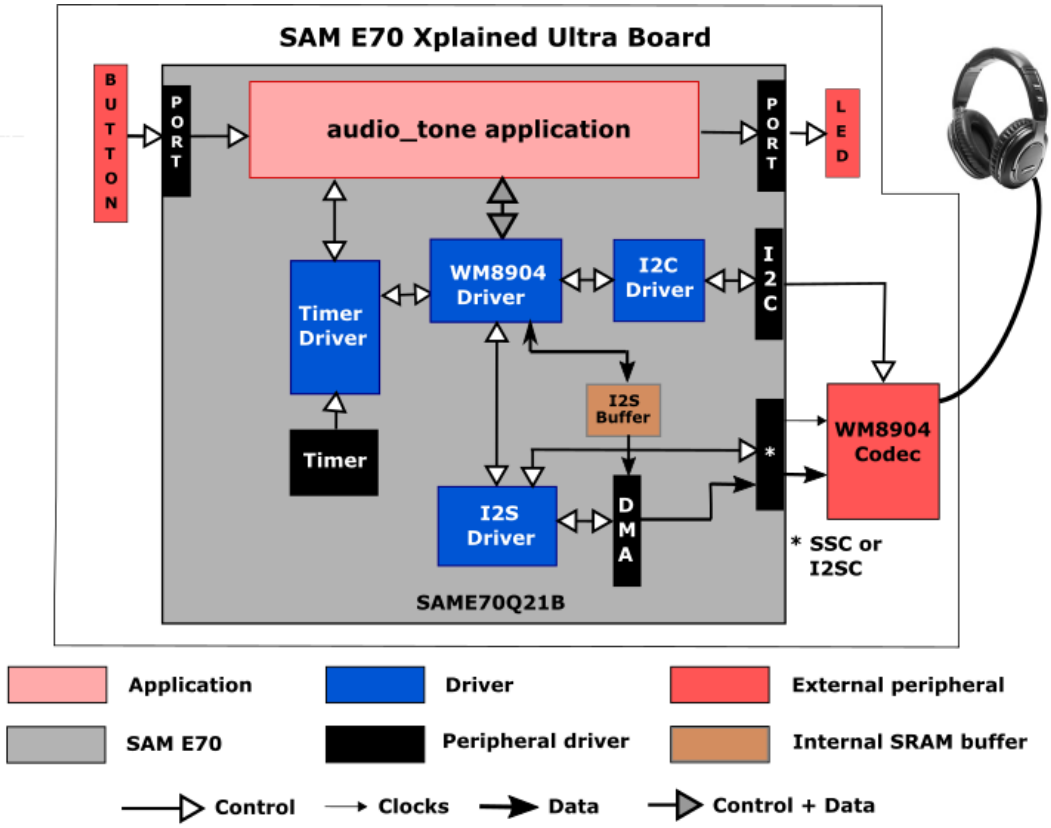
The two projects run on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- One push button (SW1)
- Two LEDs (LED1 and 2)
- WM8904 Codec Daughter Board mounted on a X32 socket

The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The program takes up to approximately 1% (17 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 2% (6 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the two SAM E70 Xplained Ultra configurations:




Depending on the project, either the SSC (Synchronous Serial Controller) or I2SC (Inter-IC Sound Controller) is used with the WM8904 codec, selected by a strapping option on the WM8904 daughterboard. When using the SSC interface, the WM8904 is configured in master mode, meaning it generates the $I^2S$ clocks (LRCLK and BCLK), and the SSC peripheral is configured as a slave. When using the I2SC interface, the WM8904 is configured in slave mode and the SSC peripheral is a master and generates the I2SC clocks. The other two possibilities (SSC as master and WM8904 as slave, or I2SC as slave and WM8904 as master) are possible, but not discussed.

The same application code is used without change between the two projects.

The SAM E70/SAM V71 microcontroller (MCU) runs the application code, and communicates with the WM8904 codec via an $I^2C$ interface. The audio interface between the SAM E70/V71 and the WM8904 codec use the $I^2S$ interface. Audio is configured as 16-bit, 48,000 samples/second, $I^2S$ format. (16-bit, 48 kHz is the standard rate used for DVD audio. An alternate that could be used is 44,100 samples/second. This is the same sample rate used for CD's. The sample rate is configurable in the MHC.)

The Master Clock (MCLK) signal used by the codec is generated by the Peripheral Clock section of the SAM E70/V71, and is fixed at 12 MHz.

The button and LEDs are interfaced using GPIO pins. There is no screen.

As with any MPLAB Harmony application, the SYS_Initialize function, which is located in the `initialization.c` source file, makes calls to initialize various subsystems as needed, such as the clock, ports, board support package (BSP), WM8904 codec, I2S, I2C, DMA, timers, and interrupts.

The codec driver and the application state machines are all updated through calls located in the SYS_Tasks function in the `tasks.c` file.

The application code is contained in the several source files. The application's state machine (`APP_Tasks`) is contained in `app.c`. It first initializes the application, which includes `APP_Tasks` then periodically calls `APP_Button_Tasks` to process any pending button presses.

Then the application state machine inside `APP_Tasks` is given control, which first gets a handle to a timer driver instance and sets up a periodic (alarm) callback. In the next state it gets a handle to the codec driver by calling the `DRV_CODEC_Open` function with a mode of `DRV_IO_INTENT_WRITE` and sets up the volume.

The application state machine then registers an event handler `APP_CODEC_BufferEventHandler` as a callback with the codec driver (which in turn is called by the DMA driver).

Two buffers are used for generating a sine wave. Each contains the data for one cycle of audio. One is currently output on a

repeated basis, and the second is filled in as necessary when the frequency changes. Then it is designated as the output buffer.

Initially, values for both buffers are calculated in advance. Two calls to the function `DRV_I2S_InitWriteLinkedListTransfer` are made each passing the address of one buffer. The buffers are each linked back to themselves such that when one is done, it starts over again. A call to `DRV_I2S_StartWriteLinkedListTransfer` is made, which starts the DMA sending data using data from the first buffer to the codec.

Only when the frequency is changed, due to a button press, does the second buffer come into play. In that case new values are calculated for the second buffer (while the first buffer continues to be used for output). When the second buffer has been filled in, a call is made to `DRV_I2S_WriteNextLinkedListTransfer` which transfers control to the second buffer after the first buffer finishes. The way the pointers are set up, it will then repeat on itself until another frequency change is made, and after which control will revert back to the first buffer.

A table called `samples` contains the number of samples for each frequency that correspond to one cycle of audio (e.g. 48 for 48000 samples/sec, and a 1 kHz tone). **Note:** the `samples` table will need to be modified if changing the sample rate to something other than 48000 samples/second.

This value with the number of samples to be generated is then passed to the function `APP_TONE_LOOKUP_TABLE_Initialize` along with which buffer to work with (1 or 2) and the sample rate. The 16-bit value for each sample is calculated based on the relative distance (angle) from 0, based in turn on the current sample number and total number of samples for one cycle. First the angle is calculated in radians:

```
double radians = (M_PI*(double)(360.0/(double)currNumSamples)*(double)i)/180.0;
```

Then the sample value is calculated using the sine function:

```
lookupTable[i].leftData = (int16_t)(0x7FFF*sin(radians));
```

If the number of samples divides into the sample rate evenly, then only 1/4 (90°) of the samples are calculated, and the remainder is filled in by reflection. Otherwise each sample is calculated individually. Before returning, the size of the buffer is calculated based on the number of samples filled in.

## Demonstration Features

- Calculation of a sine wave based on the number of samples and sample rate using the sin function
- Uses the Codec Driver Library to write audio samples to the WM8904
- At a lower level, uses the I2S Driver Library between the codec library and the chosen peripheral (SSC or I2SC) to send the audio to the codec
- Use of a alternate buffers (one active, one standby) and linked DMA
- Use of two timers: one as a periodic 1 ms timer for the application for button debouncing, and a second used by the Codec Driver (see Timer Driver Library)

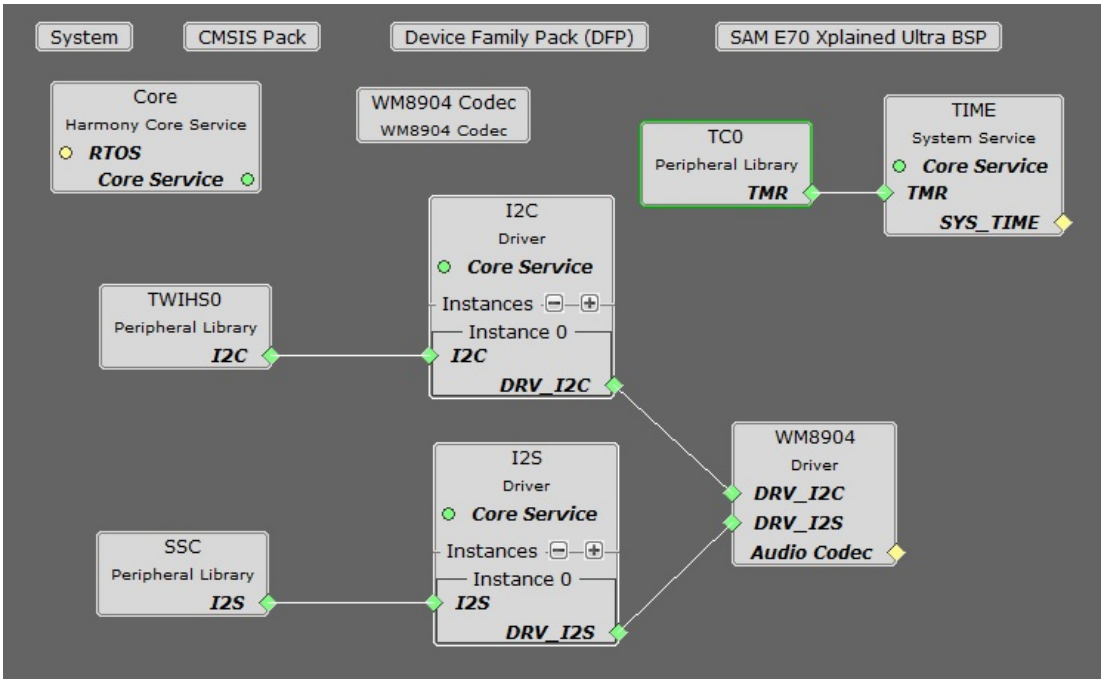
## Tools Setup Differences

For projects using the SSC interface and the WM8904 as a Master (the WM8904 codec generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the same based on the BSP used. Select the appropriate processor (ATSAME70Q21B or ATSAMV71Q21B) depending on your board. Click Finish.

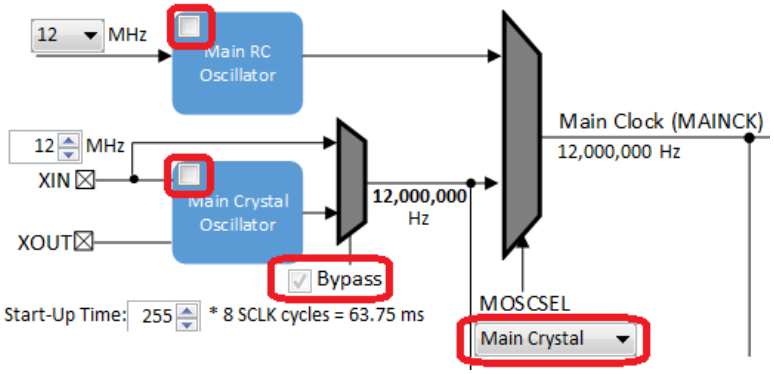
In the MHC, under Available Components select the appropriate BSP (SAM E70 Xplained Ultra or SAM V71 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:




Click on the WM8904 Driver. In the Configurations Options, set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




If using the ATSAMV71Q21B, in the Clock Diagram set MOSCEL to Main Crystal, uncheck the Bypass checkbox and RC Crystal Oscillator checkbox, and check the Main Crystal Oscillator box.

Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz). Then check the SSC checkbox in the **Peripheral Clock Controller** section.

It is also possible to change the audio format from 16 to 32-bits, and from I2S to Left Justified (SSC only). These changes need to be done in the MHC in both the WM8904, and SSC/I2SC Peripherals. In the current application code (app.h), a #define is also set to the current width.

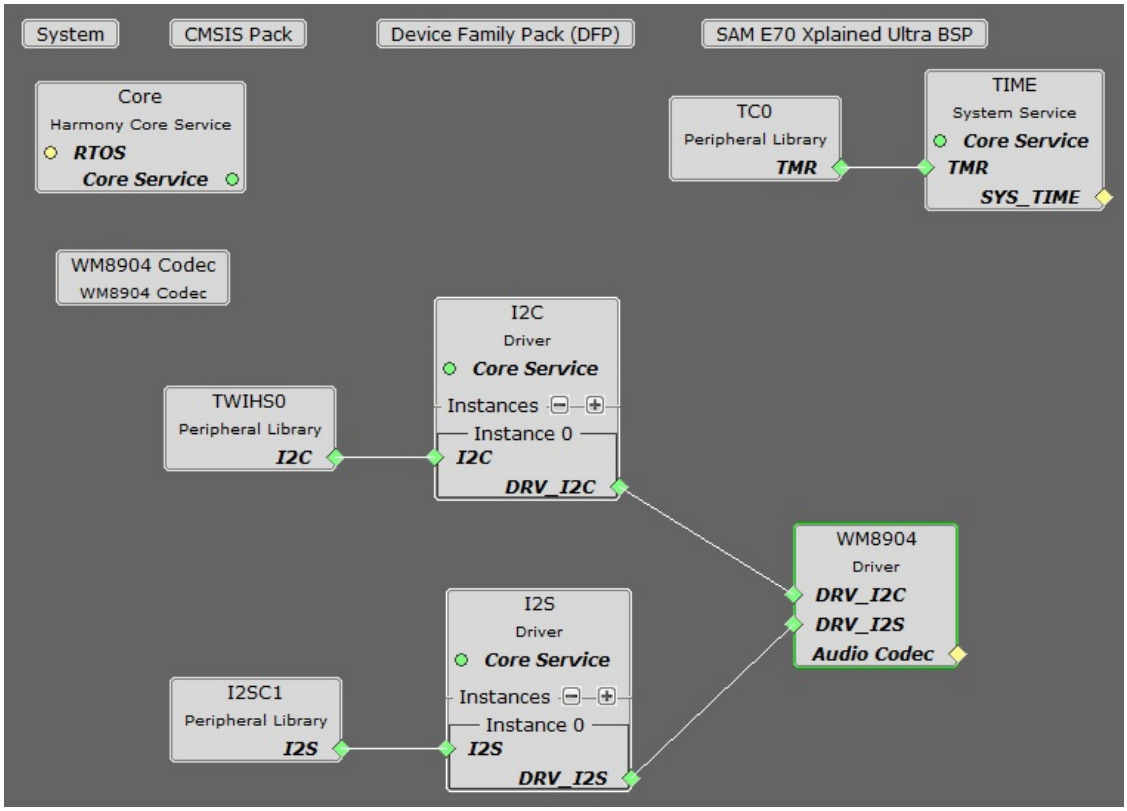
If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the `SYS_Tasks` function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of `_APP_Tasks` (just before the call to `APP_Tasks`).

For projects using the I2SC interface and the WM8904 as a Slave (the SAM E70 generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File* > *New Project*. Chose the Configuration name the based on the BSP. Select the appropriate processor (ATSAME70Q21B). (The WM8904 on the SAM V71 Xplained Ultra cannot be used with I2SC.) Click Finish.

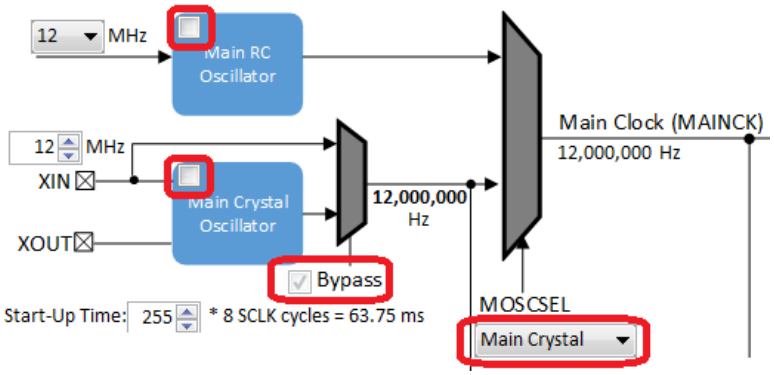
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead of SSC. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:



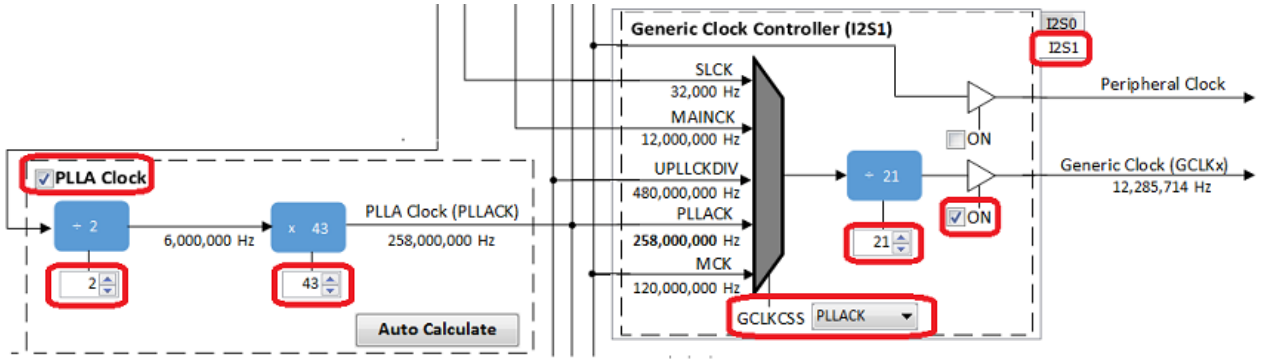

Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz).

The following tables show suggested settings for various sample rates in the Clock Diagram when using the I2SC Peripheral in Master mode. Make sure **PLLA Clock** checkbox is checked, and fill in the values for the PLLA Multiplier and Divider boxes. Select the I2S1 tab under **Generic Clock Controller**, set GCLKCSS to PLLACK, fill in the Divider value as shown, and check the checkbox next to it.




The values in the first table give the lowest error rate, but have varying PLLACK values so it is best to use the UPPCLKDIV selection for CSS under **Master Clock Controller**, for a Processor Clock of 240 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 43 | 258 MHz | 126 | 7998 | -0.025% |
| 16000 | 2 | 43 | 258 MHz | 63 | 15997 | -0.0187% |
| 44100 | 1 | 16 | 192 MHz | 17 | 41117 | 0.0385% |
| 48000 | 2 | 43 | 258 MHz | 21 | 47991 | -0.0187% |
| 96000 | 3 | 43 | 172 MHz | 7 | 95982 | -0.0187% |

The values in the second table have somewhat higher error rates, but use a PLLACK value of 294 MHz which is suitable to be used as a Processor Clock (using the PLLACK selection for CSS) which is closer to the maximum of 300 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 49 | 294 MHz | 144 | 7975 | -0.3125% |
| 16000 | 2 | 49 | 294 MHz | 72 | 15950 | -0.3125% |
| 44100 | 2 | 49 | 294 MHz | 26 | 41170 | 0.1587% |
| 48000 | 2 | 49 | 294 MHz | 24 | 47851 | -0.3104% |
| 96000 | 3 | 49 | 294 MHz | 12 | 95703 | -0.3094% |

It is also possible to change the audio format from 16 to 32-bits. This changes need to be done in the MHC in both the WM8904 Driver and SSC Peripheral. In the current application code (app.h), a #define is also set to the current width.

If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the `SYS_Tasks` function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of `_APP_Tasks` (just before the call to `APP_Tasks`).

## *Bulding the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is audio/apps/audio_tone_linkeddma. To build this project, you must open the audio/apps/audio/audio_tone_linkeddma/firmware/*.X project file in MPLAB X IDE that corresponds to your hardware configuration.

**MPLAB X IDE Project Configurations**

The following table lists and describes supported configurations.

| Project Name | BSP Used | Description |
|---|---|---|
| audio_tone_ld_sam_e70_xult_wm8904_ssc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using linked DMA. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |
| audio_tone_ld_sam_e70_xult_wm8904_i2sc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using linked DMA. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. |

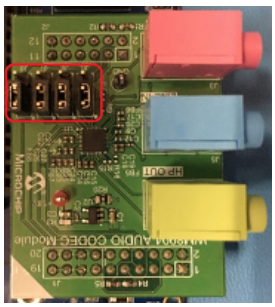
## *Configuring the Hardware*

This section describes how to configure the supported hardware.

### Description

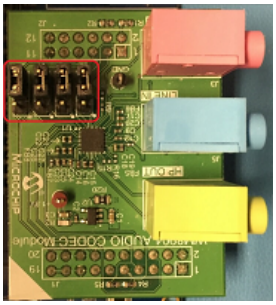
Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, with the SSC PLIB:

All jumpers on the WM8904 should be toward the **front**.



Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, with the I2SC PLIB:

All jumpers on the WM8904 should be toward the **back**.



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

## *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

All configurations:

Continuous sine tones of four frequencies can be generated. **Table 1** provides a summary of the button actions that can used to control the volume and frequency.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based. Refer to Building the Applications for details.

1. Connect headphones to the HP OUT jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).
2. The tone can be quite loud, especially when using a pair of headphones.
3. Initially the program will be in volume-setting mode (LED1 off) at a medium volume setting. Pressing SW1 with LED1 off will cycle through four volume settings (including mute).
4. Pressing SW1 longer than one second will change to frequency-setting mode (LED1 on). Pressing SW1 with LED1 on will cycle through four frequency settings -- 250 Hz, 500 Hz, 1 kHz, and 2 kHz.
5. Pressing SW1 longer than one second again will switch back to volume-setting mode again (LED1 off).

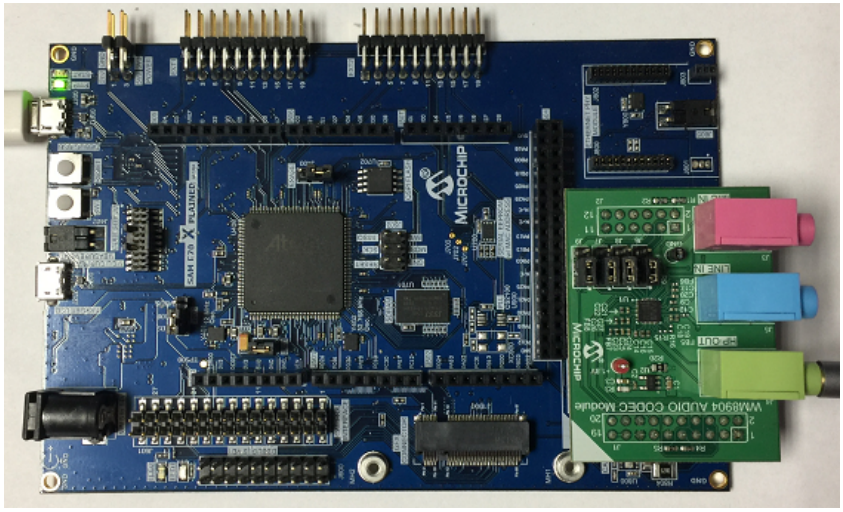



**Figure 1: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board**

### Control Descriptions

**Table 1: Button Controls for SAM E70 Xplained Ultra board**

| Control | Description |
|---|---|
| SW1 short press | If LED1 if off, SW1 cycles through four volume levels (one muted). If LED1 is on, SW1 cycles through four frequencies of sine tone. |
| SW1 long press<br>(> 1 second) | Alternates between modes (LED1 on or off). |

# microphone_loopback

This topic provides instructions and information about the MPLAB Harmony 3 Microphone Loopback demonstration application, which is included in the MPLAB Harmony Library distribution.

## Description

In this demonstration application, the AK4954 or WM8904 Codec Driver sets up the codec so it can receive audio data through the microphone input on the daughter board and sends this same audio data back out through the on-board headphones, after a delay. The volume and delay are configurable using the on-board pushbutton. The audio by default is sampled at 48,000 samples/second, which is modifiable in the MHC as described below.

To know more about the MPLAB Harmony Codec Drivers, configuring the Codec Drivers, and the APIs provided by the Codec Drivers, refer to Codec Driver Libraries.

## Architecture

There are eight different projects packaged in this application.

PIC32 MZ Bluetooth Audio Development Kit Project:

One project runs on the Bluetooth Audio Development Kit (BTADK) and PIC32MZ EF PIM, which contains a PIC32MZ2048EFH144 microcontroller with 2 MB of Flash memory and 512 KB of RAM running at 198 MHz. The BTADK includes the following features:

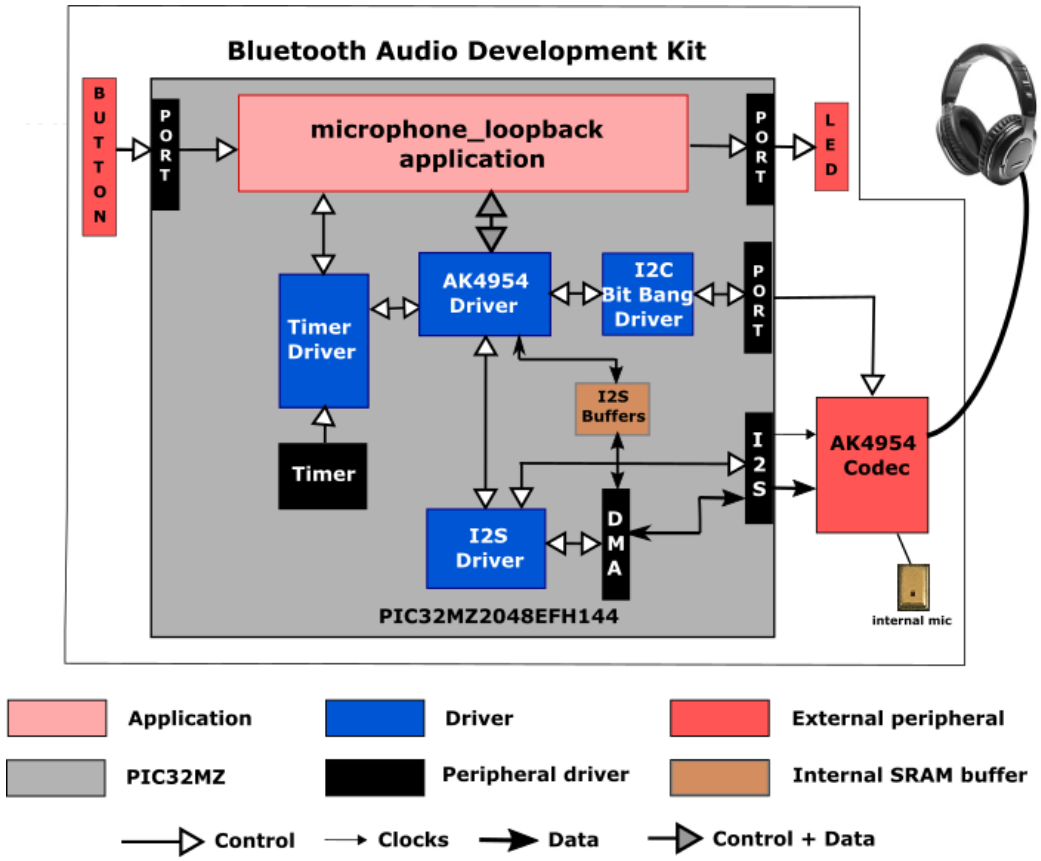
- Six push buttons (SW1-SW6, only SW1 is used)
- Five LEDs (D5-D9, only D5 is used)
- AK4954 Codec Daughter Board mounted on the rear X32 socket

The BTADK also includes an LCD display, which is not used in these projects.

The BTADK does not include the AK4954 Audio Codec daughterboard or the PIC32MZ EF Audio PIM, which are sold separately on microchipDIRECT as part number AC324954 and MA320018 respectively.

The program takes up to approximately 3% (55 KB) of the PIC32MZ2048EFH144 microcontroller's program space, and 23% (115 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the PIC32 Bluetooth Audio Development Kit configuration:




The I2S (Inter-IC Sound Controller) is used with the AK4954 codec. The AK4954 is configured in slave mode, meaning it receives $I^2S$ clocks (LRCLK and BCLK) from the PIC32, and the I2S peripheral is configured as a master. The PIC32MZ2048EFH144 uses the I2C Bit-Banged Library.

SAM E54 Curiosity Ultra Projects:

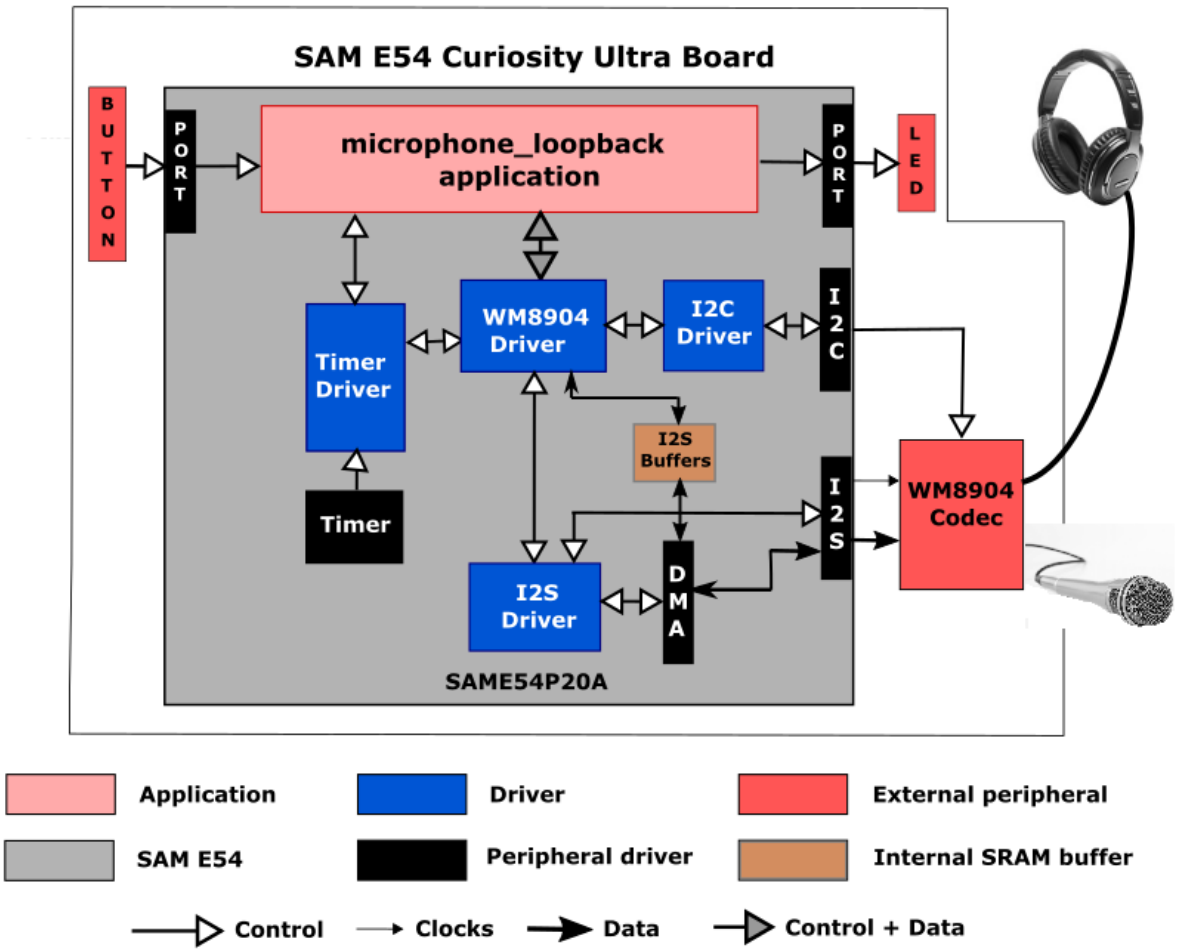
Two projects run on the SAM E54 Curiosity Ultra Board, which contains a ATSAME54P20A microcontroller with 1 MB of Flash memory and 256 KB of RAM running at 48 MHz using the following features:

- Two push buttons (SW1 and SW2, only SW1is used)
- Two LEDs (LED1 and 2, only LED1 is used)
- WM8904 Codec Daughter Board mounted on a X32 socket

The SAM E54 Curiosity Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The non-RTOS version of the program takes up to approximately 2% (23 KB) of the ATSAME54P20A microcontroller's program space. The 16-bit configuration uses 45% (115 KB) of the RAM. No heap is used. For the FreeRTOS project, the program takes up to approximately 3% (31 KB) of the ATSAME54P20A microcontroller's program space, and the 16-bit configuration uses 60% (155 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the two SAM E54 Xplained Ultra configurations (RTOS not shown):




The I2S (Inter-IC Sound Controller) is used with the WM8904 codec. The WM8904 is configured in master mode, meaning it generates the $I^2S$ clocks (LRCLK and BCLK), and the I2S peripheral is configured as a slave.

SAM E70 Xplained Ultra Projects (no display):

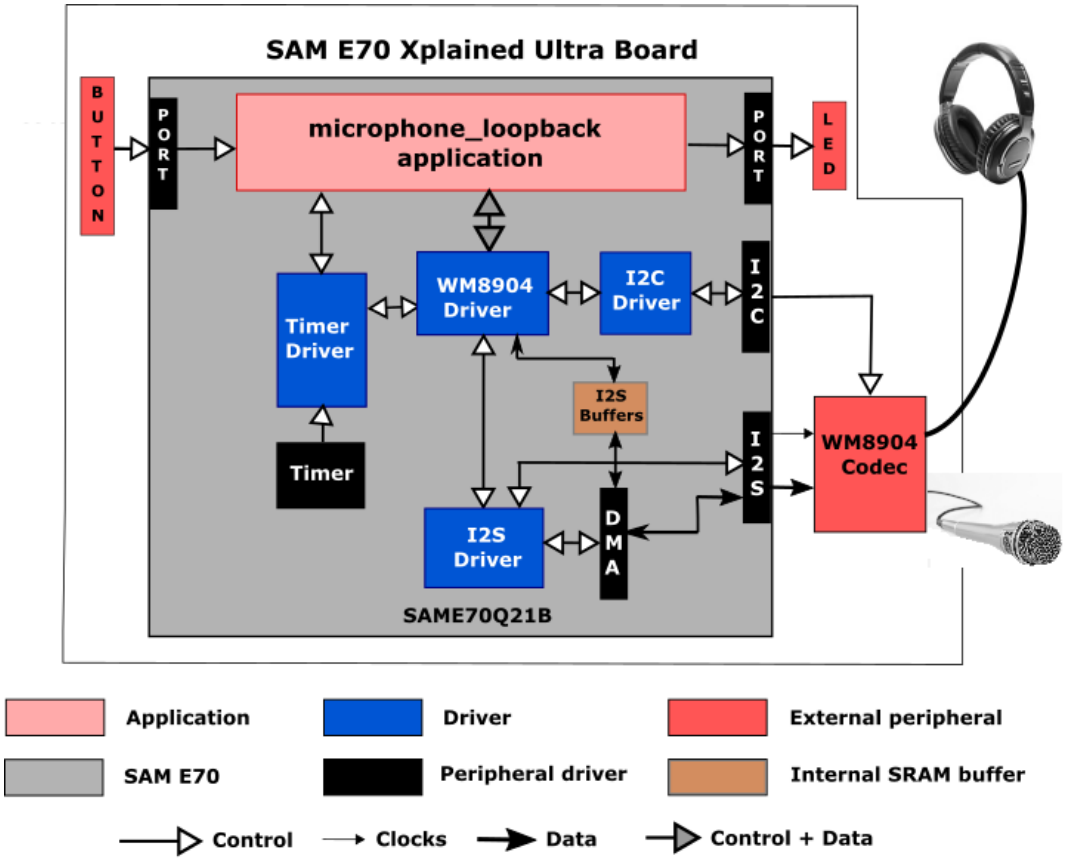
Three projects run on the SAM E70 Xplained Ultra Board without a display, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- One push button (SW1, may be labeled as SW0 on some boards)
- Two LEDs (LED1 and 2, only LED1 is used)
- AK4954 or WM8904 Codec Daughter Board mounted on a X32 socket

The SAM E70 Xplained Ultra board does not include the AK4954 or WM8904 Audio Codec daughterboards, which are sold separately on microchipDIRECT as part numbers AC324954 and AC328904, respectively.

The two non-RTOS versions of the program take up to approximately 1% (18 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 30% (115 KB) of the RAM. No heap is used. For the FreeRTOS project, the program takes up to approximately 1% (22 KB) of the ATSAME70Q21B microcontroller's program space, and the 16-bit configuration uses 41% (155 KB) of the RAM. No heap is used.

The following figure illustrates the application architecture for the three SAM E70 Xplained Ultra configurations using the WM8904 (RTOS not shown). The AK4954 version is the same with the substitution of the AK4954 Driver and Codec blocks.




The I2SC (Inter-IC Sound Controller) is used with the WM8904 codec, selected by a strapping option on the WM8904 daughterboard. The WM8904 is configured in slave mode and the I2SC peripheral is a master and generates the I2SC (LRCLK and BCLK) clocks. Other possible configurations are possible but not discussed.

SAM E70 Xplained Ultra Project with LCD display:

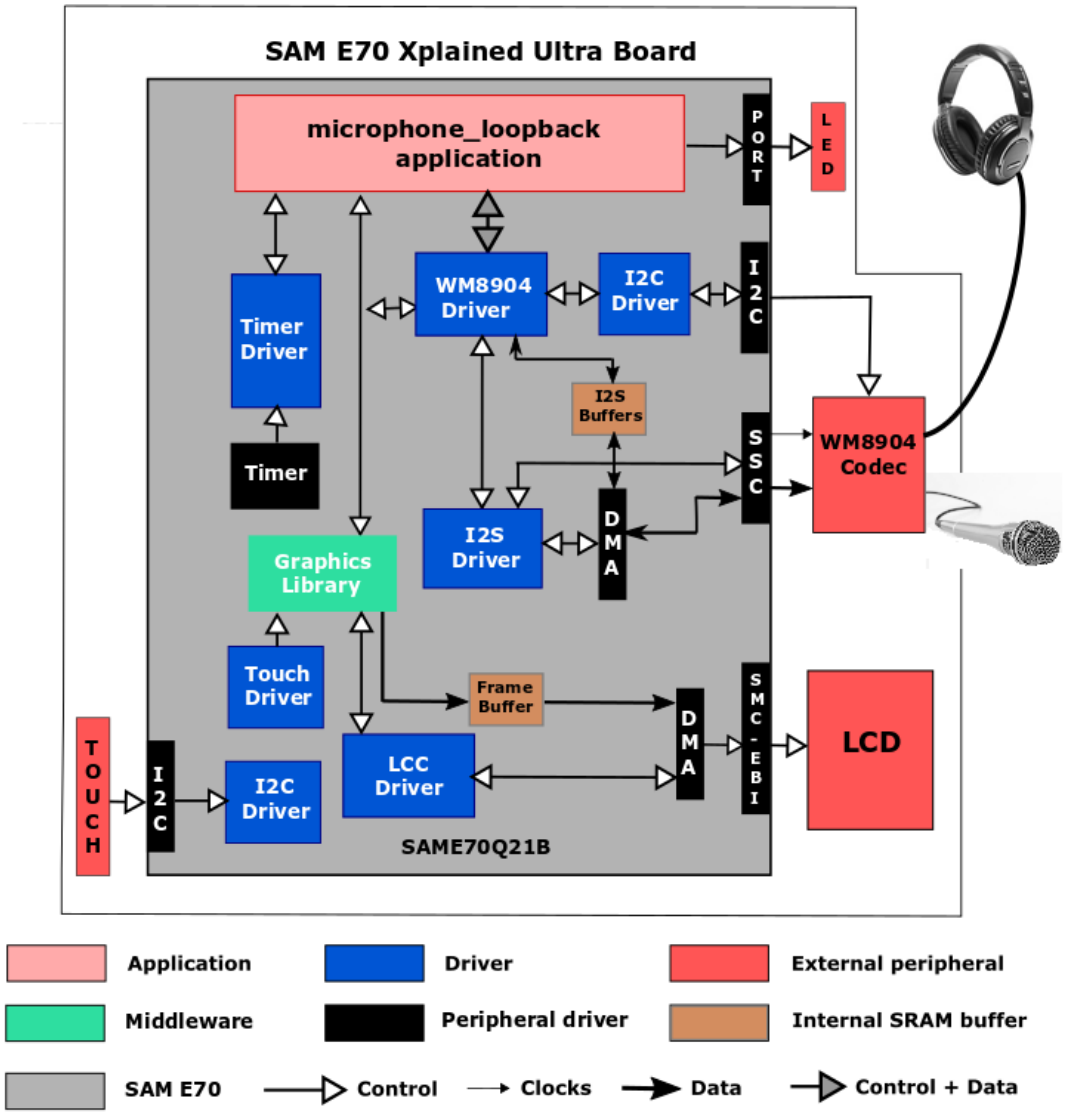
This project runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- WM8904 Codec Daughter Board mounted on a X32 socket
- PDA TM4301B 480x272 (WQVGA) Display

The SAM E70 Xplained Ultra board does not include either the WM8904 Audio Codec daughterboard or the TM4301B graphics card, which are sold separately on microchipDIRECT as part numbers AC328904 and AC320005-4, respectively.

The program takes up to approximately 5% (398 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 88% (338 KB) of the RAM, including a 32 KB heap.

The following figure illustrates the application architecture, for this configuration:




SAM V71 Xplained Ultra Project:

This project runs on the SAM V71 Xplained Ultra Board, which contains a ATSAMV71Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- Two push buttons (SW0 and 1, only SW0 used)
- Two LEDs (LED0 and 1, only LED0 used)
- WM8904 codec (on board)

The program takes up to approximately 1% (13 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 74% (284 KB) of the RAM. No heap is used.

The architecture is the same as for the first bare-metal SAM E70 Ultra board configuration; however only it uses the SSC peripheral instead of the I2SC. The WM8904 is configured in master mode and generates the I2SC (LRCLK and BCLK) clocks, and the I2SC peripheral is configured as a slave.

Except for the user interface, the same application code is used without change between the various projects.

The PIC32 or SAM microcontroller (MCU) runs the application code, and communicates with the AK4954 or WM8904 codec via an $I^2C$ interface. The audio interface between the microcontroller and the WM8904 codec use the $I^2S$ interface. Audio is configured as 16-bit, 48,000 or 16,000 samples/second, $I^2S$ format. (16-bit, 48 kHz is the standard rate used for DVD audio. 16 kHz still provides excellent voice quality audio. Another alternative that could be used is 44,100 samples/second. This is the same sample rate used for CD's.

The Master Clock (MCLK) signal used by the codec is generated by the Generic Clock section of the E54, or the Peripheral Clock section of the SAM E70/V71, and is either 12 MHz (when the codec is configured as a master), or close to 12,288,000 (when the codec is configured as a slave, this frequency being 256 times the sample rate).

In all but one project, the button and LEDs are interfaced using GPIO pins. In the other project, a 480x272 LCD screen is used as the user interface.

As with any MPLAB Harmony application, the SYS_Initialize function, which is located in the initialization.c source file, makes calls to initialize various subsystems as needed, such as the clock, ports, board support package (BSP), screen (if applicable), AK4954 or WM8904 codec, I2S, I2C, DMA, timers, and interrupts.

The codec driver and the application state machines are all updated through calls located in the SYS_Tasks function in the tasks.c file.

When the application's state machine (APP_Tasks), contained in app.c, is given control it first gets a handle to a timer driver instance and sets up a periodic (alarm) callback. In the next state it gets a handle to the codec driver by calling the DRV_CODEC_Open function with a mode of DRV_IO_INTENT_WRITE and sets up the volume. The application state machine then registers an event handler APP_CODEC_BufferEventHandler as a callback with the codec driver (which in turn is called by the DMA driver).

For the non-graphical version of the app, a total of MAX_BUFFERS (#defined as 150) buffers are allocated, each able to hold 10 ms of audio (480 samples at 48,000 samples/second), which is enough for a 1.5 second delay.). For the graphical version, which has less memory available available due to the frame buffer, MAX_BUFFERS is set at 101, each able to hold 10 ms of audio based on 160 samples at 16,000 samples/second, therefore enough for one second of delay.

Two indicies, appData.rxBufferIdx and appData.txBufferIdx are used to track the current buffers for both input and output. Initially all buffers are zeroed out. A call to DRV_CODEC_BufferAddWriteRead is made, which writes out the data from the buffer pointed to by appData.txBufferIdx (initially zero), while at the same time data is read from the codec into the buffer pointed to by appData.rxBufferIdx.

On each callback from the DMA, the buffer pointers are advanced, wrapping around at the ends. After the prescribed delay, the audio from the first buffer filled in will be output.

In the non-graphical version, if the button is held down for more than one second, the mode changes, and there is no delay while the volume is being adjusted. Instead of a circular array of buffers, the first two buffers are just toggled back and forth in a ping-pong fashion. In the graphical version, this mode change is handled through the Enable Delay button on the GUI.

## Demonstration Features

- Uses the Codec Driver Library to input audio samples from the AK4954 or WM8904 codec, and later write them back out to the codec
- At a lower level, uses the I2S Driver Library between the codec library and the chosen peripheral (SSC or I2SC or I2S) to send the audio to the codec
- Using either an array of circular buffers to create an audio delay, or two buffers in ping-pong fashion for no delay, both using DMA
- Use of two timers: one as a periodic 1 ms timer for the application for button debouncing, and a second used by the Codec Driver (see Timer Driver Library)
- For the version using the display, use of the touch screen as a UI

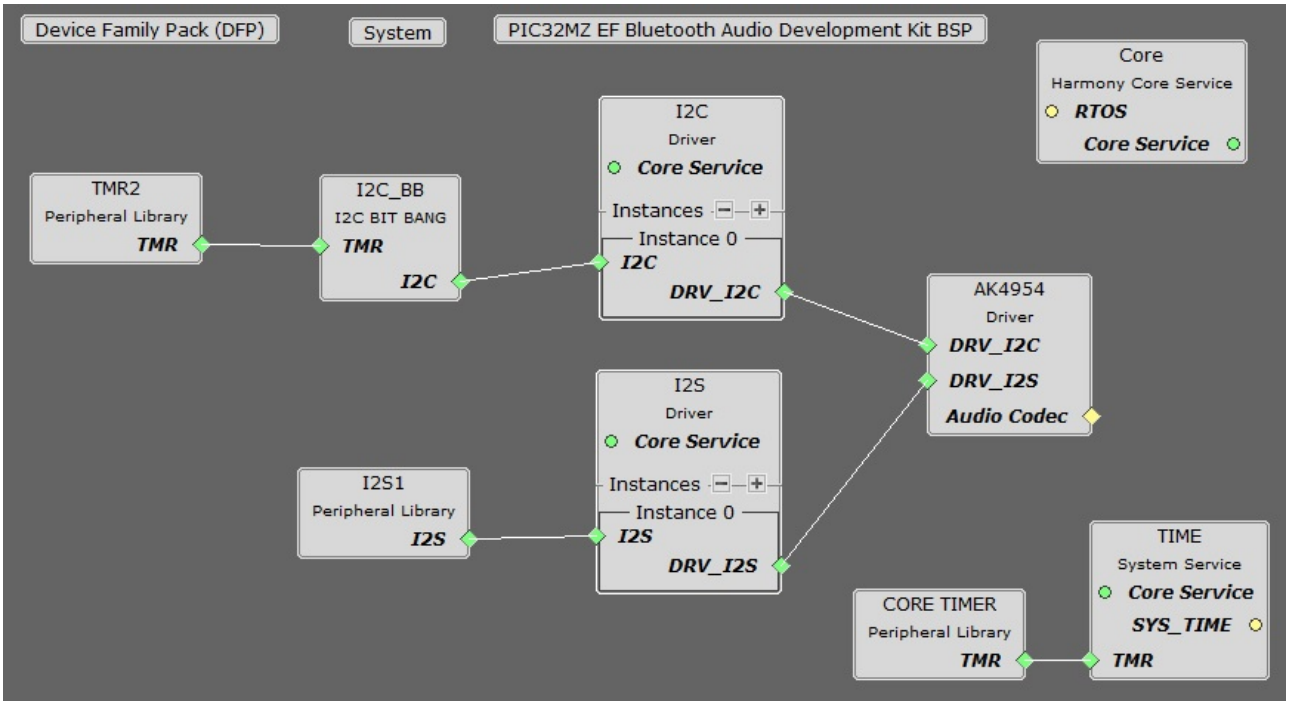
## Tools Setup Differences

For the project using the PIC32MZ EF, the I2S interface and the AK4954 as a Slave (the I2S peripheral generates the $I^2S$ clocks):

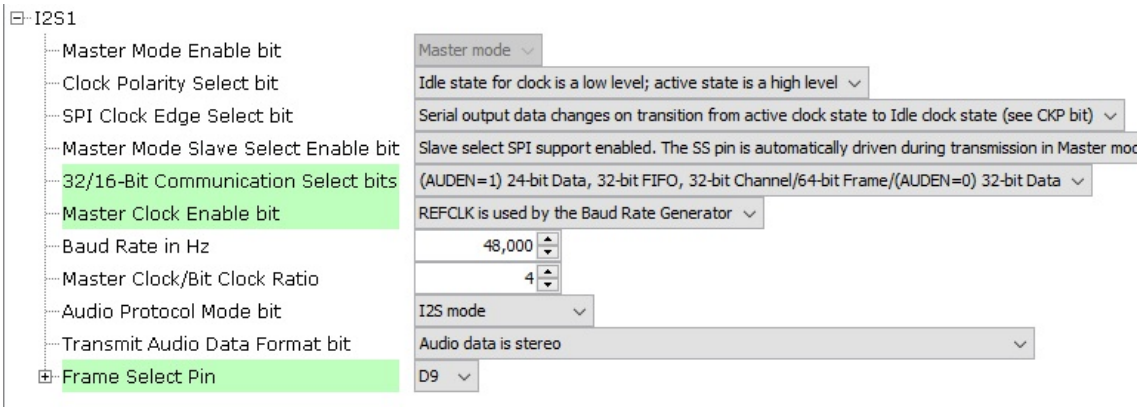
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (PIC32MZ2048EFH144). Click Finish.

In the MHC, under Available Components select the BSP PIC32MZ Bluetooth Audio Development Kit. Under *Audio>Templates*, double-click on AK4954 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

You should end up with a project graph that looks like this, after rearranging the boxes:



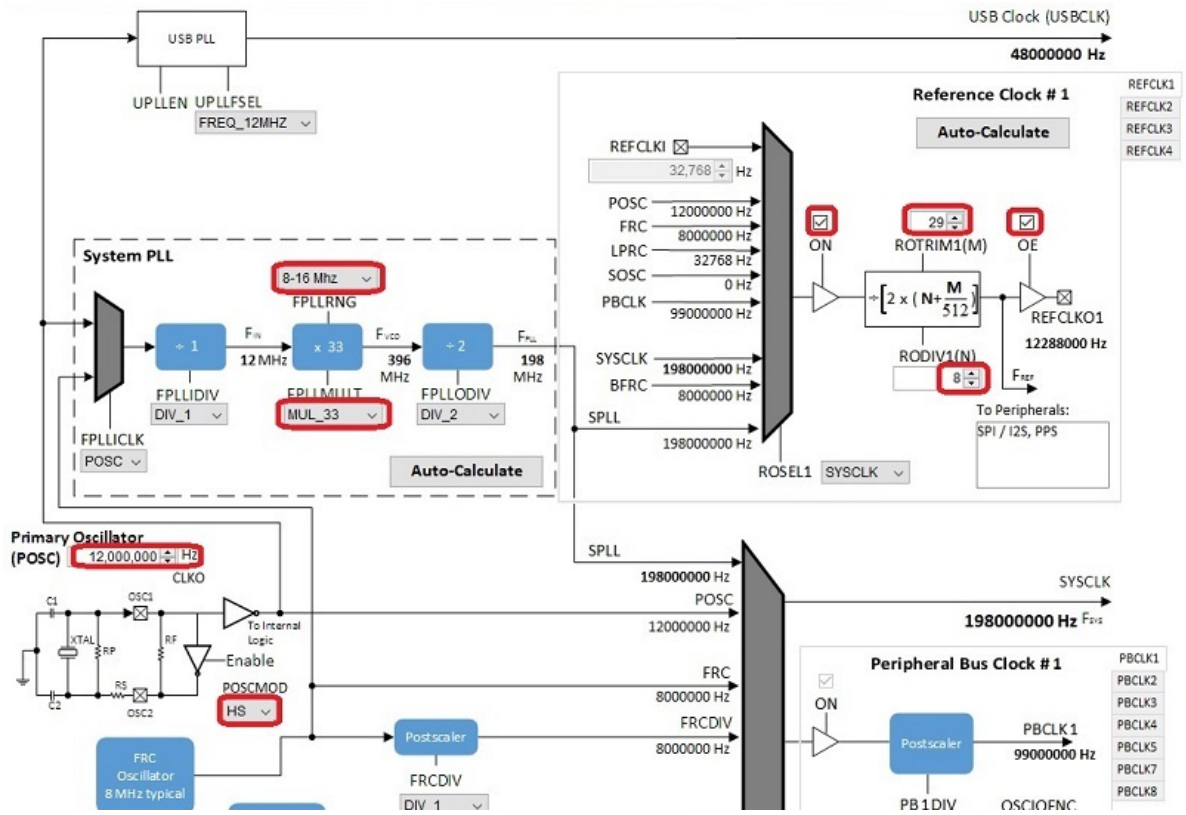

Click on the I2S Peripheral. In the Configurations Options, under 32/16-Bit Communication Select bits, select (AUDEN=1) 24-bit data, 32-bit FIFO, 32-bit Channel/64-bit Frame. Change Master Clock Enable bit to REFCLK. Set the Frame Select Pin to D9. The configuration should look like this:




Under Tools, click on Clock Configuration. In the Clock Diagram:

- Change the Primary Oscillator frequency to 12,000,000 Hz, and select HS under POSCMOD.
- Change the FPLLRNG to 8-16 MHz, and FPLLMULT to MUL_33.
- In the Reference CLock section, check the ON and OE checkboxes,, and set ROTRIM1 to 29 and RODIV1 to 8. This should give a REFCLKO1 output of 12,288,000 Hz, which is 256 times the 48 KHz sample rate.

You should end up with a clock diagram like this:

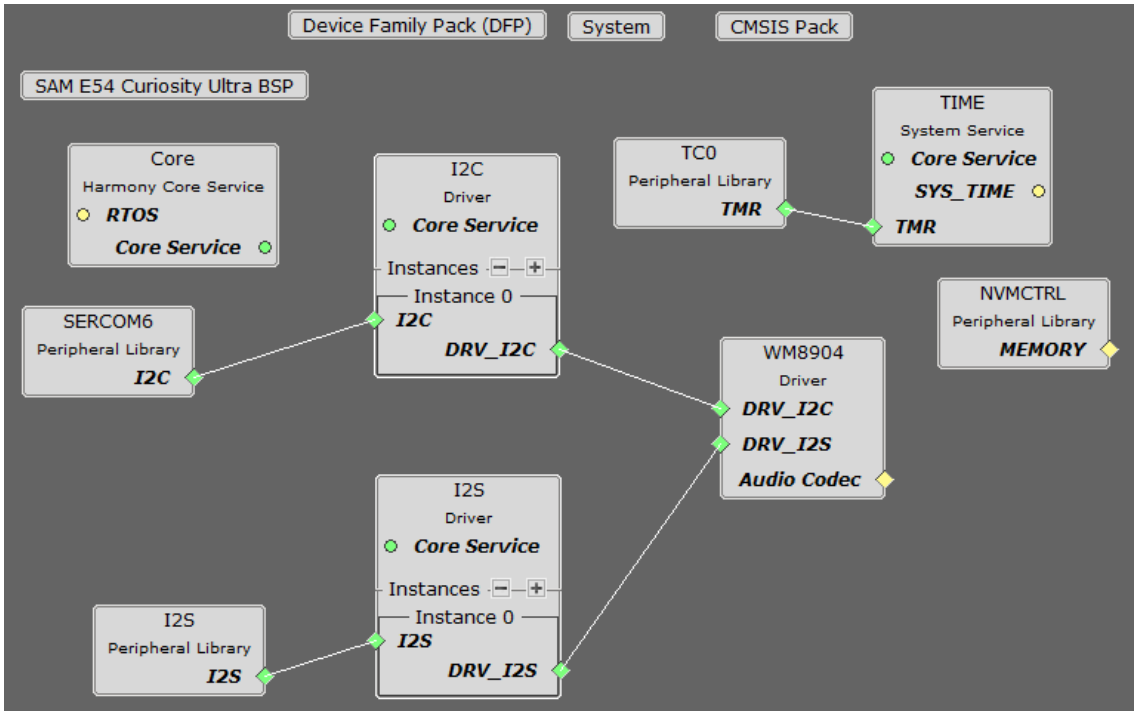



For projects using the E54, the I2S interface and the WM8904 as a Master (the WM8904 codec generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME54P20A). Click Finish.

In the MHC, under Available Components select the BSP SAM E54 Curiosity Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes:



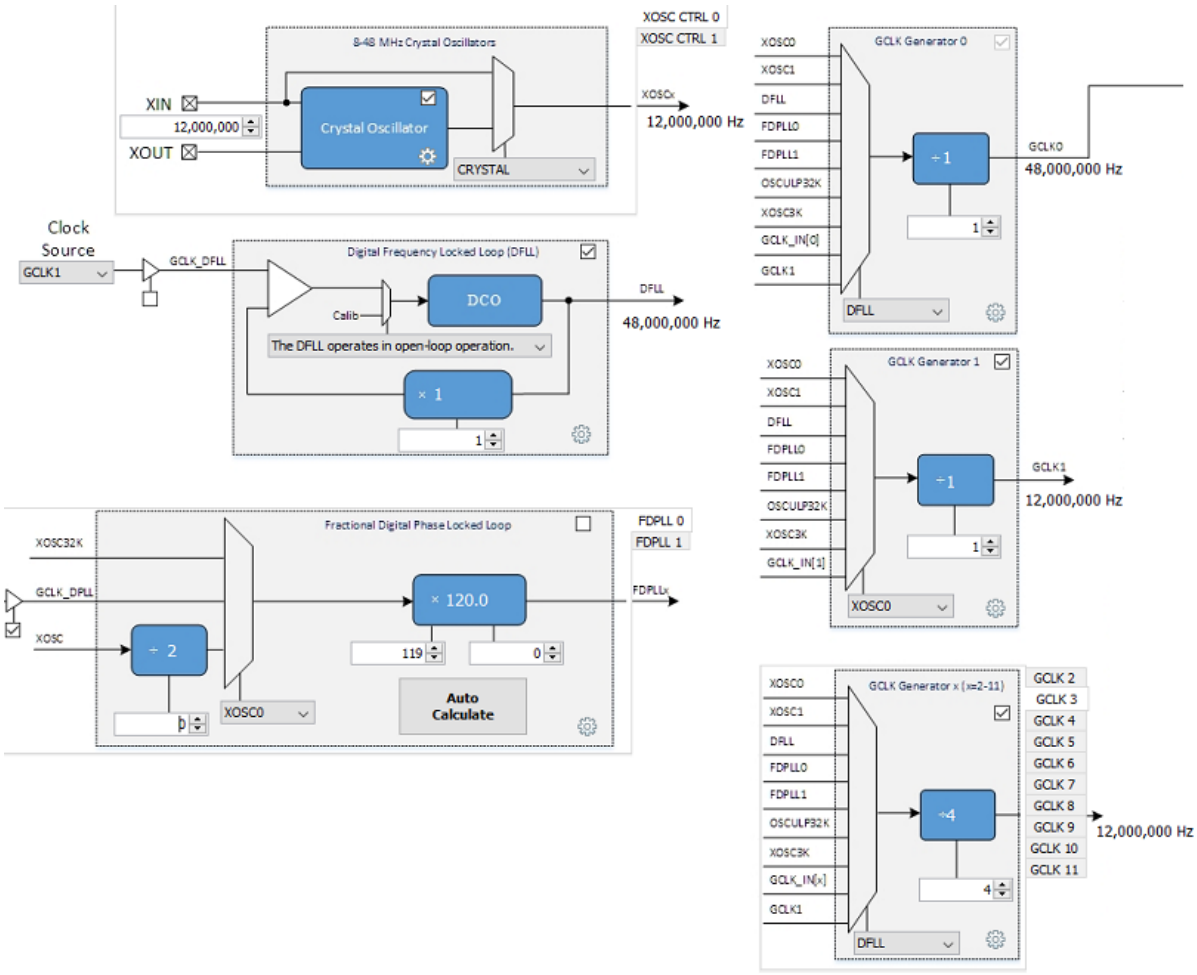

Click on the WM8904 Driver. In the Configurations Options, under Audio Data Format, change it to 32-bit I2S. Set the desired

Sample Rate if different from the default (48,000) under Sampling Rate.

In the Clock Diagram:

- Set enable the Crystal Oscillator, change it frequency to 12,000,000 Hz, and select CRYSTAL.
- Uncheck the Fractional Digital Phase Locked Loop enable (FDPLL 0).
- In the CLK Generator 0 box, change the input to DFLL for an output of 48 MHz.
- In the CLK Generator 1 box, change the input to XOSC0 with a divider of 1 for an output of 12 MHz.
- In the GCLK Generator, uncheck the selection for GCLK 2, and then select the GCLK 3 tab. Choose the DFLL as the input, with a divide by 4 for an output of 12 MHz.

You should end up with a clock diagram like this:

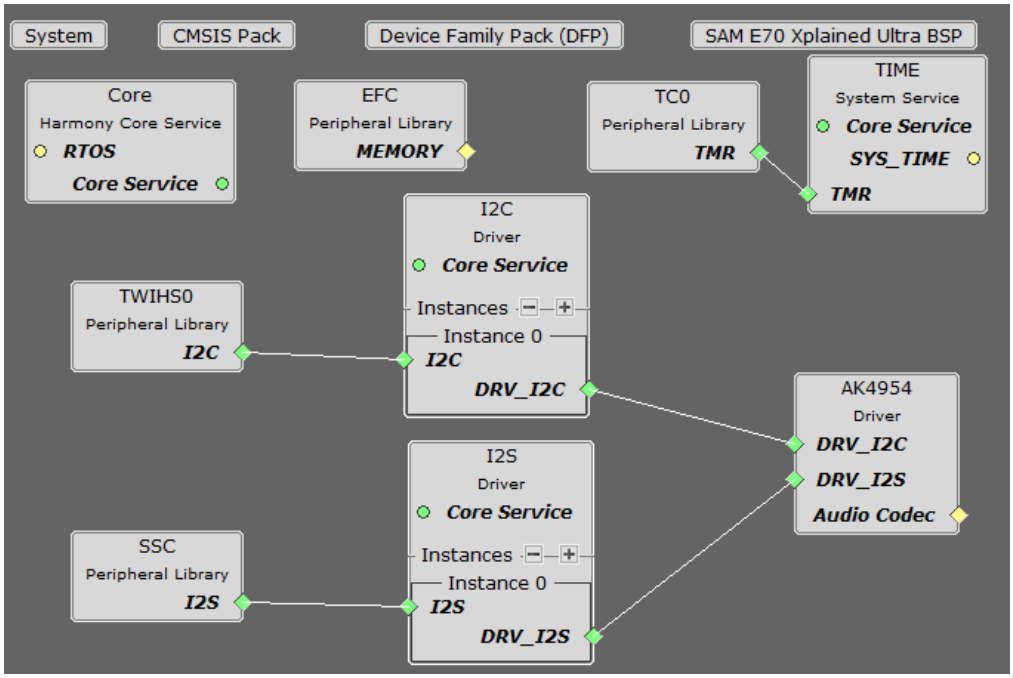



For projects using the E70, the SSC interface and the AK4954 as a Slave (the SAM E70 generates the I$^2$S clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME70Q21B). Click Finish.

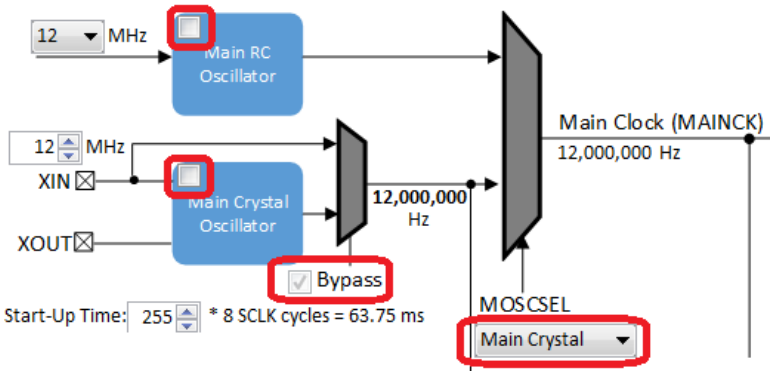
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on AK4954 Codec. Answer Yes to all questions. Click on the AK4954 Codec component (*not* the AK4954 Driver). In the Configuration Options, under AK4954 Interface, select I2SC instead of SSC. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes:

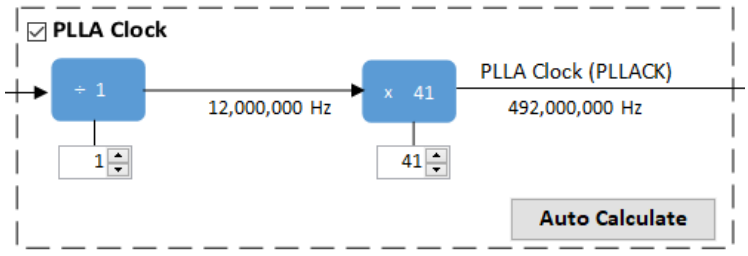



Click on the AK4954 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

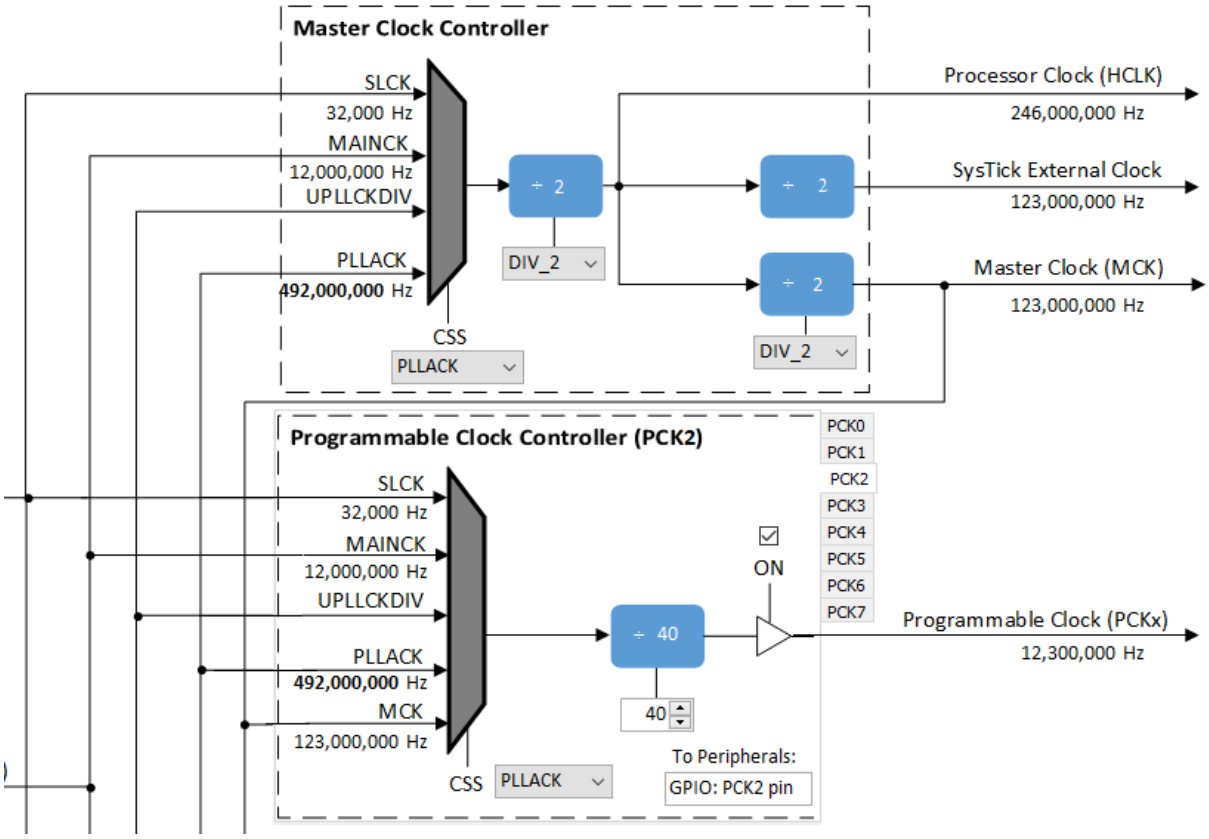
In the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




Still in the Clock Diagram, enable the PLLA Clock. Leave the divider at 1 (12 MHz) and change the multiplier to 41 for an output of 492 MHz.




In the Master Clock Controller, select the source (CSS) as PLLACK (492 MHz), all all three dividers as divide by 2 to generate a Processor Clock of 246 MHz and a Master Clock of 123 MHz. In the Programmable Clock Controller section, select the PCK2 tab, select the source (CSS) as PLLACK (492 MHz), the divider of 40 for a PCK2 output of 12,300,000 Hz



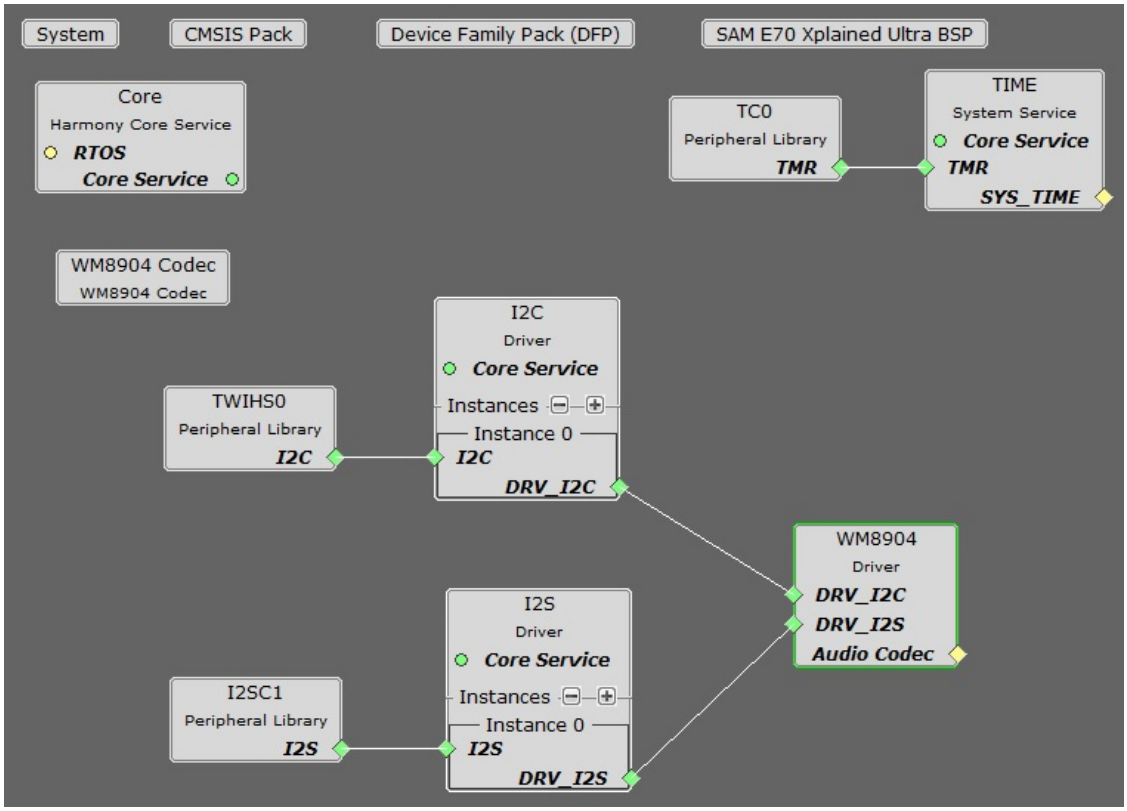

Then check the SSC checkbox in the **Peripheral Clock Controller** section..

For projects using the I2SC interface and the WM8904 as a Slave (the SAM E70 generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name based on the BSP. Select the appropriate processor (ATSAME70Q21B). (The WM8904 on the SAM V71 Xplained Ultra cannot be used with I2SC.) Click Finish.

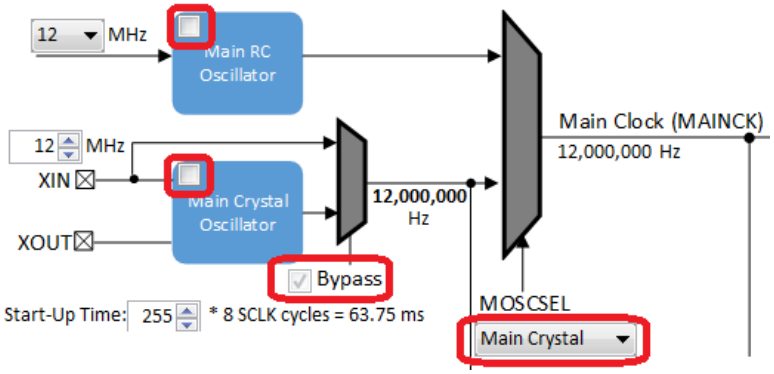
In the the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead of SSC. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:



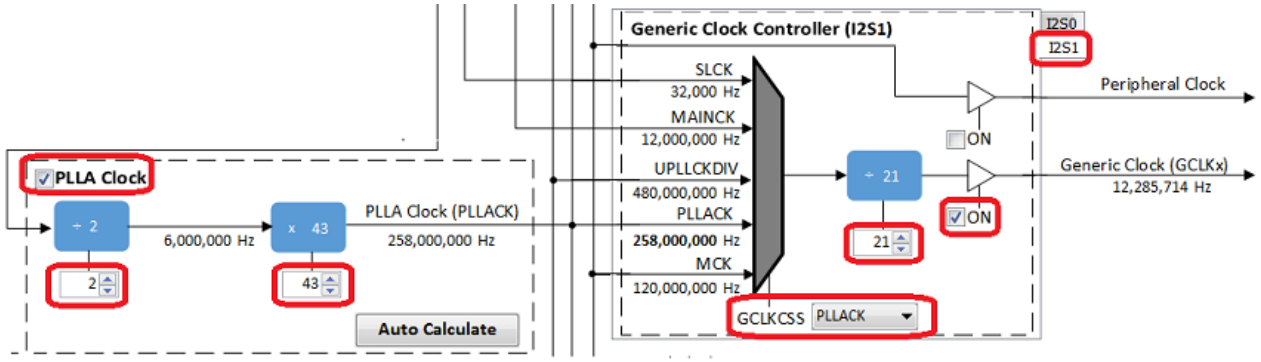

Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate. Select Enable Microphone Input, and Enable Microphone Bias for elecret microphones if appropriate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




In the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the **On** checkbox, and set CSS to MAINCLK (12 MHz).

The following tables show suggested settings for various sample rates in the Clock Diagram when using the I2SC Peripheral in Master mode. Make sure **PLLA Clock** checkbox is checked, and fill in the values for the PLLA Multiplier and Divider boxes. Select the I2S1 tab under **Generic Clock Controller**, set GCLKCSS to PLLACK, fill in the Divider value as shown, and check the checkbox next to it.




The values in the first table give the lowest error rate, but have varying PLLACK values so it is best to use the UPPCLKDIV selection for CSS under **Master Clock Controller**, for a Processor Clock of 240 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 43 | 258 MHz | 126 | 7998 | -0.025% |
| 16000 | 2 | 43 | 258 MHz | 63 | 15997 | -0.0187% |
| 44100 | 1 | 16 | 192 MHz | 17 | 41117 | 0.0385% |
| 48000 | 2 | 43 | 258 MHz | 21 | 47991 | -0.0187% |
| 96000 | 3 | 43 | 172 MHz | 7 | 95982 | -0.0187% |

The values in the second table have somewhat higher error rates, but use a PLLACK value of 294 MHz which is suitable to be used as a Processor Clock (using the PLLACK selection for CSS) which is closer to the maximum of 300 MHz.

| Desired Sample Rate | PLLA Multipler | PLLA Divider | PLLACK | I2SC Generic Clock Divider | Calculated Sample Rate | Error |
|---|---|---|---|---|---|---|
| 8000 | 2 | 49 | 294 MHz | 144 | 7975 | -0.3125% |
| 16000 | 2 | 49 | 294 MHz | 72 | 15950 | -0.3125% |
| 44100 | 2 | 49 | 294 MHz | 26 | 41170 | 0.1587% |
| 48000 | 2 | 49 | 294 MHz | 24 | 47851 | -0.3104% |
| 96000 | 3 | 49 | 294 MHz | 12 | 95703 | -0.3094% |

It is also possible to change the audio format from 16 to 32-bits. This changes need to be done in the MHC in both the WM8904 Driver and SSC Peripheral. In the current application code (app.h), a #define is also set to the current width.

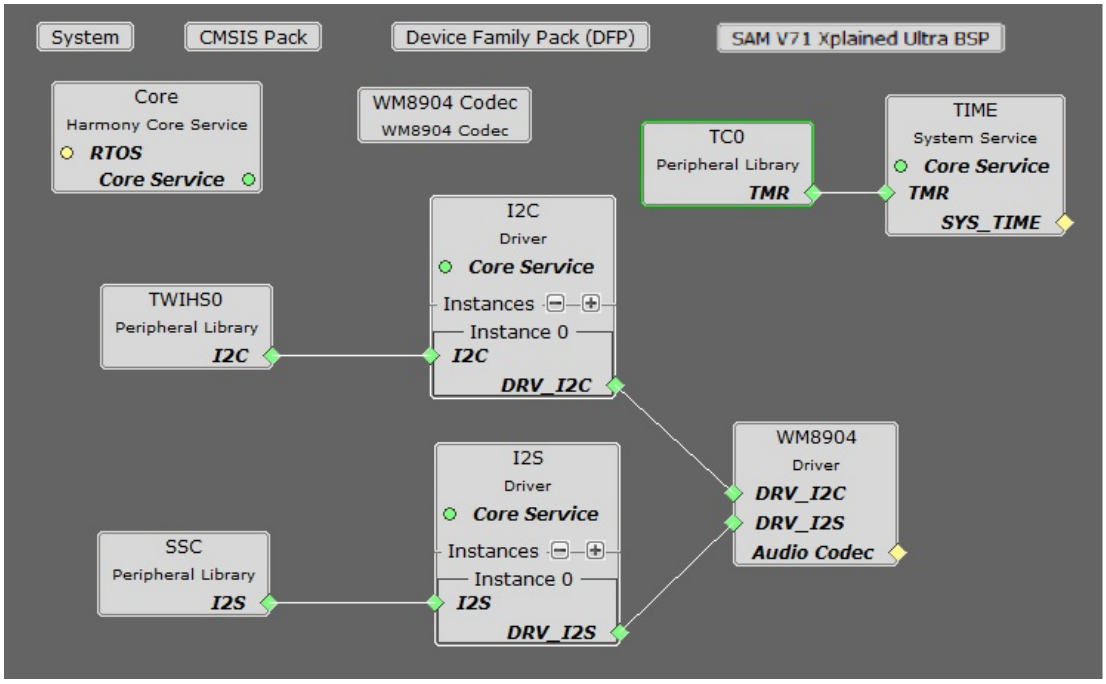
If using FreeRTOS, in the code you will need to move the call to DRV_WM8904_Tasks(sysObj.drvwm8904Codec0); from the SYS_Tasks function in src/config/<config_name>/tasks.c to inside the while(1) loop of _APP_Tasks (just before the call to APP_Tasks).

For projects using the SSC interface and the WM8904 as a Master (the WM8904 codec generates the $I^2S$ clocks):

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name same based on the BSP used. Select the appropriate processor (e.g. ATSAMV71Q21B) depending on your board. Click Finish.

In the MHC, under Available Components select the appropriate BSP (e.g. SAM V71 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be using FreeRTOS or not.

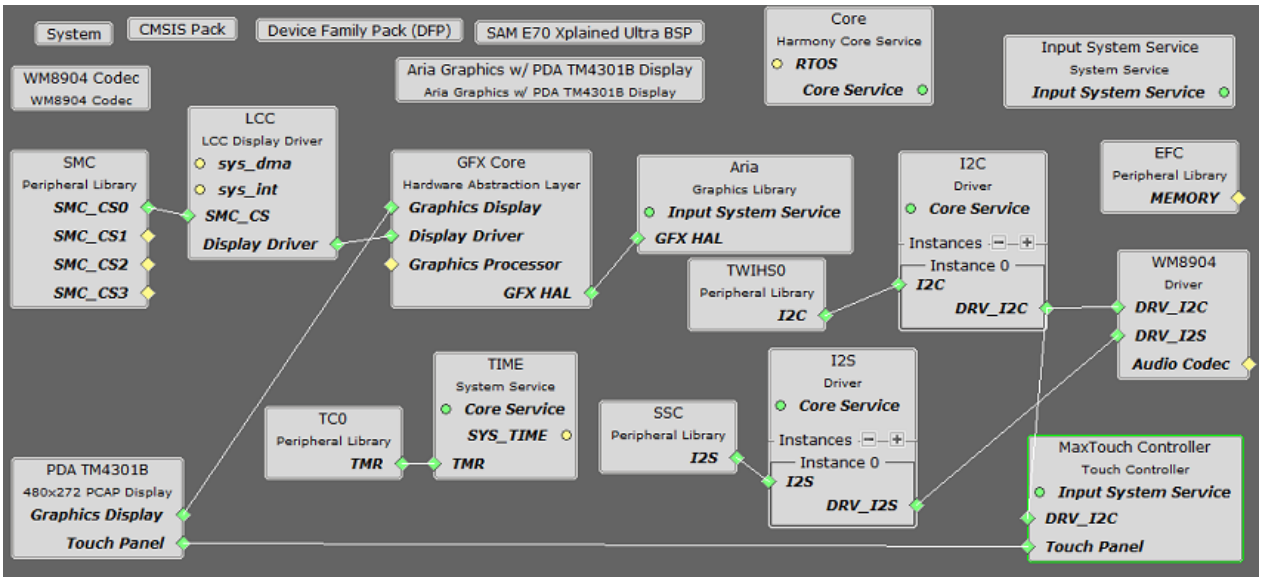
You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:




If building an application that is going to use the 480x272 display, then do the following instead. In the MHC, under Available Components select the BSP (SAM E70 Xplained Ultra). Under *Graphics>Templates*, double-click on Aria Graphics w/ PDA TM4301B Display. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

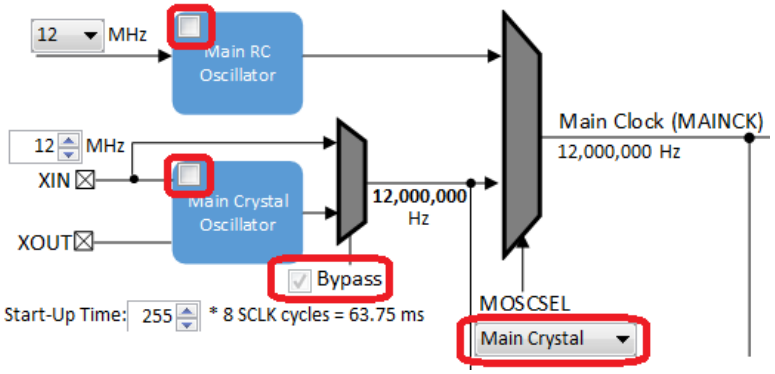
Then under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




In either case, click on the WM8904 Driver. In the Configurations Options, set the desired Sample Rate if different from the default (48,000) under Sampling Rate. Select Enable Microphone Input, and Enable Microphone Bias for elecret microphones if appropriate.

If using the SAM E70 Xplained Ultra board, in the Clock Diagram, in the Clock Diagram, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use of the 12 MHz external oscillator:




If using the ATSAMV71Q21B, in the Clock Diagram set MOSCEL to Main Crystal, uncheck the Bypass checkbox and RC Crystal Oscillator checkbox, and check the Main Crystal Oscillator box.

Also in the Clock Diagram, in the PCK2 tab of the **Programmable Clock Controller** section, check the On checkbox, and set CSS to MAINCLK (12 MHz). Then check the SSC checkbox in the **Peripheral Clock Controller** section.

It is also possible to change the audio format from 16 to 32-bits, and from I2S to Left Justified (SSC only). These changes need to be done in the MHC in both the WM8904, and SSC/I2SC Peripherals. In the current application code (app.h), a #define is also set to the current width.

If using FreeRTOS, in the code you will need to move the call to DRV_WM8904_Tasks(sysObj.drvwm8904Codec0); from the SYS_Tasks function in src/config/<config_name>/tasks.c to inside the while(1) loop of _APP_Tasks (just before the call to APP_Tasks).

## *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/audio_microphone_loopback`. To build this project, you must open the audio/apps/audio/audio_microphone_loopback/firmware/*.X project file in MPLAB X IDE that corresponds to your hardware configuration.

### MPLAB X IDE Project Configurations

The following table lists and describes supported configurations.

| Project Name | BSP Used | Description |
|---|---|---|
| mic_loopback_ pic32mz_ef_btadk_ ak4954 | pic32mz_ef_btadk | This demonstration runs on the PIC32MZ2048EFH144 processor on the Bluetooth Audio Development Kit (BTADK) and the AK4954 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and I$^2$S audio protocol using the I2S PLIB. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. |
| mic_loopback_ sam_e54_cult_ wm8904_ssc | sam_e54_cult | This demonstration runs on the ATSAME54P20A processor on the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and I$^2$S audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |
| mic_loopback_ sam_e54_cult_ wm8904_ssc_freertos | sam_e54_cult | This demonstration runs on the ATSAME54P20A processor on the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and I$^2$S audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. This demonstration also uses FreeRTOS. |
| mic_loopback_ sam_e70_xult_ ak4954_ssc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the AK4954 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and I$^2$S audio protocol using the SSC PLIB.. The AK4954 codec is configured as the slave, and the SSC peripheral as the master. |

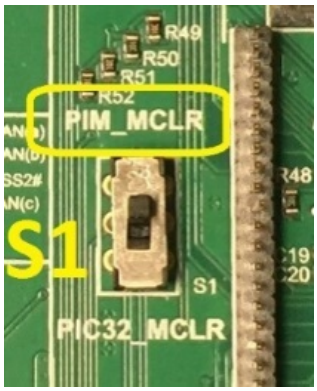
| | | |
|---|---|---|
| mic_loopback_ sam_e70_xult _wm8904_i2sc | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. |
| mic_loopback_ sam_e70_xult_ wm8904_i2sc_freertos | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the I2SC PLIB. The WM8904 codec is configured as the slave, and the I2SC peripheral as the master. This demonstration also uses FreeRTOS. |
| mic_loopback_ sam_e70_xult_ wm8904_ssc_wqvga | sam_e70_xult | This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, plus a PDA TM4301B 480x272 (WQVGA) display. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. This demonstration also uses FreeRTOS. |
| mic_loopback_ sam_v71_xult | sam_v71_xult | This demonstration runs on the ATSAMV71Q21B processor on the SAMV71 Xplained Ultra board along with the on-board WM8904 codec. The configuration is for a 16-bit data width, 48000 Hz sampling frequency, and $I^2S$ audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. |

### *Configuring the Hardware*

This section describes how to configure the supported hardware. SAM V71 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the SSC PLIB. No configuration is necessary.

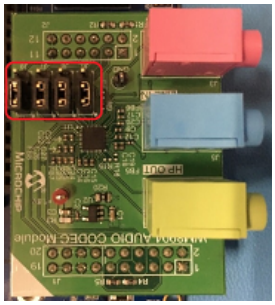
## Description

Using the Bluetooth Audio Development Kit and the PIC32MZ EF Audio PIM with the AK4954 Audio Codec Daughter Board and I2S PLIB. Plug the PIM onto the center square connector P1. Double-check to make sure the pins are aligned correctly. The AK4954 daughter board is plugged into the rear set of X32 connectors (J9/J10). Switch S1 on the PIC32 Bluetooth Audio Development Board (between the audio codec daughter board and the microcontroller PIM) should be set to PIM_MCLR.

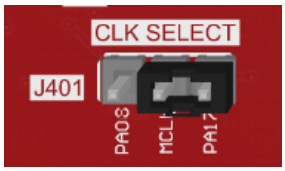



**Note:** The Bluetooth Audio Development Kit does not include the either the PIC32MZ EF Audio PIM or the AK4954 Audio Codec daughterboard, which are sold separately on microchipDIRECT as part numbers MA320018 and AC324954 respectively.

Using the SAM E54 Curiosity Ultra board and the WM8904 Audio Codec Daughter Board, and the I2S PLIB. All jumpers on the WM8904 should be toward the **front:**



In addition, make sure the J401 jumper (CLK SELECT) is set for the PA17 pin:

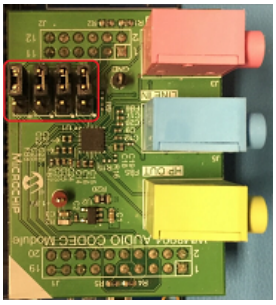



**Note:** The SAM E54 Curiosity Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the AK4954 Audio Codec Daughter Board, and the SSC PLIB. No special configuration needed.

**Note:** The SAM E70 Xplained Ultra board does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

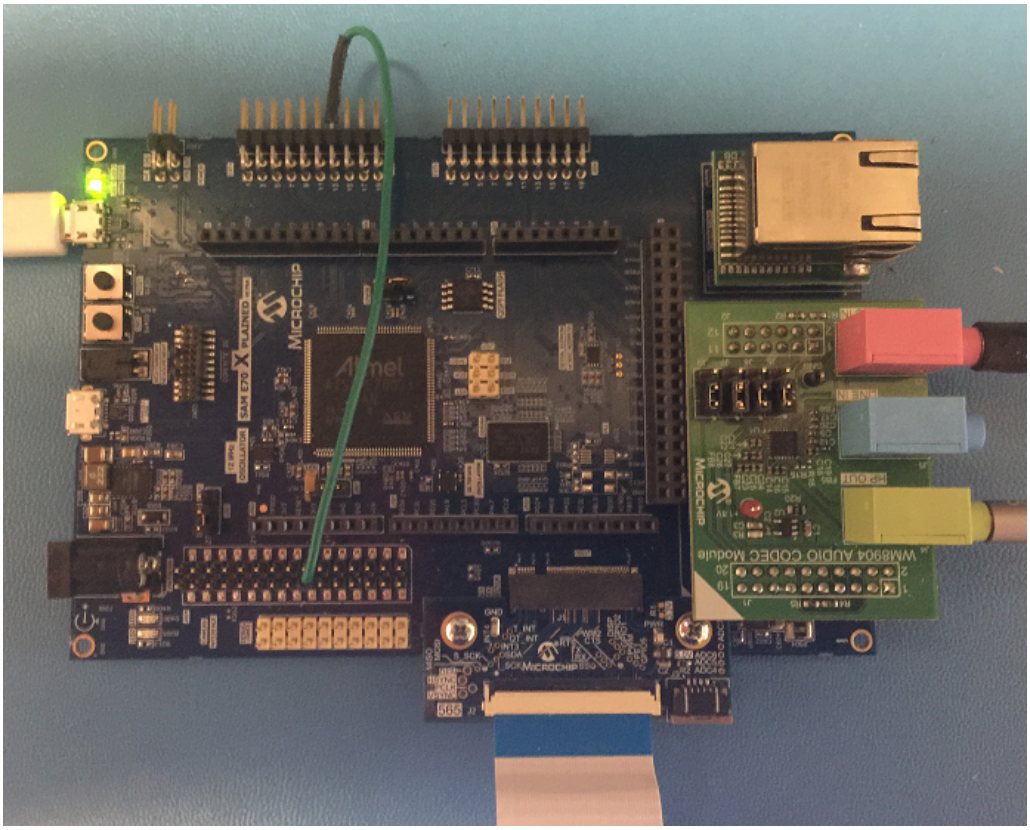
Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, and the I2SC PLIB. All jumpers on the WM8904 should be toward the **back:**



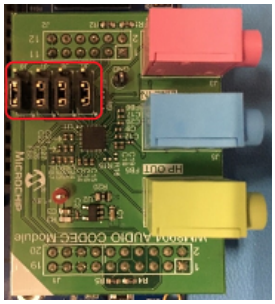
**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the AK4954 Audio Codec Daughter Board, and the SSC PLIB and the 480x272 display.

- Attach the flat cable of the PDA TM4301B 480x272 (WQVGA) display to the 565 daughterboard connected to the SAM E70 Xplained Ultra board GFX CONNECTOR.
- Add a jumper from connector EXT1 pin 13 to J601 (CAMERA INTERFACE) pin 14 as shown:



The WM8904 Audio Codec Daughter Board will be using the SSC PLIB; all jumpers on the WM8904 should be toward the **front:**



**Note:** The SAM E70 Xplained Ultra board does not include the PDA TM4301B 480x272 (WQVGA) display, which is sold separately on microchipDIRECT as part number AC320005-4, or the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM V71 Xplained Ultra board with on-board WM8904, with the SSC PLIB. No special configuration needed.

### *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

**Projects using a Pushbutton LED User interface:**

Compile the program using MPLAB X, and program the target device using the EDBG interface (ICD4 for the project using the Bluetooth Audio Development Kit). While compiling, select the appropriate MPLAB X IDE project. Refer to Building the Application for details.

Four different delays can be generated. **Tables 1-3** provides a summary of the button actions that can used to control the volume and delay amount.

1. Connect headphones to the HP OUT jack of the AK4954 or WM8904 Audio Codec Daughter Board (see **Figures 1-2**), or the HEADPHONE jack of the SAM V71 Xplained Ultra board.
2. If applicable, connect a microphone to the MIC IN jack of the WM8904 Audio Codec Daughter Board (see **Figure 2**), or the MICROPHONE jack of the SAM V71 Xplained Ultra board. (For the AK4954, the internal microphone contained on the daughterboard is used instead, see **Figure 1**.)
3. The lowest-numbered pushbutton (SW0 or SW1) and LED are used as the user interface to control the program, as shown in Tables 1-3 below, depending on the board used. Initially the program will be in delay-setting mode (LED on) at a medium volume setting. Pressing the pushbutton with the LED on will cycle through four delay settings.
4. Pressing the pushbutton longer than one second will change to volume-setting mode (LED off). Pressing the pushbutton with the LED off will cycle through four volume settings (including mute). There is never any delay when setting the volume.
5. Pressing the pushbutton longer than one second again will switch back to delay-setting mode again (LED on).

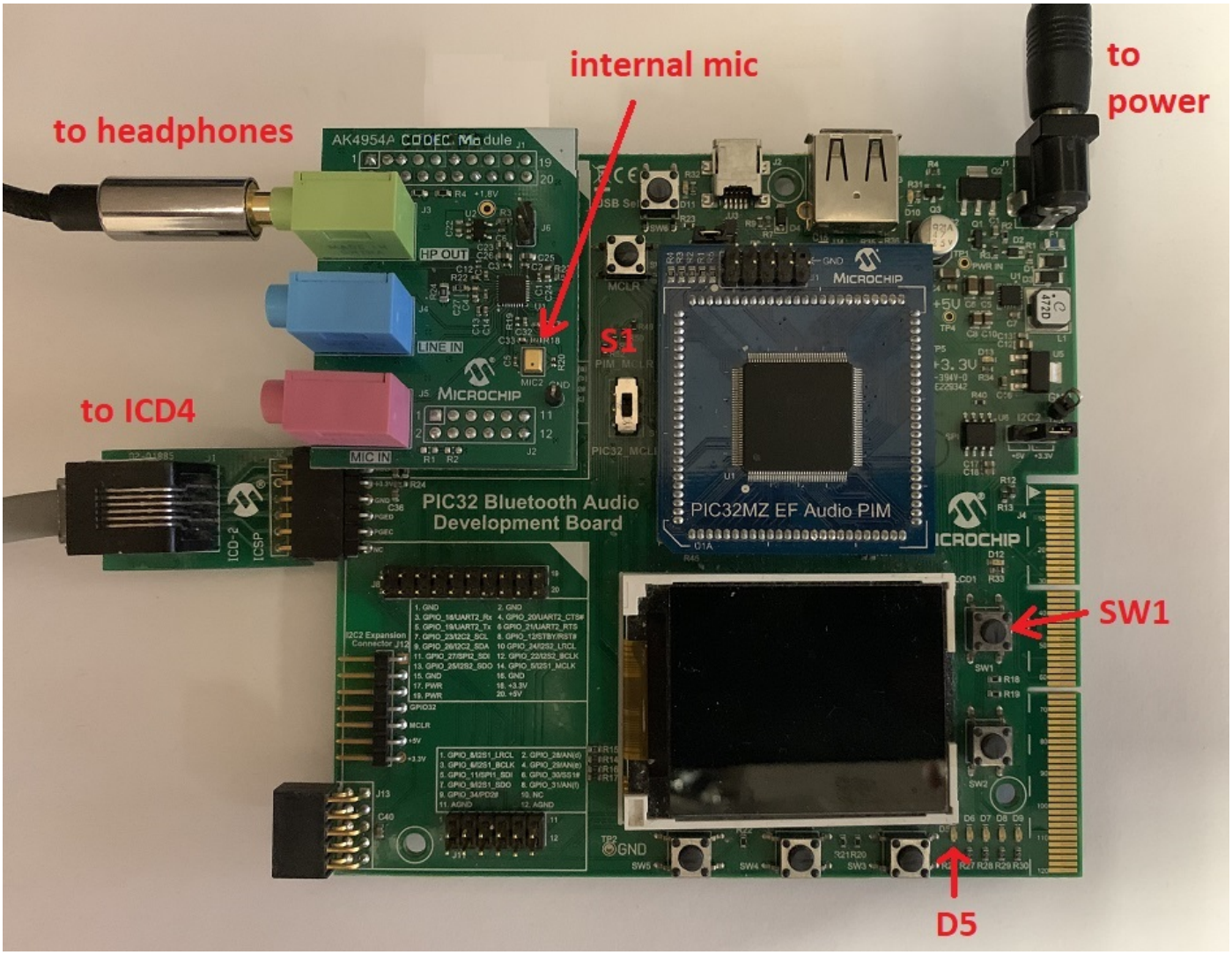



**Figure 1: AK4954 Audio Codec Daughter Board on Bluetooth Audio Development Kit with PIC32MZ EF Audio PIM**

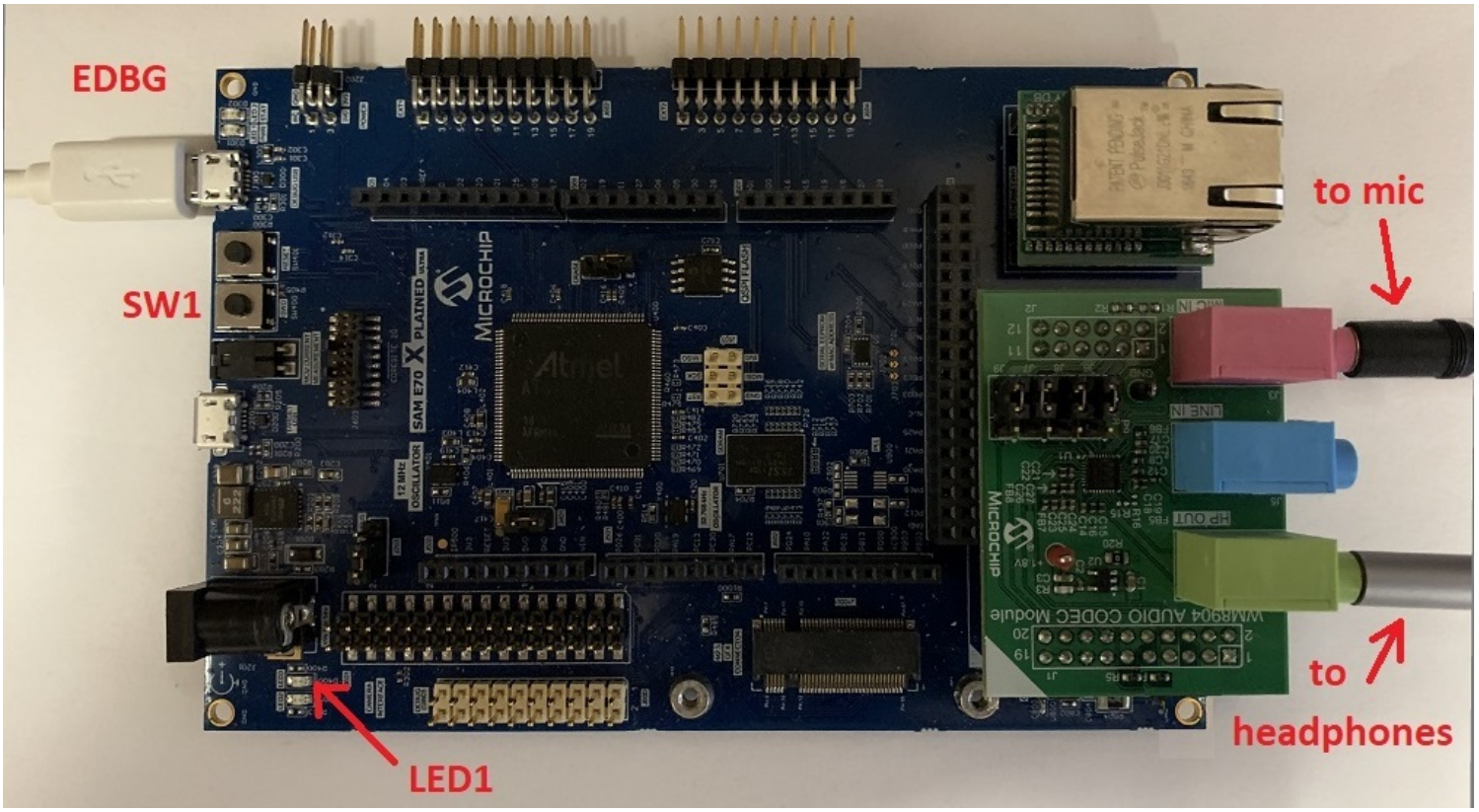



**Figure 2: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board**

## Control Descriptions

**Table 1: Button Controls for Bluetooth Audio Development Kit**

| Control | Description |
|---|---|
| SW1 short press | If LED D5 is off, SW1 cycles through four volume levels (one muted). If LED D5 is on, SW1 cycles through four delays: ¼ second, ½ second, 1 second and 1½ seconds. |
| SW1 long press (> 1 second) | Alternates between modes (LED D5 on or off). |

**Table 2: Button Controls for SAM E54 Curiosity Ultra board and SAM E70 Xplained Ultra board**

(On some E70 boards, SW0/LED0 are the lowest numbered pushbutton and LED, so use Table 3 instead.)

| Control | Description |
|---|---|
| SW1 short press | If LED1 if off, SW1 cycles through four volume levels (one muted). If LED1 is on, SW1 cycles through four delays: ¼ second, ½ second, 1 second and 1½ seconds. |
| SW1 long press (> 1 second) | Alternates between modes (LED1 on or off). |

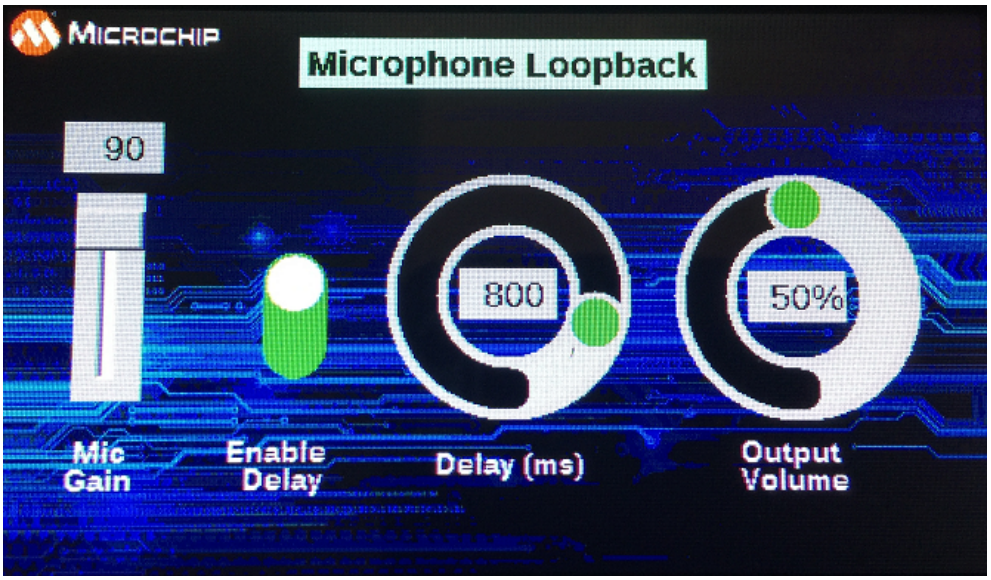
**Table 3: Button Controls for SAM V71 Xplained Ultra board**

| Control | Description |
|---|---|
| SW0 short press | If LED0 if off, SW0 cycles through four volume levels (one muted). If LED0 is on, SW0 cycles through four delays: ¼ second, ½ second, 1 second and 1½ seconds. |
| SW0 long press (> 1 second) | Alternates between modes (LED0 on or off). |

**Project using a Graphical User Interface:**

Compile the program using MPLAB X, and program the target device using the EDBG interface. While compiling, select the appropriate MPLAB X IDE project. Refer to Building the Application for details.

1. Connect headphones to the HP OUT jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).
2. Connect a microphone to the MIC IN jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).
3. Initially the program will be in delay-setting mode, with a 800 ms delay and a medium volume setting.
4. The Delay can be adjusted from 10 ms to 1000 ms (1 second) using the center circular control.
5. The Output Volume can be adjusted from 0% to 100% using the circular control on the right.
6. The Microphone Gain can be adjusted from 0% to 100% using the slider on the left.
7. Adjusting the microphone gain or output volume causes the delay to be temporarily disabled. Re-enable the previous value using the Enable Delay switch.
8. The Enable Delay swtich can also be manually toggled at any time to disable the delay and provide immediate microphone loopback to the headphones.




## usb_microphone

This topic provides instructions and information about the MPLAB Harmony 3 USB Microphone demonstration application, which is included in the MPLAB Harmony Library distribution.

### Description

This demonstration application configures the development board to implement a USB Microphone device configured to run at 16 Khz sampling rate at 16 bit per sample.

The USB Device driver in Full Speed mode will interface to a USB Host (such as a personal computer) via the USB Device Stack using the V1.0 Audio Function Driver. The embedded system will enumerate with a USB audio device endpoint and enable the host system to input audio from the USB port using a standard USB Full-Speed implementation. The embedded system will stream USB playback audio to the Codec. The Codec Driver sets up the audio interface, timing, DMA channels and buffer queue to enable a continuous audio stream. The digital audio is transmitted to the codec through an I2S data module for playback through a headphone jack.

#### Architecture

The application runs on the SAM E70 Xplained Ultra Board (E70XULT), which contains can operate at 300 MHz using the following features:

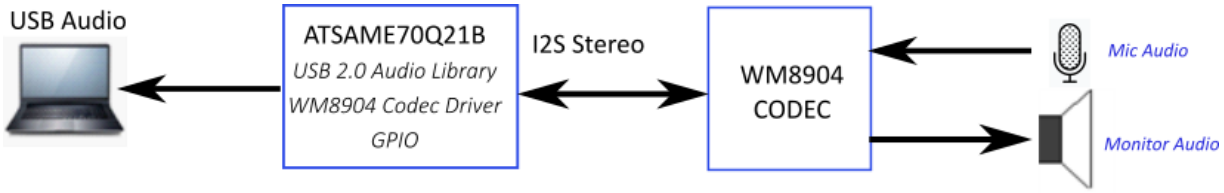
- One push button (SW1)
- Two LEDs (amber LED1 and green LED2). Only LED 1 can be used for a USB Device, that requires VBUS sense.
- WM8904 Codec Daughter Board mounted on a X32 socket
- USB Device interface

**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The usb_microphone application uses the MPLAB Harmony Configurator to setup the USB Audio Device, codec, and other items in order to play back the USB audio through the WM8904 Codec Module.

A USB device is connected to the micro-mini USB device connector. The application detects the cable connection, which can also supply device power; recognizes the type of connection (Full Speed); enumerates its functions with the host, isochronous audio streaming playback or recording through the device. Audio stream data is buffered in 1 ms frames for playback or recording using the WM8904 Codec daughter board. Audio is can be heard through the Headphone jack (HP OUT).

The following figure shows the basic architecture for the demonstration.



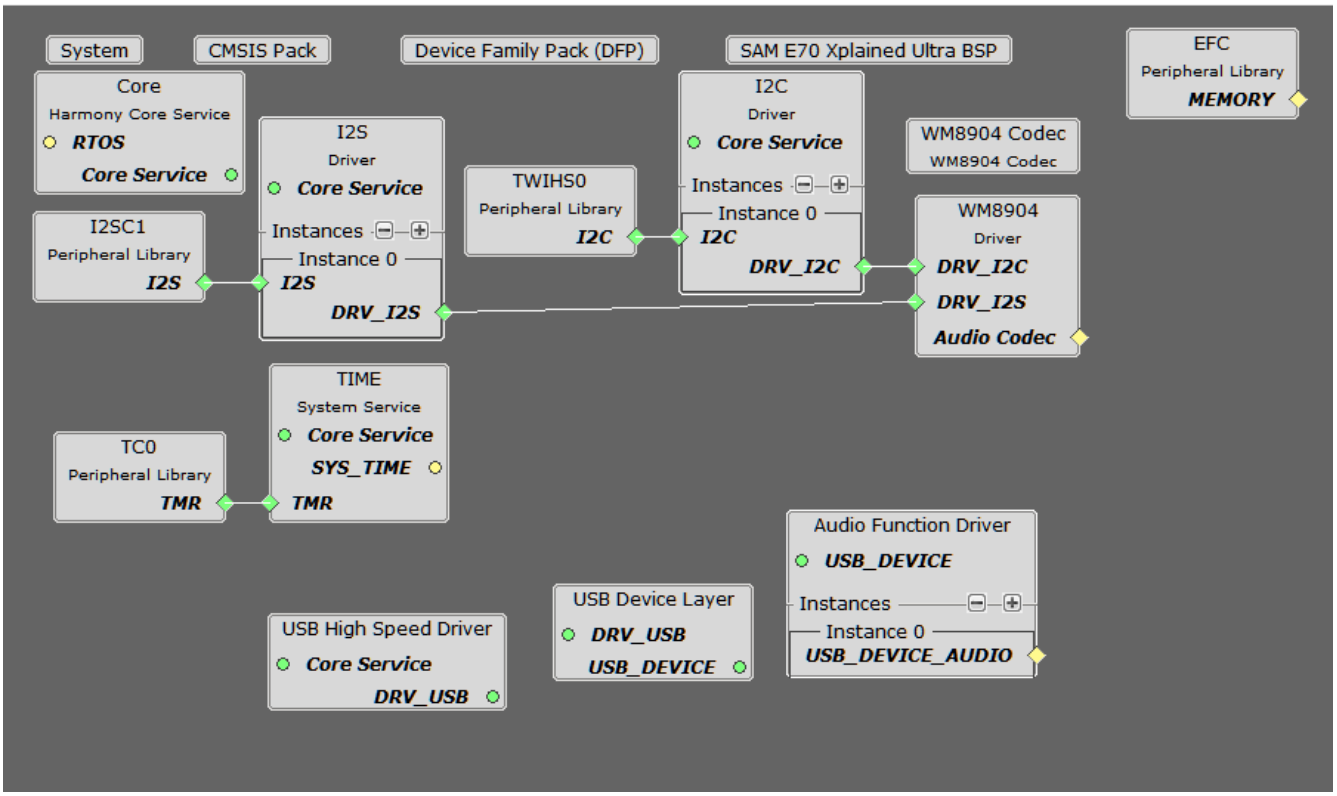

*USB Microphone Application Block Diagram*

## Demonstration Features

- Audio recording using an WM8904 codec daughterboard on the SAM E70 Xplained Ultra (E70XULT) board.
- USB device connection to a host system using the USB Library Device Stack for a USB Microphone device implemented on the SAM E70XULTZ
- E70 XULT Button processing for microphone gain control
- Using the USB Library
- USB Attach/Detach and mute status using an LED.
- Looping back of the microphone data to the codec headphone output for for playback monitoring of the input audio.

**Harmony Configuration and Code Generation**

The MPLAB-X Harmony Configurator (MHC) Project Graph for usb_microphone_basic is shown below.




*USB Microphone Application MPLAB-X Harmony Configurator Project Graph*

Each block provides configuration parameters to generate the application framework code. This includes all the needed drivers, middleware, libraries. The generated framework code is placed under the firmware/src/config directory under the Harmony 3 project configuration. This Harmony 3 application has two configurations, as described below :

1. usb_microphone_basic_sam_e70_xult_wm8904_i2sc1_usb - A bare-metal configuration
2. usb_microphone_basic_sam_e70_xult_wm8904_i2sc1_usb_freertos> - A configuration that uses the 3rd party FreeRTOS operating system.

The harmony application system components, drivers and mideleware is configured based on the project graph.

In many cases the default configuration values are used, but in general each application requires modifications to these parameter.

The usb_microphone_basic custom application code code is located in the firmware/src directory of the application folder. The app.c and app.h files are initially generated as stub files in this folder. The middleware and library APIs located in definitions.h (as generated by MHC), for the given project graph.

The usb_microphone application utilizes a state machine with the following functions:

1. Setup the drivers and USB Library interface as used by the application

a. WM8904 Codec Driver

b. Timer

c. USB Audio

2. Respond to USB Host control commands (“Attach”, “Detach”, “Suspend”, etc.)
3. Initiate and maintains the audio data streaming for the "USB Record" function as data is received from the microphone codec.

All Harmony applications use the function SYS_Initialize function located in the MHC generated file initialization.c. This is executed from main to initialize various subsystems such as the clock, ports, BSP (board support package), codec, usb, timers, and interrupts. The application APP_Initialize function in app.c is also executed at the end of this routine to setup the application code state machine.

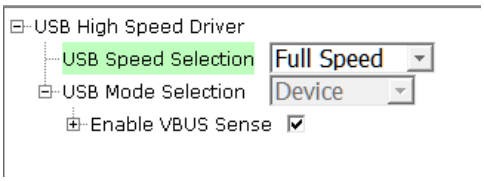
For the bare-metal configuration The USB, WM8904 driver, and the application code state machines(APP_Tasks routine in app.c) are all updated via calls located in the function SYS_Tasks, executed from the main polling loop, located in the MHC generated tasks.c.

For the FreeRTOS configuration 4 tasks are configured within 3 processes of a round-robin scheduler. The APP_Tasks and Codec update tasks is in 1 of the scheduled process, while the USB High Speed Driver and USB Device Driver are in there own separate tasks.

## Tools Setup Differences

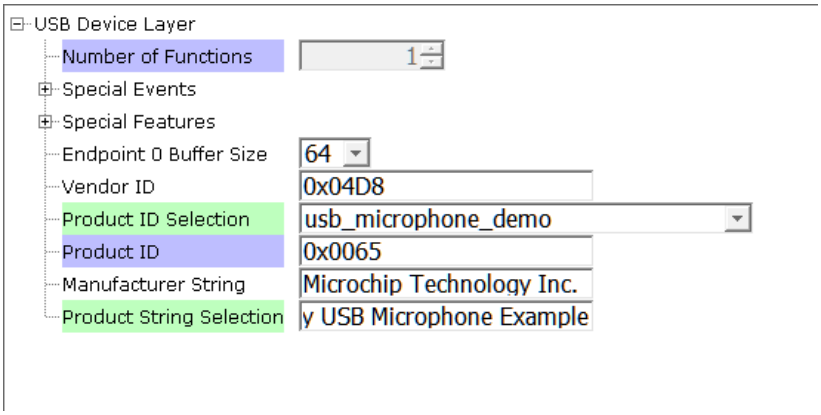
*USB Configuration*

The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is selected to by “Full Speed” (not “High Speed”):



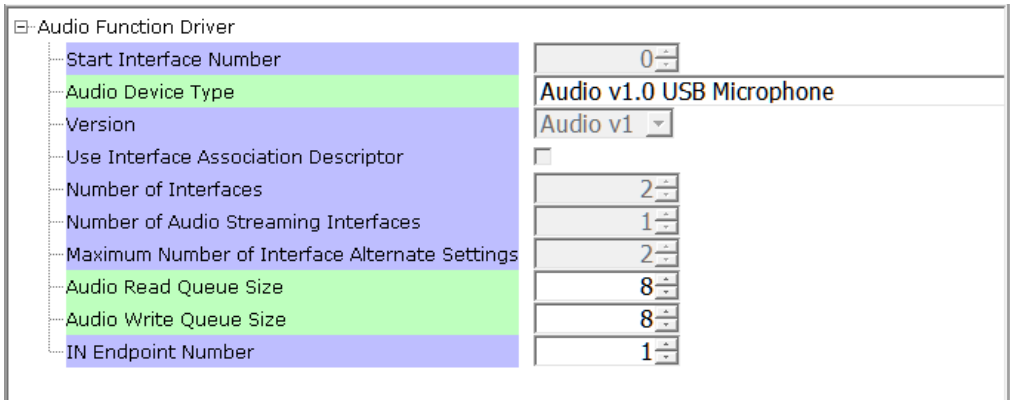

*USB High Speed Driver Configuration*

The USB Device Layer is configured by selecting the “usb_microphone_demo” with an endpoint buffer size of 64 (bytes). The Audio Function Driver is configured for 5 USB endpoints with a single function having 3 interfaces. All of these are defined for a USB Microphone device in the fullSpeedConfigurationDescriptor array variable structure (located in initialization.c under the config folder). This structure defines the connection to the host at 48 Khz with 16 bit stereo channel data. A packet queue of length APP_QUEUE_SIZE (set to 2) is used for playback data. The maximum USB packets size is set to 16 samples * 2 channels per sample * 2 bytes per channel= 64 bytes, which gives a 1ms stereo sample packet size at 16Khz (the standard data frame length at this rate), thus the buffer size needs to be of the same size.




*USB Device Layer Configuration*

The Audio Function Driver is configured with a Audio Read Queue that matches that of the codec driver write queue for this Audio V1.0 USB Microphone interface.



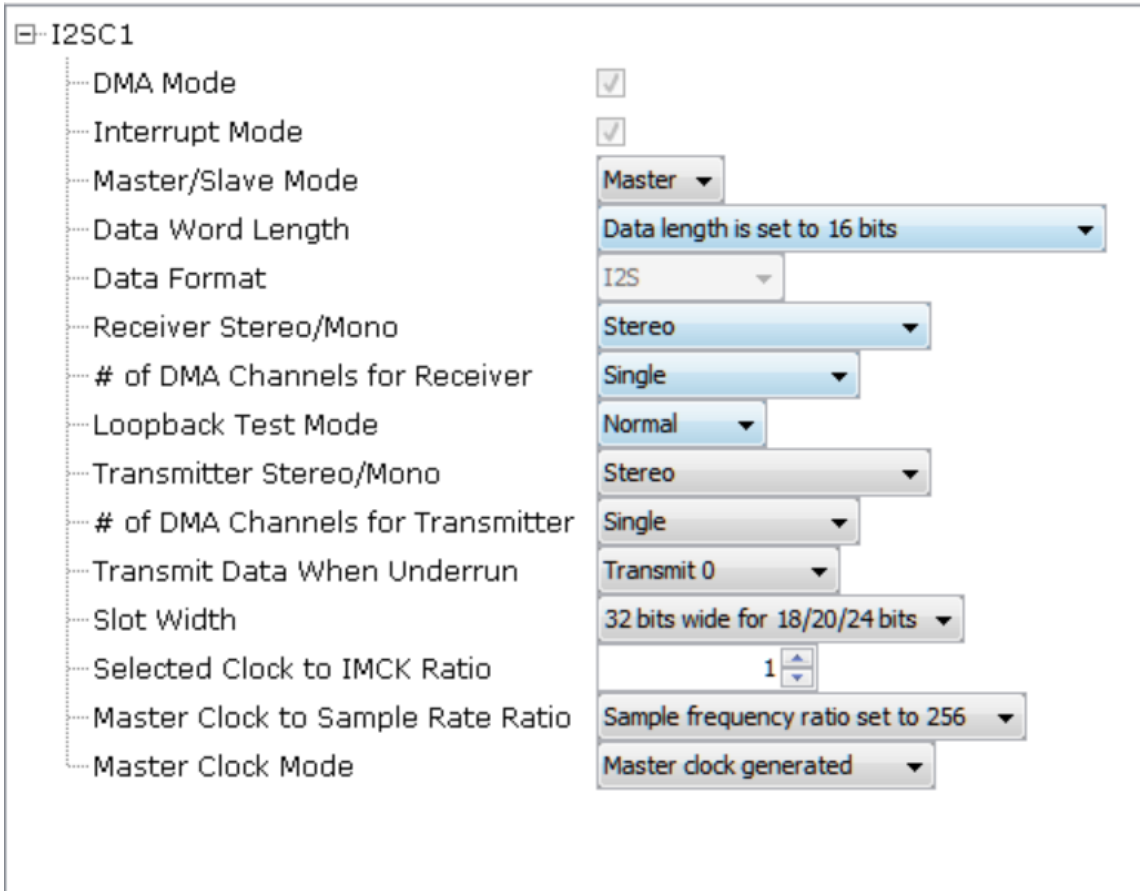

*USB Audio Function Configuration*

*The WM8904 Codec Configuration*

The WM8904 codec uses a TWIHS0 (I2C) interface for module configuration and control register setting and a I2SC1 (I2S) interface for audio data. The default settings are used.

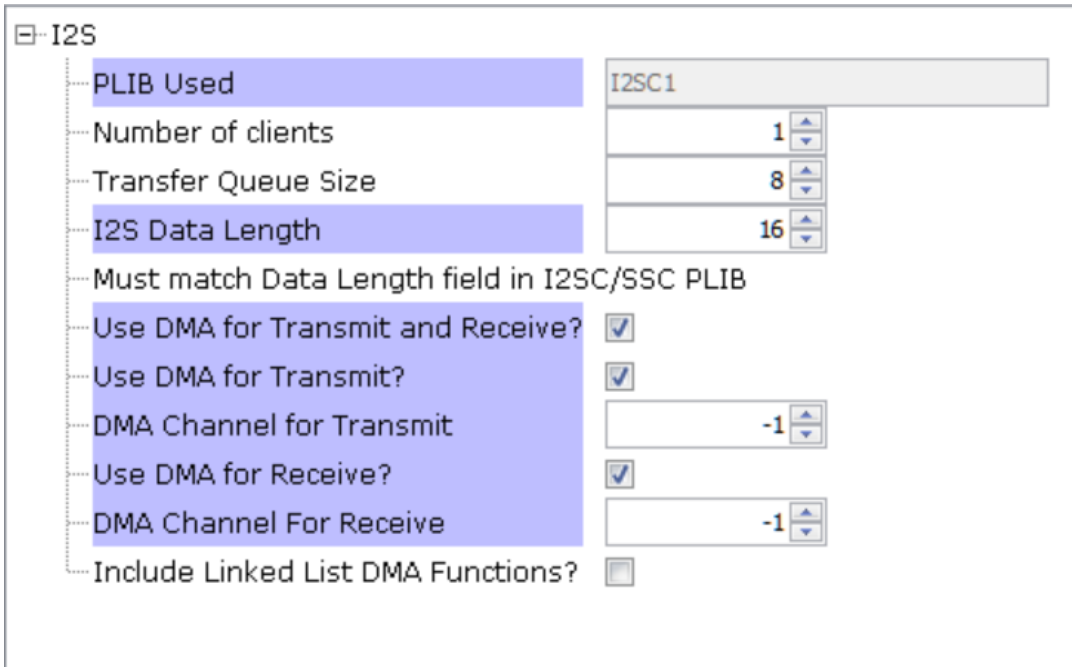
The default values are used for the TWIHS and I2C drivers.

The E70 I2SC1 driver is set to be the master with a data length of 16, 2 channels (stereo) and a MCLK divider value of of 256. These a default values, as shown below:



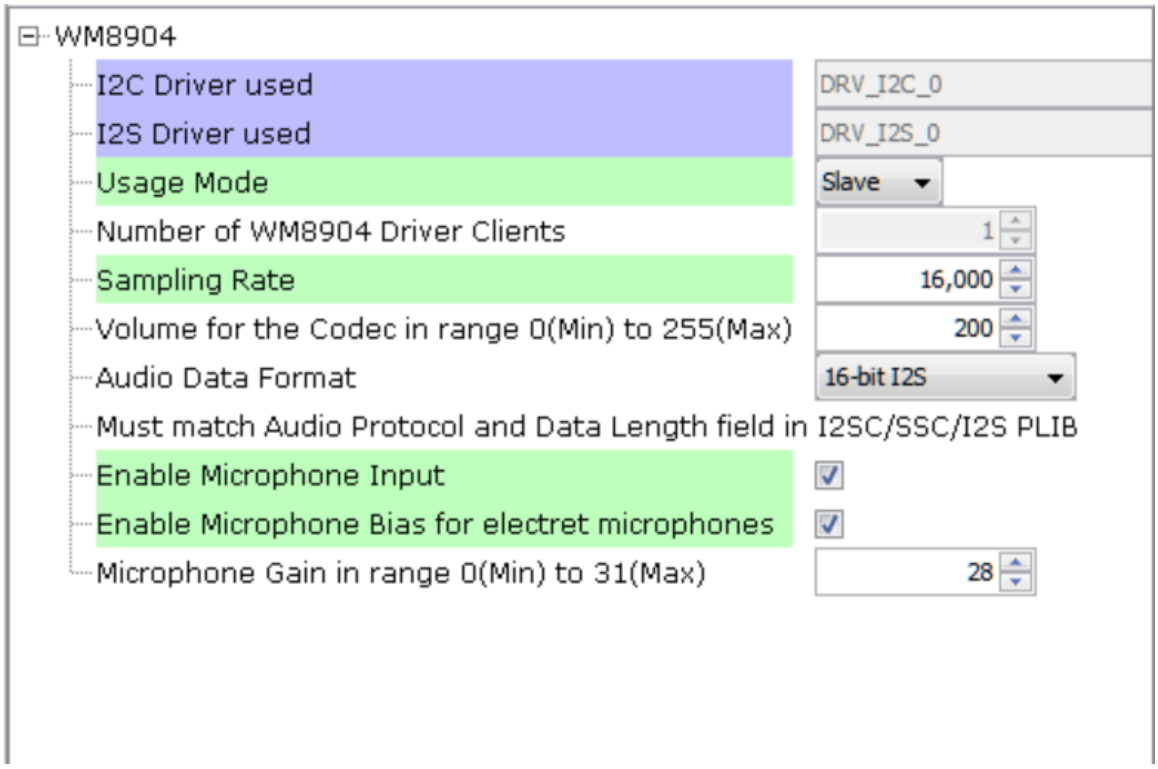

*I2SC1 Peripheral Configuration*

The I2S driver is configured with transfer queue size of 8 (default value) that matches that of the USB Read Queue, as shown below:




*USB Microphone I2S Configuration*

The WM8904 Codec is configured with an I2S slave interface operating at 16000 Hz sampling rate, as shown below:




*USB Microphone I2S Configuration*

Also, the microphone input is enabled with bias for an electret microphone.

*Pin Manager Configuration*

The buttons, LED, Switch, I2S and I2C interfaces using GPIO pins via the Microchip Harmony Configurator (MHC) Pin Manager, as follows:

When the I2SC1 is used the following pins are used:

| NAME | PORT | E70 PIN | Notes |
|---|---|---|---|
| I2SC1_WS | PE00 | 4 | I2S LRCK (Word Select) |
| I2SC1_DO0 | PE01 | 6 | I2S DO (Data Out) |
| I2SC1_DI0 | PE02 | 7 | I2S DI (Data In) |
| I2SC1_CK | PA20 | 22 | I2S BCLK (Bit Clock) |
| I2SC1_GCLK | PA19 | 23 | I2S MCLK (Bit Clock as used by the E70 I2SC1, I2S Master) |
| PMC_PCK2 | PA18 | 24 | I2S MCLK (Master Clock as used by the WM8904 Codec I2S Slave) |
| SWITCH | PA11 | 66 | Push Button |
| LED1 | PA05 | 73 | |
| TWIHS0_TWCK0 | PA04 | 77 | I2C |
| TWIHS0_TWD0 | PA03 | 91 | I2C |
| STDBY | PD11 | 98 | |
| LED2/VBUS DETECT(J204) | PB08 | 141 | J204 set to VBUS DETECT for USB Device |

*Clock Manager*

All clocks are generated from the 12 Mhz Main Clock oscillator, including the following.

| Clock | Value | Description |
|---|---|---|
| HCLK | 240 MHz | Processor Clock |
| USB FS | 48 MHz | USB Full Speed Clock |

The I2SC1 peripheral is set as the I2S bus master. It uses the I2SC1_GCLK to generate the I2S timing to the codec slave I2S interface.

The I2SC clocks are setup for 16Khz sampling rate (LRCK), with stereo 16 bit samples, giving a 32 bit sample frame. The frame bit clock (BCLK) and the I2S Master clock (MCLK) is calculated as follows:

LRCK = 16 KHz FRAME_SIZE = 16 bits/channel * 2 channels BCLK = LRCK * 16 * 2 = .512 KHz MCLK_MULT = 256 MCLK = MCLK_MULT * LRCK = 4.096 KHz

The generated MCLK should be set as close as possible to the calculated values as given below:

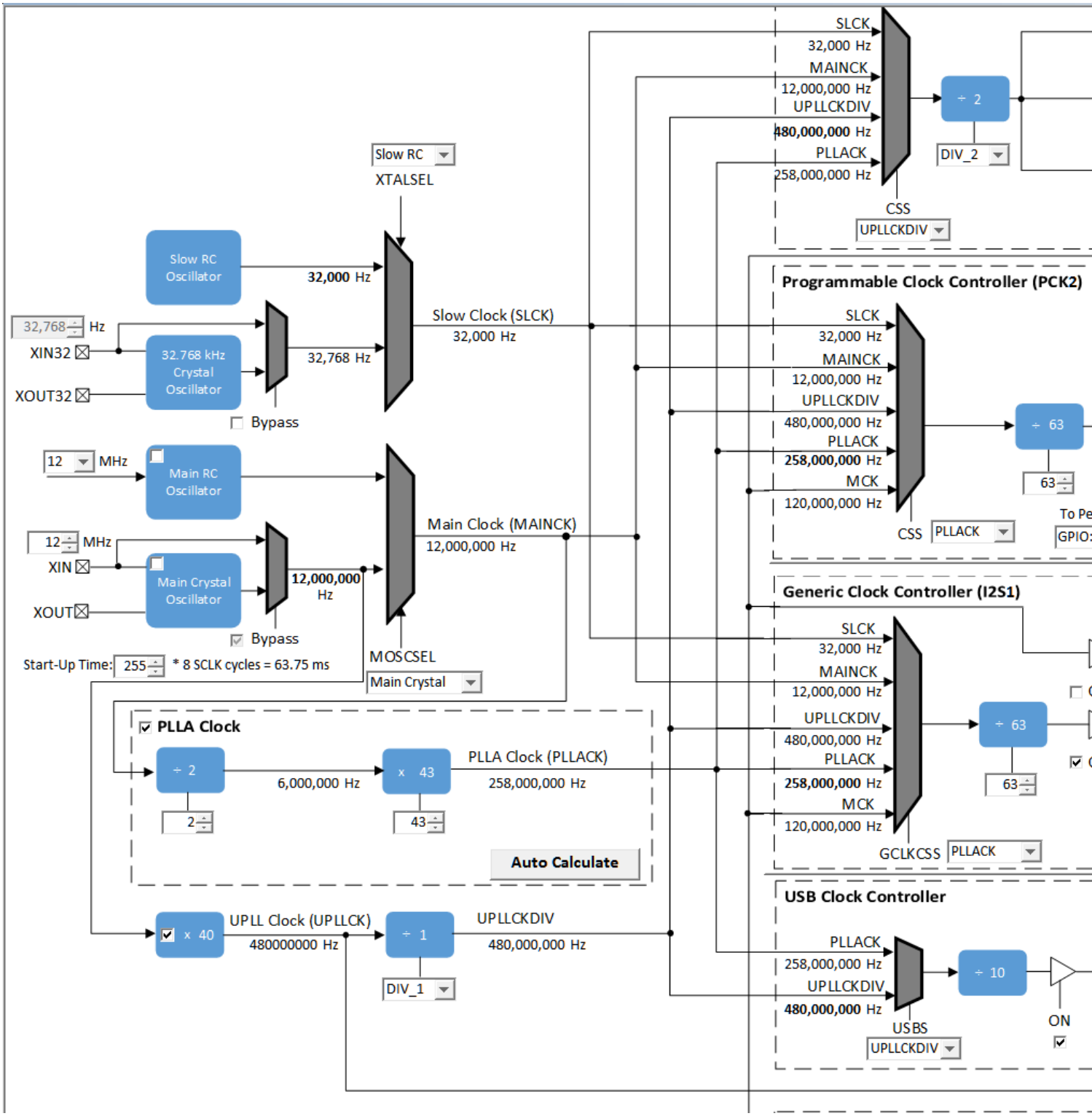
| CLOCK | I2S Function |
|---|---|
| I2SC1_WS | LRCK |
| I2SC1_CK | BCLK |
| I2SC1_GCLK | MCLK(I2SC1) |
| PCK2 | MCLK(X32) |

The E70 XULT board only provides PCK2 as the source of the MCLK to the X32 WM8904 Codec Daughter Board connector. It must be generated the same as I2SC1_GCLK.

The actual generated value utilizes the PLLA clock as the source for both the I2SC1_GCLK and the PCK2. These values generate the following clocks These I2S clocks are generated from the I2SC1 GCLK peripheral acting as I2S master.

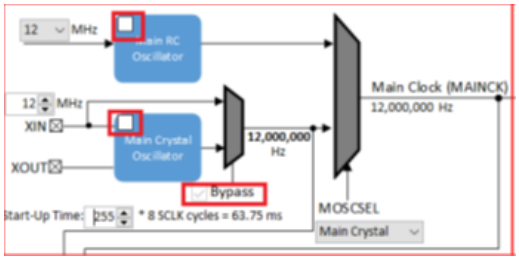
*MPLAB Harmony Configurator: Clock Diagram Configuration*

The MHC Clock Diagram to generate the processor (HCK), PCK2 (I2S MCLK), and the I2SC1 MCLK (I2SC1_GCLK) is given below.



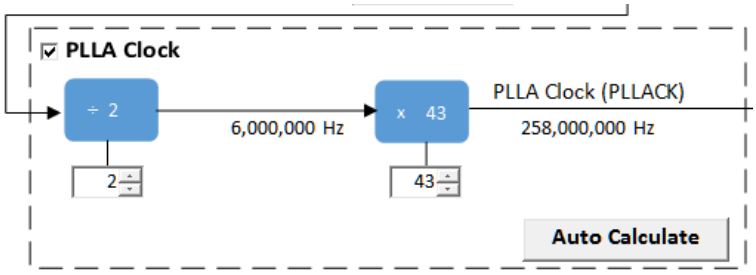

*USB Microphone MHC Clock Diagram*

Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is checked, it will cause the Main Crystal Oscillator to become disabled. An external MEMS oscillator input on the XIN pin is used for Main Clock generation.



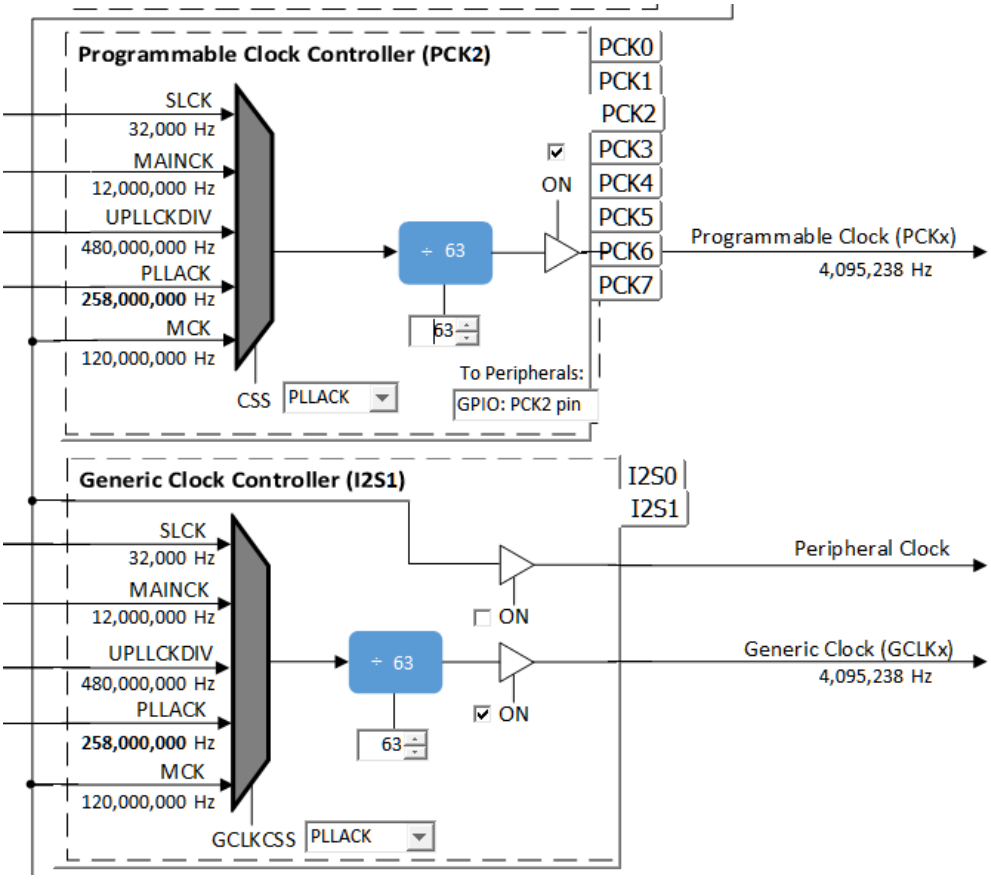

*USB Microphone Main Clock*

The PLLA clock is generated from 12 Mhz Main Clock using the selected DIVA and MULA values, as shown below:




*USB Microphone PLLA Clock*

The I2S MCLK is generated using both the PCK2 output and the I2SC1_GCLK. These are both enabled and generated using the PLLA clock source to give the same value of 4,095,238 Hz (to approximate the 4096000 Hz sampling rate), as shown below:




*USB Microphone I2S1 GCLK and PCK2 Configuration*

*Timer Driver*

The Timer driver configuration, Timer driver instance 0, is used by a system for button processing (debounce and long press) and LED blink delay. It needs to be set to “Enable Period Interrupt”. It is also required by the WM8904 Codec Driver. The default configuration values are used.

### *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/usb_microphone`. To build this project, you must open the `audio/apps/usb_microphone/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

**MPLAB X IDE Project**

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

**MPLAB X IDE Project Configurations**

This table lists and describes the supported configurations of the demonstration, which are located within `./firmware/src/config`.

| Project Name | BSP Used | Description |
|---|---|---|
| usb_microphone_basic_sam_e70_xult_wm8904_i2sc_usb | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board |
| usb_microphone_basic_sam_e70_xult_wm8904_i2sc_usb_freertos | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board. It also uses FreeRTOS. |

### *Configuring the Hardware*

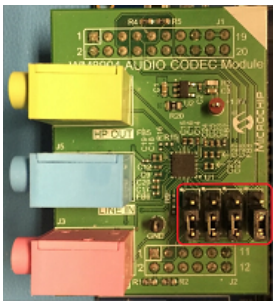
This section describes how to configure the supported hardware.

## Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the I2SC PLIB:

Jumper J204, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS Detect.

To connect to the I2SC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented towards the pink, mic in, connector. See the red outlined jumpers in the below image as reference.




**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

### *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project. Refer to Building the Application for details.

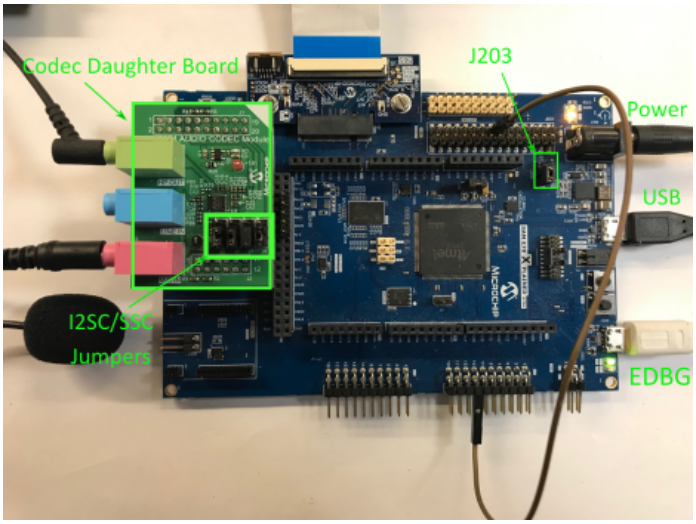



*Figure 1. WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green. Microphone In Jack is pink.*

*Note: the brown wire is a jumper wire which is not relevant for this app.*

Do the following to run the demonstration:

1. Attach the WM8904 Daughter Board to the X32 connector. Connect headphones (monitor headphones) to the HP OUT jack of the WM8904 Audio Codec Daughter Board (see Figure 1) and a microphone to the pink MIC IN jack (Refer to Figure 1.
2. Connect power to the board and connect the USB cable (EDBG) used for programming the device (Refer to Figure 1). Compile the application and program the target device using the EDBG. Run the device. The system then will be in a waiting for USB to be connected (amber LED1 off).
3. Connect to the USB Host via the micro-mini connector (Refer to Figure 1) located above the push-button switches on the right side of the board using a standard USB cable. The LED1 should turn to a solid amber color as the host enumerates and then initiates the audio stream.
4. After the Host computer acknowledges the connection, it will install drivers (if needed), No special software is necessary on the Host side.

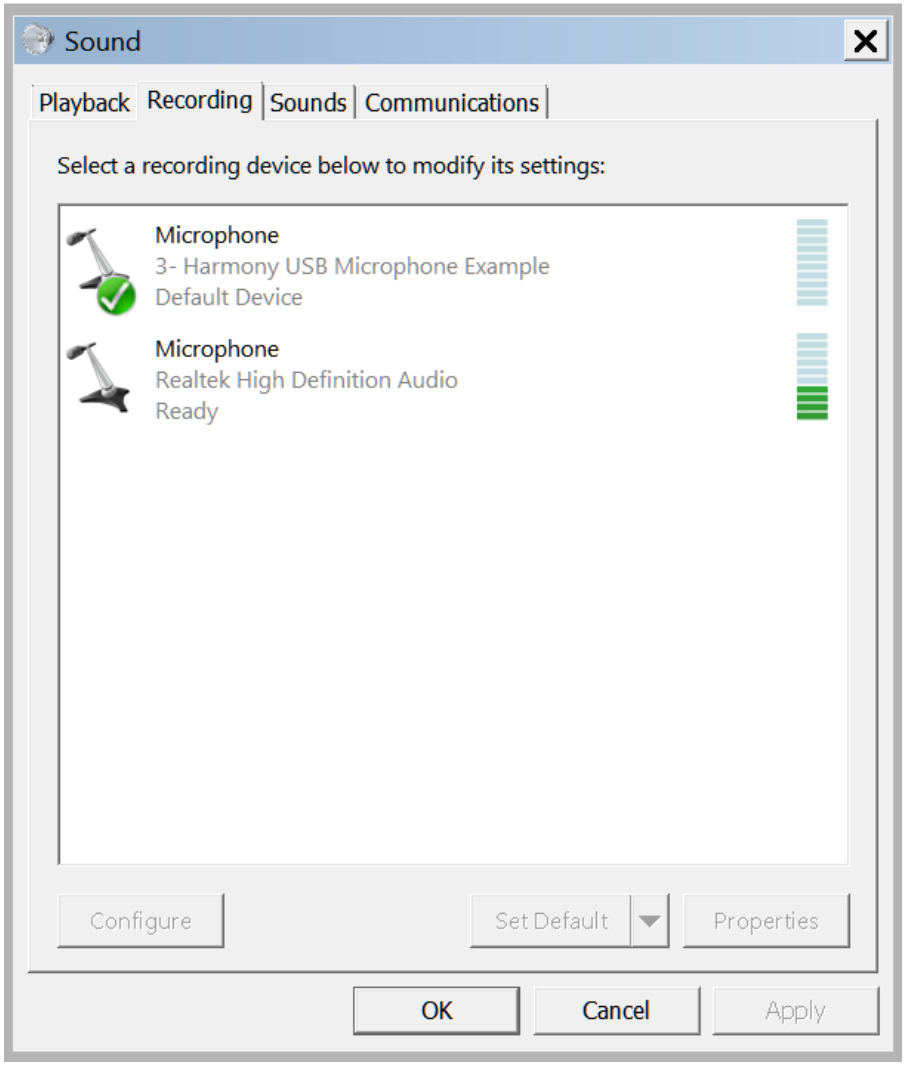



*Figure 2. Windows 7 Sound Dialog showing USB Microphone with Sound Level Meter*

5. If needed, configure the Host computer to use the usb_microphone as the selected audio recording device. For Windows, this is done in the "Recording" tab in the "Sound" dialog (as shown in Figure 2) accessed by right clicking the loudspeaker icon on the taskbar.

**Note:** The device "Harmony USB Microphone Example" should be available along with a sound level meter indication audio when playing. If no sound level is registering, uninstall the driver and reboot the host computer, since it may have incorrect configuration set by a similar connection to one of the other MPLAB-X Harmony Audio Demos.

6. The Microphone audio should be heard through the monitor headphones
7. Open a recording application (such as Audacity) and initiate a recording sessiont through the USB Microphone.

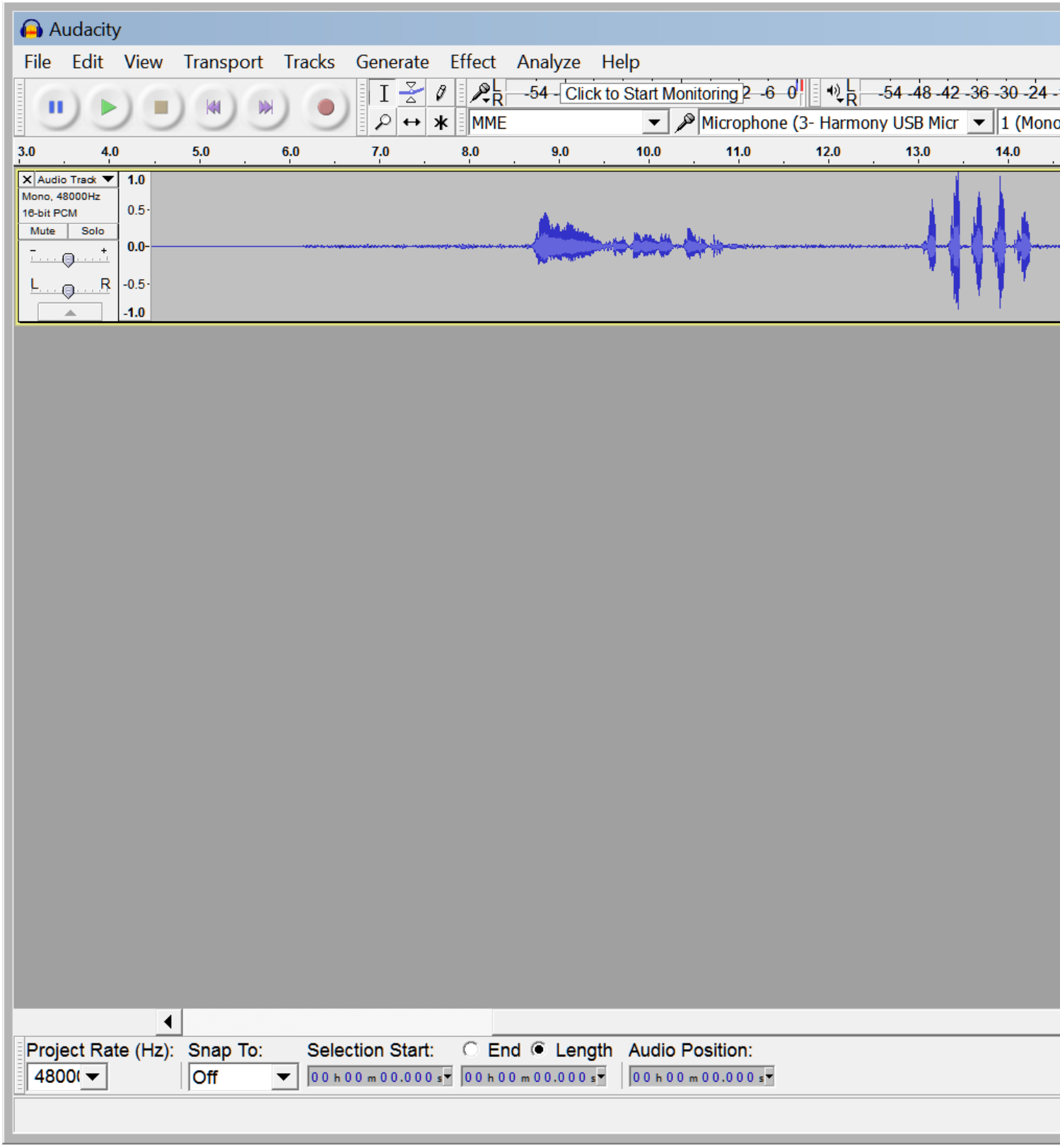



*Figure 3. Audacity Recording Session using USB Microphone*

7. Playback of recording session audio is being heard in the USB Microphone headphones.

## Control Description

Button control uses SW1 in a push button function sequence given in the table below:

| Function # | Function | Press |
|---|---|---|
| 1 | Gain Cont | Low Gain |
| 2 | Gain Cont | High Gain |

**Note:** Function 2 will transition back to Function 1 on the next button push

USB operational status is given by LED, as shown below:

| LED Status | Status |
|---|---|
| OFF | USB cable detached |
| ON | USB cable attached |
| Blinking | Attached/not Streaming |

# usb_speaker

This topic provides instructions and information about the MPLAB Harmony 3 USB Speaker demonstration application, which is included in the MPLAB Harmony Library distribution.

## Description

The USB Device driver in Full Speed mode will interface to a USB Host (such as a personal computer) via the USB Device Stack using the V1.0 Audio Function Driver. The embedded system will enumerate with a USB audio device endpoint and enable the host system to input audio from the USB port using a standard USB Full-Speed implementation. The embedded system will stream USB playback audio to the Codec. The Codec Driver sets up the audio interface, timing, DMA channels and buffer queue to enable a continuous audio stream. The digital audio is transmitted to the codec through an I2S data module for playback through a headphone jack.

### Architecture

The application runs on the SAM E70 Xplained Ultra Board, which contains a at 300 MHz using the following features:

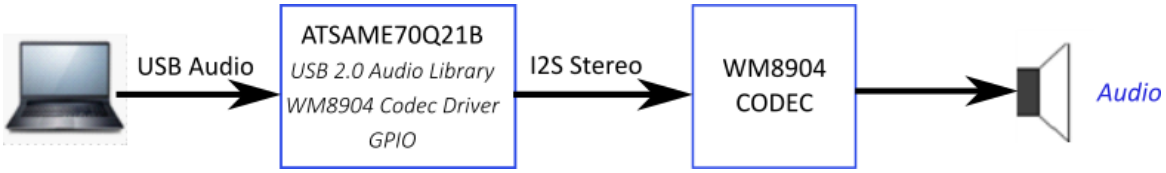
- One push button
- Two LEDs (amber and green). Only the amber LED can be used for a USB Device requiring VBUS sense.
- WM8904 Codec Daughter Board mounted on a X32 socket
- USB Device interface

**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The usb_speaker application uses the MPLAB Harmony Configurator to setup the USB Audio Device, codec, and other items in order to play back the USB audio through the WM8904 Codec Module.

A USB Host system is connected to the micro-mini USB device connector. The application detects the cable connection, which can also supply device power; recognizes the type of connection (Full Speed); enumerates its functions with the host, isochronous audio streaming playback through device. Audio stream data is buffered in 1 ms frames for playback using the WM8904 Codec daughter board. Audio is heard through the Headphone jack (HP OUT).

The following figure shows the basic architecture for the demonstration.




### Demonstration Features

- Playback using an WM8904 codec daughterboard on the SAM E70 Xplained Ultra (E70 XULT) board.

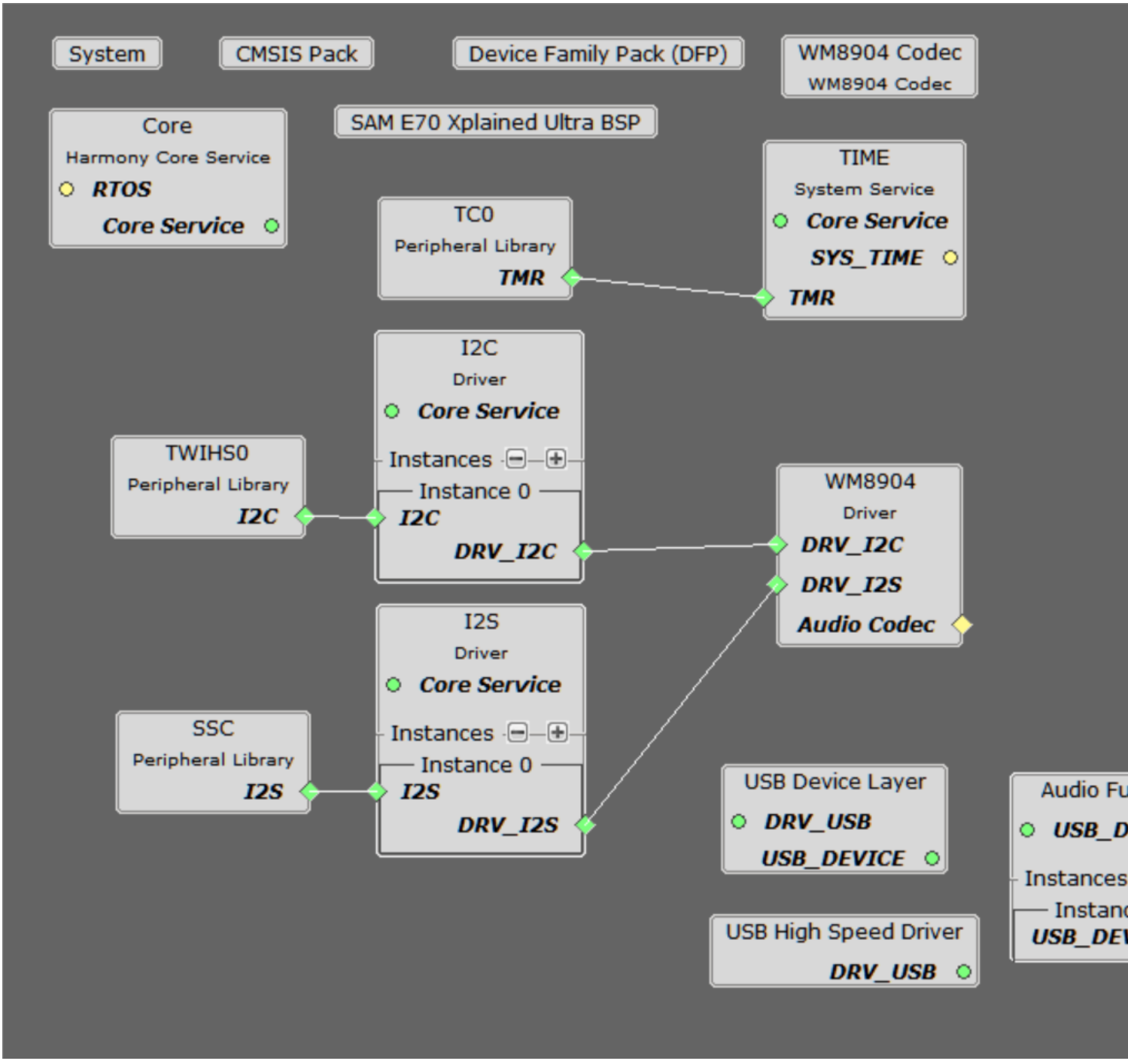
- USB connection to a host system using the USB Library Device Stack for a USB Speaker device using E70 XULT
- E70 XULT Button processing for volume/mute control
- Using the USB Library
- USB Attach/Detach and mute status using an LED
- Utilization of the of I2S peripherals SAM E70 SSC (as slave) and I2SC (as master)

Note that all the calls to the WM8904 codec driver use the form DRV_CODEC_xxx rather than DRV_WM8904_xxx. This is to make the code more generic, such that another codec could be substituted for another without having to make changes to the application code except for the location of the driver's public header file.

## Tools Setup Differences

### Harmony Configuration

The MHC Project Graph for usb_speaker_basic is shown below.



*USB Speaker Basic MHC Project Graph*

Each block provides configuration parameters to generate the application framework code. This includes all the needed drivers, middleware, libraries. The generated framework code is placed under the firmware/src/config directory for this usb_speaker application (usb_speaker_basic_sam_e70_xult_wm8904_usb). The usb_speaker_basic application code is located in the firmware/src directory app.c and app.h files, which utilize the framework drivers, middleware and library APIs located in definitions.h (as generated by the configurator).

A configuration utilizing the I2SC1 peripheral is used to replace the SSC in the above diagram. Also, a configuration that adds the FreeRTOS block to the above project is also given.

## Harmony Code Generation

All Harmony applications use the function SYS_Initialize function located in the source file also located in initialization.c. This is executed from main to initialize various subsystems such as the clock, ports, BSP (board support package), codec, usb, timers, and interrupts. The application APP_Initialize function in app.c is also executed in this routine to setup the application code state machine.

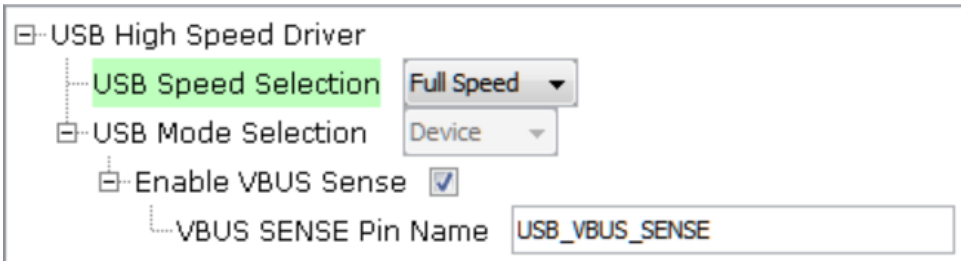
The USB, WM8904 driver, and the application state machines (APP_Tasks routine) are all updated via calls located in the function SYS_Tasks, executed from the main polling loop, located in tasks.c.

The application code is contained in the standard source file app.c. The application utilizes a simple state machine (APP_Tasks executed from SYS_Tasks) with the following functions

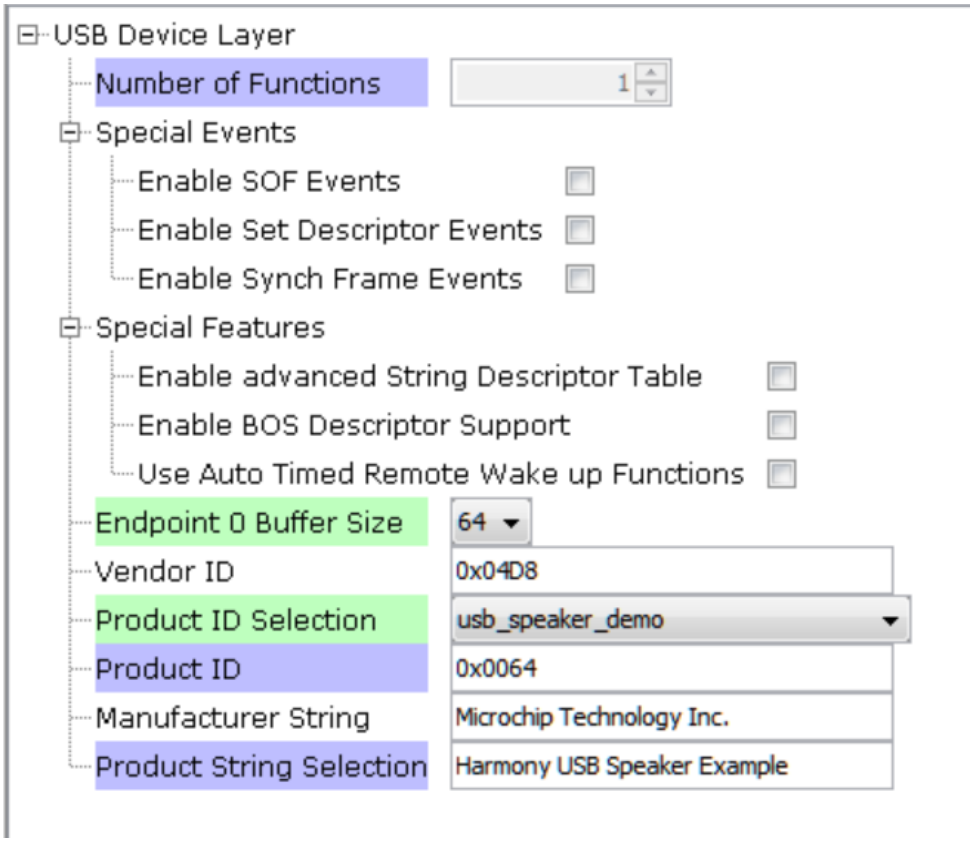
1. Setup the drivers and USB Library interface
2. Respond to USB Host control commands (“Attach”, “Detach”, “Suspend”)
3. Initiate and maintains the bidirectional data audio streaming for the "USB Playback" function.

## USB Configuration

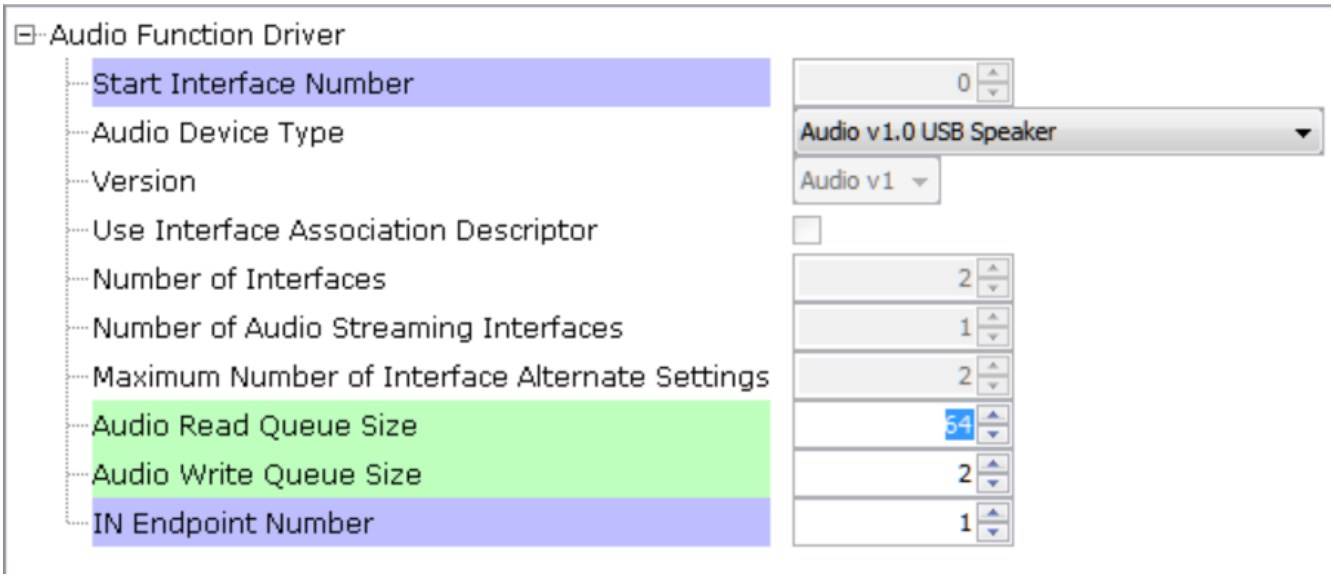
The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is selected to by “Full Speed” (not “High Speed”):




The USB Device Layer is configured by selecting the “usb_speaker_demo” with an endpoint buffer size of 64 (bytes). The Audio Function Driver is configured for 5 USB endpoints with a single function having 3 interfaces. All of these are defined for a USB Speaker device in the fullSpeedConfigurationDescriptor array variable structure (located in initialization.c under the config folder). This structure defines the connection to the host at 48 Khz with 16 bit stereo channel data. A packet queue of length APP_QUEUE_SIZE (set to 32) is used for playback data. The maximum USB packets size is set to 48 * 2 channels/sample * 2 bytes/channel= 192 bytes, which gives a 1ms stereo sample packet size at 48Khz (the standard data frame length at this rate), thus the buffer size needs to be of the same size.

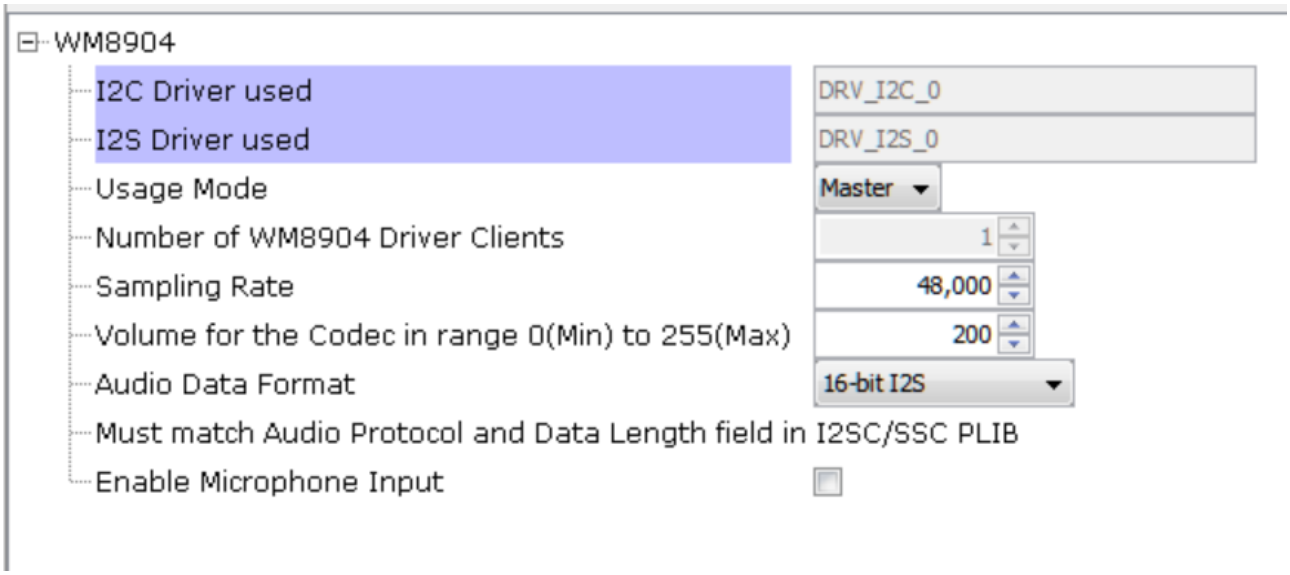



The Audio Function Driver is configured with a Audio Read Queue that matches that of the codec driver write queue for this Audio V1.0 USB Speaker interface.

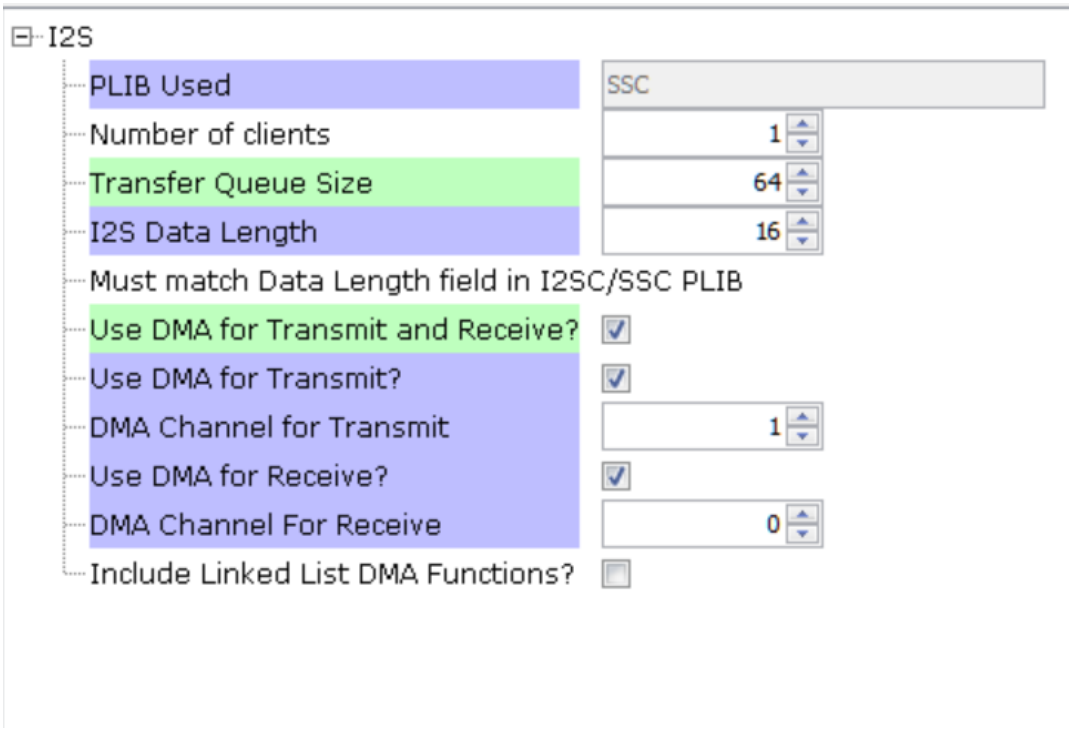



## The WM8904 Codec

The WM8904 codec uses a TWIHS (I2C) interface for configuration and control register setting and either a SSC for audio data or I2SC I2S interface. The default settings are used as shown below:




The SSC uses a Transfer queue size that matches that of the USB Read Queue:




## Pin Manager

The buttons, LED, Switch, I2S and I2C interfaces using GPIO pins via the Microchip Harmony Configurator (MHC) Pin Manager, as follows for the SSC:

| NAME | PORT | E70 PIN | Notes |
|---|---|---|---|
| SSC_TK | PB01 | 20 | I2S BCLK |
| SSC_TF | PB00 | 21 | I2S LRCK |
| PMC_PCK2 | PA18 | 24 | I2S MCLK (Master Clock) |
| SSC_TD | PD26 | 53 | I2S Data |
| SWITCH | PA11 | 66 | Push Button |
| LED | PA05 | 73 | |
| TWIHS0_TWCK0 | PA04 | 77 | I2C |

| | | | |
|---|---|---|---|
| TWIHS0_TWD0 | PA03 | 91 | I2C |
| STDBY | PD11 | 98 | |
| LED2/VBUS DETECT(J204) | PB08 | 141 | J204 set to VBUS DETECT for USB Device |

When the I2SC1 is used the following pins are used:

| NAME | PORT | E70 PIN | Notes |
|---|---|---|---|
| I2SC1_WS | PE00 | 4 | I2S LRCK (Word Select) |
| I2SC1_DO0 | PE01 | 6 | I2S DO (Data Out) |
| I2SC1_DI0 | PE02 | 7 | I2S DI (Data In) |
| I2SC1_CK | PA20 | 22 | I2S BCLK (Bit Clock) |
| I2SC1_GCLK | PA19 | 23 | I2S MCLK (Bit Clock as used by the E70 I2SC1, I2S Master) |
| PMC_PCK2 | PA18 | 24 | I2S MCLK (Master Clock as used by the WM8904 Codec I2S Slave) |
| SWITCH | PA11 | 66 | Push Button |
| LED1 | PA05 | 73 | |
| TWIHS0_TWCK0 | PA04 | 77 | I2C |
| TWIHS0_TWD0 | PA03 | 91 | I2C |
| STDBY | PD11 | 98 | |
| LED2/VBUS DETECT(J204) | PB08 | 141 | J204 set to VBUS DETECT for USB Device |

## Clock Manager

All clocks are generated from the 12 MHz Main Clock oscillator. From this clock is derived the following clocks:

| Clock | Value | Description |
|---|---|---|
| HCLK | 300 MHz | Processor Clock |
| PCK2 | 12 MHz | Peripheral Clock 2 |
| USB FS | 48 MHz | USB Full Speed Clock |

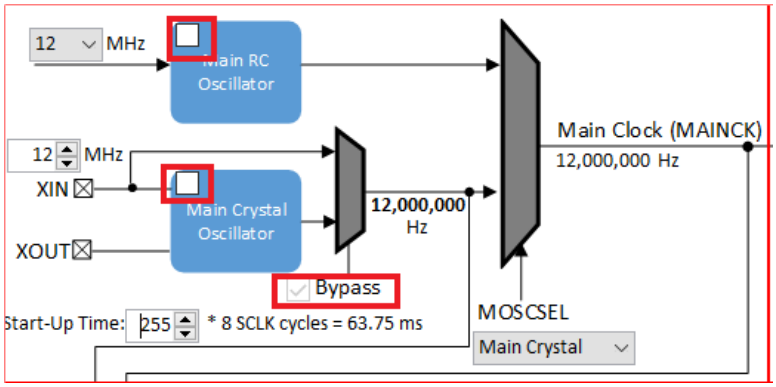
The I2S clocks are setup for 48KHz sampling rate, with stereo 16 bit samples, giving a 32 bit sample frame. The I2S clocks are generated from the WM8904 acting as I2S master using the 12.288 Mhz master clock obtained from Peripheral Clock 2 (PCK2). The I2S clocks will then be generated, as follows:

| I2S Function | Value | Description |
|---|---|---|
| LRCK | 48.000000 K | Sample rate clock |
| BCLK | 3072000 Hz | Bit Rate Clock |
| MCLK | 12.288000 MHz | Master Clock |

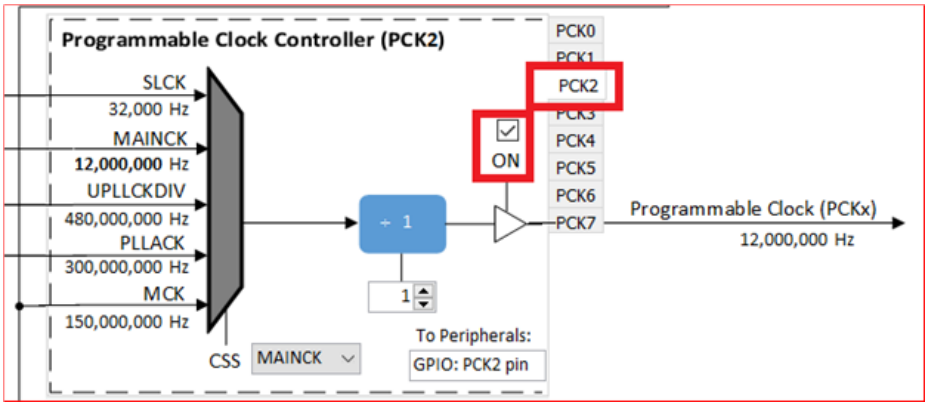
## MPLAB Harmony Configurator: Tools>Clock Configuration

### SSC Clock Configuration

Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is checked, it will cause the Main Crystal Oscillator to become disabled. An external MEMS oscillator input on the XIN pin is used for Main Clock generation.

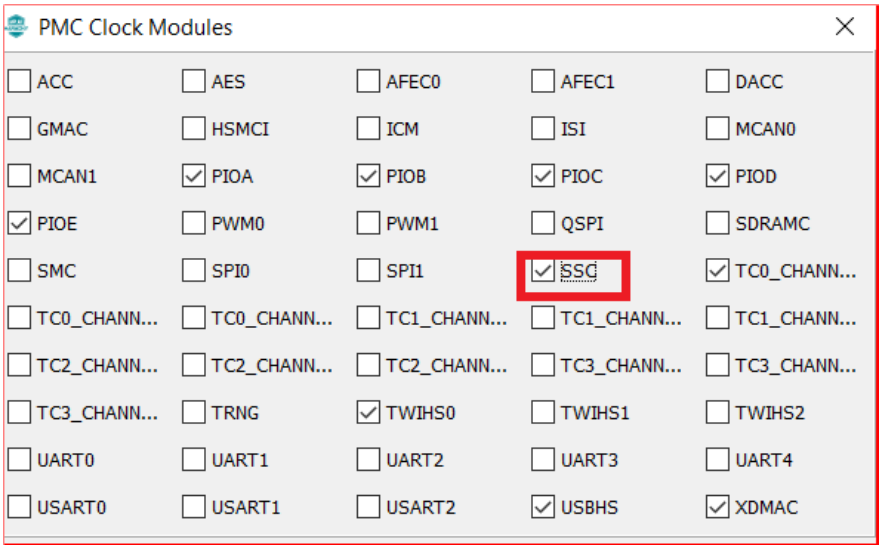



Enable the PCK2 output to enable the WM8904 master clock generation, to enable clocking for the SSC operating as a slave I2S:



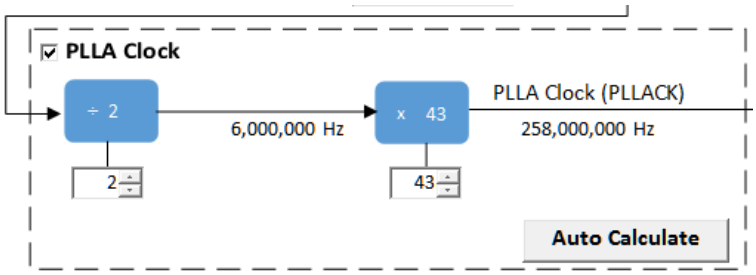

*Clock Diagram>Peripheral Clock Enable*

Enable the peripheral clock to the SSC using the Peripheral Clock Enable of the Peripheral Clock Controller:

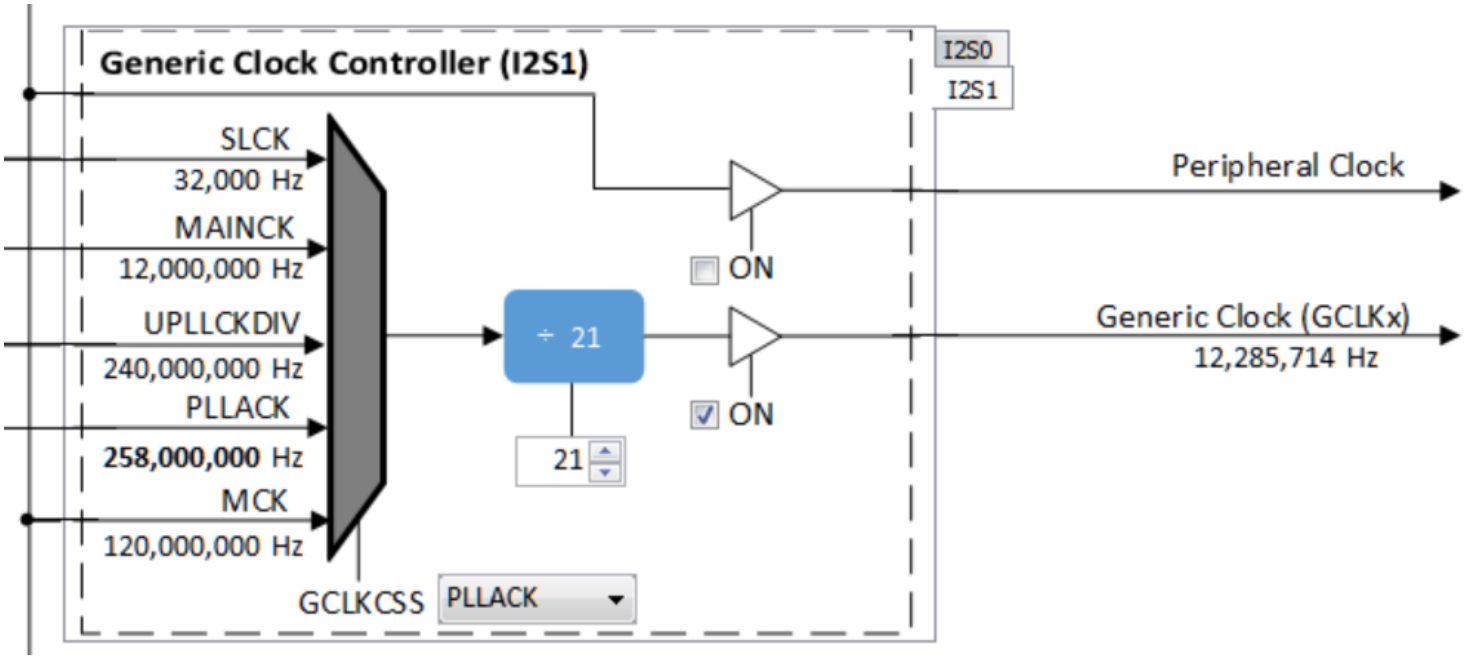



**I2SC1 Clock Configuration**

The I2SC1 master requires the I2SC1_GCLK generate an MCLK to approximate 12.288Mhz. This is sourced using the PLLA Clock (PLLACK), as shown below.



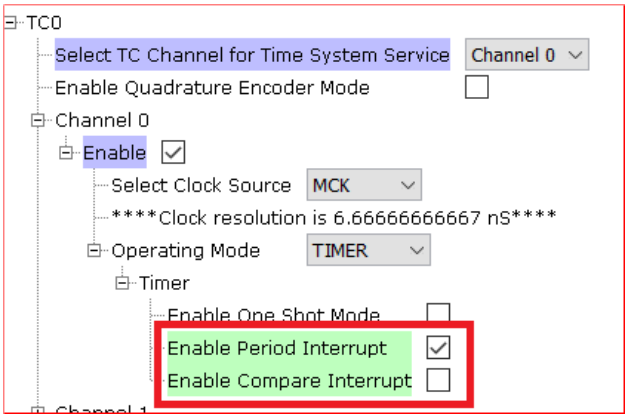
*USB Speaker PLLA Clock*




*USB Speaker I2SC1 GCLK Configuration*

*Timer Driver (TC0)*

The Timer driver configuration, Timer driver instance 0, is used by a system for button processing (debounce and long press) and LED blink delay. It needs to be set to “Enable Period Interrupt”.




## *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/usb_speaker`. To build this project, you must open the `audio/apps/usb_speaker/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

**MPLAB X IDE Project**

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

**MPLAB X IDE Project Configurations**

This table lists and describes the supported configurations of the demonstration, which are located within `./firmware/src/config`.

| Project Name | BSP Used | Description |
|---|---|---|
| us_basic_sam_e70_xult_ wm8904_i2sc_usb | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the I2SC PLIB. |
| us_basic_sam_e70_xult_ wm8904_ssc_usb | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the SSC PLIB. |
| us_basic_sam_e70_xult_ wm8904_ssc_usb_freertos | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the I2SC PLIB. It also uses FreeRTOS. |

## *Configuring the Hardware*

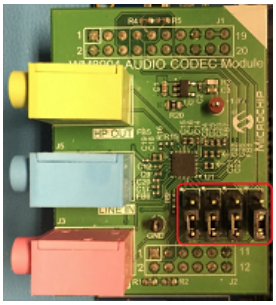
This section describes how to configure the supported hardware.

### Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the I2SC PLIB:

Jumper J204, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS.

To connect to the SSC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented towards the pink, mic in, connector. See the red outlined jumpers in the below image as reference.

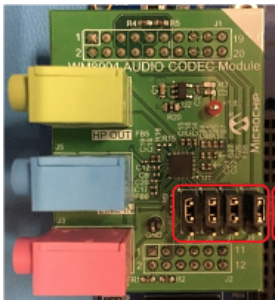



**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the SSC PLIB:

Jumper J204, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS.

To connect to the SSC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented away from the pink, mic in, connector. See the red outlined jumpers in the below image as reference.




**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

## *Running the Demonstration*

This section demonstrates how to run the demonstration.

## Description

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project. Refer to Building the Application for details.

Do the following to run the demonstration:

1. Attach the WM8904 Daughter Board to the X32 connector. Connect headphones to the HP OUT (green) jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).

**Important:** The I2SC/SSC jumpers must be in the correct position for the configuration being run.

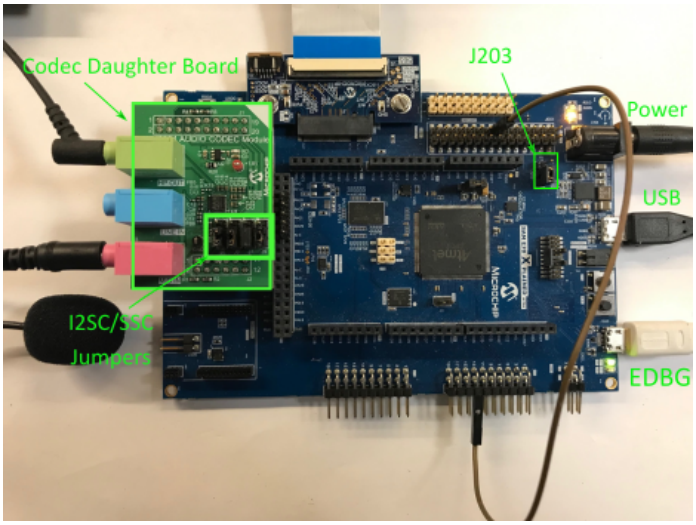



*Figure 1. WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green.*

*Note: the brown wire is a jumper which is not relevant for this app.*

2. Connect power to the board, compiles the application and program the target device. Run the device. The system will be in a waiting for USB to be connected (amber LED1 off).
3. Connect to the USB Host via the micro-mini connector located above the push-button switches on the right side of the board using a standard USB cable.
4. Allow the Host computer to acknowledge, install drivers (if needed), and enumerate the device. No special software is necessary on the Host side.

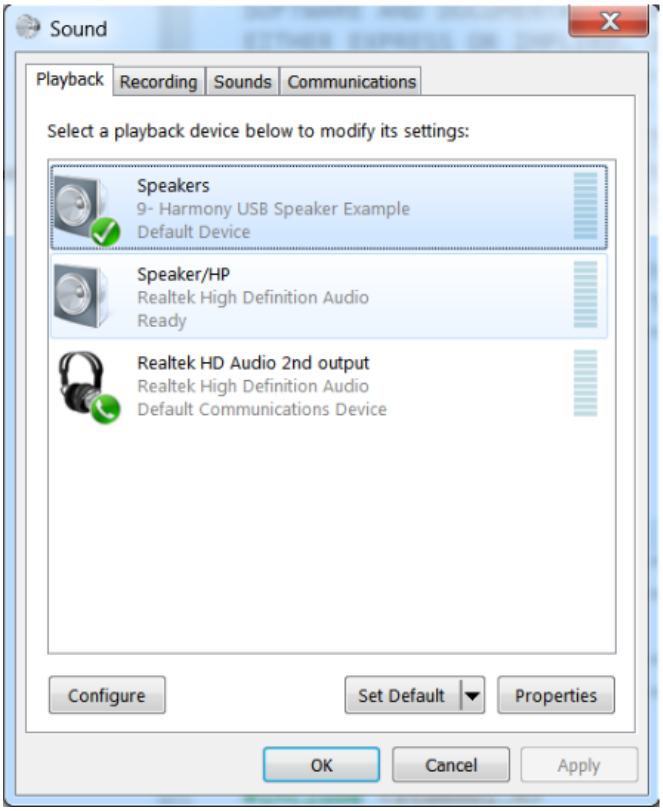


*Figure 2. Windows 7 Sound Dialog showing USB Microphone with Sound Level Meter*

5. If needed, configure the Host computer to use the usb_speaker as the selected audio playback device. For Windows, this is done in the "Playback" tab in the "Sound" dialog (as shown in Figure 2) accessed by right clicking the loudspeaker icon in the taskbar.

**Note:** The device "Harmony USB Hi Res Speaker Example" should be available along with a sound level meter indication audio when playing. If no sound level is registering, uninstall the driver, since it may have incorrect configuration set by a similar connection to one of the other MPLAB-X Harmony Audio Demos. Disconnect and reconnect the usb cable to the PC. The reconfigured driver will then be installed for the correct USB device.

6. Open a playback application (such as Window Media Player) and initiate playback through the USB Speaker
7. Playback will demonstrate that the audio is being heard in the USB Speaker headphones. Use the pushbutton switch to mute and change volume levels to the headphone.

### Control Description

Button control uses the push button function sequence given in the table below:

| Function | Press |
|---|---|
| Volume Control Level 1 | Low (-66 dB) |
| Volume Control Level 2 | Medium (-48 dB) |
| Volume Control Level 3 | High (-0 dB) |
| Mute | Mute |

**Note:** Mute will transition to Volume Control Level 1 on the next button push.

USB operational status is given by LED, as shown below:

| LED Status | Status |
|---|---|
| OFF | USB cable detached |
| ON | USB cable attached |
| Blinking | Playback muted |

## usb_speaker_hi_res

This topic provides instructions and information about the MPLAB Harmony 3 USB Speaker Hi-Res demonstration application, which is included in the MPLAB Harmony Library distribution.

### Description

This demonstration application configures the development board to implement a USB Speaker device configured to run at 96 Khz sampling rate at 24 bit per sample.

The USB Device driver in Full Speed mode will interface to a USB Host (such as a personal computer) via the USB Device Stack using the V1.0 Audio Function Driver. The embedded system will enumerate with a USB audio device endpoint and enable the host system to input audio from the USB port using a standard USB Full-Speed implementation. The embedded system will stream USB playback audio to the Codec. The Codec Driver sets up the audio interface, timing, DMA channels and buffer queue to enable a continuous audio stream. The digital audio is transmitted to the codec through an I2S data module for playback through a headphone jack.

### Architecture

The application runs on the SAM E70 Xplained Ultra Board, with the following features:

- One push button (SW1)
- Two LEDs (amber LED1 and green LED2). Only LED 1 can be used for a USB Device requiring VBUS sense.

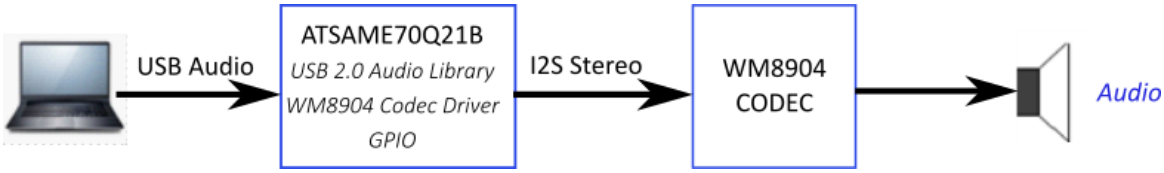
- WM8904 Codec Daughter Board mounted on a X32 socket

**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

The hi-res application uses the MPLAB Harmony Configurator to setup the USB Audio Device, codec, and other items in order to play back the USB audio through the WM8904 Codec Module.

A USB Host system is connected to the micro-mini USB device connector. The application detects the cable connection, which can also supply device power; recognizes the type of connection (Full Speed); enumerates its functions with the host, isochronous audio streaming playback through device. Audio stream data is buffered in 1 ms frames for playback using the WM8904 Codec daughter board. Audio is heard through the Headphone jack (HP OUT).

The following figure shows the basic architecture for the demonstration.



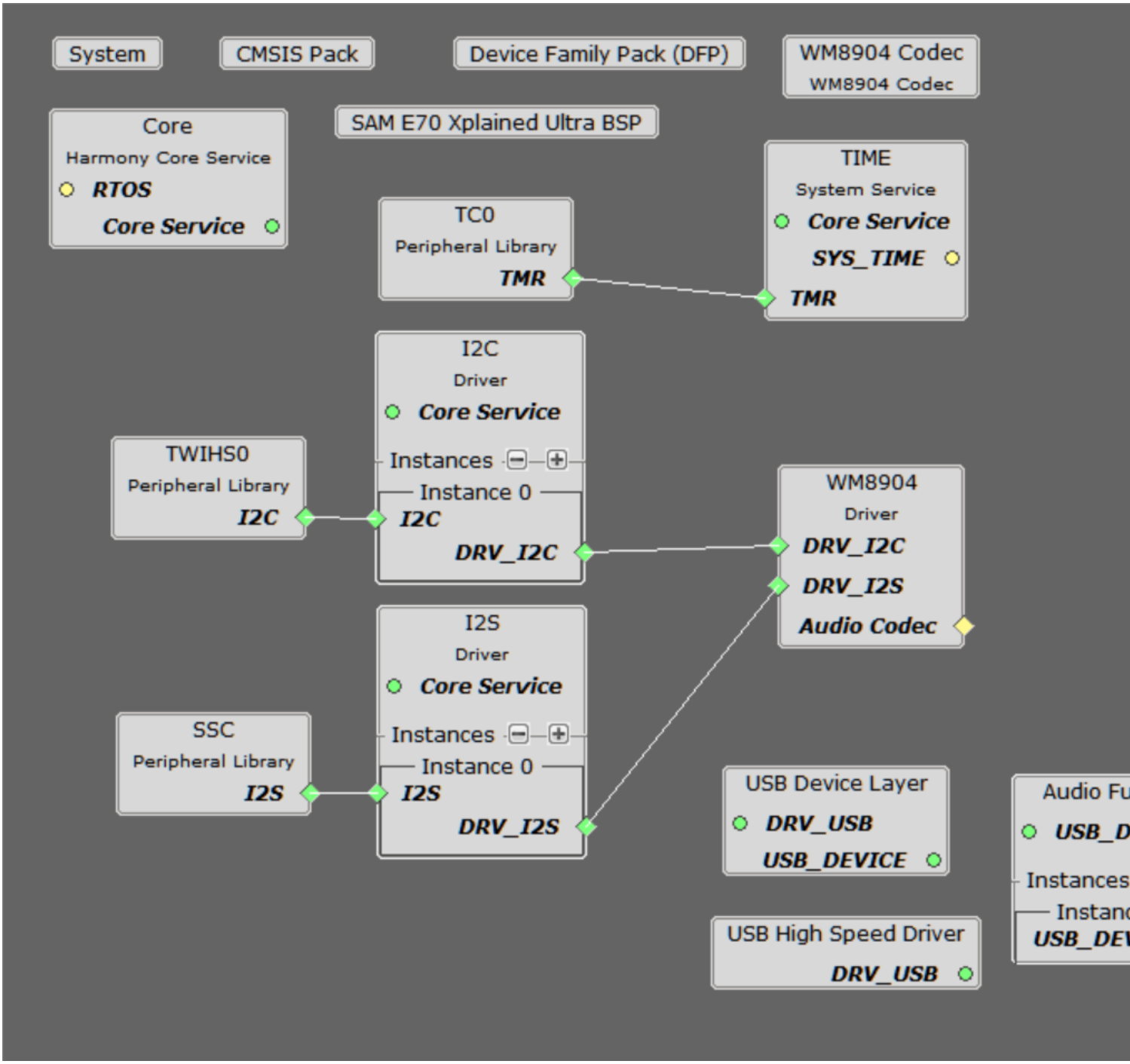

## Demonstration Features

- Playback using an WM8904 codec daughterboard on the SAM E70 xPlained Ultra (E70XULT) board.
- USB connection to a host system using the USB Library Device Stack for a USB Speaker device using E70XULT
- E70XULT Button processing for volume/mute control
- USB Attach/Detach and mute status using an LED.
- Utilization of the of I2S peripheral I2SC (as master)

## Tools Setup Differences

### Harmony Configuration

The MHC Project Graph for usb_speaker_hi-res is shown below.




Each block provides configuration parameters to generate the application framework code. This includes all the needed drivers, middleware, libraries. The generated framework code is placed under the firmware/src/config directory for this usb_speaker application (usb_speaker_basic_sam_e70_xult_wm8904_usb). The usb_speaker_basic application code is located in the firmware/src directory app.c and app.h files, which utilize the framework drivers, middleware and library APIs located in definitions.h (as generated by the configurator).

**Harmony Code Generation**

All Harmony applications use the function SYS_Initialize function located in the source file also located in initialization.c. This is executed from main to initialize various subsystems such as the clock, ports, BSP (board support package), codec, usb, timers, and interrupts. The application APP_Initialize function in app.c is also executed in this routine to setup the application code state machine.

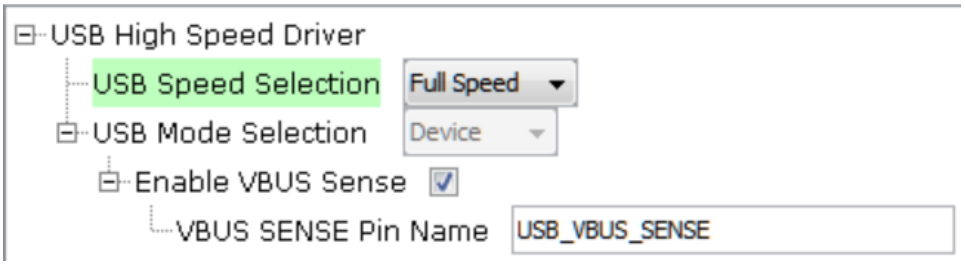
The USB, WM8904 driver, and the application state machines (APP_Tasks routine) are all updated via calls located in the function SYS_Tasks, executed from the main polling loop, located in tasks.c.

The application code is contained in the standard source file app.c. The application utilizes a simple state machine (APP_Tasks executed from SYS_Tasks) with the following functions

1. Setup the drivers and USB Library interface
2. Respond to USB Host control commands (“Attach”, “Detach”, “Suspend”)
3. Initiate and maintains the bidirectional data audio streaming for the "USB Playback" function.

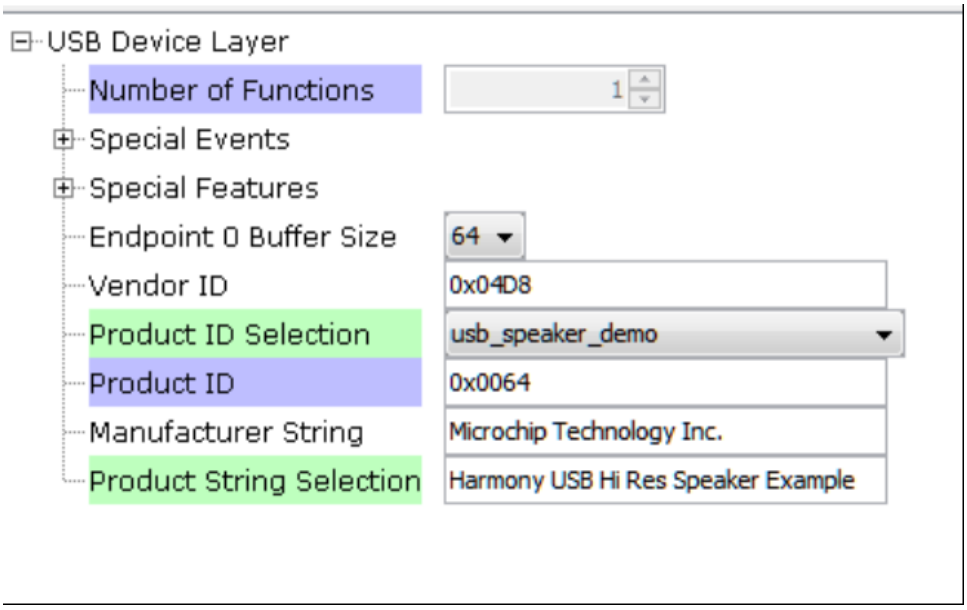
**USB Configuration**

The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is selected to by “Full Speed” (not “High Speed”):

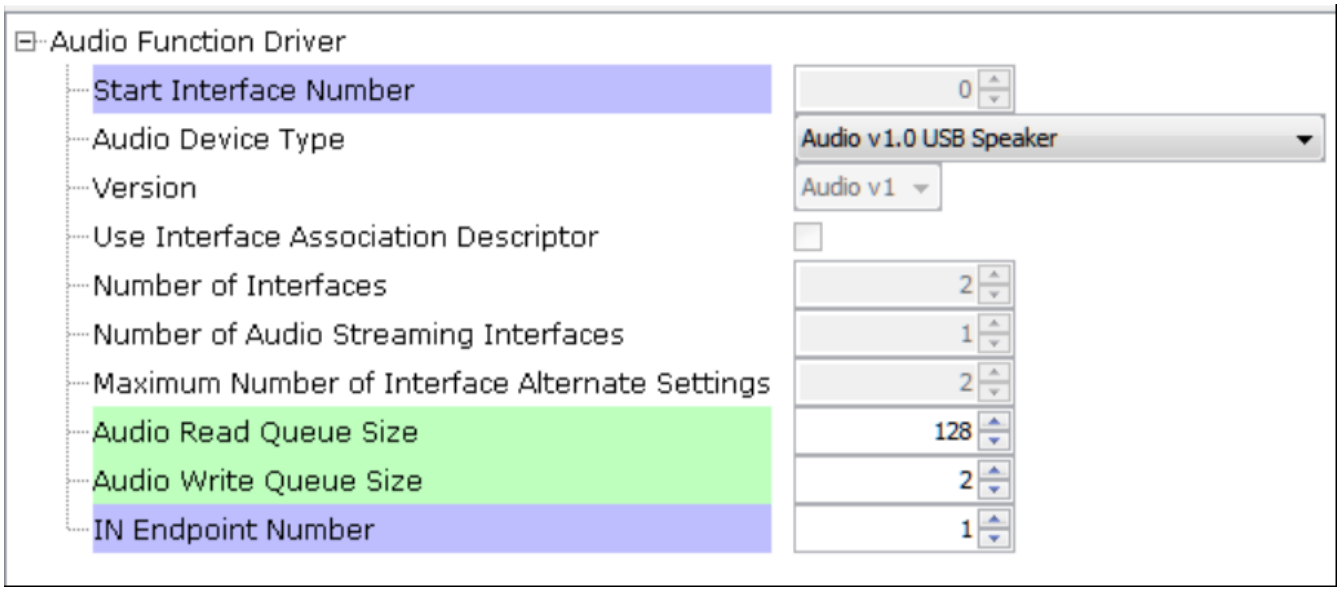



The USB Device Layer is configured by selecting the “usb_speaker_demo” with an endpoint buffer size of 64 (bytes). The Audio Function Driver is configured for 5 USB endpoints with a single function having 3 interfaces. All of these are defined for a USB Speaker device in the fullSpeedConfigurationDescriptor array variable structure (located in initialization.c under the config folder). This structure defines the connection to the host at 96 Khz with 24 bit stereo channel data. A packet queue of length APP_QUEUE_SIZE (set to 32) is used for playback data. The maximum USB packets size is set to 96 * 3 channels/sample * 2 bytes/channel= 576 bytes, which gives a 1ms stereo sample packet size at 96 Khz (the standard data frame length at this rate), thus the buffer size needs to be of the same size.

The Audio Function Driver is configured with a Audio Read Queue that matches that of the codec driver write queue for this Audio V1.0 USB Speaker interface.



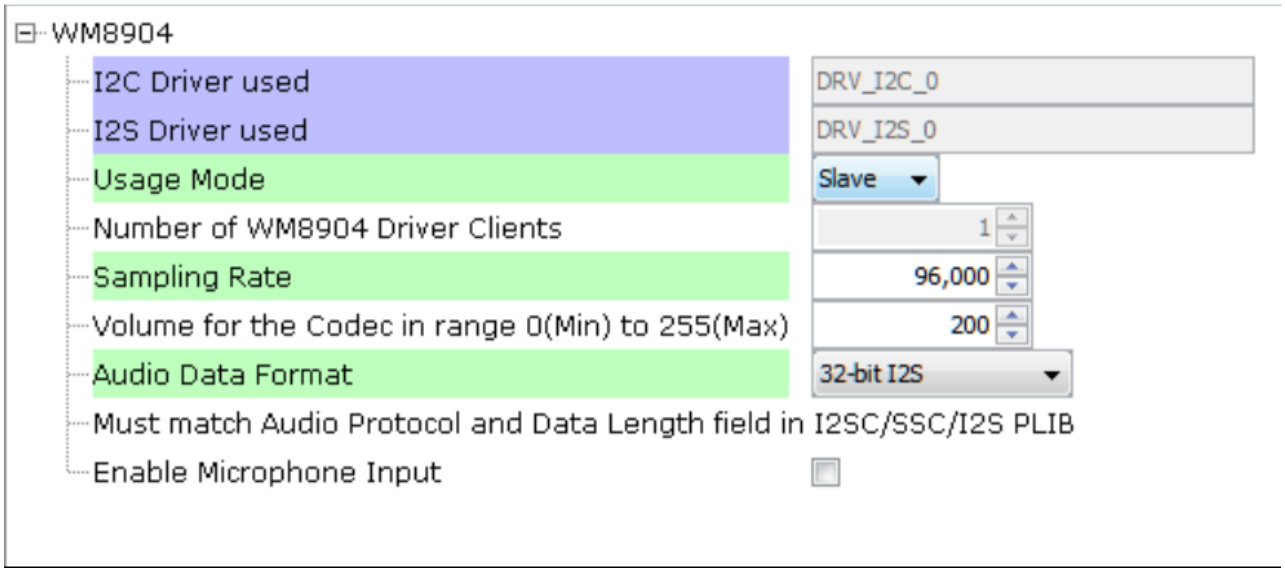

The Audio Function Driver is configured with a Audio Read Queue that matches that of the codec driver write queue for this Audio V1.0 USB Speaker interface.




**Note:** The USB Speaker selection generates a USB Full Speed Descriptor with an isochronous audio stream at 48 Khz/2 Channel/16 bits per channel. This code is modified manually to enumerate a 96Khz/ 2 Channel/ 24 bits per channel isochronous audio data stream.

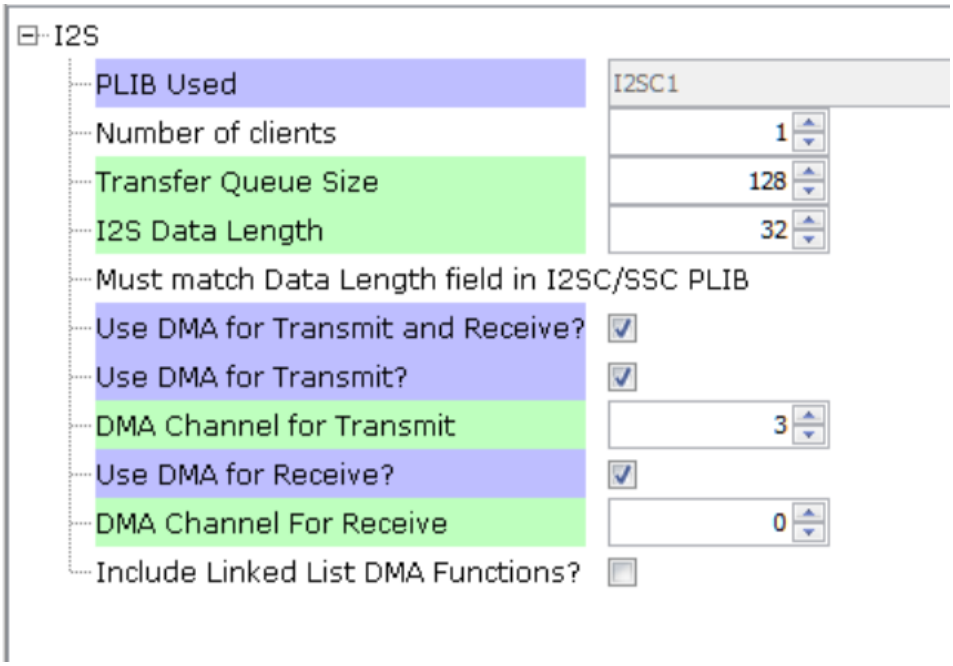
**The WM8904 Codec**

The WM8904 codec uses a TWIHS (I2C) interface for configuration and control register setting and either a SSC for audio data or I2SC I2S interface. The default settings are used as shown below:

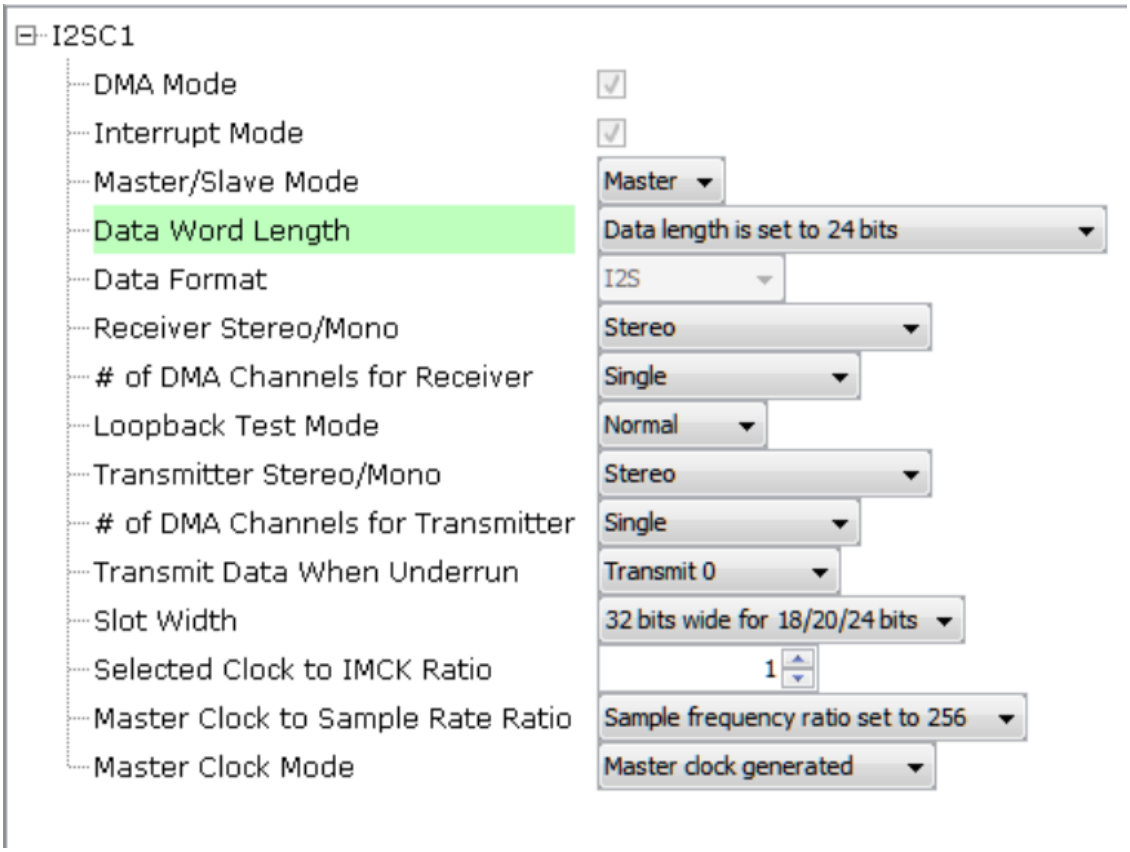



The I2SC1 driver is in “Slave” mode, thus the codec usage mode changes to “Slave”

The I2S configuration uses a Transfer queue size of 128, matching that that of the USB Read Queue:




The I2S data channel is 32 bit long, although only 24 bits is used. The DMA should be selected to match what is configured in the system block.




The I2SC1 driver is configured for 24 bit data, although a 32 bit channel frame is used. This implies that the 24 bit 3 byte packed data buffer received from the USB audio data stream must be unpacked to 32 bit words (4 byte), since a 24 bit frame is not available for the WM8904.

**Pin Manager**

The buttons, LED, Switch, I2S and I2C interfaces using GPIO pins via the Microchip Harmony Configurator (MHC) Pin Manager, as follows for the I2SC1:

| NAME | PORT | E70 PIN | Notes |
|---|---|---|---|
| I2SC1_WS | PE00 | 4 | I2S LRCK (Word Select) |
| I2SC1_DO0 | PE01 | 6 | I2S DO (Data Out) |
| I2SC1_DI0 | PE02 | 7 | I2S DI (Data In) |
| I2SC1_CK | PA20 | 22 | I2S BCLK (Bit Clock) |
| I2SC1_GCLK | PA19 | 23 | I2S MCLK (Bit Clock as used by the E70 I2SC1, I2S Master) |
| PMC_PCK2 | PA18 | 24 | I2S MCLK (Master Clock as used by the WM8904 Codec I2S Slave) |
| SWITCH | PA11 | 66 | Push Button |
| LED1 | PA05 | 73 | |
| TWIHS0_TWCK0 | PA04 | 77 | I2C |
| TWIHS0_TWD0 | PA03 | 91 | I2C |
| STDBY | PD11 | 98 | |
| LED2/VBUS DETECT(J204) | PB08 | 141 | J204 set to VBUS DETECT for USB Device |

**Clock Manager**

All clocks are generated from the 12 MHz Main Clock oscillator. From this clock is derived the following clocks:

| Clock | Value | Description |
|---|---|---|
| HCLK | 300 MHz | Processor Clock |
| PCK2 | 12 MHz | Peripheral Clock 2 |
| USB FS | 48 MHz | USB Full Speed Clock |

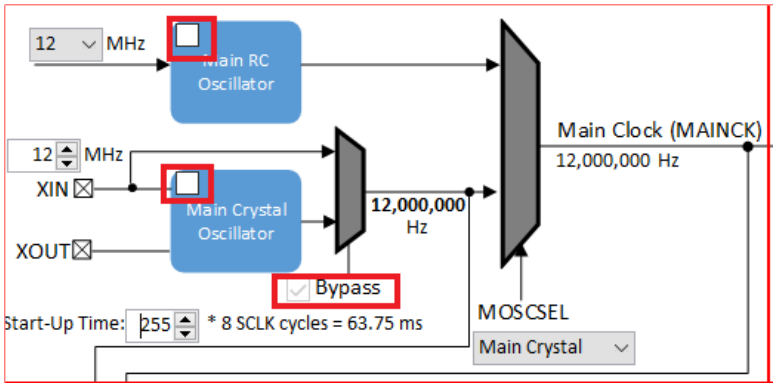
The I2S clocks are setup for 48Khz sampling rate, with stereo 32 bit samples, giving a 64 bit stereo sample frame. The I2S clocks are generated from the WM8904 acting as I2S master using the 24.576 Mhz master clock obtained from I2SC1_GCLK. The I2S clocks will then be generated from the MCLK value below to approximate the following clock rates:

| I2S Function | Value | Description |
|---|---|---|
| LRCK | 96.000000 K | Sample rate clock |
| BCLK | 6114000 Hz | Bit Rate Clock |
| MCLK | 24.576000 MHz | Master Clock |

**MPLAB Harmony Configurator: Tools>Clock Configuration**

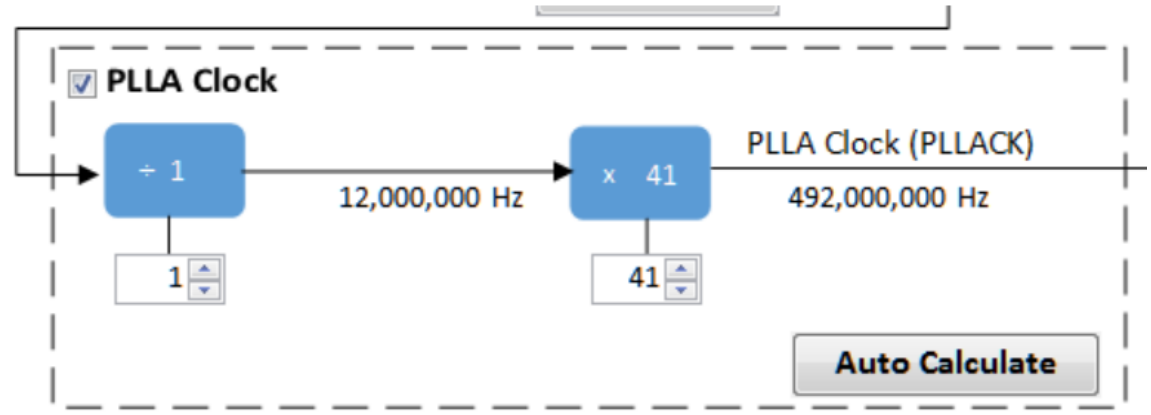
**I2SC1 Clock Configuration**

Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is checked, it will cause the Main Crystal Oscillator to become disabled. An external MEMS oscillator input on the XIN pin is used for Main Clock generation.

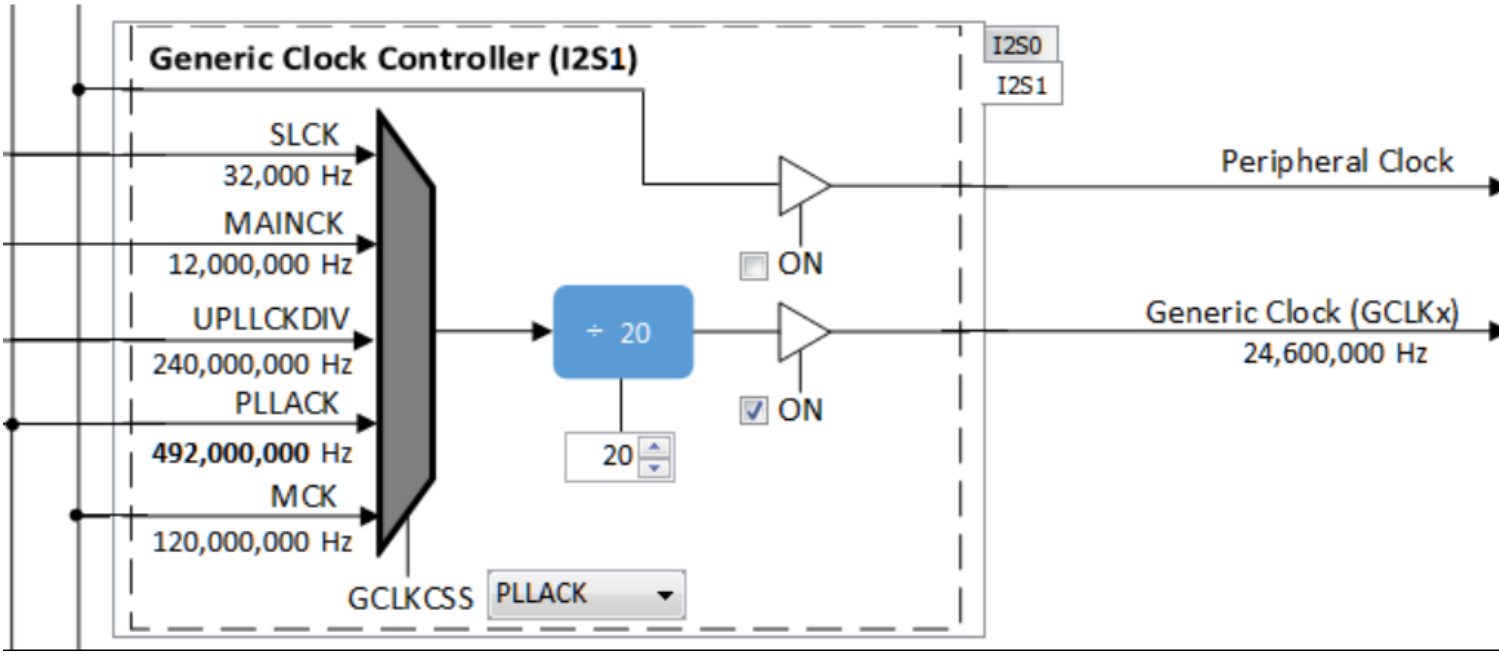



*Clock Diagram>Peripheral Clock Enable*

The I2SC1 master requires the I2SC1_GCLK generate an MCLK to approximate 24.576 hz. This clock is generated from the PLLA Clock (PLLACK), as shown below.



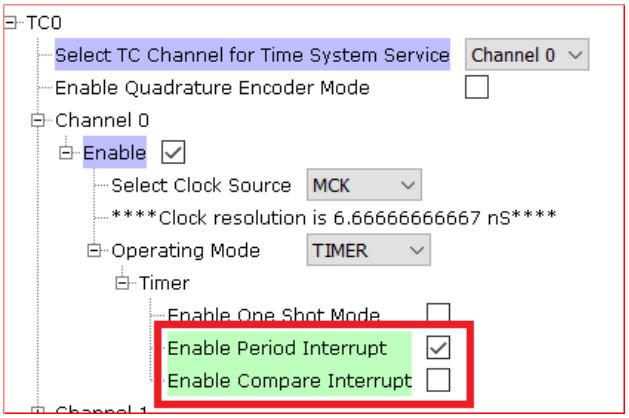

*I2SC1 Configuration of PLLA Clock (PLLACK)*




*I2SC1 GCLK Configuration*

**MPLAB Harmony Configurator: Timer Driver (TC0)**

The Timer driver configuration, Timer driver instance 0, is used by a system for button processing (debounce and long press) and LED blink delay. It needs to be set to “Enable Period Interrupt”.



### *Building the Application*

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the demonstration.

## Description

The parent folder for these files is `audio/apps/usb_speaker_hi_res`. To build this project, you must open the `audio/apps/usb_speaker_hi_res/firmware/*.X` project file in MPLAB X IDE that corresponds to your hardware configuration.

**MPLAB X IDE Project**

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

**MPLAB X IDE Project Configurations**

This table lists and describes the supported configurations of the demonstration, which are located within `./firmware/src/config`.

| Project Name | BSP Used | Description |
|---|---|---|
| us_hi_res_basic_sam_e70_xult_wm8904_i2sc_usb.X | sam_e70_xult | This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the I2SC PLIB. |

### *Configuring the Hardware*

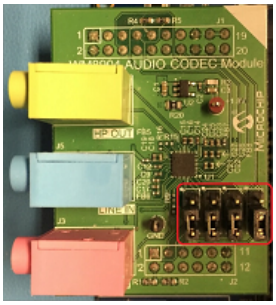
This section describes how to configure the supported hardware.

## Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the I2SC PLIB:

Jumper J204, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS.

To connect to the SSC, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented towards the pink, mic in, connector. See the red outlined jumpers in the below image as reference.




**Note:** The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

### *Running the Application*

This section demonstrates how to run the demonstration.

## Description

**Important!** Prior to using this demonstration, it is recommended to review the MPLAB Harmony 3 Release Notes for any known issues.

Compile and program the target device. While compiling, select the appropriate MPLAB X IDE project. Refer to Building the

Application for details.

Do the following to run the demonstration:

1. Attach the WM8904 Daughter Board to the X32 connector. Connect headphones to the HP OUT (green) jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**).

**Important:** The I2SC/SSC jumpers must be in the correct position for the configuration being run.

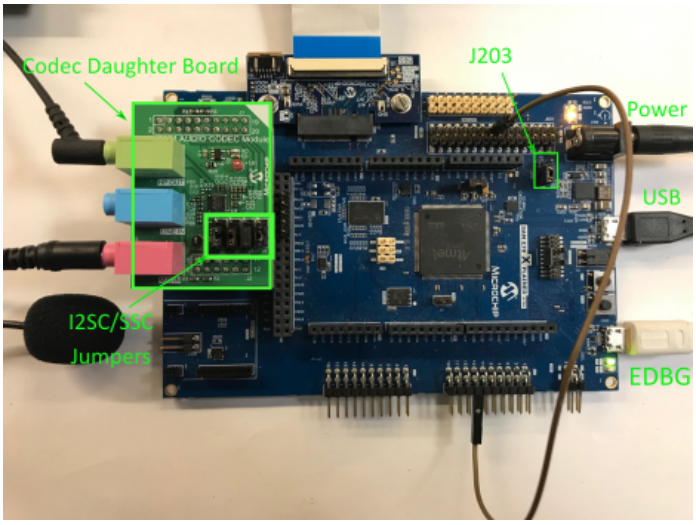



*Figure 1. WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green.*

*Note: the brown wire is a jumper which is not relevant for this app.*

2. Connect power to the board, compiles the application and program the target device. Run the device. The system will be in a waiting for USB to be connected (amber LED1 off).
3. Connect to the USB Host via the micro-mini connector located above the push-button switches on the right side of the board using a standard USB cable.
4. Allow the Host computer to acknowledge, install drivers (if needed), and enumerate the device. No special software is necessary on the Host side.

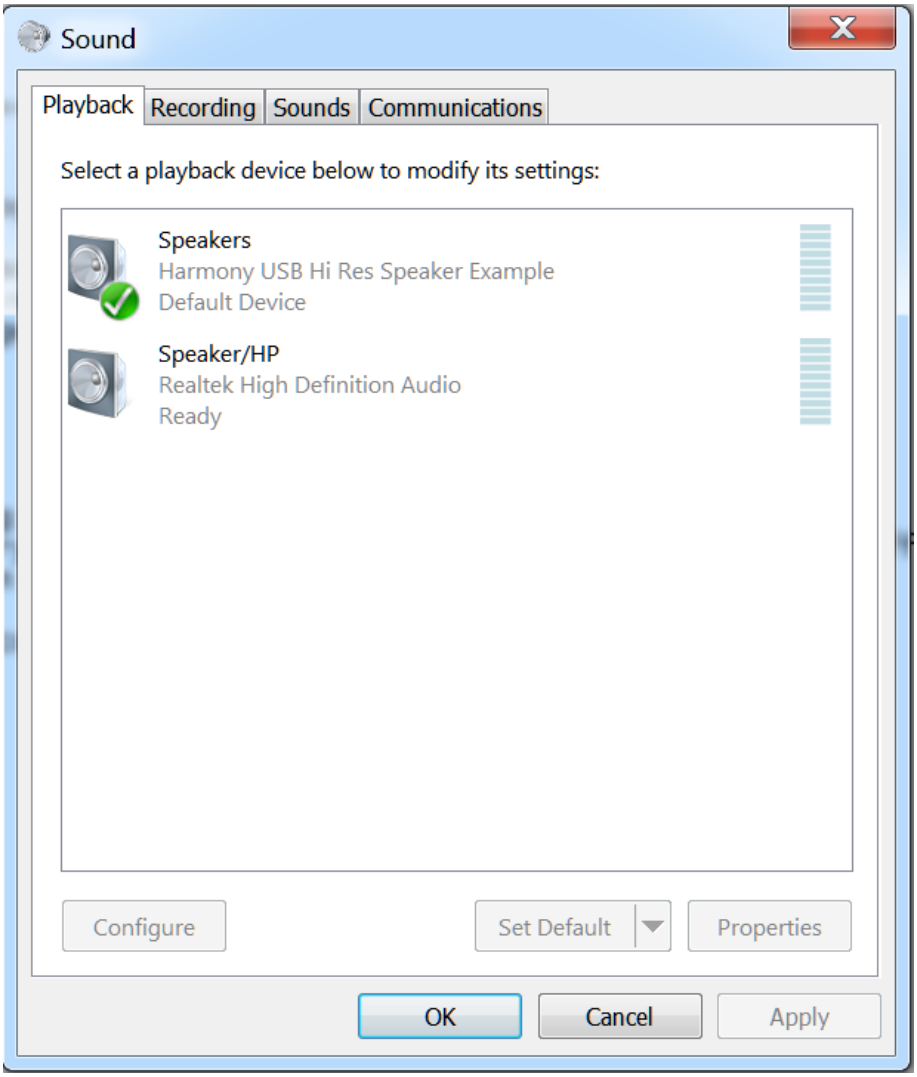



*Figure 2. Windows 7 Sound Dialog showing USB Microphone with Sound Level Meter*

5. If needed, configure the Host computer to use the usb_speaker as the selected audio playback device. For Windows, this is done in the "Playback" tab in the "Sound" dialog (as shown in Figure 2) accessed by right clicking the loudspeaker icon in the taskbar.

**Note:** The device "Harmony USB Hi Res Speaker Example" should be available along with a sound level meter indication audio when playing. If no sound level is registering, uninstall the driver, since it may have incorrect configuration set by a similar connection to one of the other MPLAB-X Harmony Audio Demos. Disconnect and reconnect the usb cable to the PC. The reconfigured driver will then be installed for the correct USB device.

6. Open a playback application (such as Window Media Player) and initiate playback through the USB Speaker
7. Playback will demonstrate that the audio is being heard in the USB Speaker headphones. Use the pushbutton switch to mute and change volume levels to the headphone.

## Control Description

Button control uses the push button function sequence given in the table below:

| Function | Press |
|---|---|
| Volume Control Level 1 | Low (-66 dB) |
| Volume Control Level 2 | Medium (-48 dB) |
| Volume Control Level 3 | High (-0 dB) |
| Mute | Mute |

**Note:** Mute will transition to Volume Control Level 1 on the next button push.

USB operational status is given by LED, as shown below:

| LED Status | Status |
|---|---|
| OFF | USB cable detached |
| ON | USB cable attached |
| Blinking | Playback muted |

# Driver Libraries Help

This section provides descriptions of the Driver libraries that are available in audio repo.

For additional information on Harmony 3 driver architecture, refer to the documentation in the core repository.

## Codec Libraries Help

### WM8904 CODEC Driver Library Help

This topic describes the WM8904 Codec Driver Library.

#### *Introduction*

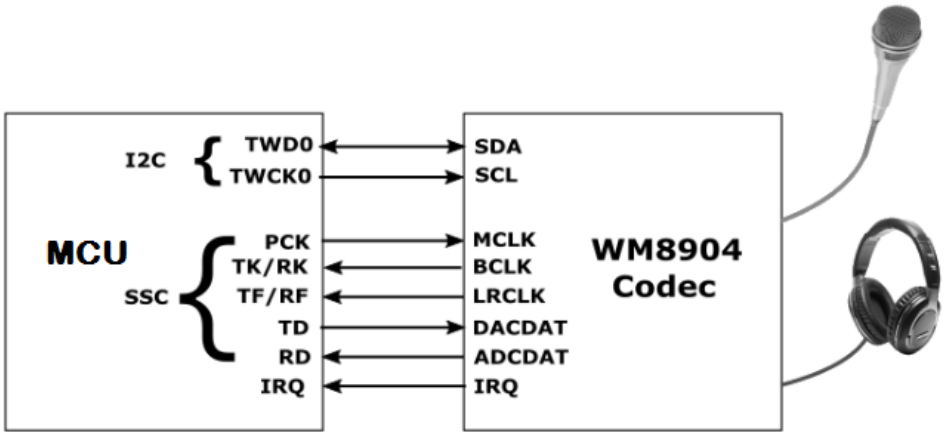
This library provides an Applications Programming Interface (API) to manage the WM8904 Codec that is serially interfaced to the $I^2C$ and $I^2S$ peripherals of a Microchip microcontroller for the purpose of providing audio solutions.

### Description

The WM8904 module is 24-bit Audio Codec from Cirrus Logic, which can operate in 16-, 20-, 24-, and 32-bit audio modes. The WM8904 can be interfaced to Microchip microcontrollers through $I^2C$ and $I^2S$ serial interfaces. The $I^2C$ interface is used to send commands and receive status, and the $I^2S$ interface is used for audio data output (to headphones or line-out) and input (from microphone or line-in).

The WM8904 can be configured as either an $I^2S$ clock slave (receives all clocks from the host), or $I^2S$ clock master (generates $I^2S$ clocks from a master clock input MCLK). Currently the driver only supports master mode with headphone output and (optionally) microphone input.

A typical interface of WM8904 to a Microchip microcontroller using an $I^2C$ and SSC interface (configured as $I^2S$), with the WM8904 set up as the $I^2S$ clock master, is provided in the following diagram:




The WM8904 Codec supports the following features:

- Audio Interface Format: 16-/20-/24-/32-bit interface, LSB justified or I2S format (only 16 and 32-bit interfaces supported in the driver)
- Sampling Frequency Range: 8 kHz to 96 kHz
- Digital Volume Control: -71.625 to 0 dB in 192 steps (converted to a linear scale 0-255 in the driver)
- Soft mute capability

#### *Using the Library*

This topic describes the basic architecture of the WM8904 Codec Driver Library and provides information and examples on its use.

## Description

**Interface Header File**: `drv_WM8904.h`

The interface to the WM8904 Codec Driver library is defined in the `audio/driver/codec/WM8904/drv_WM8904.h` header file. Any C language source (.c) file that uses the WM8904 Codec Driver library should include this header.

### Library Source Files:

The WM8904 Codec Driver library source files are provided in the `audio/driver/codec/WM8904/src` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features and to **Building the Library** for instructions on how to build the library.

**Example Applications:**

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

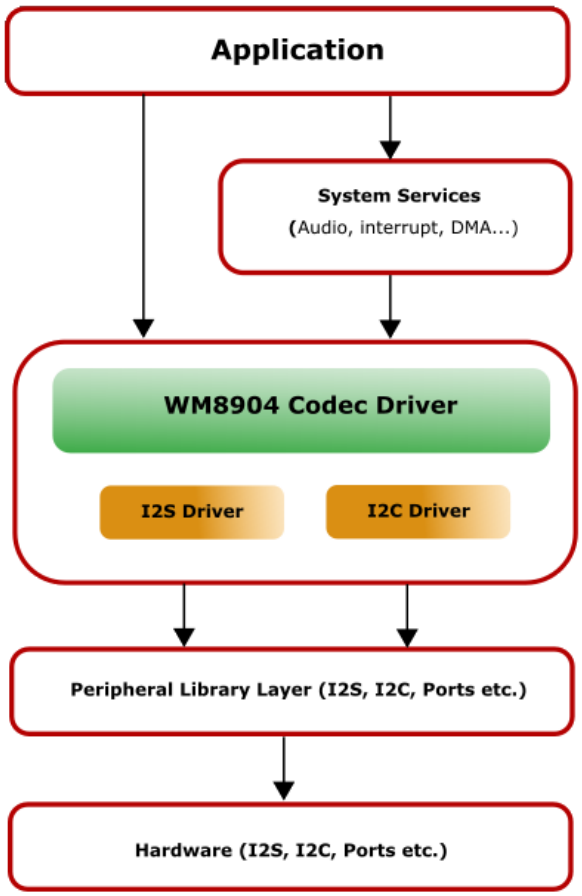
## Abstraction Model

This library provides a low-level abstraction of the WM8904 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

## Description

The abstraction model shown in the following diagram depicts how the WM8904 Codec Driver is positioned in the MPLAB Harmony framework. The WM8904 Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the WM8904 module.

**WM8904 Driver Abstraction Model**



## Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The WM8904 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced WM8904 Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the WM8904 Codec Driver Library.

| Library Interface Section | Description |
|---|---|
| System Functions | Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions. |
| Client Setup Functions | Provides open and close functions. |
| Data Transfer Functions | Provides data transfer functions, such as Buffer Read and Write. |
| Settings Functions | Provides driver specific functions for settings, such as volume control and sampling rate. |
| Other Functions | Miscellaneous functions, such as getting the driver's version number and syncing to the LRCLK signal. |
| Data Types and Constants | These data types and constants are required while interacting and setting up the WM8904 Codec Driver Library. |

**Note:** All functions and constants in this section are named with the format `DRV_ WM8904_xxx`, where 'xxx' is a function name or constant. These names are redefined in the appropriate configuration's `configuration.h` file to the format `DRV_CODEC_xxx` using #defines so that code in the application that references the library can be written as generically as possible (e.g., by writing `DRV_CODEC_Open` instead of `DRV_ WM8904_Open` etc.). This allows the codec type to be changed in the MHC without having to modify the application's source code.

## How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

## *Setup (Initialization)*

This topic describes system initialization, implementations, and includes a system access code example.

### Description

#### System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization in the system_init.c file, each instance of the WM8904 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_WM8904_INIT or by using Initialization Overrides) that are supported by the specific WM8904 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization

- Determines whether or not the microphone input is enabled

The DRV_WM8904_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ WM8904_Deinitialize, DRV_ WM8904_Status and DRV_I2S_Tasks.

### *Client Access*

This topic describes driver initialization and provides a code example.

## Description

For the application to start using an instance of the module, it must call the DRV_WM8904_Open function. The DRV_WM8904_Open function provides a driver handle to the WM8904 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_WM8904_Deinitialize, the application must call the DRV_WM8904_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to Data Types and Constants in the Library Interface section.

**Note:** It is necessary to check the status of driver initialization before opening a driver instance. The status of the WM8904 Codec Driver can be known by calling DRV_ WM8904_Status.

### Example:

```
DRV_HANDLE handle;
SYS_STATUS wm8904Status;
wm8904Status Status = DRV_WM8904_Status(sysObjects.wm8904Status DevObject);
if (SYS_STATUS_READY == wm8904Status)
{
// The driver can now be opened.
appData.wm8904Client.handle =  DRV_WM8904_Open
( DRV_WM8904_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.wm8904Client.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_WM8904_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* WM8904 Driver Is not ready */
}
```

### *Client Operations*

This topic provides information on client operations.

## Description

Client operations provide the API interface for control command and audio data transfer to the WM8904 Codec.

The following WM8904 Codec specific control command functions are provided:

- DRV_WM8904_SamplingRateSet
- DRV_WM8904_SamplingRateGet
- DRV_WM8904_VolumeSet

- DRV_WM8904_VolumeGet
- DRV_WM8904_MuteOn
- DRV_WM8904_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the WM8904 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_WM8904_BufferAddWrite, DRV_WM8904_BufferAddRead, and DRV_WM8904_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_WM8904_BUFFER_EVENT_COMPLETE, DRV_WM8904_BUFFER_EVENT_ERROR, or DRV_WM8904_BUFFER_EVENT_ABORT events.

**Note:** It is not necessary to close and reopen the client between multiple transfers.

### *Configuring the Library*

The configuration of the I2S Driver Library is based on the file `configurations.h`, as generated by the MHC.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

### System Configuration

## Macros

| | Name | Description |
|---|---|---|
| | DRV_WM8904_AUDIO_DATA_FORMAT_MACRO | Specifies the audio data format for the codec. |
| | DRV_WM8904_AUDIO_SAMPLING_RATE | Specifies the initial baud rate for the codec. |
| | DRV_WM8904_CLIENTS_NUMBER | Sets up the maximum number of clients that can be connected to any hardware instance. |
| | DRV_WM8904_ENABLE_MIC_BIAS | Specifies whether to enable the microphone bias. |
| | DRV_WM8904_ENABLE_MIC_INPUT | Specifies whether to enable the microphone input. |
| | DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2C interface. |
| | DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2S interface. |
| | DRV_WM8904_INSTANCES_NUMBER | Sets up the maximum number of hardware instances that can be supported |
| | DRV_WM8904_MASTER_MODE | Specifies if codec is in Master or Slave mode. |
| | DRV_WM8904_VOLUME | Specifies the initial volume level. |

## Description

### *DRV_WM8904_AUDIO_DATA_FORMAT_MACRO Macro*

Specifies the audio data format for the codec.

## Description

WM8904 Audio Data Format

Sets up the length of each sample plus the format (I2S or left-justified) for the audio.

Valid choices are: "DATA_16_BIT_LEFT_JUSTIFIED" "DATA_16_BIT_I2S" "DATA_32_BIT_LEFT_JUSTIFIED" "DATA_32_BIT_I2S"

### Remarks

If 24-bit audio is needed, it should be sent, left-justified, in a 32-bit format.

### C

```
#define DRV_WM8904_AUDIO_DATA_FORMAT_MACRO
```

## *DRV_WM8904_AUDIO_SAMPLING_RATE Macro*

Specifies the initial baud rate for the codec.

### Description

WM8904 Baud Rate

Sets the initial baud rate (sampling rate) for the codec. Typical values are 8000, 16000, 44100, 48000, 88200 and 96000.

### C

```
#define DRV_WM8904_AUDIO_SAMPLING_RATE
```

## *DRV_WM8904_CLIENTS_NUMBER Macro*

Sets up the maximum number of clients that can be connected to any hardware instance.

### Description

WM8904 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances.

### C

```
#define DRV_WM8904_CLIENTS_NUMBER
```

## *DRV_WM8904_ENABLE_MIC_BIAS Macro*

Specifies whether to enable the microphone bias.

### Description

WM8904 Microphone Enable

Indicates whether the bias voltage needed for electret microphones should be enabled.

### C

```
#define DRV_WM8904_ENABLE_MIC_BIAS
```

## *DRV_WM8904_ENABLE_MIC_INPUT Macro*

Specifies whether to enable the microphone input.

### Description

WM8904 Microphone Enable

Indicates whether the ADC inputs for the two microphone channels (L-R) should be enabled.

**C**

```
#define DRV_WM8904_ENABLE_MIC_INPUT
```

### *DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2C interface.

## Description

WM8904 I2C instance number

Specifies the instance number of the I2C interface being used by the MCU to send commands and receive status to and from the WM8904. enabled.

**C**

```
#define DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2S interface.

## Description

WM8904 I2S instance number

Specifies the instance number of the I2S interface being used by the MCU to send and receive audio data to and from the WM8904. enabled.

**C**

```
#define DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_WM8904_INSTANCES_NUMBER Macro*

Sets up the maximum number of hardware instances that can be supported

## Description

WM8904 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of WM8904 Codec modules that are needed by an application, namely one.

**C**

```
#define DRV_WM8904_INSTANCES_NUMBER
```

### *DRV_WM8904_MASTER_MODE Macro*

Specifies if codec is in Master or Slave mode.

## Description

WM8904 Codec Master/Slave Mode

Indicates whether the codec is to be operating in a Master mode (generating word and bit clock as outputs) or Slave mode receiving word and bit clock as inputs).

**C**

```
#define DRV_WM8904_MASTER_MODE
```

### *DRV_WM8904_VOLUME Macro*

Specifies the initial volume level.

## Description

WM8904 Volume

Sets the initial volume level, in the range 0-255.

## Remarks

The value is mapped to an internal WM8904 volume level in the range 0-192 using a logarithmic table so the input scale appears linear (128 is half volume).

## C

```
#define DRV_WM8904_VOLUME
```

### *Configuring MHC*

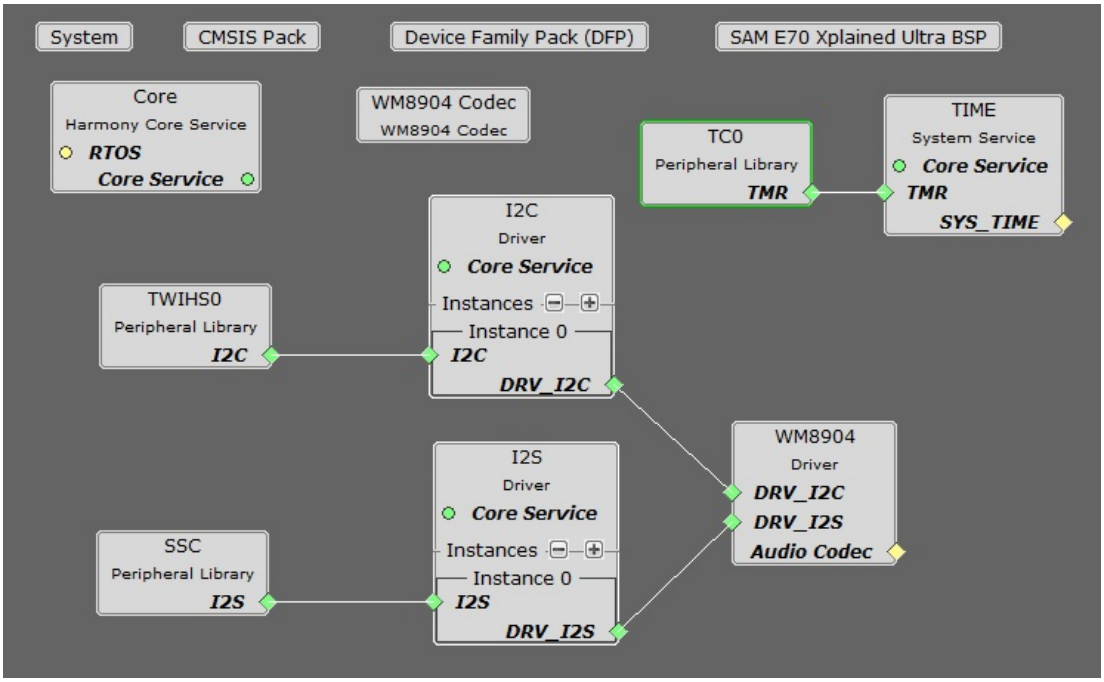
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

## Description

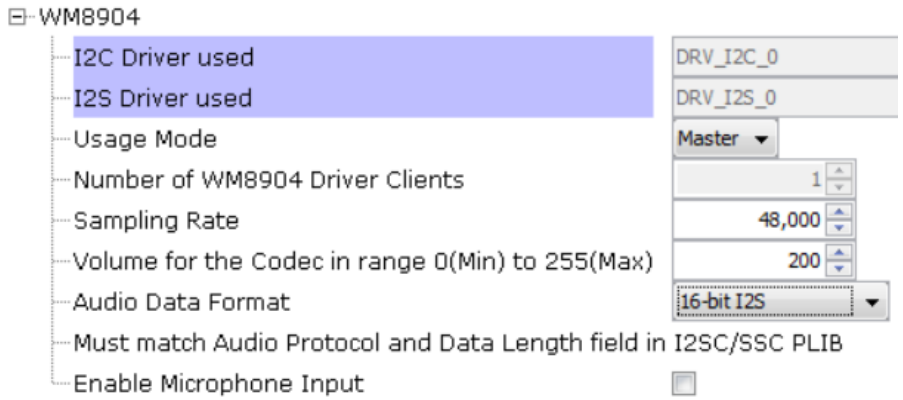
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In the MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on a codec template such as WM8904. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




Click on the WM8904 Driver component (not WM8904 Codec) and the following menu will be displayed in the Configurations Options:




- **I2C Driver Used** will display the driver instance used for the $I^2C$ interface.
- **I2S Driver Used** will display the driver instance used for the $I^2S$ interface.
- **Usage Mode** indicates whether the WM8904 is a Master (supplies $I^2S$ clocks) or a Slave (MCU supplies $I^2S$ clocks).
- **Number of WM8904 Clients** indicates the maximum number of clients that can be connected to the WM8904 Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either 16-bit Left Justified, 16-bit I2S, 32-bit Left Justified, or 32-bit I2S. It must match the audio protocol and data length set up in either the SSC or I2S PLIB.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Enable Microphone Input** should be checked a microphone is being used. If checked, another option,
- **Enable Microphone Bias** should be checked if using an electret microphone.

You can also bring in the WM8904 Driver by itself, by double clicking WM8904 under Audio->*Driver->Codec* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

Note that the WM8904 requires the TCx Peripheral Library and TIME System Service in order to perform some of its internal timing sequences.

## *Building the Library*

This section lists the files that are available in the WM8904 Codec Driver Library.

## Description

This section lists the files that are available in the `src` folder of the WM8904 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `audio/driver/codec/WM8904.`

### Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

| Source File Name | Description |
|---|---|
| `drv_wm8904.h` | Header file that exports the driver API. |

### Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

| Source File Name | Description |
|---|---|
| `/src/drv_wm8904.c` | This file contains implementation of the WM8904 Codec Driver. |

## Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

| Source File Name | Description |
|---|---|
| N/A | No optional files are available for this library. |

## Module Dependencies

The WM8904 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

***Library Interface***

### Client Setup Functions

| | Name | Description |
|---|---|---|
| | DRV_WM8904_Open | Opens the specified WM8904 driver instance and returns a handle to it |
| | DRV_WM8904_Close | Closes an opened-instance of the WM8904 driver |
| | DRV_WM8904_BufferEventHandlerSet | This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. |
| | DRV_WM8904_CommandEventHandlerSet | This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. |

### Data Transfer Functions

| | Name | Description |
|---|---|---|
| | DRV_WM8904_BufferAddRead | Schedule a non-blocking driver read operation. |
| | DRV_WM8904_BufferAddWrite | Schedule a non-blocking driver write operation. |
| | DRV_WM8904_BufferAddWriteRead | Schedule a non-blocking driver write-read operation. |
| | DRV_WM8904_ReadQueuePurge | Removes all buffer requests from the read queue. |
| | DRV_WM8904_WriteQueuePurge | Removes all buffer requests from the write queue. |

### Data Types and Constants

| | Name | Description |
|---|---|---|
| | DATA_LENGTH | in bits |
| | DRV_WM8904_AUDIO_DATA_FORMAT | Identifies the Serial Audio data interface format. |
| | DRV_WM8904_BUFFER_EVENT | Identifies the possible events that can result from a buffer add request. |
| | DRV_WM8904_BUFFER_EVENT_HANDLER | Pointer to a WM8904 Driver Buffer Event handler function |
| | DRV_WM8904_BUFFER_HANDLE | Handle identifying a write buffer passed to the driver. |
| | DRV_WM8904_CHANNEL | Identifies Left/Right Audio channel |
| | DRV_WM8904_COMMAND_EVENT_HANDLER | Pointer to a WM8904 Driver Command Event Handler Function |
| | DRV_WM8904_INIT | Defines the data required to initialize or reinitialize the WM8904 driver |
| | DRV_WM8904_BUFFER_HANDLE_INVALID | Definition of an invalid buffer handle. |
| | DRV_WM8904_COUNT | Number of valid WM8904 driver indices |
| | DRV_WM8904_INDEX_0 | WM8904 driver index definitions |
| | DRV_I2C_INDEX | This is macro DRV_I2C_INDEX. |

## Other Functions

| | Name | Description |
|---|---|---|
| | DRV_WM8904_GetI2SDriver | Get the handle to the I2S driver for this codec instance. |
| | DRV_WM8904_VersionGet | This function returns the version of WM8904 driver |
| | DRV_WM8904_VersionStrGet | This function returns the version of WM8904 driver in string format. |
| | DRV_WM8904_LRCLK_Sync | Synchronize to the start of the I2S LRCLK (left/right clock) signal |

## Settings Functions

| | Name | Description |
|---|---|---|
| | DRV_WM8904_MuteOff | This function disables WM8904 output for soft mute. |
| | DRV_WM8904_MuteOn | This function allows WM8904 output for soft mute on. |
| | DRV_WM8904_SamplingRateGet | This function gets the sampling rate set on the WM8904. |
| | DRV_WM8904_SamplingRateSet | This function sets the sampling rate of the media stream. |
| | DRV_WM8904_VolumeGet | This function gets the volume for WM8904 Codec. |
| | DRV_WM8904_VolumeSet | This function sets the volume for WM8904 Codec. |

## System Interaction Functions

| | Name | Description |
|---|---|---|
| | DRV_WM8904_Initialize | Initializes hardware and data for the instance of the WM8904 DAC module |
| | DRV_WM8904_Deinitialize | Deinitializes the specified instance of the WM8904 driver module |
| | DRV_WM8904_Status | Gets the current status of the WM8904 driver module. |
| | DRV_WM8904_Tasks | Maintains the driver's control and data interface state machine. |

### System Interaction Functions

### *DRV_WM8904_Initialize Function*

```
SYS_MODULE_OBJ  DRV_WM8904_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);
```

## Summary

Initializes hardware and data for the instance of the WM8904 DAC module

## Description

This routine initializes the WM8904 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

## Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this Codec driver.
DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this Codec driver.

## Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the driver instance to be initialized |

| | |
|---|---|
| init | Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used. |

### Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

### Remarks

This routine must be called before any other WM8904 routine is called.

This routine should only be called once during system initialization unless DRV_WM8904_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

### Example

```
DRV_WM8904_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = wm8904Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = wm8904Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_WM8904_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_WM8904_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_WM8904_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_WM8904_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_WM8904_Initialize(DRV_WM8904_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

### C

```
SYS_MODULE_OBJ DRV_WM8904_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

## *DRV_WM8904_Deinitialize Function*

void DRV_WM8904_Deinitialize( SYS_MODULE_OBJ object)

### Summary

Deinitializes the specified instance of the WM8904 driver module

### Description

Deinitializes the specified instance of the WM8904 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

### Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_WM8904_Initialize routine |

### Returns

None.

### Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

### Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_WM8904_Initialize
SYS_STATUS          status;


DRV_WM8904_Deinitialize(object);

status = DRV_WM8904_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

### C

```
void DRV_WM8904_Deinitialize(SYS_MODULE_OBJ object);
```

## *DRV_WM8904_Status Function*

SYS_STATUS DRV_WM8904_Status( SYS_MODULE_OBJ object)

### Summary

Gets the current status of the WM8904 driver module.

### Description

This routine provides the current status of the WM8904 driver module.

### Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_WM8904_Initialize routine |

### Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

### Remarks

A driver can opened only when its status is SYS_STATUS_READY.

### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize
SYS_STATUS          WM8904Status;

WM8904Status = DRV_WM8904_Status(object);
if (SYS_STATUS_READY == WM8904Status)
{
    // This means the driver can be opened using the
```

```
    // DRV_WM8904_Open() function.
}
```

**C**

```
SYS_STATUS DRV_WM8904_Status(SYS_MODULE_OBJ object);
```

### *DRV_WM8904_Tasks Function*

void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);

#### Summary

Maintains the driver's control and data interface state machine.

#### Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

#### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

#### Parameters

| Parameters | Description |
|---|---|
| object | Object handle for the specified driver instance (returned from DRV_WM8904_Initialize) |

#### Returns

None.

#### Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

#### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize

while (true)
{
    DRV_WM8904_Tasks (object);

    // Do other tasks
}
```

**C**

```
void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);
```

## Client Setup Functions

### *DRV_WM8904_Open Function*

DRV_HANDLE DRV_WM8904_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

## Summary

Opens the specified WM8904 driver instance and returns a handle to it

## Description

This routine opens the specified WM8904 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

The WM8904 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

## Preconditions

Function DRV_WM8904_Initialize must have been called before calling this function.

## Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the object instance to be opened |
| ioIntent | Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details. |

## Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur:

- if the number of client objects allocated via DRV_WM8904_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

## Remarks

The handle returned is valid until the DRV_WM8904_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

## Example

```
DRV_HANDLE handle;

handle = DRV_WM8904_Open(DRV_WM8904_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

## C

```
DRV_HANDLE DRV_WM8904_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

### *DRV_WM8904_Close Function*

void DRV_WM8904_Close( DRV_Handle handle )

### Summary

Closes an opened-instance of the WM8904 driver

### Description

This routine closes an opened-instance of the WM8904 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_WM8904_Open before the caller may use the driver again

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

- None

### Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

### Example

```
DRV_HANDLE handle;  // Returned from DRV_WM8904_Open

DRV_WM8904_Close(handle);
```

### C

```
void DRV_WM8904_Close(const DRV_HANDLE handle);
```

## *DRV_WM8904_BufferEventHandlerSet Function*

void DRV_WM8904_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_WM8904_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

### Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

### Description

When a client calls DRV_WM8904_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

## Returns

None.

## Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

## C

```
void DRV_WM8904_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_WM8904_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

### *DRV_WM8904_CommandEventHandlerSet Function*

void DRV_WM8904_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_WM8904_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

## Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

## Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

The event handler should be set before the client performs any "WM8904 Codec Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

## Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

## Returns

None.

## Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_CommandEventHandlerSet(myWM8904Handle,
                APP_WM8904CommandEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_WM8904_DeEmphasisFilterSet(myWM8904Handle, DRV_WM8904_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_WM8904CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

**C**

```
void DRV_WM8904_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_WM8904_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

## Data Transfer Functions

### *DRV_WM8904_BufferAddRead Function*

void DRV_WM8904_BufferAddRead

(

const DRV_HANDLE handle,

DRV_WM8904_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

### Summary

Schedule a non-blocking driver read operation.

### Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_WM8904_Open call.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the WM8904 instance as return by the DRV_WM8904_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

### C

```
void DRV_WM8904_BufferAddRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

## *DRV_WM8904_BufferAddWrite Function*

void DRV_WM8904_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_WM8904_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

### Summary

Schedule a non-blocking driver write operation.

### Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_WM8904_Open call.

## Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the WM8904 instance as return by the DRV_WM8904_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

## Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

## Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_WM8904_BufferAddWrite(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

### *DRV_WM8904_BufferAddWriteRead Function*

void DRV_WM8904_BufferAddWriteRead
(
const DRV_HANDLE handle,
DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)

## Summary

Schedule a non-blocking driver write-read operation.

## Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

## Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_WM8904_Open call.

## Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the WM8904 instance as returned by the DRV_WM8904_Open function |
| bufferHandle | Pointer to an argument that will contain the return buffer handle |
| transmitBuffer | The buffer where the transmit data will be stored |
| receiveBuffer | The buffer where the received data will be stored |
| size | Buffer size in bytes |

## Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

## Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not

otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every WM8904 write. The transmit and receive size must be same.

### Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// mywm8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(mywm8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWriteRead(mywm8904handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_WM8904_BufferAddWriteRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

## *DRV_WM8904_ReadQueuePurge Function*

bool DRV_WM8904_ReadQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the read queue.

### Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout

or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_WM8904_Open function. |

### Returns

True - Read queue purge is successful. False - Read queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myCodecHandle is the handle returned by the DRV_WM8904_Open function.
// Use DRV_WM8904_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_WM8904_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_WM8904_ReadQueuePurge(const DRV_HANDLE handle);
```

## *DRV_WM8904_WriteQueuePurge Function*

bool DRV_WM8904_WriteQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the write queue.

### Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_WM8904_Open function. |

### Returns

True - Write queue purge is successful. False - Write queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myCodecHandle is the handle returned by the DRV_WM8904_Open function.
// Use DRV_WM8904_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_WM8904_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_WM8904_WriteQueuePurge(const DRV_HANDLE handle);
```

## Settings Functions

## *DRV_WM8904_MuteOff Function*

void DRV_WM8904_MuteOff(DRV_HANDLE handle)

### Summary

This function disables WM8904 output for soft mute.

### Description

This function disables WM8904 output for soft mute.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;
```

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

    DRV_WM8904_MuteOff(myWM8904Handle); //WM8904 output soft mute disabled
```

### C

```
void DRV_WM8904_MuteOff(DRV_HANDLE handle);
```

## *DRV_WM8904_MuteOn Function*

void DRV_WM8904_MuteOn(DRV_HANDLE handle);

### Summary

This function allows WM8904 output for soft mute on.

### Description

This function Enables WM8904 output for soft mute.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_MuteOn(myWM8904Handle);  //WM8904 output soft muted
```

### C

```
void DRV_WM8904_MuteOn(DRV_HANDLE handle);
```

## *DRV_WM8904_SamplingRateGet Function*

uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle)

### Summary

This function gets the sampling rate set on the WM8904.

### Description

This function gets the sampling rate set on the DAC WM8904.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Remarks

None.

### Example

```
uint32_t baudRate;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

baudRate = DRV_WM8904_SamplingRateGet(myWM8904Handle);
```

### C

```
uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle);
```

## *DRV_WM8904_SamplingRateSet Function*

void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

### Summary

This function sets the sampling rate of the media stream.

### Description

This function sets the media sampling rate for the client handle.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| samplingRate | Sampling frequency in Hz |

### Returns

None.

### Remarks

None.

### Example

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_SamplingRateSet(myWM8904Handle, 48000);  //Sets 48000 media sampling rate
```

### C

```
void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

## *DRV_WM8904_VolumeGet Function*

uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel)

### Summary

This function gets the volume for WM8904 Codec.

### Description

This functions gets the current volume programmed to the Codec WM8904.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| channel | argument indicating Left or Right or Both channel volume to be modified |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

  volume = DRV_WM8904_VolumeGet(myWM8904Handle, DRV_WM8904_CHANNEL_LEFT);
```

### C

```
uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel);
```

## *DRV_WM8904_VolumeSet Function*

void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);

### Summary

This function sets the volume for WM8904 Codec.

### Description

This function sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| channel | argument indicating Left or Right or Both channel volume to be modified |

| | |
|---|---|
| volume | volume value specified in the range 0-255 (0x00 to 0xFF) |

### Returns

None

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

    DRV_WM8904_VolumeSet(myWM8904Handle,DRV_WM8904_CHANNEL_LEFT, 120);
```

### C

```
void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);
```

## Other Functions

## *DRV_WM8904_GetI2SDriver Function*

DRV_HANDLE DRV_WM8904_GetI2SDriver(DRV_HANDLE codecHandle)

### Summary

Get the handle to the I2S driver for this codec instance.

### Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

### Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

A handle to the I2S driver for this codec instance

### Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

### C

```
DRV_HANDLE DRV_WM8904_GetI2SDriver(DRV_HANDLE codecHandle);
```

### *DRV_WM8904_VersionGet Function*

uint32_t DRV_WM8904_VersionGet( void )

#### Summary

This function returns the version of WM8904 driver

#### Description

The version number returned from the DRV_WM8904_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

#### Preconditions

None.

#### Returns

returns the version of WM8904 driver.

#### Remarks

None.

#### Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

#### Example 2

```
uint32_t WM8904version;
WM8904version = DRV_WM8904_VersionGet();
```

#### C

```
uint32_t DRV_WM8904_VersionGet();
```

### *DRV_WM8904_VersionStrGet Function*

int8_t* DRV_WM8904_VersionStrGet(void)

#### Summary

This function returns the version of WM8904 driver in string format.

#### Description

The DRV_WM8904_VersionStrGet function returns a string in the format: ".[.][]" Where: is the WM8904 driver's version number. is the WM8904 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

#### Preconditions

None.

#### Returns

returns a string containing the version of WM8904 driver.

#### Remarks

None

### Example

```
int8_t *WM8904string;
WM8904string = DRV_WM8904_VersionStrGet();
```

### C

```
int8_t* DRV_WM8904_VersionStrGet();
```

## *DRV_WM8904_LRCLK_Sync Function*

uint32_t DRV_WM8904_LRCLK_Sync (const DRV_HANDLE handle);

### Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

### Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

### Preconditions

None.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_LRCLK_Sync(myWM8904Handle);
```

### C

```
bool DRV_WM8904_LRCLK_Sync(const DRV_HANDLE handle);
```

## Data Types and Constants

## *DATA_LENGTH Type*

in bits

### C

```
typedef enum DATA_LENGTH@3 DATA_LENGTH;
```

### *DRV_WM8904_AUDIO_DATA_FORMAT Enumeration*

Identifies the Serial Audio data interface format.

## Description

WM8904 Audio data format

This enumeration identifies Serial Audio data interface format.

## C

```
typedef enum {
  DATA_16_BIT_LEFT_JUSTIFIED,
  DATA_16_BIT_I2S,
  DATA_32_BIT_LEFT_JUSTIFIED,
  DATA_32_BIT_I2S
} DRV_WM8904_AUDIO_DATA_FORMAT;
```

### *DRV_WM8904_BUFFER_EVENT Enumeration*

Identifies the possible events that can result from a buffer add request.

## Description

WM8904 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_WM8904_BufferAddWrite() or the DRV_WM8904_BufferAddRead() function.

## Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_WM8904_BufferEventHandlerSet function when a buffer transfer request is completed.

## C

```
typedef enum {
  DRV_WM8904_BUFFER_EVENT_COMPLETE,
  DRV_WM8904_BUFFER_EVENT_ERROR,
  DRV_WM8904_BUFFER_EVENT_ABORT
} DRV_WM8904_BUFFER_EVENT;
```

### *DRV_WM8904_BUFFER_EVENT_HANDLER Type*

Pointer to a WM8904 Driver Buffer Event handler function

## Description

WM8904 Driver Buffer Event Handler Function

This data type defines the required function signature for the WM8904 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

## Parameters

| Parameters | Description |
|---|---|
| event | Identifies the type of event |
| bufferHandle | Handle identifying the buffer to which the event relates |
| context | Value identifying the context of the application that registered the event handling function. |

### Returns

None.

### Remarks

If the event is DRV_WM8904_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_WM8904_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_WM8904_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_WM8904_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

### Example

```
void APP_MyBufferEventHandler( DRV_WM8904_BUFFER_EVENT event,
                               DRV_WM8904_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

### C

```
typedef void (* DRV_WM8904_BUFFER_EVENT_HANDLER)(DRV_WM8904_BUFFER_EVENT event,
DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

## *DRV_WM8904_BUFFER_HANDLE Type*

Handle identifying a write buffer passed to the driver.

### Description

WM8904 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_WM8904_BufferAddWrite() or DRV_WM8904_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

### Remarks

None

### C

```
typedef uintptr_t DRV_WM8904_BUFFER_HANDLE;
```

## *DRV_WM8904_CHANNEL Enumeration*

Identifies Left/Right Audio channel

### Description

WM8904 Audio Channel

This enumeration identifies Left/Right Audio channel

### Remarks

None.

### C

```
typedef enum {
  DRV_WM8904_CHANNEL_LEFT,
  DRV_WM8904_CHANNEL_RIGHT,
  DRV_WM8904_CHANNEL_LEFT_RIGHT,
  DRV_WM8904_NUMBER_OF_CHANNELS
} DRV_WM8904_CHANNEL;
```

## *DRV_WM8904_COMMAND_EVENT_HANDLER Type*

Pointer to a WM8904 Driver Command Event Handler Function

### Description

WM8904 Driver Command Event Handler Function

This data type defines the required function signature for the WM8904 driver command event handling callback function.

A command is a control instruction to the WM8904 Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

### Parameters

| Parameters | Description |
|---|---|
| context | Value identifying the context of the application that registered the event handling function. |

### Returns

None.

### Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

### Example

```
void APP_WM8904CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

### C

```
typedef void (* DRV_WM8904_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

## *DRV_WM8904_INIT Structure*

Defines the data required to initialize or reinitialize the WM8904 driver

### Description

WM8904 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the WM8904 Codec driver.

### Remarks

None.

### C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  bool masterMode;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_WM8904_AUDIO_DATA_FORMAT audioDataFormat;
  bool enableMicInput;
  bool enableMicBias;
} DRV_WM8904_INIT;
```

## *DRV_WM8904_BUFFER_HANDLE_INVALID Macro*

Definition of an invalid buffer handle.

### Description

WM8904 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_WM8904_BufferAddWrite() and the DRV_WM8904_BufferAddRead() function if the buffer add request was not successful.

### Remarks

None.

### C

```
#define DRV_WM8904_BUFFER_HANDLE_INVALID ((DRV_WM8904_BUFFER_HANDLE)(-1))
```

## *DRV_WM8904_COUNT Macro*

Number of valid WM8904 driver indices

## Description

WM8904 Driver Module Count

This constant identifies the maximum number of WM8904 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of WM8904 instances on this microcontroller.

## Remarks

This value is part-specific.

## C

```
#define DRV_WM8904_COUNT
```

### *DRV_WM8904_INDEX_0 Macro*

WM8904 driver index definitions

## Description

Driver WM8904 Module Index

These constants provide WM8904 driver index definition.

## Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_WM8904_Initialize and DRV_WM8904_Open routines to identify the driver instance in use.

## C

```
#define DRV_WM8904_INDEX_0 0
```

### *DRV_I2C_INDEX Macro*

This is macro DRV_I2C_INDEX.

## C

```
#define DRV_I2C_INDEX DRV_WM8904_I2C_INSTANCES_NUMBER
```

### *Files*

## Files

| Name | Description |
|---|---|
| drv_wm8904.h | WM8904 Codec Driver Interface header file |
| drv_wm8904_config_template.h | WM8904 Codec Driver Configuration Template. |

## Description

This section will list only the library's interface header file(s).

### drv_wm8904.h

drv_wm8904.h

## Summary

WM8904 Codec Driver Interface header file

## Description

WM8904 Codec Driver Interface

The WM8904 Codec device driver interface provides a simple interface to manage the WM8904 16/24/32-Bit Codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the WM8904 Codec device driver.

### drv_wm8904_config_template.h

drv_wm8904_config_template.h

## Summary

WM8904 Codec Driver Configuration Template.

## Description

WM8904 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

# AK4954 CODEC Driver Library Help

This topic describes the AK4954 Codec Driver Library.

### *Introduction*

This topic describes the basic architecture of the AK4954 Codec Driver Library and provides information and examples on its use.

## Description

**Interface Header File**: `drv_AK4954.h`

The interface to the AK4954 Codec Driver library is defined in the `audio/driver/codec/AK4954/drv_AK4954.h` header file. Any C language source (.c) file that uses the AK4954 Codec Driver library should include this header.

### Library Source Files:

The AK4954 Codec Driver library source files are provided in the `audio/driver/codec/AK4954/src` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features and to **Building the Library** for instructions on how to build the library.

**Example Applications:**

This library is used by the following applications:

- `audio/apps/audio_tone`
- `audio/apps/microphone_loopback`

### *Using the Library*

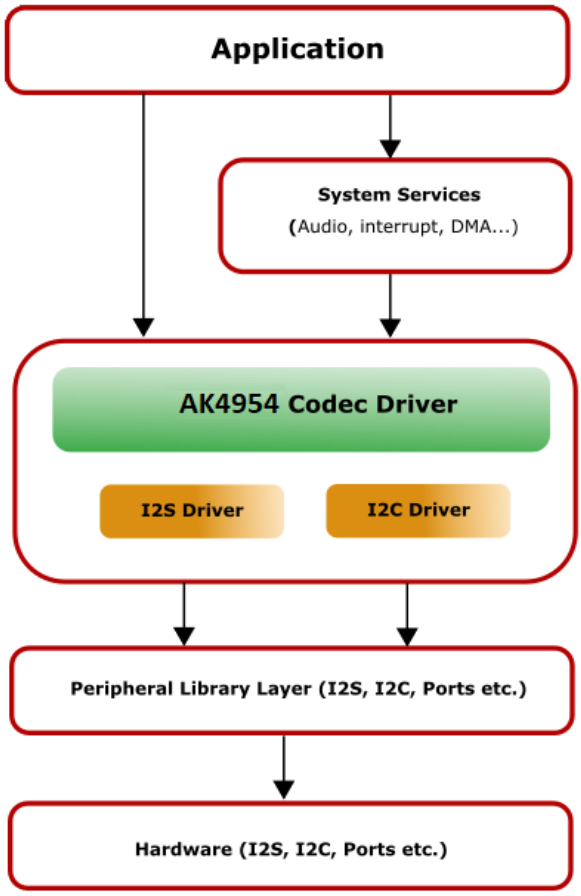
### Abstration Model

This library provides a low-level abstraction of the AK4954 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

## Description

The abstraction model shown in the following diagram depicts how the AK4954 Codec Driver is positioned in the MPLAB Harmony framework. The AK4954 Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the AK4954 module.

**AK4954 Driver Abstraction Model**




## Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4954 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4954 Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4954 Codec Driver Library.

| Library Interface Section | Description |
|---|---|
| System Functions | Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions. |
| Client Setup Functions | Provides open and close functions. |
| Data Transfer Functions | Provides data transfer functions, such as Buffer Read and Write. |
| Settings Functions | Provides driver specific functions for settings, such as volume control and sampling rate. |
| Other Functions | Miscellaneous functions, such as getting the driver's version number and syncing to the LRCLK signal. |
| Data Types and Constants | These data types and constants are required while interacting and setting up the AK4954 Codec Driver Library. |

**Note:** All functions and constants in this section are named with the format DRV_ AK4954_xxx, where 'xxx' is a function name or constant. These names are redefined in the appropriate configuration's configuration.h file to the format DRV_CODEC_xxx using #defines so that code in the application that references the library can be written as generically as possible (e.g., by writing DRV_CODEC_Open instead of DRV_ AK4954_Open etc.). This allows the codec type to be changed in the MHC without having to modify the application's source code.

## How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

## *Setup (Initialization)*

This topic describes system initialization, implementations, and includes a system access code example.

### Description

#### System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization in the system_init.c file, each instance of the AK4954 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4954_INIT or by using Initialization Overrides) that are supported by the specific AK4954 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

The DRV_AK4954_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4954_Deinitialize, DRV_ AK4954_Status and DRV_I2S_Tasks

## *Client Access*

This topic describes driver initialization and provides a code example.

### Description

For the application to start using an instance of the module, it must call the DRV_AK4954_Open function. The DRV_AK4954_Open function provides a driver handle to the AK4954 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4954_Deinitialize, the application must call the DRV_AK4954_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to Data Types and Constants in the Library Interface section.

**Note:** It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4954 Codec Driver can be known by calling DRV_ AK4954_Status.

#### Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4954Status;
```

```
ak4954Status Status = DRV_AK4954_Status(sysObjects.ak4954Status DevObject);
if (SYS_STATUS_READY == ak4954Status)
{
// The driver can now be opened.
appData.ak4954Client.handle =  DRV_AK4954_Open
( DRV_AK4954_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.ak4954Client.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_AK4954_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* AK4954 Driver Is not ready */
}
```

## *Client Operations*

This topic provides information on client operations.

### Description

Client operations provide the API interface for control command and audio data transfer to the AK4954 Codec.

The following AK4954 Codec specific control command functions are provided:

- DRV_AK4954_SamplingRateSet
- DRV_AK4954_SamplingRateGet
- DRV_AK4954_VolumeSet
- DRV_AK4954_VolumeGet
- DRV_AK4954_MuteOn
- DRV_AK4954_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4954 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_AK4954_BufferAddWrite, DRV_AK4954_BufferAddRead, and DRV_AK4954_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK4954_BUFFER_EVENT_COMPLETE, DRV_AK4954_BUFFER_EVENT_ERROR, or DRV_AK4954_BUFFER_EVENT_ABORT events.

**Note:** It is not necessary to close and reopen the client between multiple transfers.

## *Configuring the Library*

The configuration of the I2S Driver Library is based on the file configurations.h, as generated by the MHC.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

## System Configuration

### Macros

| | Name | Description |
|---|---|---|
| | DRV_AK4954_AUDIO_DATA_FORMAT_MACRO | Specifies the audio data format for the codec. |
| | DRV_AK4954_AUDIO_SAMPLING_RATE | Specifies the initial baud rate for the codec. |
| | DRV_AK4954_CLIENTS_NUMBER | Sets up the maximum number of clients that can be connected to any hardware instance. |
| | DRV_AK4954_ENABLE_MIC_BIAS | Specifies whether to enable the microphone bias. |
| | DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2C interface. |
| | DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2S interface. |
| | DRV_AK4954_INSTANCES_NUMBER | Sets up the maximum number of hardware instances that can be supported |
| | DRV_AK4954_MASTER_MODE | Specifies if codec is in Master or Slave mode. |
| | DRV_AK4954_MIC_GAIN | Specifies the gain of the microphone (external or line input only) |
| | DRV_AK4954_VOLUME | Specifies the initial volume level. |
| | DRV_AK4954_WHICH_MIC_INPUT | Specifies whether to enable the microphone input. |

### Description

## *DRV_AK4954_AUDIO_DATA_FORMAT_MACRO Macro*

Specifies the audio data format for the codec.

### Description

AK4954 Audio Data Format

Sets up the length of each sample plus the format (I2S or left-justified) for the audio.

Valid choices are: "DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_LSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_16BIT_LSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_MSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_I2S_16BIT_24BIT"
"DRV_AK4954_AUDIO_DATA_FORMAT_32BIT_MSB_SDTO_32BIT_MSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_I2S_32BIT" where SDTO is input line (ADC) and STDI is output line (DAC)

### Remarks

If 24-bit audio is needed, it should be sent, left-justified, in a 32-bit format.

### C

```
#define DRV_AK4954_AUDIO_DATA_FORMAT_MACRO
```

## *DRV_AK4954_AUDIO_SAMPLING_RATE Macro*

Specifies the initial baud rate for the codec.

### Description

AK4954 Baud Rate

Sets the initial baud rate (sampling rate) for the codec. Typical values are 8000, 16000, 44100, 48000, 88200 and 96000.

**C**

```
#define DRV_AK4954_AUDIO_SAMPLING_RATE
```

### *DRV_AK4954_CLIENTS_NUMBER Macro*

Sets up the maximum number of clients that can be connected to any hardware instance.

## Description

AK4954 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances.

**C**

```
#define DRV_AK4954_CLIENTS_NUMBER
```

### *DRV_AK4954_ENABLE_MIC_BIAS Macro*

Specifies whether to enable the microphone bias.

## Description

AK4954 Microphone Enable

Indicates whether the bias voltage needed for electret microphones should be enabled.

**C**

```
#define DRV_AK4954_ENABLE_MIC_BIAS
```

### *DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2C interface.

## Description

AK4954 I2C instance number

Specifies the instance number of the I2C interface being used by the MCU to send commands and receive status to and from the AK4954. enabled.

**C**

```
#define DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2S interface.

## Description

AK4954 I2S instance number

Specifies the instance number of the I2S interface being used by the MCU to send and receive audio data to and from the AK4954. enabled.

**C**

```
#define DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_AK4954_INSTANCES_NUMBER Macro*

Sets up the maximum number of hardware instances that can be supported

## Description

AK4954 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4954 Codec modules that are needed by an application, namely one.

## C

```
#define DRV_AK4954_INSTANCES_NUMBER
```

### *DRV_AK4954_MASTER_MODE Macro*

Specifies if codec is in Master or Slave mode.

## Description

AK4954 Codec Master/Slave Mode

Indicates whether the codec is to be operating in a Master mode (generating word and bit clock as outputs) or Slave mode receiving word and bit clock as inputs).

## C

```
#define DRV_AK4954_MASTER_MODE
```

### *DRV_AK4954_MIC_GAIN Macro*

Specifies the gain of the microphone (external or line input only)

## Description

AK4954 Microphone Gain

Specifies the gain of the microphone (external or line input only), on a scale of 0-31

## C

```
#define DRV_AK4954_MIC_GAIN
```

### *DRV_AK4954_VOLUME Macro*

Specifies the initial volume level.

## Description

AK4954 Volume

Sets the initial volume level, in the range 0-255.

## Remarks

The value is mapped to an internal AK4954 volume level in the range 0-192 using a logarithmic table so the input scale appears linear (128 is half volume).

## C

```
#define DRV_AK4954_VOLUME
```

### *DRV_AK4954_WHICH_MIC_INPUT Macro*

Specifies whether to enable the microphone input.

## Description

AK4954 Which Microphone

Indicates which microphone (or line input) is chosen

Valid choices are: "MIC1" (Internal Mic on board) "MIC2" (External Mic Input) "MIC3" (Line Input)

## C

```
#define DRV_AK4954_WHICH_MIC_INPUT
```

### *Configuring MHC*

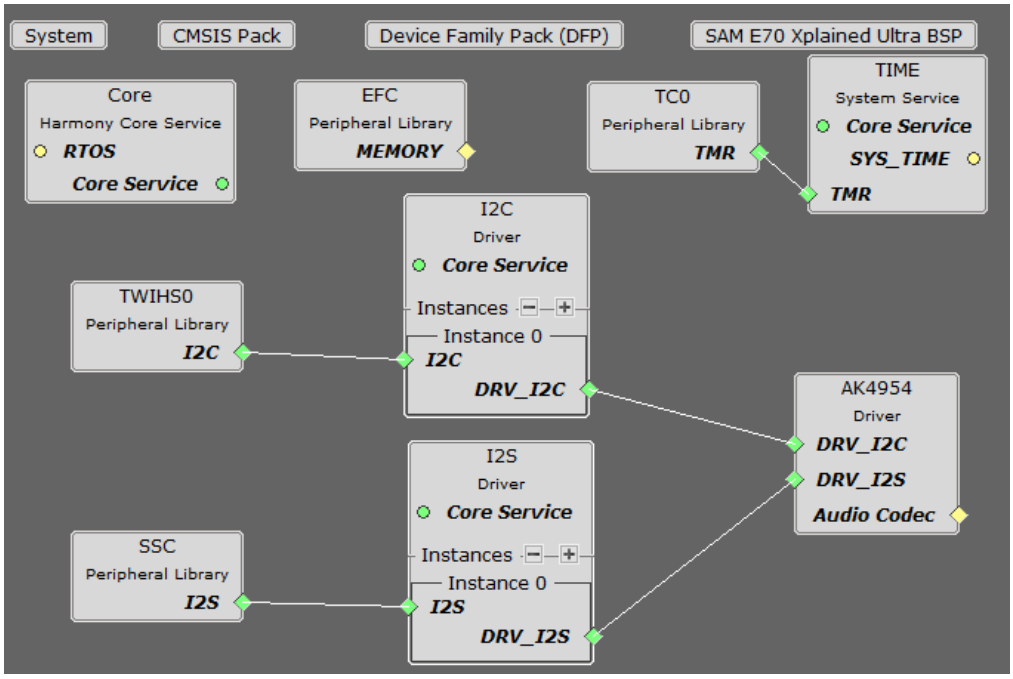
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

## Description

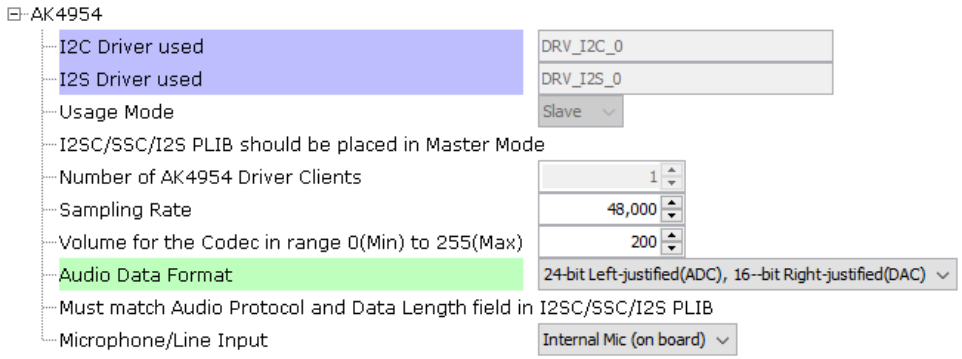
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In the MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on a codec template such as AK4954. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




Click on the AK4954 Driver component (not AK4954 Codec) and the following menu will be displayed in the Configurations Options:




- **I2C Driver Used** will display the driver instance used for the $I^2C$ interface.
- **I2S Driver Used** will display the driver instance used for the $I^2S$ interface.
- **Usage Mode** indicates whether the AK4954 is a Master (supplies $I^2S$ clocks) or a Slave (MCU supplies $I^2S$ clocks).
- **Number of AK4954 Clients** indicates the maximum number of clients that can be connected to the AK4954 Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either
  - 24-bit Left Justified (ADC), 24-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 16-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 24-bit Left-justified(DAC)
  - 24/16-bit I2S
  - 32-bit Left Justified (ADC), 32-bit Left-justified(DAC)
  - 32-bit I2S

It must match the audio protocol and data length set up in either the SSC or I2S PLIB.

- **Microphone/Line Input** selects which microphone or line input is selected, either:
  - Internal Mic (mounted on the AK4954 daughterboard)
  - External Mic Input
  - Line Input

If External Mic input or Line Input is selected, then the following option is provided:

- **Ext Mic Gain in dB range 0(min) to 31(max)**

If External Mic input is selected, then the following option is provided:

- **Enable Microphone Bias** should be checked if using an electret microphone.

You can also bring in the AK4954 Driver by itself, by double clicking AK4954 under Audio->*Driver*->*Codec* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

Note that the AK4954 requires the TCx Peripheral Library and TIME System Service in order to perform some of its internal timing sequences.

## *Building the Library*

This section lists the files that are available in the AK4954 Codec Driver Library.

## Description

This section lists the files that are available in the `src` folder of the AK4954 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `audio/driver/codec/AK4954.`

### Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

| Source File Name | Description |
|---|---|
| `drv_ak4954.h` | Header file that exports the driver API. |

## Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

| Source File Name | Description |
|---|---|
| /src/drv_ak4954.c | This file contains implementation of the AK4954 Codec Driver. |

## Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

| Source File Name | Description |
|---|---|
| N/A | No optional files are available for this library. |

## Module Dependencies

The AK4954 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

### *Library Interface*

### Client Setup Functions

| | Name | Description |
|---|---|---|
| | DRV_AK4954_Open | Opens the specified AK4954 driver instance and returns a handle to it. |
| | DRV_AK4954_Close | Closes an opened-instance of the AK4954 driver |
| | DRV_AK4954_BufferEventHandlerSet | This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. |
| | DRV_AK4954_CommandEventHandlerSet | This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. |

### Data Transfer Functions

| | Name | Description |
|---|---|---|
| | DRV_AK4954_BufferAddRead | Schedule a non-blocking driver read operation. |
| | DRV_AK4954_BufferAddWrite | Schedule a non-blocking driver write operation. |
| | DRV_AK4954_BufferAddWriteRead | Schedule a non-blocking driver write-read operation. |
| | DRV_AK4954_ReadQueuePurge | Removes all buffer requests from the read queue. |
| | DRV_AK4954_WriteQueuePurge | Removes all buffer requests from the write queue. |

### Data Types and Constants

| | Name | Description |
|---|---|---|
| | DRV_AK4954_AUDIO_DATA_FORMAT | Identifies the Serial Audio data interface format. |
| | DRV_AK4954_BUFFER_EVENT | Identifies the possible events that can result from a buffer add request. |
| | DRV_AK4954_BUFFER_EVENT_HANDLER | Pointer to a AK4954 Driver Buffer Event handler function |
| | DRV_AK4954_BUFFER_HANDLE | Handle identifying a write buffer passed to the driver. |
| | DRV_AK4954_CHANNEL | Identifies Left/Right Audio channel |
| | DRV_AK4954_COMMAND_EVENT_HANDLER | Pointer to a AK4954 Driver Command Event Handler Function |

| | | |
|---|---|---|
| | DRV_AK4954_DIGITAL_BLOCK_CONTROL | Identifies Bass-Boost Control function |
| | DRV_AK4954_INIT | Defines the data required to initialize or reinitialize the AK4954 driver |
| | DRV_AK4954_INT_EXT_MIC | Identifies the Mic input source. |
| | DRV_AK4954_MIC | This is type DRV_AK4954_MIC. |
| | DRV_AK4954_MONO_STEREO_MIC | Identifies the Mic input as Mono / Stereo. |
| | SAMPLE_LENGTH | in bits |
| | DRV_AK4954_AUDIO_DATA_FORMAT_I2S | for compatability with old code |
| | DRV_AK4954_BUFFER_HANDLE_INVALID | Definition of an invalid buffer handle. |
| | DRV_AK4954_COUNT | Number of valid AK4954 driver indices |
| | DRV_AK4954_INDEX_0 | AK4954 driver index definitions |
| | DRV_AK4954_INDEX_1 | This is macro DRV_AK4954_INDEX_1. |
| | DRV_AK4954_INDEX_2 | This is macro DRV_AK4954_INDEX_2. |
| | DRV_AK4954_INDEX_3 | This is macro DRV_AK4954_INDEX_3. |
| | DRV_AK4954_INDEX_4 | This is macro DRV_AK4954_INDEX_4. |
| | DRV_AK4954_INDEX_5 | This is macro DRV_AK4954_INDEX_5. |

## Other Functions

| | Name | Description |
|---|---|---|
| | DRV_AK4954_LRCLK_Sync | Synchronize to the start of the I2S LRCLK (left/right clock) signal |
| | DRV_AK4954_GetI2SDriver | Get the handle to the I2S driver for this codec instance. |
| | DRV_AK4954_VersionStrGet | This function returns the version of AK4954 driver in string format. |
| | DRV_AK4954_VersionGet | This function returns the version of AK4954 driver. |

## Settings Functions

| | Name | Description |
|---|---|---|
| | DRV_AK4954_MicGainGet | This function gets the microphone gain for the AK4954 Codec. |
| | DRV_AK4954_MicGainSet | This function sets the microphone gain for the AK4954 CODEC. |
| | DRV_AK4954_MicMuteOff | Umutes th AK4954's microphone input. |
| | DRV_AK4954_MicMuteOn | Mutes the AK4954's microphone input |
| | DRV_AK4954_MicSet | This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input. |
| | DRV_AK4954_MonoStereoMicSet | This function sets up the codec for the Mono or Stereo microphone mode. |
| | DRV_AK4954_MuteOff | This function disables AK4954 output for soft mute. |
| | DRV_AK4954_MuteOn | This function allows AK4954 output for soft mute on. |
| | DRV_AK4954_IntExtMicSet | This function sets up the codec for the X32 DB internal or the external microphone use. |
| | DRV_AK4954_SamplingRateGet | This function gets the sampling rate set on the DAC AK4954. |
| | DRV_AK4954_SamplingRateSet | This function sets the sampling rate of the media stream. |
| | DRV_AK4954_VolumeGet | This function gets the volume for AK4954 Codec. |
| | DRV_AK4954_VolumeSet | This function sets the volume for AK4954 Codec. |

## System Interaction Functions

| | Name | Description |
|---|---|---|
| | DRV_AK4954_Initialize | Initializes hardware and data for the instance of the AK4954 Codec module. |
| | DRV_AK4954_EnableInitialization | Enable delayed initialization of the driver. |
| | DRV_AK4954_IsInitializationDelayed | Checks if delayed initialization of the driver has been requested. |
| | DRV_AK4954_Deinitialize | Deinitializes the specified instance of the AK4954 driver module. |
| | DRV_AK4954_Status | Gets the current status of the AK4954 driver module. |
| | DRV_AK4954_Tasks | Maintains the driver's control and data interface state machine. |

## Description

## System Interaction Functions

### *DRV_AK4954_Initialize Function*

SYS_MODULE_OBJ DRV_AK4954_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);

### Summary

Initializes hardware and data for the instance of the AK4954 Codec module.

### Description

This routine initializes the AK4954 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

### Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this Codec driver.
DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this Codec driver.

### Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the driver instance to be initialized |
| init | Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used. |

### Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

### Remarks

This routine must be called before any other AK4954 routine is called.

This routine should only be called once during system initialization unless DRV_AK4954_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

### Example

```
DRV_AK4954_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                             = true;
init->status                            = SYS_STATUS_BUSY;
init->numClients                        = 0;
init->i2sDriverModuleIndex              = ak4954Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex              = ak4954Init->i2cDriverModuleIndex;
init->samplingRate                      = DRV_AK4954_AUDIO_SAMPLING_RATE;
init->audioDataFormat                   = DRV_AK4954_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4954_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4954Init->volume;
}
init->isInInterruptContext              = false;

init->commandCompleteCallback = (DRV_AK4954_COMMAND_EVENT_HANDLER)0;
```

```
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4954_Initialize(DRV_AK4954_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

**C**

```
SYS_MODULE_OBJ DRV_AK4954_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

### *DRV_AK4954_EnableInitialization Function*

void DRV_AK4954_EnableInitialization(SYS_MODULE_OBJ object);

#### Summary

Enable delayed initialization of the driver.

#### Description

If the AK4954 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. Once this has been accomplished, this function should be called to kick-start the codec driver initialization.

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

#### Parameters

| Parameters | Description |
|---|---|
| object | Object handle for the specified driver instance (returned from DRV_AK4954_Initialize) |

#### Returns

None.

#### Remarks

This is not needed for audio-only applications without a Bluetooth module.

**C**

```
void DRV_AK4954_EnableInitialization(SYS_MODULE_OBJ object);
```

### *DRV_AK4954_IsInitializationDelayed Function*

bool DRV_AK4954_IsInitializationDelayed(SYS_MODULE_OBJ object);

#### Summary

Checks if delayed initialization of the driver has been requested.

#### Description

If the AK4954 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. This function returns true if that option has been selected in MHC in the checkbox: "Delay driver initialization (due to shared RESET pin)"

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

### Parameters

| Parameters | Description |
|---|---|
| object | Object handle for the specified driver instance (returned from DRV_AK4954_Initialize) |

### Returns

true if the delayed initialization option has been enabled

### Remarks

This is not needed for audio-only applications without a Bluetooth module.

### C

```
bool DRV_AK4954_IsInitializationDelayed(SYS_MODULE_OBJ object);
```

## *DRV_AK4954_Deinitialize Function*

void DRV_AK4954_Deinitialize( SYS_MODULE_OBJ object)

### Summary

Deinitializes the specified instance of the AK4954 driver module.

### Description

Deinitializes the specified instance of the AK4954 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

### Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_AK4954_Initialize routine |

### Returns

None.

### Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

### Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4954_Initialize
SYS_STATUS          status;


DRV_AK4954_Deinitialize(object);

status = DRV_AK4954_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

### C

```
void DRV_AK4954_Deinitialize(SYS_MODULE_OBJ object);
```

## *DRV_AK4954_Status Function*

SYS_STATUS DRV_AK4954_Status( SYS_MODULE_OBJ object)

### Summary

Gets the current status of the AK4954 driver module.

### Description

This routine provides the current status of the AK4954 driver module.

### Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_AK4954_Initialize routine |

### Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

### Remarks

A driver can opened only when its status is SYS_STATUS_READY.

### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize
SYS_STATUS          AK4954Status;

AK4954Status = DRV_AK4954_Status(object);
if (SYS_STATUS_READY == AK4954Status)
{
    // This means the driver can be opened using the
    // DRV_AK4954_Open() function.
}
```

### C

```
SYS_STATUS DRV_AK4954_Status(SYS_MODULE_OBJ object);
```

## *DRV_AK4954_Tasks Function*

void  DRV_AK4954_Tasks(SYS_MODULE_OBJ object);

### Summary

Maintains the driver's control and data interface state machine.

### Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

### Parameters

| Parameters | Description |
|---|---|
| object | Object handle for the specified driver instance (returned from DRV_AK4954_Initialize) |

### Returns

None.

### Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize

while (true)
{
    DRV_AK4954_Tasks (object);

    // Do other tasks
}
```

### C

```
void DRV_AK4954_Tasks(SYS_MODULE_OBJ object);
```

## Client Setup Functions

### *DRV_AK4954_Open Function*

DRV_HANDLE DRV_AK4954_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

### Summary

Opens the specified AK4954 driver instance and returns a handle to it.

### Description

This routine opens the specified AK4954 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4954 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

### Preconditions

Function DRV_AK4954_Initialize must have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the object instance to be opened |
| ioIntent | Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details. |

### Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4954_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

### Remarks

The handle returned is valid until the DRV_AK4954_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

### Example

```
DRV_HANDLE handle;

handle = DRV_AK4954_Open(DRV_AK4954_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

### C

```
DRV_HANDLE DRV_AK4954_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

## *DRV_AK4954_Close Function*

void DRV_AK4954_Close( DRV_Handle handle )

### Summary

Closes an opened-instance of the AK4954 driver

### Description

This routine closes an opened-instance of the AK4954 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4954_Open before the caller may use the driver again

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

### Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4954_Open

DRV_AK4954_Close(handle);
```

### C

```
void DRV_AK4954_Close(const DRV_HANDLE handle);
```

## *DRV_AK4954_BufferEventHandlerSet Function*

void DRV_AK4954_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4954_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

### Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

### Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4954_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

### Returns

None.

### Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

### Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddRead(myAK4954handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_AK4954_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4954_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

## *DRV_AK4954_CommandEventHandlerSet Function*

void DRV_AK4954_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4954_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

### Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted

command have finished.

## Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4954_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4954 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

## Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

## Returns

None.

## Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_CommandEventHandlerSet(myAK4954Handle,
                APP_AK4954CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_DeEmphasisFilterSet(myAK4954Handle, DRV_AK4954_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4954CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

### C

```
void DRV_AK4954_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4954_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

## Data Transfer Functions

### *DRV_AK4954_BufferAddRead Function*

void DRV_AK4954_BufferAddRead

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

### Summary

Schedule a non-blocking driver read operation.

### Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4954_Open call.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the AK4954 instance as return by the DRV_AK4954_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not

otherwise be called directly in an ISR.

### C

```
void DRV_AK4954_BufferAddRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

## *DRV_AK4954_BufferAddWrite Function*

void DRV_AK4954_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

### Summary

Schedule a non-blocking driver write operation.

### Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4954_Open call.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the AK4954 instance as return by the DRV_AK4954_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

### Example

```
MY_APP_OBJ myAppObj;
```

```
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWrite(myAK4954handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

**C**

```
void DRV_AK4954_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

### *DRV_AK4954_BufferAddWriteRead Function*

```
void DRV_AK4954_BufferAddWriteRead
(
const DRV_HANDLE handle,
        DRV_AK4954_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)
```

### Summary

Schedule a non-blocking driver write-read operation.

## Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

## Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4954_Open call.

## Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the AK4954 instance as returned by the DRV_AK4954_Open function |
| bufferHandle | Pointer to an argument that will contain the return buffer handle |
| transmitBuffer | The buffer where the transmit data will be stored |
| receiveBuffer | The buffer where the received data will be stored |
| size | Buffer size in bytes |

## Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

## Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4954 write. The transmit and receive size must be same.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myak4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myak4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWriteRead(myak4954handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
```

```
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

**C**

```
void DRV_AK4954_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

### *DRV_AK4954_ReadQueuePurge Function*

bool DRV_AK4954_ReadQueuePurge( const DRV_HANDLE handle )

#### Summary

Removes all buffer requests from the read queue.

#### Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

#### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_AK4954_Open function. |

#### Returns

True - Read queue purge is successful. False - Read queue purge has failed.

#### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

#### Example

```
// myCodecHandle is the handle returned by the DRV_AK4954_Open function.
// Use DRV_AK4954_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
```

```
void APP_TimeOut(void)
{
    if(false == DRV_AK4954_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

**C**

```
bool DRV_AK4954_ReadQueuePurge(const DRV_HANDLE handle);
```

### *DRV_AK4954_WriteQueuePurge Function*

bool DRV_AK4954_WriteQueuePurge( const DRV_HANDLE handle )

#### Summary

Removes all buffer requests from the write queue.

#### Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

#### Preconditions

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_AK4954_Open function. |

#### Returns

True - Write queue purge is successful. False - Write queue purge has failed.

#### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

#### Example

```
// myCodecHandle is the handle returned by the DRV_AK4954_Open function.
// Use DRV_AK4954_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_AK4954_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

**C**

```
bool DRV_AK4954_WriteQueuePurge(const DRV_HANDLE handle);
```

## Settings Functions

### *DRV_AK4954_MicGainGet Function*

uint8_t DRV_AK4954_MicGainGet(DRV_HANDLE handle)

#### Summary

This function gets the microphone gain for the AK4954 Codec.

#### Description

This functions gets the current microphone gain programmed to the Codec AK4954.

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

#### Returns

Microphone gain, in range 0-31.

#### Remarks

None.

#### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t gain;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

  gain = DRV_AK4954_MicGainGet(myAK4954Handle);
```

#### C

```
uint8_t DRV_AK4954_MicGainGet(DRV_HANDLE handle);
```

### *DRV_AK4954_MicGainSet Function*

void DRV_AK4954_MicGainSet(DRV_HANDLE handle, uint8_t gain)

#### Summary

This function sets the microphone gain for the AK4954 CODEC.

#### Description

This functions sets the microphone gain value from 0-31 which can range from -1.5 to 28.3 dB

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| gain | Gain value, in range 0-31 |

### Returns

None.

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MicGainSet(myAK4954Handle, 15);  //AK4954 mic gain set to 15
```

### Remarks

None.

### C

```
void DRV_AK4954_MicGainSet(DRV_HANDLE handle, uint8_t gain);
```

## *DRV_AK4954_MicMuteOff Function*

void DRV_AK4954_MicMuteOff(DRV_HANDLE handle)

### Summary

Umutes th AK4954's microphone input.

### Description

This function unmutes the AK4954's microphone input.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MicMuteOff(myAK4954Handle);  //AK4954 microphone unmuted
```

### C

```
void DRV_AK4954_MicMuteOff(DRV_HANDLE handle);
```

### *DRV_AK4954_MicMuteOn Function*

void DRV_AK4954_MicMuteOn(DRV_HANDLE handle);

#### Summary

Mutes the AK4954's microphone input

#### Description

This function mutes the AK4954's microphone input

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

#### Returns

None.

#### Remarks

None.

#### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_MicMuteOn(myAK4954Handle);   //AK4954 microphone muted
```

#### C

```
void DRV_AK4954_MicMuteOn(DRV_HANDLE handle);
```

### *DRV_AK4954_MicSet Function*

void DRV_AK4954_IntMic12Set

#### Summary

This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.

#### Description

This function sets up the codec.

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| micInput | Internal vs External mic input |

### Returns

None

### Remarks

None.

### C

```
void DRV_AK4954_MicSet(DRV_HANDLE handle, DRV_AK4954_MIC micInput);
```

## *DRV_AK4954_MonoStereoMicSet Function*

void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle);

### Summary

This function sets up the codec for the Mono or Stereo microphone mode.

### Description

This function sets up the codec for the Mono or Stereo microphone mode.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None

### Remarks

None.

### C

```
void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4954_MONO_STEREO_MIC mono_stereo_mic);
```

## *DRV_AK4954_MuteOff Function*

void DRV_AK4954_MuteOff(DRV_HANDLE handle)

### Summary

This function disables AK4954 output for soft mute.

### Description

This function disables AK4954 output for soft mute.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MuteOff(myAK4954Handle); //AK4954 output soft mute disabled
```

### C

```
void DRV_AK4954_MuteOff(DRV_HANDLE handle);
```

## *DRV_AK4954_MuteOn Function*

void DRV_AK4954_MuteOn(DRV_HANDLE handle);

### Summary

This function allows AK4954 output for soft mute on.

### Description

This function Enables AK4954 output for soft mute.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_MuteOn(myAK4954Handle);  //AK4954 output soft muted
```

### C

```
void DRV_AK4954_MuteOn(DRV_HANDLE handle);
```

## *DRV_AK4954_IntExtMicSet Function*

void DRV_AK4954_IntExtMicSet

### Summary

This function sets up the codec for the X32 DB internal or the external microphone use.

### Description

This function sets up the codec for the internal or the external microphone use.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| micInput | Internal vs External mic input |

### Returns

None

### Remarks

None.

### C

```
void DRV_AK4954_IntExtMicSet(DRV_HANDLE handle, DRV_AK4954_INT_EXT_MIC micInput);
```

## *DRV_AK4954_SamplingRateGet Function*

uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle)

### Summary

This function gets the sampling rate set on the DAC AK4954.

### Description

This function gets the sampling rate set on the DAC AK4954.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
uint32_t baudRate;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

baudRate = DRV_AK4954_SamplingRateGet(myAK4954Handle);
```

### C

```
uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle);
```

## *DRV_AK4954_SamplingRateSet Function*

void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

### Summary

This function sets the sampling rate of the media stream.

### Description

This function sets the media sampling rate for the client handle.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_SamplingRateSet(myAK4954Handle, 48000);  //Sets 48000 media sampling rate
```

### C

```
void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

## *DRV_AK4954_VolumeGet Function*

uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan)

### Summary

This function gets the volume for AK4954 Codec.

### Description

This functions gets the current volume programmed to the CODEC AK4954.

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| chan | Audio channel volume to be set |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

  volume = DRV_AK4954_VolumeGet(myAK4954Handle,DRV_AK4954_CHANNEL_LEFT);
```

### C

```
uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan);
```

## *DRV_AK4954_VolumeSet Function*

void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);

### Summary

This function sets the volume for AK4954 Codec.

### Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| chan | Audio channel volume to be set |
| volume | volume value specified in the range 0-255 (0x00 to 0xFF) |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_VolumeSet(myAK4954Handle, DRV_AK4954_CHANNEL_LEFT, 120);
```

### C

```
void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);
```

## Other Functions

### *DRV_AK4954_LRCLK_Sync Function*

uint32_t DRV_AK4954_LRCLK_Sync (const DRV_HANDLE handle);

### Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

### Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

### Preconditions

None.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
    // myAK4954Handle is the handle returned
    // by the DRV_AK4954_Open function.

    DRV_AK4954_LRCLK_Sync(myAK4954Handle);
```

### C

```
bool DRV_AK4954_LRCLK_Sync(const DRV_HANDLE handle);
```

### *DRV_AK4954_GetI2SDriver Function*

DRV_HANDLE DRV_AK4954_GetI2SDriver(DRV_HANDLE codecHandle)

#### Summary

Get the handle to the I2S driver for this codec instance.

#### Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

#### Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

#### Returns

- A handle to the I2S driver for this codec instance

#### Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

#### C

```
DRV_HANDLE DRV_AK4954_GetI2SDriver(DRV_HANDLE codecHandle);
```

### *DRV_AK4954_VersionStrGet Function*

int8_t* DRV_AK4954_VersionStrGet(void)

#### Summary

This function returns the version of AK4954 driver in string format.

#### Description

The DRV_AK4954_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4954 driver's version number. is the AK4954 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

#### Preconditions

None.

#### Returns

returns a string containing the version of AK4954 driver.

#### Remarks

None.

#### Example 1

"0.03a" "1.00"

### Example 2

```
int8_t *AK4954string;
AK4954string = DRV_AK4954_VersionStrGet();
```

### C

```
int8_t* DRV_AK4954_VersionStrGet();
```

## *DRV_AK4954_VersionGet Function*

uint32_t DRV_AK4954_VersionGet( void )

### Summary

This function returns the version of AK4954 driver.

### Description

The version number returned from the DRV_AK4954_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

### Preconditions

None.

### Returns

returns the version of AK4954 driver.

### Remarks

None.

### Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

### Example 2

```
uint32_t AK4954version;
AK4954version = DRV_AK4954_VersionGet();
```

### C

```
uint32_t DRV_AK4954_VersionGet();
```

## Data Types and Constants

## *DRV_AK4954_AUDIO_DATA_FORMAT Type*

Identifies the Serial Audio data interface format.

### Description

AK4954 Audio data format

This enumeration identifies Serial Audio data interface format.

### C

```
typedef enum DRV_AK4954_AUDIO_DATA_FORMAT@1 DRV_AK4954_AUDIO_DATA_FORMAT;
```

### *DRV_AK4954_BUFFER_EVENT Type*

Identifies the possible events that can result from a buffer add request.

## Description

AK4954 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4954_BufferAddWrite() or the DRV_AK4954_BufferAddRead() function.

## Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4954_BufferEventHandlerSet function when a buffer transfer request is completed.

## C

```
typedef enum DRV_AK4954_BUFFER_EVENT@1 DRV_AK4954_BUFFER_EVENT;
```

### *DRV_AK4954_BUFFER_EVENT_HANDLER Type*

Pointer to a AK4954 Driver Buffer Event handler function

## Description

AK4954 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4954 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

## Parameters

| Parameters | Description |
|---|---|
| event | Identifies the type of event |
| bufferHandle | Handle identifying the buffer to which the event relates |
| context | Value identifying the context of the application that registered the event handling function. |

## Returns

None.

## Remarks

If the event is DRV_AK4954_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4954_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4954_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4954_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

### Example

```
void APP_MyBufferEventHandler( DRV_AK4954_BUFFER_EVENT event,
                               DRV_AK4954_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

### C

```
typedef void (* DRV_AK4954_BUFFER_EVENT_HANDLER)(DRV_AK4954_BUFFER_EVENT event,
DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

## *DRV_AK4954_BUFFER_HANDLE Type*

Handle identifying a write buffer passed to the driver.

### Description

AK4954 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4954_BufferAddWrite() or DRV_AK4954_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

### Remarks

None

### C

```
typedef uintptr_t DRV_AK4954_BUFFER_HANDLE;
```

## *DRV_AK4954_CHANNEL Type*

Identifies Left/Right Audio channel

### Description

AK4954 Audio Channel

This enumeration identifies Left/Right Audio channel

### Remarks

None.

### C

```
typedef enum DRV_AK4954_CHANNEL@1 DRV_AK4954_CHANNEL;
```

## *DRV_AK4954_COMMAND_EVENT_HANDLER Type*

Pointer to a AK4954 Driver Command Event Handler Function

### Description

AK4954 Driver Command Event Handler Function

This data type defines the required function signature for the AK4954 driver command event handling callback function.

A command is a control instruction to the AK4954 Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

### Parameters

| Parameters | Description |
|---|---|
| context | Value identifying the context of the application that registered the event handling function. |

### Returns

None.

### Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

### Example

```
void APP_AK4954CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

### C

```
typedef void (* DRV_AK4954_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

## *DRV_AK4954_DIGITAL_BLOCK_CONTROL Type*

Identifies Bass-Boost Control function

### Description

AK4954 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

### Remarks

None.

**C**

```
typedef enum DRV_AK4954_DIGITAL_BLOCK_CONTROL@1 DRV_AK4954_DIGITAL_BLOCK_CONTROL;
```

### *DRV_AK4954_INIT Type*

Defines the data required to initialize or reinitialize the AK4954 driver

## Description

AK4954 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4954 Codec driver.

## Remarks

None.

**C**

```
typedef struct DRV_AK4954_INIT@1 DRV_AK4954_INIT;
```

### *DRV_AK4954_INT_EXT_MIC Type*

Identifies the Mic input source.

## Description

AK4954 Mic Internal / External Input

This enumeration identifies the Mic input source.

**C**

```
typedef enum DRV_AK4954_INT_EXT_MIC@1 DRV_AK4954_INT_EXT_MIC;
```

### *DRV_AK4954_MIC Type*

This is type DRV_AK4954_MIC.

**C**

```
typedef enum DRV_AK4954_MIC@1 DRV_AK4954_MIC;
```

### *DRV_AK4954_MONO_STEREO_MIC Type*

Identifies the Mic input as Mono / Stereo.

## Description

AK4954 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

**C**

```
typedef enum DRV_AK4954_MONO_STEREO_MIC@1 DRV_AK4954_MONO_STEREO_MIC;
```

### *SAMPLE_LENGTH Type*

in bits

**C**

```
typedef enum SAMPLE_LENGTH@1 SAMPLE_LENGTH;
```

### *DRV_AK4954_AUDIO_DATA_FORMAT_I2S Macro*

for compatability with old code

**C**

```
#define DRV_AK4954_AUDIO_DATA_FORMAT_I2S DRV_AK4954_AUDIO_DATA_FORMAT_I2S_16BIT_24BIT  // for compatability with old code
```

### *DRV_AK4954_BUFFER_HANDLE_INVALID Macro*

Definition of an invalid buffer handle.

## Description

AK4954 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4954_BufferAddWrite() and the DRV_AK4954_BufferAddRead() function if the buffer add request was not successful.

## Remarks

None

**C**

```
#define DRV_AK4954_BUFFER_HANDLE_INVALID ((DRV_AK4954_BUFFER_HANDLE)(-1))
```

### *DRV_AK4954_COUNT Macro*

Number of valid AK4954 driver indices

## Description

AK4954 Driver Module Count

This constant identifies the maximum number of AK4954 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4954 instances on this microcontroller.

## Remarks

This value is part-specific.

**C**

```
#define DRV_AK4954_COUNT
```

### *DRV_AK4954_INDEX_0 Macro*

AK4954 driver index definitions

## Description

Driver AK4954 Module Index

These constants provide AK4954 driver index definition.

## Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the

DRV_AK4954_Initialize and DRV_AK4954_Open routines to identify the driver instance in use.

**C**

```
#define DRV_AK4954_INDEX_0 0
```

### *DRV_AK4954_INDEX_1 Macro*

This is macro DRV_AK4954_INDEX_1.

**C**

```
#define DRV_AK4954_INDEX_1 1
```

### *DRV_AK4954_INDEX_2 Macro*

This is macro DRV_AK4954_INDEX_2.

**C**

```
#define DRV_AK4954_INDEX_2 2
```

### *DRV_AK4954_INDEX_3 Macro*

This is macro DRV_AK4954_INDEX_3.

**C**

```
#define DRV_AK4954_INDEX_3 3
```

### *DRV_AK4954_INDEX_4 Macro*

This is macro DRV_AK4954_INDEX_4.

**C**

```
#define DRV_AK4954_INDEX_4 4
```

### *DRV_AK4954_INDEX_5 Macro*

This is macro DRV_AK4954_INDEX_5.

**C**

```
#define DRV_AK4954_INDEX_5 5
```

### *Files*

## Files

| Name | Description |
|---|---|
| drv_ak4954.h | AK4954 Codec Driver Interface header file |
| drv_ak4954_config_template.h | AK4954 Codec Driver Configuration Template. |

## Description

### drv_ak4954.h

drv_ak4954.h

## Summary

AK4954 Codec Driver Interface header file

## Description

AK4954 Codec Driver Interface

The AK4954 Codec device driver interface provides a simple interface to manage the AK4954 16/24/32-Bit Codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the AK4954 Codec device driver.

### drv_ak4954_config_template.h

drv_ak4954_config_template.h

## Summary

AK4954 Codec Driver Configuration Template.

## Description

AK4954 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

# GENERIC AUDIO DRIVER CODEC Library

This topic describes the Generic Codec Driver Library.

### *Introduction*

#### Summary

This library provides an Applications Programming Interface (API) to manage the Generic Codec that is serially interfaced to the $I^2C$ and $I^2S$ peripherals of a Microchip microcontroller for the purpose of providing audio solutions.

## Description

#### Description

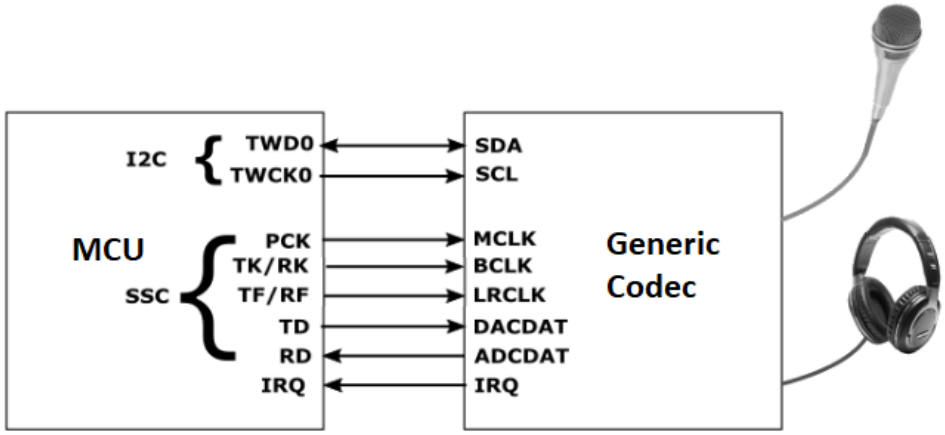
This file contains the implementation of the Generic Codec driver, which provides a simple interface to manage a codec that can be interfaced to Microchip microcontroller. The user will need to modify it to match the requirements of their codec. Areas where code needs to be added or changed are marked with TO-DO!!

Note: this module assumes the codec is controlled over an $I^2$2C interface. The $I^2C$ Driver will need to be enabled in the MHC Project Graph. If another type of interface is used, the user will need to modify the code to suit. This module makes use of SYS_TIME. It will need to be enabled in the Project Graph.

This module assumes the an $I^2S$ interface is used for audio data output (to headphones or line-out) and input (from microphone or line-in).

The Generic Codec can be configured as either an $I^2S$ clock slave (receives all clocks from the host), or $I^2S$ clock master (generates $I^2S$ clocks from a master clock input MCLK).

A typical interface of Generic Codec to a Microchip microcontroller using an $I^2C$ and SSC interface (configured as $I^2S$), with the Generic Codec set up as the $I^2S$ clock slave, is provided in the following diagram:




## *Using the Library*

This topic describes the basic architecture of the Generic Codec Driver Library and provides information and examples on its use.

## Description

**Interface Header File**: `drv_Generic.h`

The interface to the Generic Codec Driver library is defined in the `audio/driver/codec/Generic/drv_Generic.h` header file. Any C language source (.c) file that uses the Generic Codec Driver library should include this header.

### Library Source Files:

The Generic Codec Driver library source files are provided in the `audio/driver/codec/Generic/src` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features and to **Building the Library** for instructions on how to build the library.

**Example Applications:**

This codec is not used directly by any demonstration applications. However the following applications could be looked at, to see how a codec such as the WM8904 or AK4954 is used:

- `audio/apps/audio_tone`
- `audio/apps/microphone_loopback`

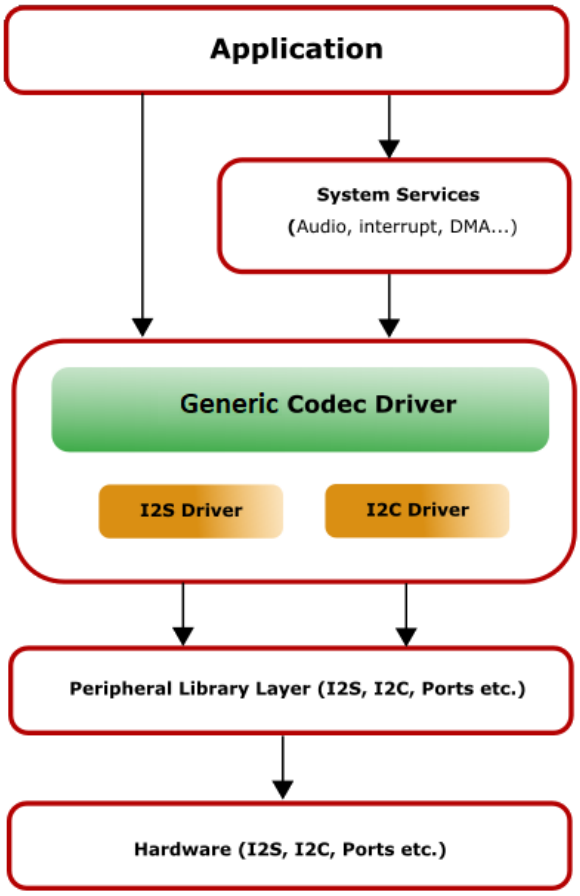
## Abstraction Model

This library provides a low-level abstraction of the Generic Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

## Description

The abstraction model shown in the following diagram depicts how the Generic Codec Driver is positioned in the MPLAB Harmony framework. The Generic Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the Generic module.

**Generic Driver Abstraction Model**




## Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The Generic Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced Generic Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Generic Codec Driver Library.

| Library Interface Section | Description |
|---|---|
| System Functions | Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions. |
| Client Setup Functions | Provides open and close functions. |
| Data Transfer Functions | Provides data transfer functions, such as Buffer Read and Write. |
| Settings Functions | Provides driver specific functions for settings, such as volume control and sampling rate. |
| Other Functions | Miscellaneous functions, such as getting the driver's version number and syncing to the LRCLK signal. |
| Data Types and Constants | These data types and constants are required while interacting and setting up the Generic Codec Driver Library. |

**Note:** All functions and constants in this section are named with the format DRV_ Generic_xxx, where 'xxx' is a function name or constant. These names are redefined in the appropriate configuration's configuration.h file to the format DRV_CODEC_xxx using #defines so that code in the application that references the library can be written as generically as possible (e.g., by writing DRV_CODEC_Open instead of DRV_ Generic_Open etc.). This allows the codec type to be changed in the MHC without having to modify the application's source code.

## How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

## *Setup (Initialization)*

This topic describes system initialization, implementations, and includes a system access code example.

### Description

#### System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization in the system_init.c file, each instance of the Generic module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_Generic_INIT or by using Initialization Overrides) that are supported by the specific Generic device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

The DRV_Generic_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ Generic_Deinitialize, DRV_ Generic_Status and DRV_I2S_Tasks.

## *Client Access*

This topic describes driver initialization and provides a code example.

### Description

For the application to start using an instance of the module, it must call the DRV_Generic_Open function. The DRV_Generic_Open function provides a driver handle to the Generic Codec Driver instance for operations. If the driver is deinitialized using the function DRV_Generic_Deinitialize, the application must call the DRV_Generic_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to Data Types and Constants in the Library Interface section.

**Note:** It is necessary to check the status of driver initialization before opening a driver instance. The status of the Generic Codec Driver can be known by calling DRV_ Generic_Status.

#### Example:

```
DRV_HANDLE handle;
SYS_STATUS genericStatus;
genericStatus Status = DRV_Generic_Status(sysObjects.genericStatus DevObject);
if (SYS_STATUS_READY == genericStatus)
{
// The driver can now be opened.
appData.genericClient.handle = DRV_Generic_Open
```

```
(DRV_Generic_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.genericClient.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_Generic_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* Generic Driver Is not ready */
}
```

### *Client Operations*

This topic provides information on client operations.

## Description

Client operations provide the API interface for control command and audio data transfer to the Generic Codec.

The following Generic Codec specific control command functions are provided:

- DRV_Generic_SamplingRateSet
- DRV_Generic_SamplingRateGet
- DRV_Generic_VolumeSet
- DRV_Generic_VolumeGet
- DRV_Generic_MuteOn
- DRV_Generic_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the Generic Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_Generic_BufferAddWrite, DRV_Generic_BufferAddRead, and DRV_Generic_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_Generic_BUFFER_EVENT_COMPLETE, DRV_Generic_BUFFER_EVENT_ERROR, or DRV_Generic_BUFFER_EVENT_ABORT events.

**Note:** It is not necessary to close and reopen the client between multiple transfers.

### *Configuring the Library*

The configuration of the I2S Driver Library is based on the file configurations.h, as generated by the MHC.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

## System Configuration

### Macros

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO | Specifies the audio data format for the codec. |
| | DRV_GENERICCODEC_AUDIO_SAMPLING_RATE | Specifies the initial baud rate for the codec. |
| | DRV_GENERICCODEC_CLIENTS_NUMBER | Sets up the maximum number of clients that can be connected to any hardware instance. |
| | DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2C interface. |
| | DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx | Specifies the instance number of the I2S interface. |
| | DRV_GENERICCODEC_INSTANCES_NUMBER | Sets up the maximum number of hardware instances that can be supported |
| | DRV_GENERICCODEC_VOLUME | Specifies the initial volume level. |

### Description

### *DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO Macro*

Specifies the audio data format for the codec.

### Description

Generic Codec Audio Data Format

Sets up the length of each sample plus the format (I2S or left-justified) for the audio.

Valid choices are: "DATA_16_BIT_LEFT_JUSTIFIED", 16-bit Left Justified "DATA_16_BIT_I2S", 16-bit I2S" "DATA_32_BIT_LEFT_JUSTIFIED", 32-bit Left Justified "DATA_32_BIT_I2S", 32-bit I2S"

### C

```
#define DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO
```

### *DRV_GENERICCODEC_AUDIO_SAMPLING_RATE Macro*

Specifies the initial baud rate for the codec.

### Description

Generic Codec Baud Rate

Sets the initial baud rate (sampling rate) for the codec. Typical values are 8000, 16000, 44100, 48000, 88200 and 96000.

### C

```
#define DRV_GENERICCODEC_AUDIO_SAMPLING_RATE
```

### *DRV_GENERICCODEC_CLIENTS_NUMBER Macro*

Sets up the maximum number of clients that can be connected to any hardware instance.

### Description

Generic Codec Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances.

**C**

```
#define DRV_GENERICCODEC_CLIENTS_NUMBER
```

### *DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2C interface.

## Description

Generic Codec I2C instance number

Specifies the instance number of the I2C interface being used by the MCU to send commands and receive status to and from the Generic Codec. enabled.

**C**

```
#define DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx Macro*

Specifies the instance number of the I2S interface.

## Description

Generic Codec I2S instance number

Specifies the instance number of the I2S interface being used by the MCU to send and receive audio data to and from the Generic Codec. enabled.

**C**

```
#define DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx
```

### *DRV_GENERICCODEC_INSTANCES_NUMBER Macro*

Sets up the maximum number of hardware instances that can be supported

## Description

Generic Codec driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of Generic Codec modules that are needed by an application, namely one.

**C**

```
#define DRV_GENERICCODEC_INSTANCES_NUMBER
```

### *DRV_GENERICCODEC_VOLUME Macro*

Specifies the initial volume level.

## Description

Generic Codec Volume

Sets the initial volume level, in the range 0-255.

## Remarks

The value is mapped to an internal Generic Codec volume level in the range 0-192 using a logarithmic table so the input scale appears linear (128 is half volume).

## C

```
#define DRV_GENERICCODEC_VOLUME
```

### *Confuguring MHC*

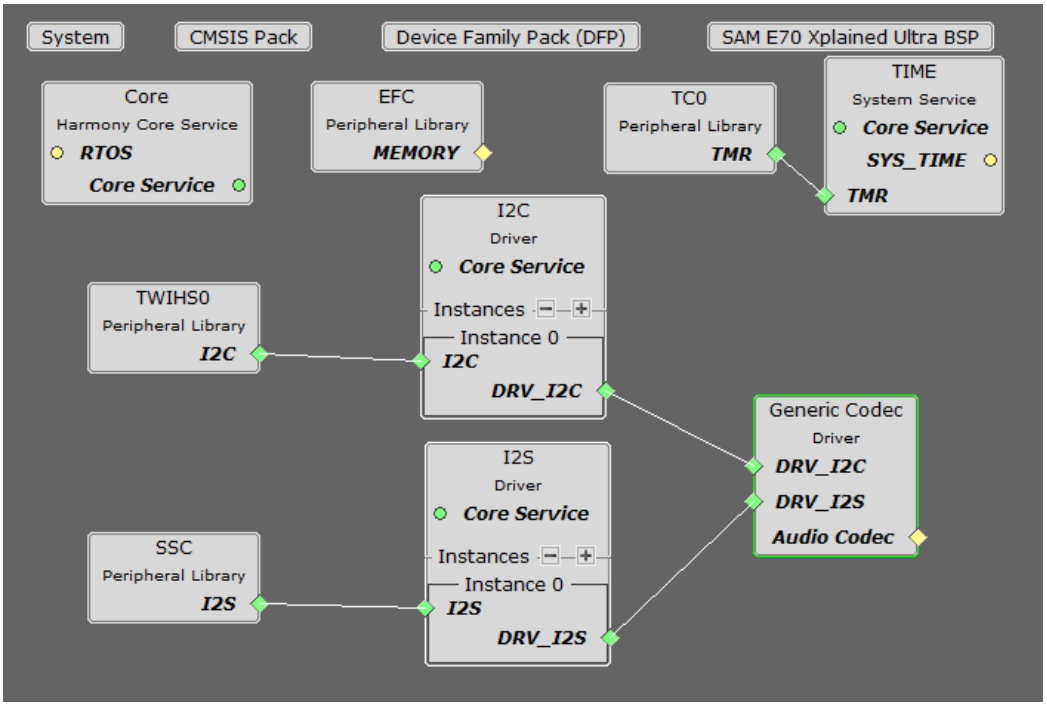
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

## Description

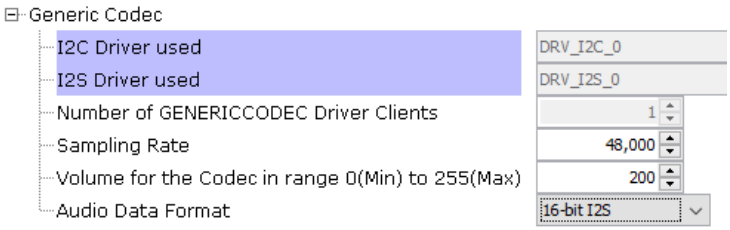
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In the MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on a codec template such as Generic. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




Click on the Generic Driver component (not Generic Codec) and the following menu will be displayed in the Configurations Options:




- **I2C Driver Used** will display the driver instance used for the $I^2C$ interface.
- **I2S Driver Used** will display the driver instance used for the $I^2S$ interface.
- **Number of Generic Clients** indicates the maximum number of clients that can be connected to the Generic Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either
  - 24-bit Left Justified (ADC), 24-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 16-bit Right-justified(DAC)

- 24-bit Left Justified (ADC), 24-bit Left-justified(DAC)
- 24/16-bit I2S
- 32-bit Left Justified (ADC), 32-bit Left-justified(DAC)
- 32-bit I2S

It must match the audio protocol and data length set up in either the SSC or I2S PLIB..

You can also bring in the Generic Driver by itself, by double clicking Generic under Audio->*Driver*->*Codec* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

Note that the Generic requires the TCx Peripheral Library and TIME System Service in order to perform some of its internal timing sequences.

## *Building the Library*

This section lists the files that are available in the Generic Codec Driver Library.

### Description

This section lists the files that are available in the `src` folder of the Generic Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `audio/driver/codec/Generic.`

#### Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

| Source File Name | Description |
|---|---|
| `drv_genericcodec.h` | Header file that exports the driver API. |

#### Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

| Source File Name | Description |
|---|---|
| `/src/drv_genericcodec.c` | This file contains implementation of the Generic Codec Driver. |

#### Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

| Source File Name | Description |
|---|---|
| `N/A` | No optional files are available for this library. |

#### Module Dependencies

The Generic Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

***Library Interface***

## Client Setup Functions

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_Open | Opens the specified Generic Codec driver instance and returns a handle to it |
| | DRV_GENERICCODEC_Close | Closes an opened-instance of the Generic Codec driver |
| | DRV_GENERICCODEC_BufferEventHandlerSet | This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. |
| | DRV_GENERICCODEC_CommandEventHandlerSet | This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. |

## Dara Types and Constants

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_BUFFER_EVENT | Identifies the possible events that can result from a buffer add request. |
| | DRV_GENERICCODEC_BUFFER_EVENT_HANDLER | Pointer to a Generic Codec Driver Buffer Event handler function |
| | DRV_GENERICCODEC_BUFFER_HANDLE | Handle identifying a write buffer passed to the driver. |
| | DRV_GENERICCODEC_CHANNEL | Identifies Left/Right Audio channel |
| | DRV_GENERICCODEC_COMMAND_EVENT_HANDLER | Pointer to a Generic Codec Driver Command Event Handler Function |
| | DRV_GENERICCODEC_INIT | Defines the data required to initialize or reinitialize the Generic Codec driver |
| | DRV_GENERICCODEC_BUFFER_HANDLE_INVALID | Definition of an invalid buffer handle. |
| | DRV_GENERICCODEC_COUNT | Number of valid Generic Codec driver indices |
| | DRV_GENERICCODEC_INDEX_0 | Generic Codec driver index definitions |
| | DRV_GENERICCODEC_INDEX_1 | This is macro DRV_GENERICCODEC_INDEX_1. |
| | DRV_GENERICCODEC_INDEX_2 | This is macro DRV_GENERICCODEC_INDEX_2. |
| | DRV_GENERICCODEC_INDEX_3 | This is macro DRV_GENERICCODEC_INDEX_3. |
| | DRV_GENERICCODEC_INDEX_4 | This is macro DRV_GENERICCODEC_INDEX_4. |
| | DRV_GENERICCODEC_INDEX_5 | This is macro DRV_GENERICCODEC_INDEX_5. |
| | DRV_GENERICCODEC_AUDIO_DATA_FORMAT | Identifies the Serial Audio data interface format. |

## Data Transfer Functions

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_BufferAddRead | Schedule a non-blocking driver read operation. |
| | DRV_GENERICCODEC_BufferAddWrite | Schedule a non-blocking driver write operation. |
| | DRV_GENERICCODEC_BufferAddWriteRead | Schedule a non-blocking driver write-read operation. |
| | DRV_GENERICCODEC_ReadQueuePurge | Removes all buffer requests from the read queue. |
| | DRV_GENERICCODEC_WriteQueuePurge | Removes all buffer requests from the write queue. |

## Other Functions

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_GetI2SDriver | Get the handle to the I2S driver for this codec instance. |
| | DRV_GENERICCODEC_LRCLK_Sync | Synchronize to the start of the I2S LRCLK (left/right clock) signal |

## Settings Functions

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_MicGainGet | This function gets the microphone gain for the Generic Codec. |
| | DRV_GENERICCODEC_MicGainSet | This function sets the microphone gain for the Generic Codec CODEC. |
| | DRV_GENERICCODEC_MicMuteOff | Umutes th Generic Codec's microphone input. |
| | DRV_GENERICCODEC_MicMuteOn | Mutes the Generic Codec's microphone input |
| | DRV_GENERICCODEC_MuteOff | This function disables Generic Codec output for soft mute. |
| | DRV_GENERICCODEC_MuteOn | This function allows Generic Codec output for soft mute on. |
| | DRV_GENERICCODEC_SamplingRateGet | This function gets the sampling rate set on the Generic Codec. |
| | DRV_GENERICCODEC_SamplingRateSet | This function sets the sampling rate of the media stream. |
| | DRV_GENERICCODEC_VersionGet | This function returns the version of Generic Codec driver |
| | DRV_GENERICCODEC_VersionStrGet | This function returns the version of Generic Codec driver in string format. |
| | DRV_GENERICCODEC_VolumeGet | This function gets the volume for Generic Codec. |
| | DRV_GENERICCODEC_VolumeSet | This function sets the volume for Generic Codec. |

## System Interaction Functions

| | Name | Description |
|---|---|---|
| | DRV_GENERICCODEC_Initialize | Initializes hardware and data for the instance of the Generic Codec module |
| | DRV_GENERICCODEC_Deinitialize | Deinitializes the specified instance of the Generic Codec driver module |
| | DRV_GENERICCODEC_Status | Gets the current status of the Generic Codec driver module. |
| | DRV_GENERICCODEC_Tasks | Maintains the driver's control and data interface state machine. |

## Description

### System Interaction Functions

### *DRV_GENERICCODEC_Initialize Function*

```
SYS_MODULE_OBJ  DRV_GENERICCODEC_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);
```

## Summary

Initializes hardware and data for the instance of the Generic Codec module

## Description

This routine initializes the Generic Codec driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

## Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this Codec driver.
DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this Codec driver.

### Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the driver instance to be initialized |
| init | Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used. |

### Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

### Remarks

This routine must be called before any other Generic Codec routine is called.

This routine should only be called once during system initialization unless DRV_GENERICCODEC_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

### Example

```
DRV_GENERICCODEC_INIT              init;
SYS_MODULE_OBJ                 objectHandle;

init->inUse                          = true;
init->status                         = SYS_STATUS_BUSY;
init->numClients                     = 0;
init->i2sDriverModuleIndex           = genercicodecInit->i2sDriverModuleIndex;
init->i2cDriverModuleIndex           = genercicodecInit->i2cDriverModuleIndex;
init->samplingRate                   = DRV_GENERICCODEC_AUDIO_SAMPLING_RATE;
init->audioDataFormat                = DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext           = false;

init->commandCompleteCallback = (DRV_GENERICCODEC_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_GENERICCODEC_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_GENERICCODEC_Initialize(DRV_GENERICCODEC_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

### C

```
SYS_MODULE_OBJ DRV_GENERICCODEC_Initialize(const SYS_MODULE_INDEX drvIndex, const
SYS_MODULE_INIT * const init);
```

## *DRV_GENERICCODEC_Deinitialize Function*

void DRV_GENERICCODEC_Deinitialize( SYS_MODULE_OBJ object)

### Summary

Deinitializes the specified instance of the Generic Codec driver module

### Description

Deinitializes the specified instance of the Generic Codec driver module, disabling its operation (and any hardware). Invalidates all the internal data.

### Preconditions

Function DRV_GENERICCODEC_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_GENERICCODEC_Initialize routine |

### Returns

None.

### Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

### Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_GENERICCODEC_Initialize
SYS_STATUS          status;


DRV_GENERICCODEC_Deinitialize(object);

status = DRV_GENERICCODEC_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

### C

```
void DRV_GENERICCODEC_Deinitialize(SYS_MODULE_OBJ object);
```

## *DRV_GENERICCODEC_Status Function*

SYS_STATUS DRV_GENERICCODEC_Status( SYS_MODULE_OBJ object)

### Summary

Gets the current status of the Generic Codec driver module.

### Description

This routine provides the current status of the Generic Codec driver module.

### Preconditions

Function DRV_GENERICCODEC_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_GENERICCODEC_Initialize routine |

### Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

### Remarks

A driver can opened only when its status is SYS_STATUS_READY.

### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_GENERICCODEC_Initialize
```

```
SYS_STATUS          GENERICCODECStatus;

GENERICCODECStatus = DRV_GENERICCODEC_Status(object);
if (SYS_STATUS_READY == GENERICCODECStatus)
{
    // This means the driver can be opened using the
    // DRV_GENERICCODEC_Open() function.
}
```

**C**

```
SYS_STATUS DRV_GENERICCODEC_Status(SYS_MODULE_OBJ object);
```

### *DRV_GENERICCODEC_Tasks Function*

void DRV_GENERICCODEC_Tasks(SYS_MODULE_OBJ object);

#### Summary

Maintains the driver's control and data interface state machine.

#### Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

#### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

#### Parameters

| Parameters | Description |
|---|---|
| object | Object handle for the specified driver instance (returned from DRV_GENERICCODEC_Initialize) |

#### Returns

None.

#### Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

#### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_GENERICCODEC_Initialize

while (true)
{
    DRV_GENERICCODEC_Tasks (object);

    // Do other tasks
}
```

**C**

```
void DRV_GENERICCODEC_Tasks(SYS_MODULE_OBJ object);
```

## Client Setup Functions

### *DRV_GENERICCODEC_Open Function*

DRV_HANDLE DRV_GENERICCODEC_Open

(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
)

## Summary

Opens the specified Generic Codec driver instance and returns a handle to it

## Description

This routine opens the specified Generic Codec driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

Generic Codec can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

## Preconditions

Function DRV_GENERICCODEC_Initialize must have been called before calling this function.

## Parameters

| Parameters | Description |
|---|---|
| drvIndex | Identifier for the object instance to be opened |
| ioIntent | Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details. |

## Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_GENERICCODEC_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

## Remarks

The handle returned is valid until the DRV_GENERICCODEC_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

## Example

```
DRV_HANDLE handle;

handle = DRV_GENERICCODEC_Open(DRV_GENERICCODEC_INDEX_0, DRV_IO_INTENT_WRITEREAD |
DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

## C

```
DRV_HANDLE DRV_GENERICCODEC_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

### *DRV_GENERICCODEC_Close Function*

void DRV_GENERICCODEC_Close( DRV_Handle handle )

## Summary

Closes an opened-instance of the Generic Codec driver

## Description

This routine closes an opened-instance of the Generic Codec driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_GENERICCODEC_Open before the caller may use the driver again

## Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

## Returns

- None

## Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

## Example

```
DRV_HANDLE handle;  // Returned from DRV_GENERICCODEC_Open

DRV_GENERICCODEC_Close(handle);
```

## C

```
void DRV_GENERICCODEC_Close(const DRV_HANDLE handle);
```

### *DRV_GENERICCODEC_BufferEventHandlerSet Function*

void DRV_GENERICCODEC_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_GENERICCODEC_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

## Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

## Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_GENERICCODEC_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

## Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

## Returns

None.

## Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWrite(myGENERICCODEChandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;
```

```
            default:
                break;
        }
    }
```

**C**

```
void DRV_GENERICCODEC_BufferEventHandlerSet(DRV_HANDLE handle, const
DRV_GENERICCODEC_BUFFER_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

### *DRV_GENERICCODEC_CommandEventHandlerSet Function*

void DRV_GENERICCODEC_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_GENERICCODEC_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

## Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

## Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

The event handler should be set before the client performs any "Generic Codec Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

## Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

## Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

## Returns

None.

## Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.
```

```
// Client registers an event handler with driver

DRV_GENERICCODEC_CommandEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECCommandEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_DeEmphasisFilterSet(myGENERICCODECHandle,
DRV_GENERICCODEC_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECCommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

**C**

```
void DRV_GENERICCODEC_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_GENERICCODEC_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

## Data Transfer Functions

### *DRV_GENERICCODEC_BufferAddRead Function*

void DRV_GENERICCODEC_BufferAddRead

(

const DRV_HANDLE handle,

DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

### Summary

Schedule a non-blocking driver read operation.

### Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_GENERICCODEC_Open call.

## Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the Generic Codec instance as return by the DRV_GENERICCODEC_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

## Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

## Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

## C

```
void DRV_GENERICCODEC_BufferAddRead(const DRV_HANDLE handle, DRV_GENERICCODEC_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

### *DRV_GENERICCODEC_BufferAddWrite Function*

void DRV_GENERICCODEC_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

## Summary

Schedule a non-blocking driver write operation.

## Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

## Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_GENERICCODEC_Open call.

## Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the Generic Codec instance as return by the DRV_GENERICCODEC_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

## Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

## Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

## Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWrite(myGENERICCODEChandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_GENERICCODEC_BufferAddWrite(const DRV_HANDLE handle, DRV_GENERICCODEC_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

## *DRV_GENERICCODEC_BufferAddWriteRead Function*

void DRV_GENERICCODEC_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

### Summary

Schedule a non-blocking driver write-read operation.

### Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_GENERICCODEC_Open call.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the Generic Codec instance as returned by the DRV_GENERICCODEC_Open function |
| bufferHandle | Pointer to an argument that will contain the return buffer handle |
| transmitBuffer | The buffer where the transmit data will be stored |
| receiveBuffer | The buffer where the received data will be stored |
| size | Buffer size in bytes |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that

is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every Generic Codec write. The transmit and receive size must be same.

### Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// mygenericcodecHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(mygenericcodecHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWriteRead(mygenericcodecHandle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_GENERICCODEC_BufferAddWriteRead(const DRV_HANDLE handle,
DRV_GENERICCODEC_BUFFER_HANDLE * bufferHandle, void * transmitBuffer, void * receiveBuffer,
size_t size);
```

## *DRV_GENERICCODEC_ReadQueuePurge Function*

bool DRV_GENERICCODEC_ReadQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the read queue.

### Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_GENERICCODEC_Open function. |

### Returns

True - Read queue purge is successful. False - Read queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myCodecHandle is the handle returned by the DRV_GENERICCODEC_Open function.
// Use DRV_GENERICCODEC_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_GENERICCODEC_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_GENERICCODEC_ReadQueuePurge(const DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_WriteQueuePurge Function*

bool DRV_GENERICCODEC_WriteQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the write queue.

### Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_GENERICCODEC_Open function. |

### Returns

True - Write queue purge is successful. False - Write queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myCodecHandle is the handle returned by the DRV_GENERICCODEC_Open function.
// Use DRV_GENERICCODEC_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_GENERICCODEC_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_GENERICCODEC_WriteQueuePurge(const DRV_HANDLE handle);
```

## Settings Functions

### *DRV_GENERICCODEC_MicGainGet Function*

uint8_t DRV_GENERICCODEC_MicGainGet(DRV_HANDLE handle)

### Summary

This function gets the microphone gain for the Generic Codec.

### Description

This functions gets the current microphone gain programmed to the Generic Codec.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

Microphone gain, in range 0-31.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
```

```
uint8_t gain;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

  gain = DRV_GENERICCODEC_MicGainGet(myGENERICCODECHandle);
```

**C**

```
uint8_t DRV_GENERICCODEC_MicGainGet(DRV_HANDLE handle);
```

### *DRV_GENERICCODEC_MicGainSet Function*

void DRV_GENERICCODEC_MicGainSet(DRV_HANDLE handle, uint8_t gain)

#### Summary

This function sets the microphone gain for the Generic Codec CODEC.

#### Description

This functions sets the microphone gain value from 0-31

#### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| gain | Gain value, in range 0-31 |

#### Returns

None.

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MicGainSet(myGENERICCODECHandle, 15);  //GENERICCODEC mic gain set to 15
```

#### Remarks

None.

**C**

```
void DRV_GENERICCODEC_MicGainSet(DRV_HANDLE handle, uint8_t gain);
```

### *DRV_GENERICCODEC_MicMuteOff Function*

void DRV_GENERICCODEC_MicMuteOff(DRV_HANDLE handle)

#### Summary

Umutes th Generic Codec's microphone input.

#### Description

This function unmutes the Generic Codec's microphone input.

#### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MicMuteOff(myGENERICCODECHandle);  //Generic Codec microphone unmuted
```

### C

```
void DRV_GENERICCODEC_MicMuteOff(DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_MicMuteOn Function*

void DRV_GENERICCODEC_MicMuteOn(DRV_HANDLE handle);

### Summary

Mutes the Generic Codec's microphone input

### Description

This function mutes the Generic Codec's microphone input

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_MicMuteOn(myGENERICCODECHandle);   //Generic Codec microphone muted
```

### C

```
void DRV_GENERICCODEC_MicMuteOn(DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_MuteOff Function*

void DRV_GENERICCODEC_MuteOff(DRV_HANDLE handle)

### Summary

This function disables Generic Codec output for soft mute.

### Description

This function disables Generic Codec output for soft mute.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MuteOff(myGENERICCODECHandle); //Generic Codec output soft mute disabled
```

### C

```
void DRV_GENERICCODEC_MuteOff(DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_MuteOn Function*

void DRV_GENERICCODEC_MuteOn(DRV_HANDLE handle);

### Summary

This function allows Generic Codec output for soft mute on.

### Description

This function Enables Generic Codec output for soft mute.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_MuteOn(myGENERICCODECHandle);  //GENERICCODEC output soft muted
```

### C

```
void DRV_GENERICCODEC_MuteOn(DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_SamplingRateGet Function*

uint32_t DRV_GENERICCODEC_SamplingRateGet(DRV_HANDLE handle)

### Summary

This function gets the sampling rate set on the Generic Codec.

### Description

This function gets the sampling rate set on the DAC Generic Codec.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Remarks

None.

### Example

```
uint32_t baudRate;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

baudRate = DRV_GENERICCODEC_SamplingRateGet(myGENERICCODECHandle);
```

### C

```
uint32_t DRV_GENERICCODEC_SamplingRateGet(DRV_HANDLE handle);
```

## *DRV_GENERICCODEC_SamplingRateSet Function*

void DRV_GENERICCODEC_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

### Summary

This function sets the sampling rate of the media stream.

### Description

This function sets the media sampling rate for the client handle.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| samplingRate | Sampling frequency in Hz |

### Returns

None.

### Remarks

None.

### Example

```
// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_SamplingRateSet(myGENERICCODECHandle, 48000);  //Sets 48000 media sampling rate
```

### C

```
void DRV_GENERICCODEC_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

## *DRV_GENERICCODEC_VersionGet Function*

uint32_t DRV_GENERICCODEC_VersionGet( void )

### Summary

This function returns the version of Generic Codec driver

### Description

The version number returned from the DRV_GENERICCODEC_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 +  Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

### Preconditions

None.

### Returns

returns the version of Generic Codec driver.

### Remarks

None.

### Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

### Example 2

```
uint32_t GENERICCODECversion;
```

```
GENERICCODECversion = DRV_GENERICCODEC_VersionGet();
```

**C**

```
uint32_t DRV_GENERICCODEC_VersionGet();
```

### *DRV_GENERICCODEC_VersionStrGet Function*

int8_t* DRV_GENERICCODEC_VersionStrGet(void)

#### Summary

This function returns the version of Generic Codec driver in string format.

#### Description

The DRV_GENERICCODEC_VersionStrGet function returns a string in the format: ".[.][]" Where: is the Generic Codec driver's version number. is the Generic Codec driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

#### Preconditions

None.

#### Returns

returns a string containing the version of Generic Codec driver.

#### Remarks

None

#### Example

```
int8_t *GENERICCODECstring;
GENERICCODECstring = DRV_GENERICCODEC_VersionStrGet();
```

**C**

```
int8_t* DRV_GENERICCODEC_VersionStrGet();
```

### *DRV_GENERICCODEC_VolumeGet Function*

uint8_t DRV_GENERICCODEC_VolumeGet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel)

#### Summary

This function gets the volume for Generic Codec.

#### Description

This functions gets the current volume programmed to the Generic Codec.

#### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| channel | argument indicating Left or Right or Both channel volume to be modified |

### Returns

None.

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

  volume = DRV_GENERICCODEC_VolumeGet(myGENERICCODECHandle, DRV_GENERICCODEC_CHANNEL_LEFT);
```

### C

```
uint8_t DRV_GENERICCODEC_VolumeGet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel);
```

## *DRV_GENERICCODEC_VolumeSet Function*

void DRV_GENERICCODEC_VolumeSet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel, uint8_t volume);

### Summary

This function sets the volume for Generic Codec.

### Description

This functions sets the volume value from 0-255.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| channel | argument indicating Left or Right or Both channel volume to be modified |
| volume | volume value specified in the range 0-255 (0x00 to 0xFF) |

### Returns

None

### Remarks

None.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_VolumeSet(myGENERICCODECHandle,DRV_GENERICCODEC_CHANNEL_LEFT, 120);
```

### C

```
void DRV_GENERICCODEC_VolumeSet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel, uint8_t
volume);
```

## Other Functions

### *DRV_GENERICCODEC_GetI2SDriver Function*

DRV_HANDLE DRV_GENERICCODEC_GetI2SDriver(DRV_HANDLE codecHandle)

### Summary

Get the handle to the I2S driver for this codec instance.

### Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

### Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

- A handle to the I2S driver for this codec instance

### Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

### C

```
DRV_HANDLE DRV_GENERICCODEC_GetI2SDriver(DRV_HANDLE codecHandle);
```

### *DRV_GENERICCODEC_LRCLK_Sync Function*

uint32_t DRV_GENERICCODEC_LRCLK_Sync (const DRV_HANDLE handle);

### Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

### Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

### Preconditions

None.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_LRCLK_Sync(myGENERICCODECHandle);
```

### C

```
bool DRV_GENERICCODEC_LRCLK_Sync(const DRV_HANDLE handle);
```

## Dara Types and Constants

### *DRV_GENERICCODEC_BUFFER_EVENT Type*

Identifies the possible events that can result from a buffer add request.

### Description

Generic Codec Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_GENERICCODEC_BufferAddWrite() or the DRV_GENERICCODEC_BufferAddRead() function.

### Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_GENERICCODEC_BufferEventHandlerSet function when a buffer transfer request is completed.

### C

```
typedef enum DRV_GENERICCODEC_BUFFER_EVENT@1 DRV_GENERICCODEC_BUFFER_EVENT;
```

### *DRV_GENERICCODEC_BUFFER_EVENT_HANDLER Type*

Pointer to a Generic Codec Driver Buffer Event handler function

### Description

Generic Codec Driver Buffer Event Handler Function

This data type defines the required function signature for the Generic Codec driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

### Parameters

| Parameters | Description |
|---|---|
| event | Identifies the type of event |
| bufferHandle | Handle identifying the buffer to which the event relates |
| context | Value identifying the context of the application that registered the event handling function. |

## Returns

None.

## Remarks

If the event is DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_GENERICCODEC_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_GENERICCODEC_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_GENERICCODEC_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_GENERICCODEC_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

## Example

```
void DRV_GENERICCODECBufferEventHandler( DRV_GENERICCODEC_BUFFER_EVENT event,
                                DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle,
                                uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

## C

```
typedef void (* DRV_GENERICCODEC_BUFFER_EVENT_HANDLER)(DRV_GENERICCODEC_BUFFER_EVENT event,
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

### *DRV_GENERICCODEC_BUFFER_HANDLE Type*

Handle identifying a write buffer passed to the driver.

## Description

Generic Codec Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_GENERICCODEC_BufferAddWrite() or DRV_GENERICCODEC_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

### Remarks

None

### C

```
typedef uintptr_t DRV_GENERICCODEC_BUFFER_HANDLE;
```

## *DRV_GENERICCODEC_CHANNEL Type*

Identifies Left/Right Audio channel

### Description

Generic Codec Audio Channel

This enumeration identifies Left/Right Audio channel

### Remarks

None.

### C

```
typedef enum DRV_GENERICCODEC_CHANNEL@1 DRV_GENERICCODEC_CHANNEL;
```

## *DRV_GENERICCODEC_COMMAND_EVENT_HANDLER Type*

Pointer to a Generic Codec Driver Command Event Handler Function

### Description

Generic Codec Driver Command Event Handler Function

This data type defines the required function signature for the Generic Codec driver command event handling callback function.

A command is a control instruction to the Generic Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

### Parameters

| Parameters | Description |
|---|---|
| context | Value identifying the context of the application that registered the event handling function. |

### Returns

None.

### Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_GENERICCODEC_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

### Example

```
void DRV_GENERICCODECCommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;
```

```
    // Last Submitted command is completed.
    // Perform further processing here
}
```

**C**

```
typedef void (* DRV_GENERICCODEC_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

### *DRV_GENERICCODEC_INIT Type*

Defines the data required to initialize or reinitialize the Generic Codec driver

#### Description

Generic Codec Driver Initialization Data

This data type defines the data required to initialize or reinitialize the Generic Codec driver.

#### Remarks

None.

**C**

```
typedef struct DRV_GENERICCODEC_INIT@1 DRV_GENERICCODEC_INIT;
```

### *DRV_GENERICCODEC_BUFFER_HANDLE_INVALID Macro*

Definition of an invalid buffer handle.

#### Description

Generic Codec Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_GENERICCODEC_BufferAddWrite() and the DRV_GENERICCODEC_BufferAddRead() function if the buffer add request was not successful.

#### Remarks

None.

**C**

```
#define DRV_GENERICCODEC_BUFFER_HANDLE_INVALID ((DRV_GENERICCODEC_BUFFER_HANDLE)(-1))
```

### *DRV_GENERICCODEC_COUNT Macro*

Number of valid Generic Codec driver indices

#### Description

Generic Codec Driver Module Count

This constant identifies the maximum number of Generic Codec Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of Generic Codec instances on this microcontroller.

#### Remarks

This value is part-specific.

**C**

```
#define DRV_GENERICCODEC_COUNT
```

### *DRV_GENERICCODEC_INDEX_0 Macro*

Generic Codec driver index definitions

## Description

Driver Generic Codec Module Index

These constants provide Generic Codec driver index definition.

## Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_GENERICCODEC_Initialize and DRV_GENERICCODEC_Open routines to identify the driver instance in use.

**C**

```
#define DRV_GENERICCODEC_INDEX_0 0
```

### *DRV_GENERICCODEC_INDEX_1 Macro*

This is macro DRV_GENERICCODEC_INDEX_1.

**C**

```
#define DRV_GENERICCODEC_INDEX_1 1
```

### *DRV_GENERICCODEC_INDEX_2 Macro*

This is macro DRV_GENERICCODEC_INDEX_2.

**C**

```
#define DRV_GENERICCODEC_INDEX_2 2
```

### *DRV_GENERICCODEC_INDEX_3 Macro*

This is macro DRV_GENERICCODEC_INDEX_3.

**C**

```
#define DRV_GENERICCODEC_INDEX_3 3
```

### *DRV_GENERICCODEC_INDEX_4 Macro*

This is macro DRV_GENERICCODEC_INDEX_4.

**C**

```
#define DRV_GENERICCODEC_INDEX_4 4
```

### *DRV_GENERICCODEC_INDEX_5 Macro*

This is macro DRV_GENERICCODEC_INDEX_5.

**C**

```
#define DRV_GENERICCODEC_INDEX_5 5
```

### *DRV_GENERICCODEC_AUDIO_DATA_FORMAT Type*

Identifies the Serial Audio data interface format.

#### Description

Generic Codec Audio data format

This enumeration identifies Serial Audio data interface format.

#### C

```
typedef enum DRV_GENERICCODEC_AUDIO_DATA_FORMAT@1 DRV_GENERICCODEC_AUDIO_DATA_FORMAT;
```

### *Files*

#### Files

| Name | Description |
|---|---|
| drv_genericcodec.h | Generic Codec Driver Interface header file |
| drv_genericcodec_config_template.h | Generic Codec Driver Configuration Template. |

#### Description

### drv_genericcodec.h

drv_genericcodec.h

#### Summary

Generic Codec Driver Interface header file

#### Description

Generic Codec Driver Interface

The Generic Codec device driver interface provides a simple interface to manage a codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the Generic Codec device driver.

### drv_genericcodec_config_template.h

drv_genericcodec_config_template.h

#### Summary

Generic Codec Driver Configuration Template.

#### Description

Generic Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

## I2S Driver Library Help

This section describes the I2S Driver Library.

## Introduction

This library provides an interface to manage the I2S Audio Protocol Interface Modes.

### Description

The I2S Driver is connected to a hardware module that provides the actual I2S stream, on some MCUs this is a Serial Peripheral Interface (SPI), on others it may be an I2S Controller (I2SC), or Serial Synchronous Controller (SSC).

The I2S hardware peripheral is then interfaced to various devices such as codecs and Bluetooth modules to provide microcontroller-based audio solutions.

## Using the Library

This topic describes the basic architecture of the I2S Driver Library and provides information and examples on its use.

### Description

**Interface Header File**: `drv_i2s.h`

The interface to the I2S Driver Library is defined in the `drv_i2s.h` header file. Any C language source (`.c`) file that uses the I2S Driver Library should include `drv_i2s.h`.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

**Example Applications:**

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

### *Abstraction Model*

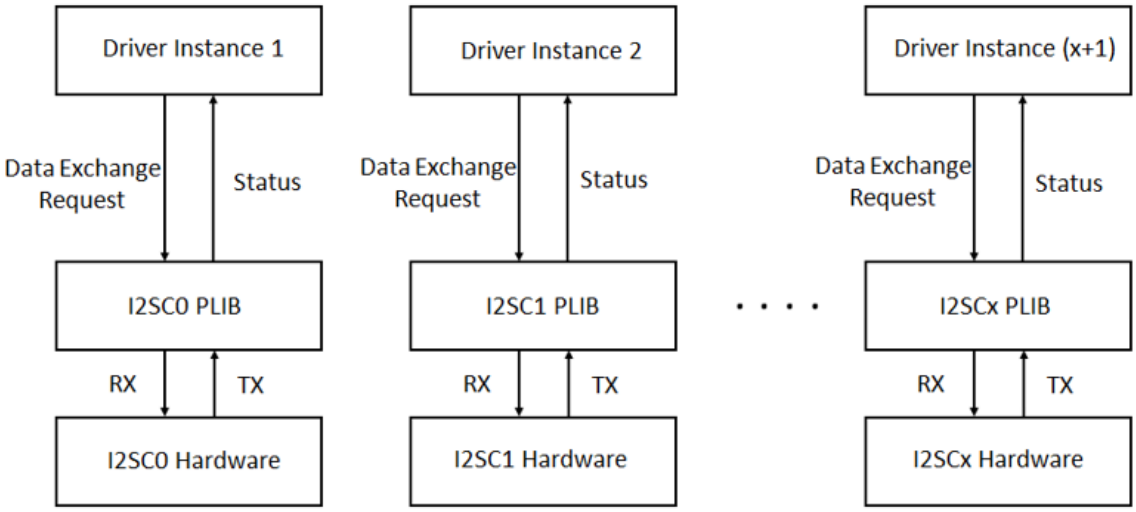
The I2S Driver provides a high level abstraction of the lower level (SPI/I2SC/SSC) I2S modules with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2S Driver Library interface.

### Description

Different types of I2S capable PLIBs are available on various Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The I2S Driver Library abstracts out these differences and provides a unified model for audio data transfer across different types of I2S modules.

Both the transmitter and receiver provide a buffer in the driver, which transmits and receives data to/from the hardware. The I2S Driver Library provides a set of interfaces to perform the read and the write. The following diagrams illustrate the abstraction model used by the I2S Driver Library. The I2SC Peripheral is used as an example of an I2S-capable PLIB.

**I2S Driver Abstraction Model**




The PLIBs currently provided, such as SSC and I2SC, only support an interrupt/DMA mode of operation. Polled mode of operation is not supported.

## *Library Overview*

Refer to the Driver Library Overwiew section for information on how the driver operates in a system.

The I2S driver library provides an API interface to transfer/receive digital audio data using supported Audio protocols. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2S Driver Library.

| Library Interface Section | Description |
|---|---|
| System Interaction Functions | Provides device initialization and status functions. |
| Client Setup Functions | Provides open and close functions. |
| Data Transfer Functions | Provides data transfer functions. |
| Miscellaneous Functions | Provides driver miscellaneous functions such as get error functions, L/R clock sync, etc. |
| Data Types and Constants | These data types and constants are required while interacting and setting up the I2S Driver Library. |

## *How the Library Works*

The library provides interfaces to support:

- System Functionality
- Client Functionality

**Note:** Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

## System Access

This section provides information on system access.

## Description

### System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2S module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_I2S_INIT or by using Initialization Overrides) that are supported by the specific I2S device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, DMA TX Channel, DMA RX Channel and Error Interrupt

The DRV_I2S_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_I2S_Deinitialize, DRV_I2S_Status, DRV_I2S_Tasks, and DRV_I2S_TasksError.

**Notes:**

1. The system initialization setting only effect the instance of the peripheral that is being initialized.
2. Configuration of the dynamic driver for DMA mode(uses DMA channel for data transfer) or Non DMA mode can be performed by appropriately setting the 'dmaChannelTransmit' and 'dmaChannelReceive' variables of the DRV_I2S_INIT structure. For example the TX will be in DMA mode when 'dmaChannelTransmit' is initialized to a valid supported channel number from the enum DMA_CHANNEL. TX will be in Non DMA mode when 'dmaChannelTransmit' is initialized to 'DMA_CHANNEL_NONE'.

**Example:**

```
DRV_I2S_INIT                  init;
SYS_MODULE_OBJ              objectHandle;

/* I2S Driver Initialization Data */
DRV_I2S_INIT drvI2S0InitData =
{
    .i2sPlib = &drvI2S0PlibAPI,
    .interruptI2S = DRV_I2S_INT_SRC_IDX0,
    .numClients = DRV_I2S_CLIENTS_NUMBER_IDX0,
    .queueSize = DRV_I2S_QUEUE_SIZE_IDX0,
    .dmaChannelTransmit = DRV_I2S_XMIT_DMA_CH_IDX0,
    .dmaChannelReceive  = DRV_I2S_RCV_DMA_CH_IDX0,
    .i2sTransmitAddress = (void *)&(SSC_REGS->SSC_THR),
    .i2sReceiveAddress = (void *)&(SSC_REGS->SSC_RHR),
    .interruptDMA = XDMAC_IRQn,
    .dmaDataLength = DRV_I2S_DATA_LENGTH_IDX0,
};

sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);
```

**Task Routine**

There is no task routine, since polled mode is not currently supported.

## Client Access

This section provides information on general client operation.

## Description

### General Client Operation

For the application to start using an instance of the module, it must call the DRV_I2S_Open function. This provides the settings required to open the I2S instance for operation.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

### Example:

```
DRV_HANDLE handle;
handle = DRV_I2S_Open(drvObj->i2sDriverModuleIndex,
```

```
        (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_NONBLOCKING));
    if (DRV_HANDLE_INVALID == handle)
    {
        // Unable to open the driver
        // May be the driver is not initialized or the initialization
        // is not complete.
    }
```

## Client Operations - Buffered

This section provides information on buffered client operations.

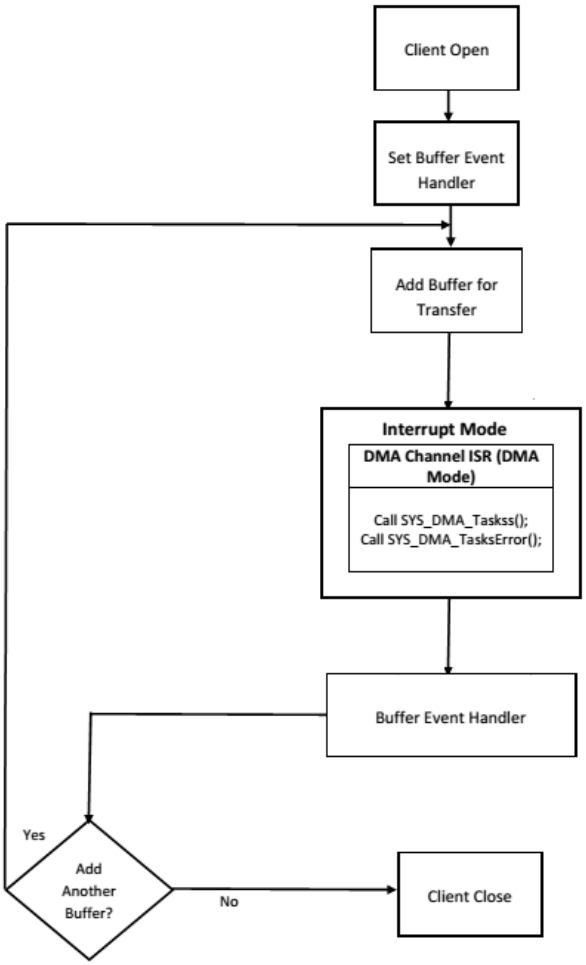
## Description

### Client Operations - Buffered

Client buffered operations provide a the typical audio interface. The functions DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead are the buffered data operation functions. The buffered functions schedules non-blocking operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_I2S_BUFFER_EVENT_COMPLETE, DRV_I2S_BUFFER_EVENT_ERROR or DRV_I2S_BUFFER_EVENT_ABORT events.

The buffer add requests are processed from the I2S channel ISR in interrupt mode.

The following diagram illustrates the buffered data operations



**Note:** It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. If DMA mode is desired, the DMA should be initialized by calling SYS_DMA_Initialize.
3. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).
4. The driver object should have been initialized by calling DRV_I2S_Initialize. If DMA mode is desired, related attributes in the init structure must be set.
5. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
6. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
7. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
8. The Register and event handler for the client handle should be set up by calling DRV_I2S_BufferEventHandlerSet.
9. Add a buffer to initiate the data transfer by calling DRV_I2S_BufferAddWrite/DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWriteRead.
10. When the DMA Channel has finished, the callback function registered in step 8 will be called.
11. Repeat step 9 through step 10 to handle multiple buffer transmission and reception.
12. When the client is done it can use DRV_I2S_Close to close the client handle.

**Example:**

```
// The following is an example for interrupt mode buffered transmit

#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj; //I2S module object
DRV_HANDLE handle;  //Client handle
uint32_t i2sClock;   //BCLK frequency
uint32_t baudrate;   //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state; //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
```

```
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);

                /* Add a buffer to write*/
                DRV_I2S_WriteBufferAdd(handle, myAudioBuffer, BUFFER_SIZE,
                    &bufferHandle);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
                    // Error handling here
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:

                state = APP_STATE_DONE;
            break;


            case  APP_STATE_DONE:
            {
                // Close done
                DRV_I2S_Close(handle);
            }
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
```

```
            // Take Action as needed
        }
        else
        {
            // Do nothing
        }
    }


    void SYS_Tasks ( void )
    {
        /* Call the application's tasks routine */
        APP_Tasks ( );
    }
```

### *Configuring the Library*

The configuration of the I2S Driver Library is based on the file `configurations.h`.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

### System Configuration

Configurations for driver instances, polled/interrupt mode, etc.

### *Configuring MHC*

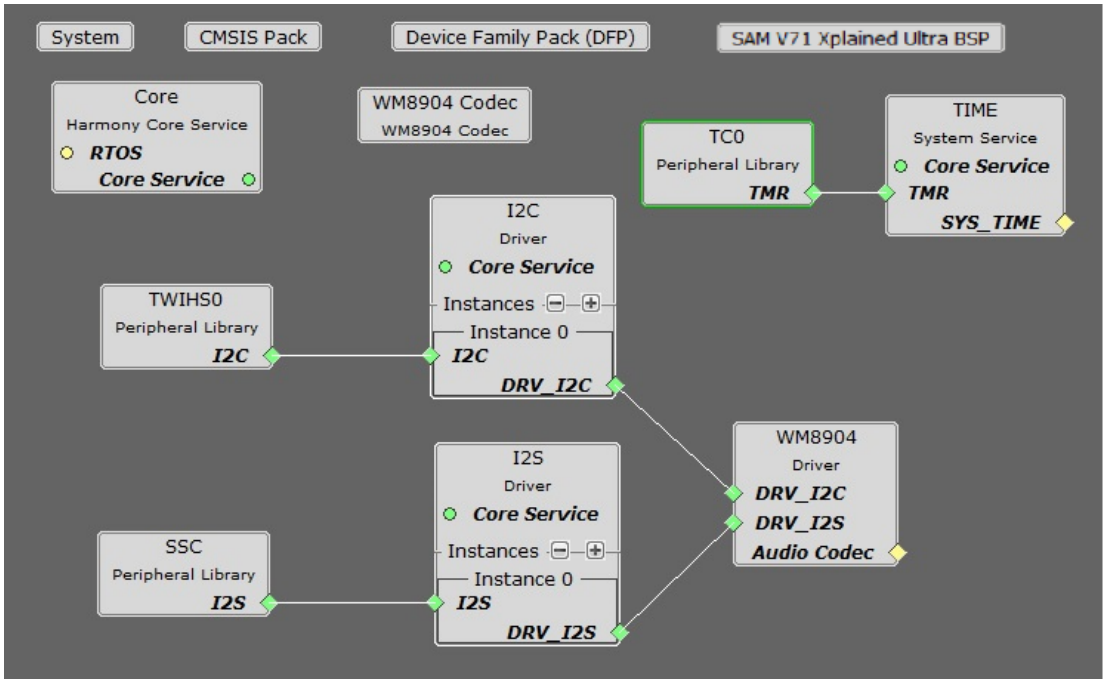
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

## Description

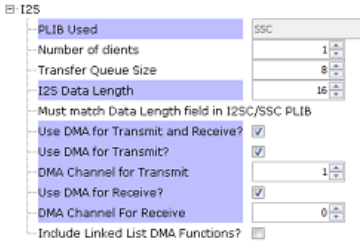
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File* > *New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under Audio->*Templates*, double-click on a codec template such as WM8904. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:




Click on the I2S Driver component, Instance 0, and the following menu will be displayed in the Configurations Options:




**PLIB Used** will display the hardware peripheral instance connected to the I2S Driver, such as SPI0, SSC, or I2SC1.

**Number of Clients** indicates the maximum number of clients that can be connected to the I2S Driver.

**Transfer Queue Size** indicates number of buffers, of each transfer queue (transmit/receive).

**I2S Data Length** is the number of bits for one channel of audio (left or right). It must match the size of the PLIB.

**Use DMA for Transmit and Receive** should always be checked if using DMA, which is currently the only supported mode.

**Use DMA for Transmit** should be checked if sending data to a codec or Bluetooth module.

**DMA Channel for Transmit** indicates the DMA channel # assigned (done automatically when you connect the PLIB).

**Use DMA for Receive** should be checked if receiving data from a codec or Bluetooth module.However if you are only writing to the I2S stream, leaving this checked won't harm anything.

**DMA Channel for Receive** indicates the DMA channel # assigned (done automatically when you connect the PLIB).

**Included Linked List DMA Functions** should be checked if using the Linked DMA feature of some MCUs.

You can also bring in the I2S Driver by itself, by double clicking I2S under Harmony->*Drivers* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

## Building the Library

This section lists the files that are available in the I2S Driver Library.

### Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is core/`driver/i2s`.

#### Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

| Source File Name | Description |
|---|---|
| drv_i2s.h | This file provides the interface definitions of the I2S driver (generated via template audio/driver/i2s/templates/drv_i2s.h.ftl) |

### Required File(s)

***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

| Source File Name | Description |
|---|---|
| /src/drv_i2s.c | This file contains the core implementation of the I2S driver with DMA support (generated via template audio/driver/i2s/templates/drv_i2s.c.ftl) |

### Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

| Source File Name | Description |
|---|---|
| N/A | |

### Module Dependencies

The I2S Driver Library depends on the following modules:

- I2S Peripheral Library, *or*
- SSC Peripheral Library
- I2SC Peripheral Library

## Library Interface

### a) System Interaction Functions

| | Name | Description |
|---|---|---|
| | DRV_I2S_Initialize | Initializes the I2S instance for the specified driver index. |
| | DRV_I2S_Status | Gets the current status of the I2S driver module. |

### b) Client Setup Functions

| | Name | Description |
|---|---|---|
| | DRV_I2S_Open | Opens the specified I2S driver instance and returns a handle to it. |
| | DRV_I2S_Close | Closes an opened-instance of the I2S driver. |

### c) Data Transfer Functions

| | Name | Description |
|---|---|---|
| | DRV_I2S_ReadBufferAdd | Queues a read operation. |
| | DRV_I2S_WriteBufferAdd | Queues a write operation. |
| | DRV_I2S_WriteReadBufferAdd | Queues a write/read operation. |
| | DRV_I2S_BufferEventHandlerSet | Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. |
| | DRV_I2S_ReadQueuePurge | Removes all buffer requests from the read queue. |
| | DRV_I2S_WriteQueuePurge | Removes all buffer requests from the write queue. |
| | DRV_I2S_BufferStatusGet | Returns the transmit/receive request status. |

| | | |
|---|---|---|
| | DRV_I2S_BufferCompletedBytesGet | Returns the number of bytes that have been processed for the specified buffer request. |

### d) Miscellaneous Functions

| | Name | Description |
|---|---|---|
| | DRV_I2S_ErrorGet | Gets the I2S hardware errors associated with the client. |
| | DRV_I2S_LRCLK_Sync | Synchronize to the start of the I2S LRCLK (left/right clock) signal |
| | DRV_I2S_SerialSetup | Sets the I2S serial communication settings dynamically. |
| | DRV_I2S_ClockGenerationSet | Set the clock(PLLA and I2SC GCLK clock) generation values |
| | DRV_I2S_ProgrammableClockSet | Set the Programmable Clock |

### e) Data Types and Constants

| | Name | Description |
|---|---|---|
| | DRV_I2S_BUFFER_EVENT | Identifies the possible events that can result from a buffer add request. |
| | DRV_I2S_BUFFER_EVENT_HANDLER | Pointer to a I2S Driver Buffer Event handler function |
| | DRV_I2S_BUFFER_HANDLE | Handle identifying a read or write buffer passed to the driver. |
| | DRV_I2S_SERIAL_SETUP | Defines the data required to dynamically set the serial settings. |
| | DRV_I2S_BUFFER_HANDLE_INVALID | Definition of an invalid buffer handle. |

## Description

This section describes the Application Programming Interface (API) functions of the I2S Driver Library.

Refer to each section for a detailed description.

### *a) System Interaction Functions*

#### DRV_I2S_Initialize Function

```
SYS_MODULE_OBJ DRV_I2S_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
)
```

## Summary

Initializes the I2S instance for the specified driver index.

## Description

This routine initializes the I2S driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the I2S module ID. For example, driver instance 0 can be assigned to I2S2.

## Preconditions

None.

## Parameters

| Parameters | Description |
|---|---|
| index | Identifier for the instance to be initialized |
| init | Pointer to the init data structure containing any data necessary to initialize the driver. |

### Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

### Remarks

This routine must be called before any other I2S routine is called.

This routine should only be called once during system initialization. This routine will NEVER block for hardware access.

### Example

```
// The following code snippet shows an example I2S driver initialization.

SYS_MODULE_OBJ              objectHandle;

I2S_PLIB_API drvUsart0PlibAPI = {
    {
        .readCallbackRegister = I2S1_ReadCallbackRegister,
        .read = I2S1_Read,
        .readIsBusy = I2S1_ReadIsBusy,
        .readCountGet = I2S1_ReadCountGet,
        .writeCallbackRegister = I2S1_WriteCallbackRegister,
        .write = I2S1_Write,
        .writeIsBusy = I2S1_WriteIsBusy,
        .writeCountGet = I2S1_WriteCountGet,
        .errorGet = I2S1_ErrorGet
    }
};

DRV_I2S_INIT drvUsart0InitData =
{
    .i2sPlib = &drvUsart0PlibAPI,
    .interruptI2S = I2S1_IRQn,
    .queueSizeTransmit = DRV_I2S_XMIT_QUEUE_SIZE_IDX0,
    .queueSizeReceive = DRV_I2S_RCV_QUEUE_SIZE_IDX0,
    .dmaChannelTransmit = SYS_DMA_CHANNEL_NONE,
    .dmaChannelReceive = SYS_DMA_CHANNEL_NONE,
    .i2sTransmitAddress = I2S1_TRANSMIT_ADDRESS,
    .i2sReceiveAddress = I2S1_RECEIVE_ADDRESS,
    .interruptDMA = XDMAC_IRQn
};

objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1,
    (SYS_MODULE_INIT*)&drvUsart0InitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

### C

```
SYS_MODULE_OBJ DRV_I2S_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

## DRV_I2S_Status Function

SYS_STATUS DRV_I2S_Status( SYS_MODULE_OBJ object )

### Summary

Gets the current status of the I2S driver module.

### Description

This routine provides the current status of the I2S driver module.

### Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| object | Driver object handle, returned from the DRV_I2S_Initialize routine |

### Returns

SYS_STATUS_READY - Initialization have succeeded and the I2S is ready for additional operations

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

### Remarks

A driver can opened only when its status is SYS_STATUS_READY.

### Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize
SYS_STATUS          i2sStatus;

i2sStatus = DRV_I2S _Status(object);
if (SYS_STATUS_READY == i2sStatus)
{
    // This means the driver can be opened using the
    // DRV_I2S_Open() function.
}
```

### C

```
SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object);
```

## *b) Client Setup Functions*

### DRV_I2S_Open Function

DRV_HANDLE DRV_I2S_Open

(

const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent

)

### Summary

Opens the specified I2S driver instance and returns a handle to it.

### Description

This routine opens the specified I2S driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

### Preconditions

Function DRV_I2S_Initialize must have been called before calling this function.

### Parameters

| Parameters | Description |
|---|---|
| index | Identifier for the object instance to be opened |

| | |
|---|---|
| intent | Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended useof the driver. See function description for details. |

### Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_I2S_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver peripheral instance being opened is not initialized or is invalid.
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.

### Remarks

The handle returned is valid until the DRV_I2S_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

### Example

```
DRV_HANDLE handle;

handle = DRV_I2S_Open(DRV_I2S_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

### C

```
DRV_HANDLE DRV_I2S_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

## DRV_I2S_Close Function

void DRV_I2S_Close( DRV_Handle handle )

### Summary

Closes an opened-instance of the I2S driver.

### Description

This routine closes an opened-instance of the I2S driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. A new handle must be obtained by calling DRV_I2S_Open before the caller may use the driver again.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

### Returns

None.

### Remarks

None.

### Example

```
// 'handle', returned from the DRV_I2S_Open

DRV_I2S_Close(handle);
```

### C

```
void DRV_I2S_Close(const DRV_HANDLE handle);
```

## *c) Data Transfer Functions*

## DRV_I2S_ReadBufferAdd Function

void DRV_I2S_ReadBufferAdd
(
const DRV_HANDLE handle,
void * buffer,
const size_t size,
DRV_I2S_BUFFER_HANDLE * bufferHandle
)

### Summary

Queues a read operation.

### Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_I2S_Open function. |
| buffer | Buffer where the received data will be stored. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The buffer handle is returned in the bufferHandle argument. This is DRV_I2S_BUFFER_HANDLE_INVALID if the request was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by the client. It should not be called in the event handler associated with another I2S driver instance. It should not be called directly in an ISR.

### Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

DRV_I2S_ReadBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.
```

### C

```
void DRV_I2S_ReadBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_I2S_BUFFER_HANDLE * const bufferHandle);
```

## DRV_I2S_WriteBufferAdd Function

void DRV_I2S_WriteBufferAdd

(

const DRV_HANDLE handle,

void * buffer,

size_t size,

DRV_I2S_BUFFER_HANDLE * bufferHandle

);

### Summary

Queues a write operation.

### Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the driver instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. On returning, the bufferHandle parameter may be DRV_I2S_BUFFER_HANDLE_INVALID for the following reasons:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the transmit queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or a DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as return by the DRV_I2S_Open function. |
| buffer | Data to be transmitted. |
| size | Buffer size in bytes. |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

### Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

DRV_I2S_WriteBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.
```

### C

```
void DRV_I2S_WriteBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_I2S_BUFFER_HANDLE * bufferHandle);
```

## DRV_I2S_WriteReadBufferAdd Function

void DRV_I2S_BufferAddWriteRead(const DRV_HANDLE handle,

void *transmitBuffer, void *receiveBuffer,

size_t size, DRV_I2S_BUFFER_HANDLE *bufferHandle)

### Summary

Queues a write/read operation.

### Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0

- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as return by the DRV_I2S_Open function. |
| transmitBuffer | Data to be transmitted. |
| receiveBuffer | Will hold data that is received. |
| size | Buffer size in bytes (same for both buffers) |
| bufferHandle | Pointer to an argument that will contain the return buffer handle. |

### Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

### Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every I2S write. The transmit and receive size must be same.

### C

```
void DRV_I2S_WriteReadBufferAdd(const DRV_HANDLE handle, void * transmitBuffer, void *
receiveBuffer, size_t size, DRV_I2S_BUFFER_HANDLE * bufferHandle);
```

## DRV_I2S_BufferEventHandlerSet Function

void DRV_I2S_BufferEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_I2S_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t context

)

### Summary

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

### Description

This function allows a client to register a buffer event handling function with the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine. |
| eventHandler | Pointer to the event handler function. |
| context | The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure). |

### Returns

None.

### Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

### Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandler,
                                (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

### C

```
void DRV_I2S_BufferEventHandlerSet(const DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

## DRV_I2S_ReadQueuePurge Function

bool DRV_I2S_ReadQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the read queue.

### Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_I2S_Open function. |

### Returns

True - Read queue purge is successful. False - Read queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myI2SHandle is the handle returned by the DRV_I2S_Open function.
// Use DRV_I2S_ReadBufferAdd to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_I2S_ReadQueuePurge(myI2Shandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_I2S_ReadQueuePurge(const DRV_HANDLE handle);
```

## DRV_I2S_WriteQueuePurge Function

bool DRV_I2S_WriteQueuePurge( const DRV_HANDLE handle )

### Summary

Removes all buffer requests from the write queue.

### Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout

or to remove unwanted stalled buffer requests or in any other use case.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

### Parameters

| Parameters | Description |
|---|---|
| handle | Handle of the communication channel as returned by the DRV_I2S_Open function. |

### Returns

True - Write queue purge is successful. False - Write queue purge has failed.

### Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

### Example

```
// myI2SHandle is the handle returned by the DRV_I2S_Open function.
// Use DRV_I2S_WriteBufferAdd to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_I2S_WriteQueuePurge(myI2Shandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

### C

```
bool DRV_I2S_WriteQueuePurge(const DRV_HANDLE handle);
```

## DRV_I2S_BufferStatusGet Function

DRV_I2S_BUFFER_EVENT DRV_I2S_BufferStatusGet
(
const DRV_I2S_BUFFER_HANDLE bufferHandle
)

### Summary

Returns the transmit/receive request status.

### Description

This function can be used to poll the status of the queued buffer request if the application doesn't prefer to use the event handler (callback) function to get notified.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function must have been called and a valid buffer handle returned.

### Parameters

| Parameters | Description |
|---|---|
| bufferhandle | Handle for the buffer of which the processed number of bytes to be obtained. |

### Returns

The success or error event of the buffer.

### Remarks

This function returns error event if the buffer handle is invalid.

### Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;
DRV_I2S_BUFFER_EVENT event;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                                (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd( myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

//Check the status of the buffer
//This call can be used to wait until the buffer is processed.
event  = DRV_I2S_BufferStatusGet(bufferHandle);
```

### C

```
DRV_I2S_BUFFER_EVENT DRV_I2S_BufferStatusGet(const DRV_I2S_BUFFER_HANDLE bufferHandle);
```

## DRV_I2S_BufferCompletedBytesGet Function

size_t DRV_I2S_BufferCompletedBytesGet

(

DRV_I2S_BUFFER_HANDLE bufferHandle

);

### Summary

Returns the number of bytes that have been processed for the specified buffer request.

### Description

The client can use this function, in a case where the buffer is terminated due to an error, to obtain the number of bytes that have been processed. Or in any other use case. This function can be used for non-DMA buffer transfers only. It cannot be used when the I2S driver is configured to use DMA.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function must have been called and a valid buffer handle returned.

## Parameters

| Parameters | Description |
|---|---|
| bufferhandle | Handle for the buffer of which the processed number of bytes to be obtained. |

## Returns

Returns the number of bytes that have been processed for this buffer.

Returns DRV_I2S_BUFFER_HANDLE_INVALID for an invalid or an expired buffer handle.

## Remarks

This function is expected to work in non-DMA mode only. This function is thread safe when used in a RTOS application. If called from the callback, it must not call an OSAL mutex or critical section.

## Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                                (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd( myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
            // Error handling here.
            // We can find out how many bytes have been processed in this
            // buffer request prior to the error.
            processedBytes= DRV_I2S_BufferCompletedBytesGet(bufferHandle);
            break;

        default:
            break;
    }
}
```

### C

```
size_t DRV_I2S_BufferCompletedBytesGet(DRV_I2S_BUFFER_HANDLE bufferHandle);
```

### *d) Miscellaneous Functions*

### DRV_I2S_ErrorGet Function

DRV_I2S_ERROR DRV_I2S_ErrorGet( const DRV_HANDLE handle )

#### Summary

Gets the I2S hardware errors associated with the client.

#### Description

This function returns the errors associated with the given client. The call to this function also clears all the associated error flags.

#### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

#### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |

#### Returns

Errors occurred as listed by DRV_I2S_ERROR. This function reports multiple I2S errors if occurred.

#### Remarks

I2S errors are normally associated with the receiver. The driver clears all the errors internally and only returns the occurred error information for the client.

#### Example

```
// 'handle', returned from the DRV_I2S_Open

if (DRV_I2S_ERROR_OVERRUN & DRV_I2S_ErrorGet(handle))
{
    //Errors are cleared by the driver, take respective action
    //for the overrun error case.
}
```

#### C

```
DRV_I2S_ERROR DRV_I2S_ErrorGet(const DRV_HANDLE handle);
```

### DRV_I2S_LRCLK_Sync Function

uint32_t DRV_I2S_LRCLK_Sync (const DRV_HANDLE handle, const uint32_t sample_rate);

#### Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

#### Description

This function waits until low-tohigh transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Jusitified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

### Preconditions

None.

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

DRV_I2S_LRCLK_Sync(myI2SHandle, 48000);
```

### C

```
bool DRV_I2S_LRCLK_Sync(const DRV_HANDLE handle, const uint32_t sample_rate);
```

## DRV_I2S_SerialSetup Function

bool DRV_I2S_SerialSetup(const DRV_HANDLE handle,
DRV_I2S_SERIAL_SETUP * setup)

### Summary

Sets the I2S serial communication settings dynamically.

### Description

This function sets the I2S serial communication settings dynamically.

### Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle. The I2S transmit or receive transfer status should not be busy.

### Parameters

| Parameters | Description |
|---|---|
| handle | A valid open-instance handle, returned from the driver's open routine |
| setup | Pointer to the structure containing the serial setup. |

### Returns

true - Serial setup was updated successfully. false - Failure while updating serial setup.

### Remarks

None.

### Example

```
// 'handle', returned from the DRV_I2S_Open

DRV_I2S_SERIAL_SETUP setup = {
        115200,
        DRV_I2S_DATA_8_BIT,
        DRV_I2S_PARITY_ODD,
        DRV_I2S_STOP_1_BIT
    };

DRV_I2S_SerialSetup(handle, &setup);
```

### C

```
bool DRV_I2S_SerialSetup(const DRV_HANDLE handle, DRV_I2S_SERIAL_SETUP * setup);
```

## DRV_I2S_ClockGenerationSet Function

bool DRV_I2S_ClockGenerationSet(DRV_HANDLE handle, uint8_t div, uint8_t mul, uint8_t div2);

### Summary

Set the clock(PLLA and I2SC GCLK clock) generation values

### Description

Set the clock(PLLA and I2SC GCLK clock) generation values

### Preconditions

None.

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.
DRV_I2S_ClockGenerationSet(myI2SHandle, 2, 40, 5);
```

### C

```
bool DRV_I2S_ClockGenerationSet(DRV_HANDLE handle, uint8_t div, uint8_t mul, uint8_t div2);
```

## DRV_I2S_ProgrammableClockSet Function

bool DRV_I2S_ProgrammableClockSet(DRV_HANDLE handle, uint8_t pClkNum, uint8_t div2);

### Summary

Set the Programmable Clock

### Description

Set the Programmable Clock ignoring glitch control

### Preconditions

None.

### Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

### Remarks

None.

### Example

```
// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.
DRV_I2S_ProgrammableClockSet(myI2SHandle, 2, 7);
```

### C

```
bool DRV_I2S_ProgrammableClockSet(DRV_HANDLE handle, uint8_t pClkNum, uint8_t div2);
```

## *e) Data Types and Constants*

### DRV_I2S_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

### Description

I2S Driver Buffer Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd functions.

### Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_I2S_BufferEventHandlerSet function when a buffer transfer request is completed.

### C

```
typedef enum {
  DRV_I2S_BUFFER_EVENT_COMPLETE,
  DRV_I2S_BUFFER_EVENT_ERROR,
  DRV_I2S_BUFFER_EVENT_ABORT
} DRV_I2S_BUFFER_EVENT;
```

### DRV_I2S_BUFFER_EVENT_HANDLER Type

Pointer to a I2S Driver Buffer Event handler function

### Description

I2S Driver Buffer Event Handler Function Pointer

This data type defines the required function signature for the I2S driver buffer event handling callback function. A client must register a pointer using the buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

### Parameters

| Parameters | Description |
|---|---|
| event | Identifies the type of event |
| bufferHandle | Handle identifying the buffer to which the vent relates |
| context | Value identifying the context of the application that registered the event handling function. |

### Returns

None.

### Remarks

If the event is DRV_I2S_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_I2S_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully. The DRV_I2S_BufferCompletedBytesGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event. And bufferHandle will be valid while the buffer request is in the queue and during callback, unless an error occurred. After callback returns, the driver will retire

the buffer handle.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_I2S_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The DRV_I2S_ReadBufferAdd and DRV_I2S_WriteBufferAdd functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running. For example, I2S2 driver buffers cannot be added in I2S1 driver event handler.

### Example

```
void APP_MyBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
                               DRV_I2S_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

### C

```
typedef void (* DRV_I2S_BUFFER_EVENT_HANDLER)(DRV_I2S_BUFFER_EVENT event, DRV_I2S_BUFFER_HANDLE
bufferHandle, uintptr_t context);
```

## DRV_I2S_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

### Description

I2S Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

### Remarks

None

### C

```
typedef uintptr_t DRV_I2S_BUFFER_HANDLE;
```

### DRV_I2S_SERIAL_SETUP Type

Defines the data required to dynamically set the serial settings.

#### Description

I2S Driver Serial Setup Data

This data type defines the data required to dynamically set the serial settings for the specific I2S driver instance.

#### Remarks

This structure is implementation specific. It is fully defined in drv_i2s_definitions.h.

#### C

```
typedef struct _DRV_I2S_SERIAL_SETUP DRV_I2S_SERIAL_SETUP;
```

### DRV_I2S_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

#### Description

I2S Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_I2S_ReadBufferAdd and DRV_I2S_WriteBufferAdd functions if the buffer add request was not successful.

#### Remarks

None

#### C

```
#define DRV_I2S_BUFFER_HANDLE_INVALID
```

## Files

### Files

| Name | Description |
|---|---|
| drv_i2s.h | I2S Driver Interface Header File |
| drv_i2s_config_template.h | I2S Driver Configuration Template. |

### Description

### *drv_i2s.h*

drv_i2s.h

#### Summary

I2S Driver Interface Header File

#### Description

I2S Driver Interface Header File

The I2S device driver provides a simple interface to manage the I2S or SSC modules on Microchip PIC32 microcontrollers. This file provides the interface definition for the I2S driver.

### *drv_i2s_config_template.h*

drv_i2s_config_template.h

## Summary

I2S Driver Configuration Template.

## Description

I2S Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

# Peripheral Libraries Help

This topic provides help for the peripheral libraries that are available in the audio repo.

For additional information on Harmony 3 peripheral libraries (PLIBs), refer to the documentation in the csp repository.

### Description

The MPLAB Harmony Configurator (MHC) is a Graphical User Interface (GUI) plug-in tool for MPLAB X IDE to configures a rich set of peripherals and functions specific to your application and generate the corresponding peripheral library code. The generated code directly accesses the Peripheral registers without any abstraction layers and it is easy to understand.

## I2S Peripheral Library Help

This section provides an interface to use the I2S peripheral.

## Introduction

This library provides a brief overview of the I2S peripheral.

### Description

The I2S module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2S module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **SDI:** Serial Data Input for receiving sample digital audio data (ADCDAT as output from the codec)
- **SDO:** Serial Data Output for transmitting digital audio data (DACDAT as input to the codec)
- **SCKn:** Serial Clock, also known as bit clock (BCLK)
- **FSn:** Frame Select, also known as Word Select or Left/Right Channel Clock (LRCK)

In addition, there is a fifth line, called **MCKn** (Master Clock), which can be used to drive the codec.

There are two clock units in the I2S module, so **n**=1 or 2 depending on which is used. However there are only two data lines (SDO/SDI), one for each serializer (transmit/receive).

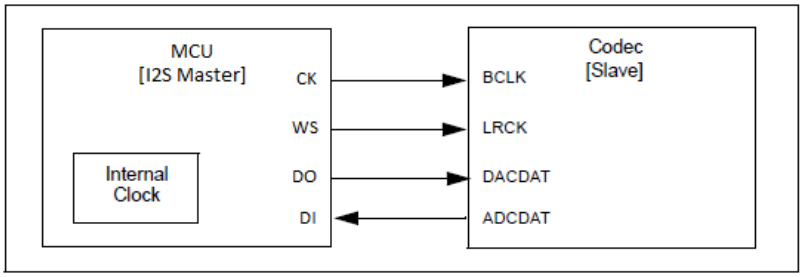
The SCK provides the clock required to drive the data out or into the module, while FS provides the synchronization of the frame based on the protocol mode selected. In I2S mode, the leading edge of audio data is driven out one SCK period of starting the frame.

In Master mode, the module generates both the SCK and FS.

In Slave mode, the peripheral generates the BCLK and LRCLK signals, driving the SCK and FS pins of the I2S module. When in Slave mode, the I2S cannot generate a master clock (MCK), so a generic clock (GCLKn) must be used.

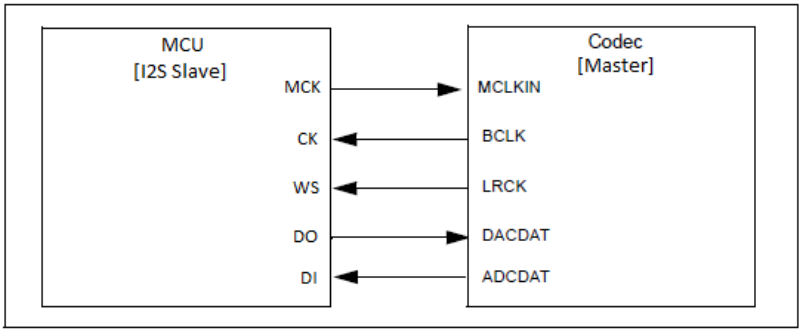
#### Master Mode

**Master Generating its Own Clock – Output BCLK and LRCK**




#### Slave Mode

**Codec Device as Master Derives MCLK from MCU Reference Clock Out**




## Configuring the Library

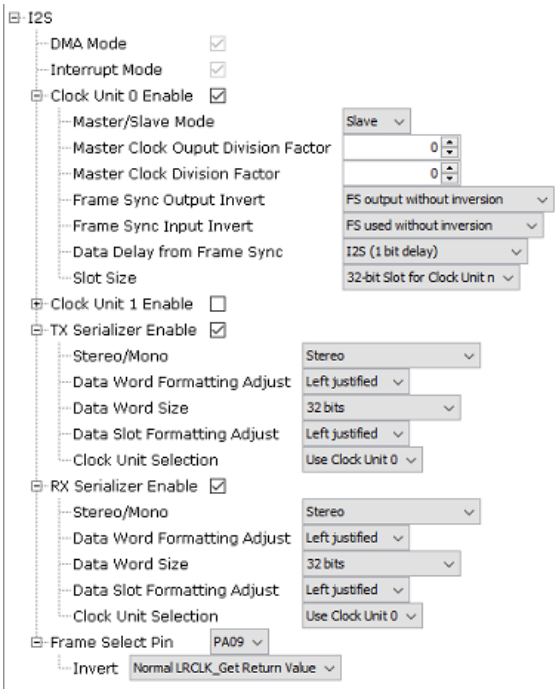
This section describes how to configure the peripheral library using the MHC.

### Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the I2S peripheral by clicking on the appropriate instance under *Peripherals->I2S* in the Available Components section of MHC. (The I2S component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2S peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options:




Typical values are shown for working in Slave mode, with a codec such as the WM8904 in Master mode. **DMA** and **Interrupt Mode** are always enabled.

For each of the two Clock Units (0 and 1, only the first is shown):

**Clock Unit n Enable --** if checked, the clock unit is enabled

**Master/Slave Mode** can be either Master -- the I2S peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

**Master Clock Output Divisor** -- the Generic I2S Clock (GCLK_I2S_n) is divided by this number to generate the Master Clock (MCKn)

**Master Clock Division Factor** -- the Master Clock (MCKn) is divided by this number to generate a serial clock SCKn

**Frame Sync Output Invert** -- the FSn signal is output with or without inversion

**Frame Sync Input Invert** -- the FSn signal is used with or without inversion on input

**Data Delay from Frame Sync** -- either 1 bit for I2S format, or 0 for left/right adjusted format

**Slot Size** selects the number of bits per slot: 8, 16, 24 or 32

For each of the Serializers (TX/RX):

**Stereo/Mono** selects stereo or mono (left channel is duplicated to the right)

**Data Word Formatting Adjust** selects left or right adjustment of data samples within the word

**Data Word Size** selects the number of bits per sample: 8, 16, 18, 20, 24 or 32 (or 8-bit compact stereo or 16-bit compact stereo)

**Data Slot Formatting Adjust** selects left or right adjustment of data samples within the slot

**Clock Unit Selection** selects which clock unit to use for this serializer, 0 or 1

**Frame Select Pin** specifies which pin is used as the FSn output as selected in the Pin Diagram

**Invert** selects whether the output of the I2S_LRCLK_Get function returns the true value or inverted one.

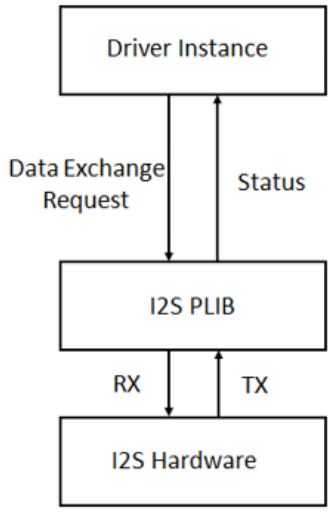
## Using the Library

This topic describes the basic architecture of the I2S Peripheral Library and provides information and examples on how to use it.

### Description

#### Abstraction Model

The I2S module sits between the I2S Driver, and the actual hardware.




**Interface Header File**: `plib_i2sc.h`

The interface to the I2S Peripheral Library is defined in the `plib_i2sc.h` header file. Any C language source (`.c`) file that uses the I2S Peripheral Library should include plib_i2sc.h.

#### Library Source Files:

The I2S Peripheral Library library source files are provided in the audio`/peripheral/i2s_xxxx/src` or audio`/peripheral/i2s_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

### Usage Model

The only usage model for the I2S Peripheral Library is to use a Interrupt/DMA model. Therefore the remaining functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right clock (LRCLK) for the I2S stream.

**Example Applications:**

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

## Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

### *a) Initialization Function*

### *b) Transaction Functions*

## Files

### Files

| Name | Description |
|---|---|
| plib_i2s_u2224.h | I2S PLIB Header File for documentation |

### Description

### *plib_i2s_u2224.h*

plib_i2s_u2224.h

### Summary

I2S PLIB Header File for documentation

### Description

I2SC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of an Inter-IC Sound Controller (I2S). This file must not be included in any MPLAB Project.

# I2SC Peripheral Library Help

This section provides an interface to use the Inter-IC Sound Controller (I2SC) peripheral.

# Introduction

This library provides a brief overview of the I2SC peripheral.

## Description

The I2SC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2SC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **DI:** Serial Data Input for receiving sample digital audio data (ADCDAT)
- **DO:** Serial Data Output for transmitting digital audio data (DACDAT)
- **CK:** Serial Clock, also known as bit clock (BCLK)
- **WS:** Word Select, also known as Left/Right Channel Clock (LRCK)

The BCLK provides the clock required to drive the data out or into the module, while the LRCK provides the synchronization of the frame based on the protocol mode selected.

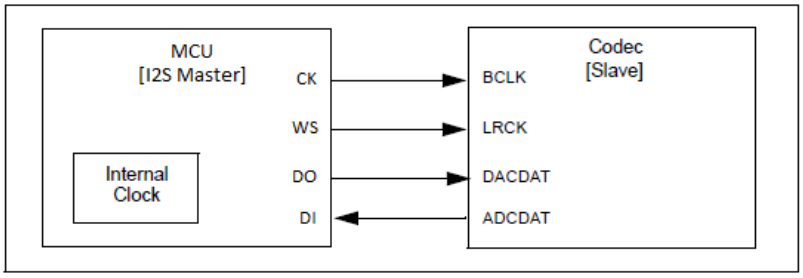
In Master mode, the module generates both the BCLK on the CK pin and the LRCK on the WS pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of CK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out DO, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

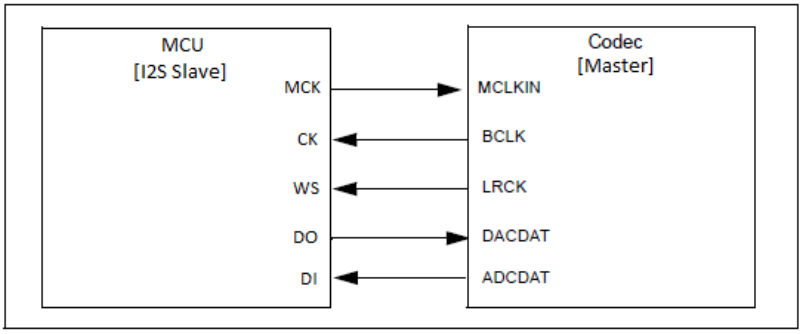
### Master Mode

**Master Generating its Own Clock – Output BCLK and LRCK**




### Slave Mode

**Codec Device as Master Derives MCLK from MCU Reference Clock Out**




**Audio Formats**

The I2SC Module supports just one audio format, I2S. Left Justified format is not available.

## Configuring the Library

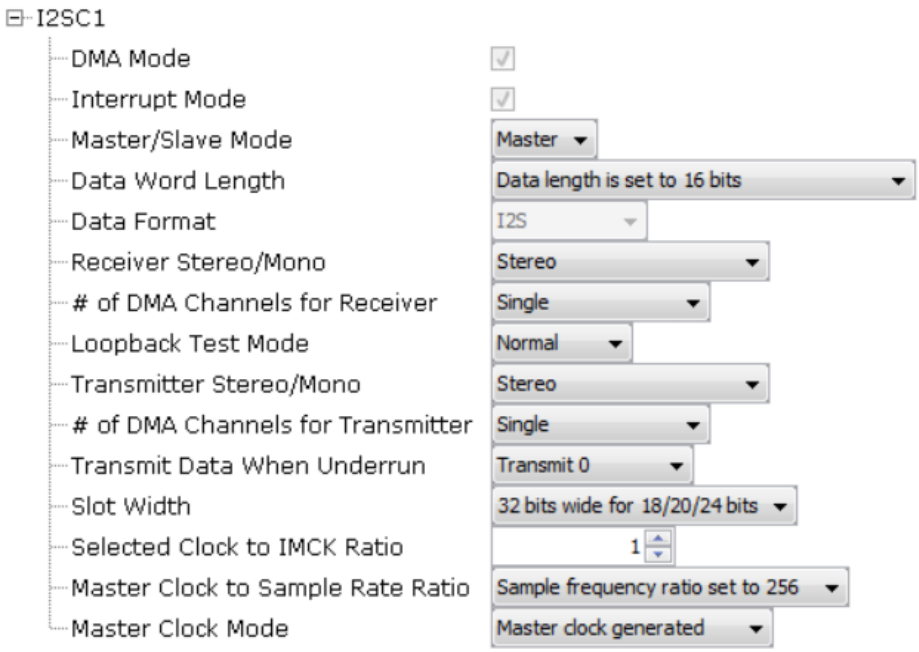
This section describes how to configure the peripheral library using the MHC.

### Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the I2SC peripheral by clicking on the appropriate instance under *Peripherals->I2SC* in the Available Components section of MHC. (The I2SC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2SCx peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options (example shown for instance I2SC1):




Default values are shown. The **DMA** and **Interrupt Mode** are always enabled.

**Master/Slave Mode** can be either Master -- the I2SC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

**Data Word Length** can be selected from 8 to 32-bits.

**Data Format** is currently always I2S.

**Receiver Stereo/Mono** can be either Stereo or Mono (left channel duplicated to right).

**# of DMA Channels for Receiver** can either Single or Multiple (1/channel).

**Loopback Test Mode** can be Normal or Loop mode.

**Transmitter Stereo/Mono** can be either Stereo or Mono (left channel duplicated to right).

**Transmit Data When Underrun** can be Transmit 0 or Transmit previous.

**Slot Width** can be 24 or 32-bits wide.

The following options are only shown if the Master/Slave mode is Master:

- **Selected Clock for IMCK Ratio** is selectable as any integer
- **Master Clock to Sample Rate Ratio** can be set from 32 to 2048
- **Master Clock Mode** can be either Master Clock Generated or not

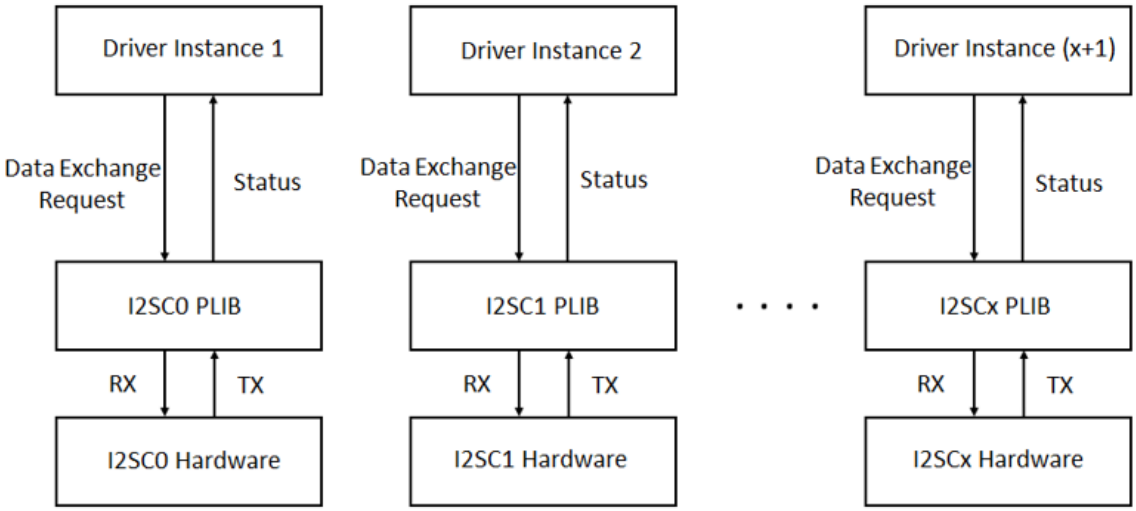
# Using the Library

This topic describes the basic architecture of the I2SC Peripheral Library and provides information and examples on how to use it.

## Description

### Abstraction Model

The I2SC module sits between the I2S Driver, and the actual hardware.




**Interface Header File**: `plib_i2sc.h`

The interface to the I2SC Peripheral Library is defined in the `plib_i2sc.h` header file. Any C language source (`.c`) file that uses the I2SC Peripheral Library should include plib_i2sc.h.

### Library Source Files:

The I2SC Peripheral Library library source files are provided in the `csp/peripheral/i2sc_xxxx/src` or `csp/peripheral/i2sc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

### Usage Model

The only usage model for the I2SC Peripheral Library is to use a Interrupt/DMA model. Therefore the remaining functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right clock (LRCLK) for the I2S stream.

**Example Applications:**

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

## Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

### a) Initialization Function

| | Name | Description |
|---|---|---|
| | I2SCx_Initialize | Initializes I2SCx module of the device |

### b) Transaction Functions

| | Name | Description |
|---|---|---|
| | I2SCx_LRCLK_Get | Get the level of the I2S LRCLK (left/right clock) signal |

### *a) Initialization Function*

#### I2SCx_Initialize Function

void I2SCx_Initialize (void);

### Summary

Initializes I2SCx module of the device

### Description

This function initializes I2SCx module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

### Preconditions

MHC GUI should be configured with the right values.

### Returns

None.

### Remarks

This function must be called only once and before any other I2SC function is called.

### Example

```
I2SC0_Initialize();
```

### C

```
void I2SCx_Initialize();
```

### *b) Transaction Functions*

#### I2SCx_LRCLK_Get Function

uint32_t I2SCx_LRCLK_Get(void);

### Summary

Get the level of the I2S LRCLK (left/right clock) signal

### Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize itself to the LRCLK signal.

### Preconditions

None.

### Returns

State of the LRCLK pin for the I2SCx module -- 1 if high, 0 if low

### Remarks

None.

### Example

```
I2SC1_LRCLK_Get();
```

### C

```
uint32_t I2SCx_LRCLK_Get();
```

#### *c) Data Types and Constants*

## Files

### Files

| Name | Description |
|---|---|
| plib_i2sc_11241.h | I2SC PLIB Header File for documentation |

### Description

#### *plib_i2sc_11241.h*

plib_i2sc_11241.h

### Summary

I2SC PLIB Header File for documentation

### Description

I2SC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of an Inter-IC Sound Controller (I2SC). This file must not be included in any MPLAB Project.

# SPI-I2S Peripheral Library Help

This section provides an interface to use the Inter-IC Sound (I2S) peripheral.

## Introduction

This library provides a brief overview of the I2S peripheral. On PIC32MX/MZ processors, this peripheral is shared with the SPI (Serial Peripheral Interface) hardware.

### Description

The I2S module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2S module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **SDI:** Serial Data Input for receiving sample digital audio data (ADCDAT as output from the codec)
- **SDO:** Serial Data Output for transmitting digital audio data (DACDAT as input to the codec)
- **SCK:** Serial Clock, also known as bit clock (BCLK)
- **FS:** Frame Select, also known as Word Select or Left/Right Channel Clock (LRCK)

In addition, there is a fifth line, called REFCLKO (Reference CLock Output, or Master Clock), which can be used to drive the codec.

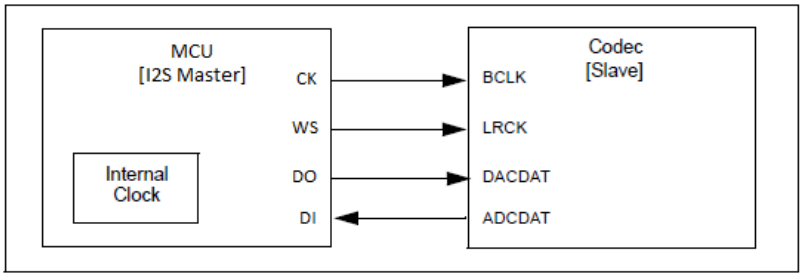
The SCK provides the clock required to drive the data out or into the module, while FS provides the synchronization of the frame based on the protocol mode selected. In I2S mode, the leading edge of audio data is driven out one SCK period of starting the frame.

In Master mode, the module generates both the SCK and FS.

In Slave mode, the peripheral generates the BCLK and LRCLK signals, driving the SCK and FS pins of the I2S module.

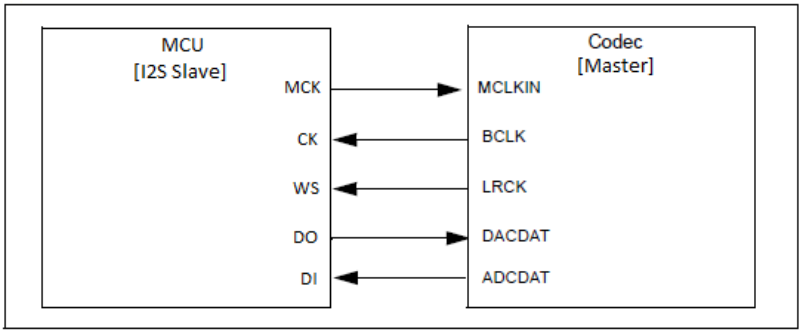
**Master Mode**

**Master Generating its Own Clock – Output BCLK and LRCK**




**Slave Mode**

**Codec Device as Master Derives MCLK from MCU Reference Clock Out**




## Configuring the Library

This section describes how to configure the peripheral library using the MHC.

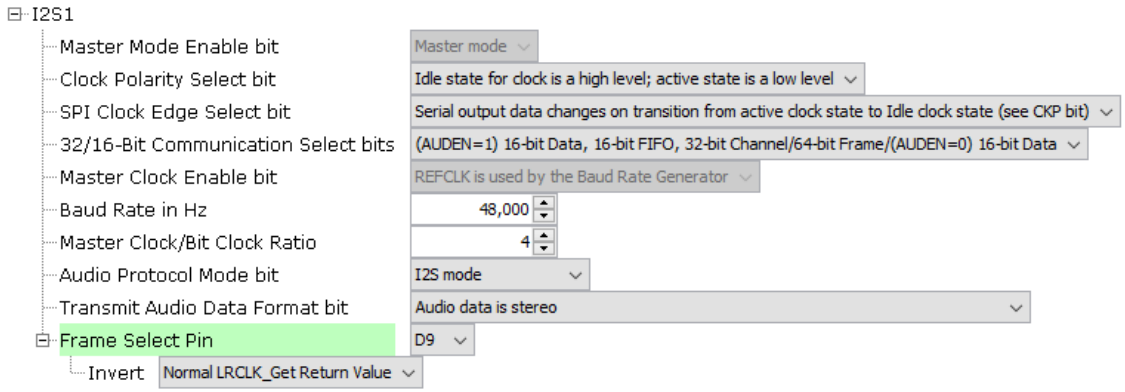
### Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip

Harmony Configurator (MHC).

Choose the I2S peripheral by clicking on the appropriate instance under *Peripherals->I2S* in the Available Components section of MHC. (The I2S component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2S peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options:




Note: DMA and Interrupt Mode are always enabled.

**Master Mode Enable Bit --** only Master Mode is currently supported.

**Clock Polarity Select Bit** -- determines whether idle state of clock is high (default) or vice versa.

**SPI Clock Edge Select Bit** -- selects whether output chamges on transistion from active to idle (default) or vice versa.

**32/16 Bit Communication Select Bits** -- selects the data, FIFO, channel, and frame sizes.

**Master CLock Enable Bit** -- always selects REFCLK to be used by the baud rate generator.

**Audio Protocol Mode** -- I2S, left or right justified, or PCM.

**Transmit Audio Data Format** -- stereo or mono.

**Frame Select Pin** -- specifies which pin is used for the frame select (aka L/R clock or word select)

**Invert** -- indicates whether the output of the LRCLK_Get function is inverted or not

## Using the Library

This section explains how to use the peripheral library.

### Description

```
Library Source Files:
The I2S Peripheral Library library source files are provided in the
audio/peripheral/i2s_xxxx/src or audio/peripheral/i2s_xxxx/templates directory. This folder may
contain optional files and alternate implementations. Please refer to Configuring the Library
for instructions on how to select optional features.

Usage Model
The only usage model for the I2S Peripheral Library is to use a Interrupt/DMA model. Therefore
the remaining functions normally associated with a PLIB like this (handling write or read
requests, or returning transfer status), will be accomplished by the I2S driver directly
communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right
clock (LRCLK) for the I2S stream.

Example Applications:
This library is used by the following applications, among others:
  * audio/apps/audio_tone
  * audio/apps/microphone_loopback
```

## Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

### a) Initialization Function

| | Name | Description |
|---|---|---|
| | I2Sx_Initialize | Initializes I2S x module of the device |

### b) Status Functions

| | Name | Description |
|---|---|---|
| | I2Sx_LRCLK_Get | Get LRCLK state |

### *a) Initialization Function*

#### I2Sx_Initialize Function

void I2Sx_Initialize (void);

#### Summary

Initializes I2S x module of the device

#### Description

This function initializes I2S x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

#### Preconditions

MHC GUI should be configured with the right values.

#### Returns

None.

#### Remarks

This function must be called only once and before any other SPI function is called.

#### Example

```
    I2S1_Initialize();
```

#### C

```
void I2Sx_Initialize();
```

### *b) Status Functions*

#### I2Sx_LRCLK_Get Function

uint32_t I2Sx_LRCLK_Get (void);

### Summary

Get LRCLK state

### Description

This function returns the state of the LRCLK (left/right clock, aka word select line) of the I2S peipheral.

### Preconditions

MHC GUI should be configured with the right values.

### Returns

None.

### Example

```
lrclkState = I2Sx_LRCLK_Get();
```

### C

```
uint32_t I2Sx_LRCLK_Get();
```

## Files

### Files

| Name | Description |
|---|---|
| plib_spi_01329.h | SPI PLIB Header File for documentation |

### Description

#### *plib_spi_01329.h*

plib_spi_01329.h

### Summary

SPI PLIB Header File for documentation

### Description

SPI-I2S PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Peripheral Interface (SPI) controller used for I2S. This file must not be included in any MPLAB Project.

# SSC Peripheral Library Help

This section provides an interface to use the Serial Synchronous Controller (SSC) peripheral.

## Introduction

This section provides a brief overview of the SSC peripheral.

### Description

The SSC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module. The SSC hardware also provides for other protocols, which can be configured manually.

The SSC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **RD:** Serial Data Input for receiving sample digital audio data (ADCDAT)
- **TD:** Serial Data Output for transmitting digital audio data (DACDAT)
- **TK/RK:** Transmit/Receive Serial Clock, also known as bit clock (BCLK)
- **TF/RF:** Transmit/Receive Frame Clock, also known as Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

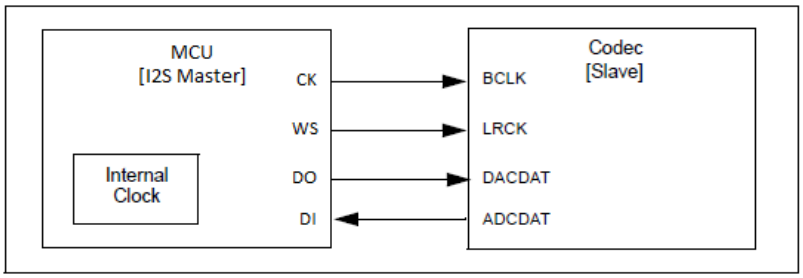
In Master mode, the module generates both the BCLK on the TK pin and the LRCK on the TF pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of BCLK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out TD, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

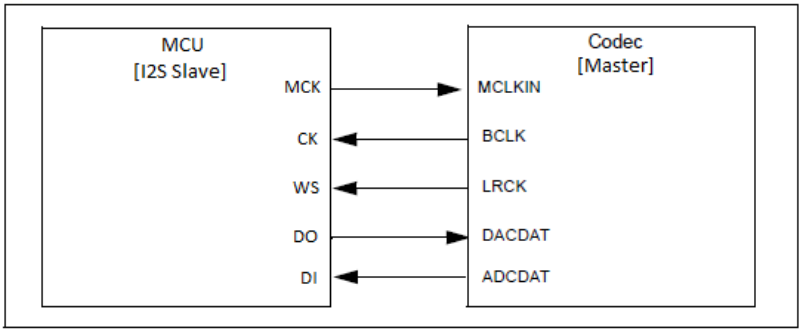
### Master Mode

**Master Generating its Own Clock – Output BCLK and LRCK**




### Slave Mode

**Codec Device as Master Derives MCLK from MCU Reference Clock Out**




**Audio Formats**

The SSC Module supports two audio formats, I2S and Left Justified, which are selectable via MHC.

## Configuring the Library

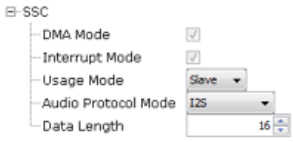
This section describes how to configure the peripheral library using the MHC.

### Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the SSC peripheral by clicking on the appropriate instance under *Peripherals->SSC* in the Available Components section of MHC. (The SSC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the SSC peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options):




Default values are shown. The **DMA** and **Interrupt Mode** are always enabled.

**Usage Mode** can be either Master -- the SSC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

**Audio Protocol Mode** can be either I2S, Left Justified, or Custom. In the latter case, a number of other options are then available, which allows the SSC peripheral to be used for a number of protocols (not currently supported by existing drivers).

**Data Length** is set to the number of data bits per channel.

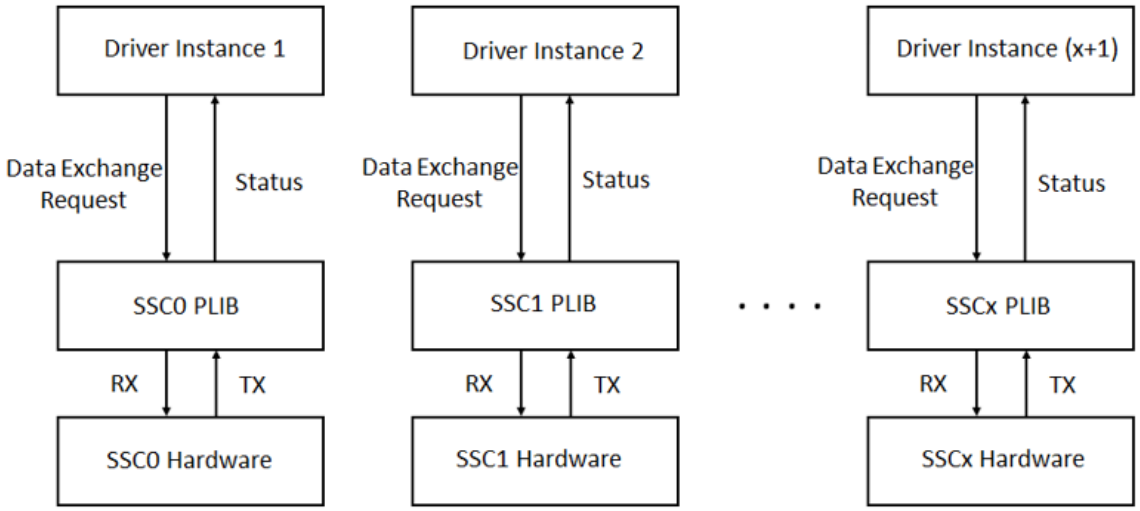
## Using the Library

This topic describes the basic architecture of the SSC Peripheral Library and provides information and examples on how to use it.

### Description

#### Abstraction Model

The SSC module sits between the I2S Driver, and the actual hardware.




Note: Some devices may have only one instance of an SSC interface (SSC0 only).

**Interface Header File**: plib_ssc.h

The interface to the SSC Peripheral Library is defined in the plib_ssc.h header file. Any C language source (.c) file that uses the SSC Peripheral Library should include plib_ssc.h.

#### Library Source Files:

The SSC Peripheral Library library source files are provided in the audio`/peripheral/ssc_xxxx/src` or audio`/peripheral/ssc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

#### Usage Model

The only usage model for the SSC Peripheral Library is to use a Interrupt/DMA model. Therefore many functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

**Example Applications:**

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`

## Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

### a) Initialization Function

| | Name | Description |
|---|---|---|
| | SSC_Initialize | Initializes SSC module of the device |

### b) Transaction Functions

| | Name | Description |
|---|---|---|
| | SSC_BaudSet | Changes the baud rate (samples/second) of the interface. |
| | SSC_LRCLK_Get | Get the level of the I2S LRCLK (left/right clock) signal |

### *a) Initialization Function*

#### SSC_Initialize Function

void SSC_Initialize (void);

### Summary

Initializes SSC module of the device

### Description

This function initializes SSC module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

### Preconditions

MHC GUI should be configured with the right values.

### Returns

None.

### Remarks

This function must be called only once and before any other SSC function is called.

### Example

```
SSC_Initialize();
```

### C

```
void SSC_Initialize();
```

### *b) Transaction Functions*

### SSC_BaudSet Function

void SSC_BaudSet (const uint32_t baud);

#### Summary

Changes the baud rate (samples/second) of the interface.

#### Description

This function Cchanges the baud rate, or samples/second of the interface.

#### Preconditions

None.

#### Returns

None.

#### Remarks

None.

#### Example

```
SSC_BaudSet(44100);
```

#### C

```
void SSC_BaudSet(const uint32_t baud);
```

### SSC_LRCLK_Get Function

uint32_t SSC_LRCLK_Get(void);

#### Summary

Get the level of the I2S LRCLK (left/right clock) signal

#### Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize itself to the LRCLK signal.

#### Preconditions

None.

#### Returns

State of the LRCLK pin for the SSC module -- 1 if high, 0 if low if the audio format is I2S, and 0 if high, 1 if low if the format is Left Jusutifed

#### Remarks

None.

#### Example

```
SSC_LRCLK_Get();
```

#### C

```
uint32_t SSC_LRCLK_Get();
```

# Files

## Files

| Name | Description |
|---|---|
| plib_ssc_6078.h | SSC PLIB Header File for documentation |

## Description

### *plib_ssc_6078.h*

plib_ssc_6078.h

## Summary

SSC PLIB Header File for documentation

## Description

SSC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Synchronous Controller (SSC). This file must not be included in any MPLAB Project.

# Audio Libraries Help

## Decoder Library Help

This section provides descriptions of the software Decoder Libraries that are available in MPLAB Harmony.

## ADPCM Decoder Library

### *Introduction*

Adaptive Differential Pulse Code Modulation (ADPCM) is a royalty-free audio codec. It is a lossy compression standard. It is a variant of Differential Pulse-Code Modulation (DPCM). The file format used to contain an ADPCM files, is the same as WAV. Decoding is fast and requires little processing power.

Typical applications using ADPCM are for speech. This implementation currently supports 16 bit stereo.

### *Using the Library*

### *Library Interface*

#### Functions

| | Name | Description |
|---|---|---|
| | ADPCM_Initialize | This is function ADPCM_Initialize. |
| | ADPCM_Decoder | Decode the ADPCM stream of inSize from input pointer into output pointer. |
| | ADPCM_GetChannels | This is function ADPCM_GetChannels. |
| | ADPCM_Decoder_ConfigByteOrder | Configure input ADPCM stream byte order. |
| | ADPCM_HdrGetFormat | This is function ADPCM_HdrGetFormat. |
| | ADPCM_HdrGetBitsPerSample | This is function ADPCM_HdrGetBitsPerSample. |
| | ADPCM_HdrGetDataLen | This is function ADPCM_HdrGetDataLen. |
| | ADPCM_HdrGetFileSize | This is function ADPCM_HdrGetFileSize. |
| | ADPCM_HdrGetBlockAlign | This is function ADPCM_HdrGetBlockAlign. |
| | ADPCM_HdrGetBytesPerSec | This is function ADPCM_HdrGetBytesPerSec. |
| | ADPCM_HdrGetNumOfChan | This is function ADPCM_HdrGetNumOfChan. |
| | ADPCM_HdrGetSamplesPerSec | This is function ADPCM_HdrGetSamplesPerSec. |

#### Data Types and Constants

| | Name | Description |
|---|---|---|
| | ADPCMHEADER | ADPCM WAV File Header |
| | ADPCM_HEADER_SIZE | WAV Header Size |

#### Description

This section describes the Application Programming Interface (API) functions of the ADPCM Decoder Library.

## Functions

### *ADPCM_Initialize Function*

This is function ADPCM_Initialize.

**C**

```
void ADPCM_Initialize(uint8_t * input);
```

### *ADPCM_Decoder Function*

bool ADPCM_Decoder
(uint8_t *input,
uint16_t inSize,
uint16_t *read,
int16_t *output,
uint16_t *written);

**Summary**

Decode the ADPCM stream of inSize from input pointer into output pointer.

**Description**

Decode the ADPCM stream of inSize from input pointer into output pointer.

**Parameters**

| Parameters | Description |
|---|---|
| inSize | Size of ADPCM input stream in bytes. |
| read | Size of ADPCM decoder has decoded after this function complete |

**Returns**

Pointer to the decoded data. written - Size of decoded data.

If successful, return true, otherwise, return false.

**C**

```
bool ADPCM_Decoder(uint8_t * input, uint16_t inSize, uint16_t * read, int16_t * output,
uint16_t * written);
```

### *ADPCM_GetChannels Function*

This is function ADPCM_GetChannels.

**C**

```
uint8_t ADPCM_GetChannels();
```

### *ADPCM_Decoder_ConfigByteOrder Function*

void ADPCM_Decoder_ConfigByteOrder (bool isLE);

### Summary

Configure input ADPCM stream byte order.

### Description

Configure input ADPCM stream byte order.

### Parameters

| Parameters | Description |
|---|---|
| isLE | True if ADPCM stream is in little endian format, otherwise, false. |

### Returns

None.

### C

```
void ADPCM_Decoder_ConfigByteOrder(bool isLE);
```

## *ADPCM_HdrGetFormat Function*

This is function ADPCM_HdrGetFormat.

### C

```
int ADPCM_HdrGetFormat();
```

## *ADPCM_HdrGetBitsPerSample Function*

This is function ADPCM_HdrGetBitsPerSample.

### C

```
int ADPCM_HdrGetBitsPerSample();
```

## *ADPCM_HdrGetDataLen Function*

This is function ADPCM_HdrGetDataLen.

### C

```
int ADPCM_HdrGetDataLen();
```

## *ADPCM_HdrGetFileSize Function*

This is function ADPCM_HdrGetFileSize.

### C

```
unsigned int ADPCM_HdrGetFileSize();
```

## *ADPCM_HdrGetBlockAlign Function*

This is function ADPCM_HdrGetBlockAlign.

### C

```
int ADPCM_HdrGetBlockAlign();
```

### *ADPCM_HdrGetBytesPerSec Function*

This is function ADPCM_HdrGetBytesPerSec.

**C**

```
int ADPCM_HdrGetBytesPerSec();
```

### *ADPCM_HdrGetNumOfChan Function*

This is function ADPCM_HdrGetNumOfChan.

**C**

```
int ADPCM_HdrGetNumOfChan();
```

### *ADPCM_HdrGetSamplesPerSec Function*

This is function ADPCM_HdrGetSamplesPerSec.

**C**

```
int ADPCM_HdrGetSamplesPerSec();
```

## Data Types and Constants

### *ADPCMHEADER Type*

ADPCM WAV File Header

**Description**

ADPCM Audio File Header Structure

This structure is header structure of wav container.

**C**

```
typedef struct ADPCMHEADER@1 ADPCMHEADER;
```

### *ADPCM_HEADER_SIZE Macro*

WAV Header Size

**Description**

ADPCM WAV File Header Size

WAV Header Size

**C**

```
#define ADPCM_HEADER_SIZE 44
```

### *Files*

## Files

| Name | Description |
|---|---|
| adpcm_dec.h | This is file adpcm_dec.h. |

## Description

### adpcm_dec.h

This is file adpcm_dec.h.

# WAV Decoder Library

### *Introduction*

### *Using the Library*

### *Library Interface*

## Functions

| | Name | Description |
|---|---|---|
| | WAV_Initialize_N | This is function WAV_Initialize_N. |
| | WAV_Decoder | This is function WAV_Decoder. |
| | WAV_GetChannels | This is function WAV_GetChannels. |
| | WAV_UpdatePlaytime | This is function WAV_UpdatePlaytime. |
| | WAV_GetDuration | This is function WAV_GetDuration. |
| | WAV_HdrGetDataLen | This is function WAV_HdrGetDataLen. |
| | WAV_HdrGetFileSize | This is function WAV_HdrGetFileSize. |
| | WAV_GetSampleRate | This is function WAV_GetSampleRate. |
| | WAV_GetBitRate | This is function WAV_GetBitRate. |
| | WAV_HdrGetBytesPerSec | This is function WAV_HdrGetBytesPerSec. |
| | WAV_HdrGetSamplesPerSec | This is function WAV_HdrGetSamplesPerSec. |
| | WAV_GetAudioSize | This is function WAV_GetAudioSize. |

## Data Types and Constants

| | Name | Description |
|---|---|---|
| | WAV_HEADER_SIZE | This is macro WAV_HEADER_SIZE. |
| | WAV_HEAP_SIZE | This is macro WAV_HEAP_SIZE. |

| | | |
|---|---|---|
| | WAVE_FORMAT_ALAW | This is macro WAVE_FORMAT_ALAW. |
| | WAVE_FORMAT_MULAW | This is macro WAVE_FORMAT_MULAW. |
| | WAVE_FORMAT_PCM | This is macro WAVE_FORMAT_PCM. |
| | WAV_INPUT_BUFFER_SIZE | This is macro WAV_INPUT_BUFFER_SIZE. |
| | WAV_OUTPUT_BUFFER_SIZE | This is macro WAV_OUTPUT_BUFFER_SIZE. |
| | dWAVHEADER | This is type dWAVHEADER. |
| | WAV_DEC | This is type WAV_DEC. |

## Description

## Functions

### *WAV_Initialize_N Function*

This is function WAV_Initialize_N.

**C**

```
void WAV_Initialize_N(uint8_t * input, SYS_FS_HANDLE wavFilehandle);
```

### *WAV_Decoder Function*

This is function WAV_Decoder.

**C**

```
bool WAV_Decoder(uint8_t * input, uint16_t inSize, uint16_t * read, int16_t * output, uint16_t
* written);
```

### *WAV_GetChannels Function*

This is function WAV_GetChannels.

**C**

```
uint8_t WAV_GetChannels();
```

### *WAV_UpdatePlaytime Function*

This is function WAV_UpdatePlaytime.

**C**

```
uint32_t WAV_UpdatePlaytime();
```

### *WAV_GetDuration Function*

This is function WAV_GetDuration.

**C**

```
uint32_t WAV_GetDuration();
```

### *WAV_HdrGetDataLen Function*

This is function WAV_HdrGetDataLen.

**C**

```
int WAV_HdrGetDataLen();
```

### *WAV_HdrGetFileSize Function*

This is function WAV_HdrGetFileSize.

**C**

```
unsigned int WAV_HdrGetFileSize();
```

### *WAV_GetSampleRate Function*

This is function WAV_GetSampleRate.

**C**

```
uint32_t WAV_GetSampleRate();
```

### *WAV_GetBitRate Function*

This is function WAV_GetBitRate.

**C**

```
uint32_t WAV_GetBitRate();
```

### *WAV_HdrGetBytesPerSec Function*

This is function WAV_HdrGetBytesPerSec.

**C**

```
int WAV_HdrGetBytesPerSec();
```

### *WAV_HdrGetSamplesPerSec Function*

This is function WAV_HdrGetSamplesPerSec.

**C**

```
int WAV_HdrGetSamplesPerSec();
```

### *WAV_GetAudioSize Function*

This is function WAV_GetAudioSize.

**C**

```
uint32_t WAV_GetAudioSize();
```

## Data Types and Constants

### *WAV_HEADER_SIZE Macro*

This is macro WAV_HEADER_SIZE.

**C**

```
#define WAV_HEADER_SIZE 44
```

### *WAV_HEAP_SIZE Macro*

This is macro WAV_HEAP_SIZE.

**C**

```
#define WAV_HEAP_SIZE 1024
```

### *WAVE_FORMAT_ALAW Macro*

This is macro WAVE_FORMAT_ALAW.

**C**

```
#define WAVE_FORMAT_ALAW 0x0006
```

### *WAVE_FORMAT_MULAW Macro*

This is macro WAVE_FORMAT_MULAW.

**C**

```
#define WAVE_FORMAT_MULAW 0x0007
```

### *WAVE_FORMAT_PCM Macro*

This is macro WAVE_FORMAT_PCM.

**C**

```
#define WAVE_FORMAT_PCM 0x0001
```

### *WAV_INPUT_BUFFER_SIZE Macro*

This is macro WAV_INPUT_BUFFER_SIZE.

**C**

```
#define WAV_INPUT_BUFFER_SIZE
(DECODER_MAX_OUTPUT_BUFFER_SIZE>DECODER_MAX_INPUT_BUFFER_SIZE?DECODER_MAX_INPUT_BUFFER_SIZE:DECO
DER_MAX_OUTPUT_BUFFER_SIZE)
```

### *WAV_OUTPUT_BUFFER_SIZE Macro*

This is macro WAV_OUTPUT_BUFFER_SIZE.

**C**

```
#define WAV_OUTPUT_BUFFER_SIZE (WAV_INPUT_BUFFER_SIZE)
```

### *dWAVHEADER Type*

This is type dWAVHEADER.

**C**

```
typedef struct dWAVHEADER@1 dWAVHEADER;
```

### *WAV_DEC Type*

This is type WAV_DEC.

**C**

```
typedef struct WAV_DEC@1 WAV_DEC;
```

### *Files*

**Files**

| Name | Description |
|---|---|
| wav_dec.h | This is file wav_dec.h. |

**Description**

### wav_dec.h

This is file wav_dec.h.

# Encoder Library Help

This section provides descriptions of the software Encoder Libraries that are available in MPLAB Harmony.

## ADPCM Ecncoder Library

### *Introduction*

Adaptive Differential Pulse Code Modulation (ADPCM) is a royalty-free audio codec. It is a lossy compression standard. It is a variant of Differential Pulse-Code Modulation (DPCM). The file format used to contain an ADPCM files, is the same as WAV.

Its popularity comes because it achieves compression of 4 to 1 but does not require high processing power.

Typical applications using ADPCM are for speech. This implementation currently supports 16 bit stereo.

### *Using the Library*

### *Library Interface*

## Functions

| | Name | Description |
|---|---|---|
| | adpcm_encoder_init | This is function adpcm_encoder_init. |
| | adpcm_encode_frame | This is function adpcm_encode_frame. |
| | adpcm_encoder_config_byteorder | This is function adpcm_encoder_config_byteorder. |
| | adpcm_encoder_free | This is function adpcm_encoder_free. |

## Description

### Functions

### *adpcm_encoder_init Function*

This is function adpcm_encoder_init.

**C**

```
bool adpcm_encoder_init();
```

### *adpcm_encode_frame Function*

This is function adpcm_encode_frame.

**C**

```
bool adpcm_encode_frame(void * pin, uint32_t insize, void * pout, uint32_t * outsize);
```

### *adpcm_encoder_config_byteorder Function*

This is function adpcm_encoder_config_byteorder.

**C**

```
void adpcm_encoder_config_byteorder(bool isLE);
```

### *adpcm_encoder_free Function*

This is function adpcm_encoder_free.

**C**

```
bool adpcm_encoder_free();
```

### Data Types and Constants

#### *Files*

## Files

| Name | Description |
|---|---|
| adpcm_enc.h | Contains the ADPCM encoder specific defintions and function prototypes. |

## Description

### adpcm_enc.h

adpcm_enc.h

## Summary

Contains the ADPCM encoder specific defintions and function prototypes.

## Description

ADPCM Decoder Header File

This file contains the ADPCM encoder specific defintions and function prototypes.

# PCM Encoder Library

#### *Introduction*

The PCM encoder is a royalty-free audio codec. It is a lossless standard and there is no compression.

It can typically be used to share speech; share music; create files to playback by older/slower processors. This implementation currently supports 16 bit stereo. The sample rate can vary widely.

#### *Using the Library*

#### *Library Interface*

## Functions

| | Name | Description |
|---|---|---|
|  | pcm_encoder_init | This is function pcm_encoder_init. |

| | | |
|---|---|---|
| | pcm_encode_frame | @Function int ExampleFunctionName ( int param1, int param2 )<br>@Summary Brief one-line description of the function.<br>@Description Full description, explaining the purpose and usage of the function.<br>Additional description in consecutive paragraphs separated by HTML paragraph breaks, as necessary.<br>Type "JavaDoc" in the "How Do I?" IDE toolbar for more information on tags.<br>@Precondition List and describe any required preconditions. If there are no preconditions, enter "None."<br>@Parameters @param param1 Describe the first parameter to the function.<br>@param param2 Describe the second parameter to the function.<br>@Returns List (if feasible) and describe the return values of the function. 1 Indicates... more |
| | pcm_encoder_free | This is function pcm_encoder_free. |

## Description

## Functions

### *pcm_encoder_init Function*

This is function pcm_encoder_init.

**C**

```
bool pcm_encoder_init(int channel, int inputSampleRate);
```

### *pcm_encode_frame Function*

@Function int ExampleFunctionName ( int param1, int param2 )

@Summary Brief one-line description of the function.

@Description Full description, explaining the purpose and usage of the function.

Additional description in consecutive paragraphs separated by HTML paragraph breaks, as necessary.

Type "JavaDoc" in the "How Do I?" IDE toolbar for more information on tags.

@Precondition List and describe any required preconditions. If there are no preconditions, enter "None."

@Parameters @param param1 Describe the first parameter to the function.

@param param2 Describe the second parameter to the function.

@Returns List (if feasible) and describe the return values of the function. 1 Indicates an error occurred 0 Indicates an error did not occur  @Remarks Describe any special behavior not described above.

Any additional remarks.

@Example @code if(ExampleFunctionName(1, 2) == 0) { return 3;

**C**

```
bool pcm_encode_frame(void * pin, uint32_t insize, void * pout, uint32_t * outsize);
```

### *pcm_encoder_free Function*

This is function pcm_encoder_free.

**C**

```
bool pcm_encoder_free();
```

## Data Types and Constants

### *Files*

## Files

| Name | Description |
|---|---|
| pcm_enc.h | • WAV Encoder Header File<br>@Company Company Name<br>@File Name wav_enc.h<br>@Summary Brief description of the file.<br>@Description Describe the purpose of this file. |

## Description

### pcm_enc.h

- WAV Encoder Header File

@Company Company Name

@File Name wav_enc.h

@Summary Brief description of the file.

@Description Describe the purpose of this file.

# Index