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 hardware codecs that can be connected to development boards, such as the WM8904 Codec Daughterboard.

Finally, it contains a BSP Audio Template 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 Codec Driver is in the `audio/driver` folder. The Audio Template is in the `audio/templates` folder.

Important! This repository only contains the files for the audio applications, 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.

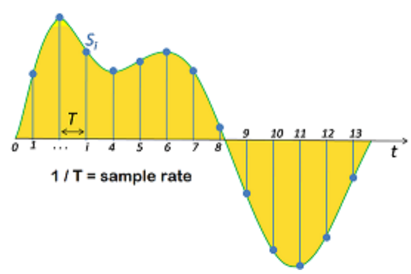
Digital Audio Basics

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.

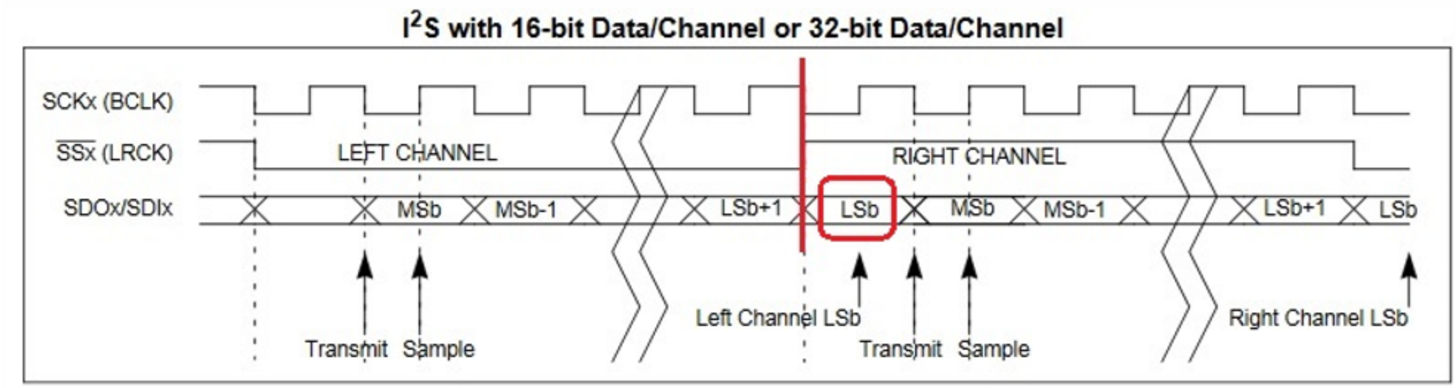
The bit clock rate = sample rate * # of channels * # of bits / channel

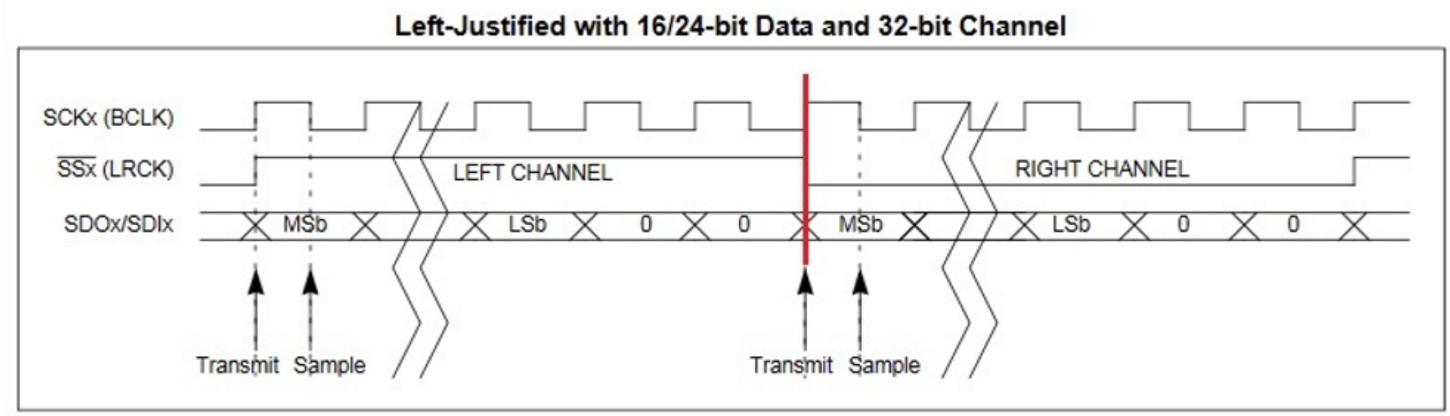
e.g. for CD audio, 44.1 kHz * 2 channels (stereo) * 16 bits/channel = 1.4112 MHz

- 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):

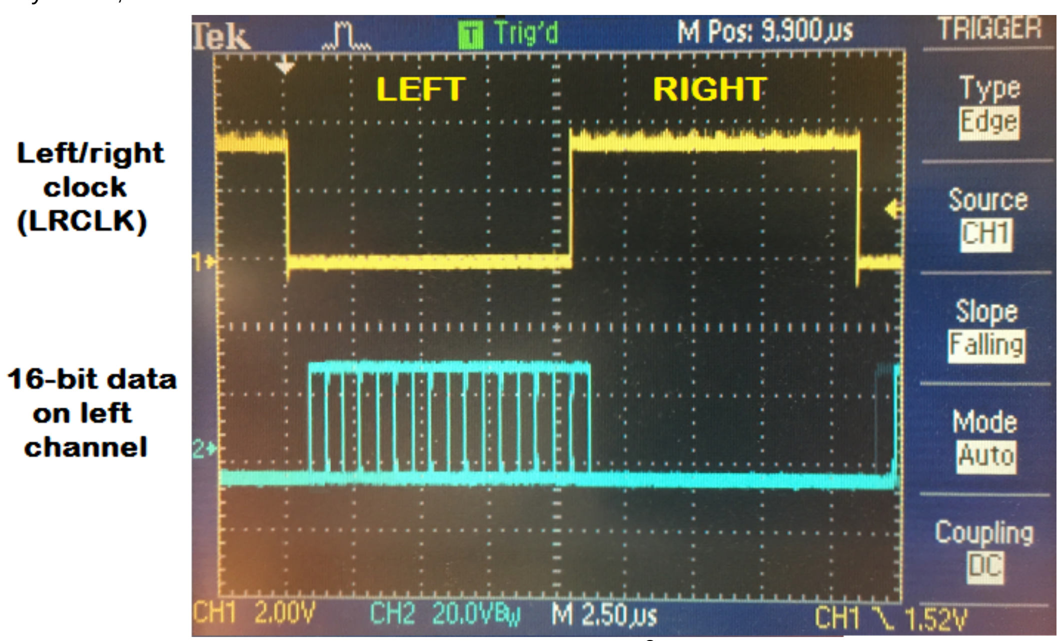




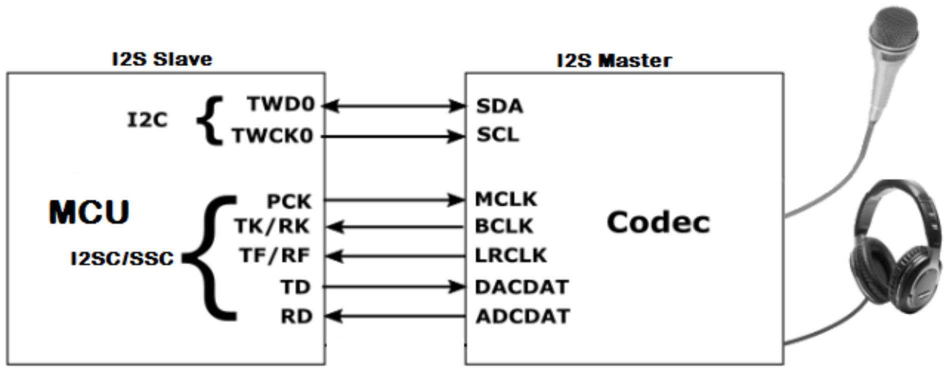
In I^{2}S format, as shown in the top half of the figure above, LRCLK is low for the left channel and high for the right channel. The most-significant bit (MSb) of the left channel data starts one bit clock (BCLK) late, such that the least significant bit (LSb) is actually in the right channel side.

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.

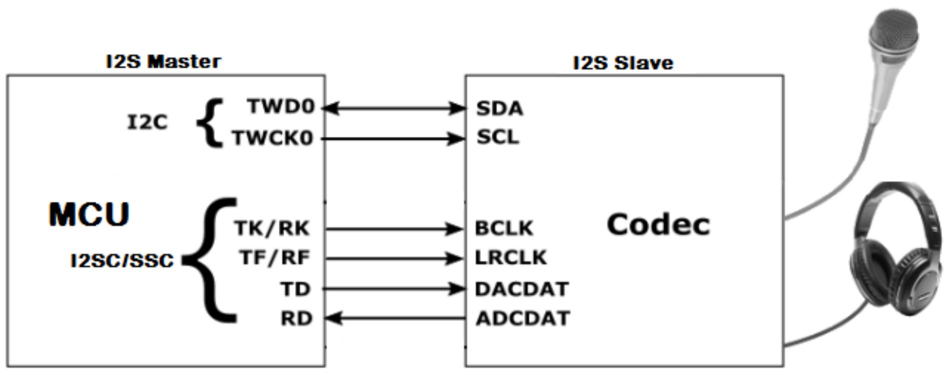
Below is a oscilloscope photo showing the LRCLK in yellow, and data in blue. Only the left channel has audio. The overlap in the I^{2}S format is clearly visible, as well as the 16 individual data bits.



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::

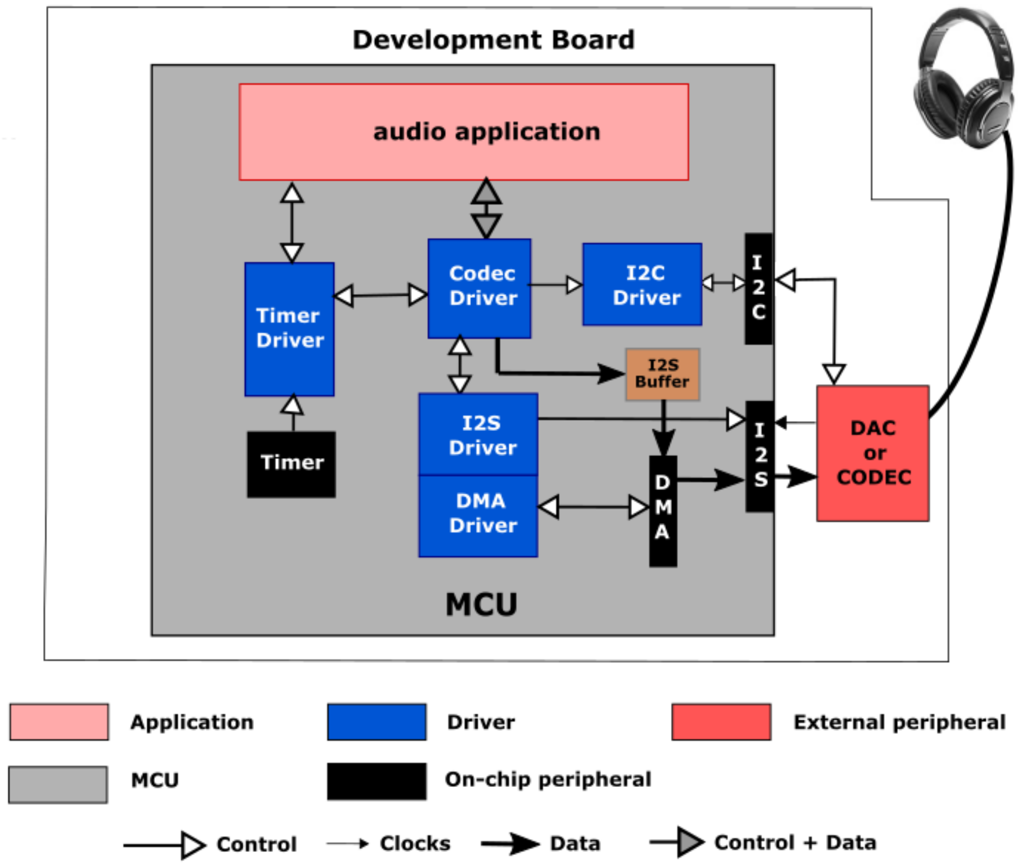


Digital Audio in Harmony

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^{2}S-capable PLIB, which may be implemented using an SPI, SPI, 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^{2}S-capable peripheral.

Creating Your First Audio Application from Scratch

This topic provides a user a quick start guide for building their first I2S audio project in Harmony 3.

Description

This tutorial provides information on how to create an audio project using MPLAB Harmony 3. To keep the application as simple as possible, no graphics or other UI are included. It simply outputs a 1000 Hz sine wave over the I2S interface at 48,000 samples/sec that generates a 1 kHz sine tone. The sine tone is generated internally using math functions (no audio inputs).

The following hardware setup is supported:

- SAM E70 Xplained Ultra Evaluation Kit
- PIC32 Audio Codec Daughter Card
- Typical 3.5 mm stereo headphones (purchased separately)

Getting Started

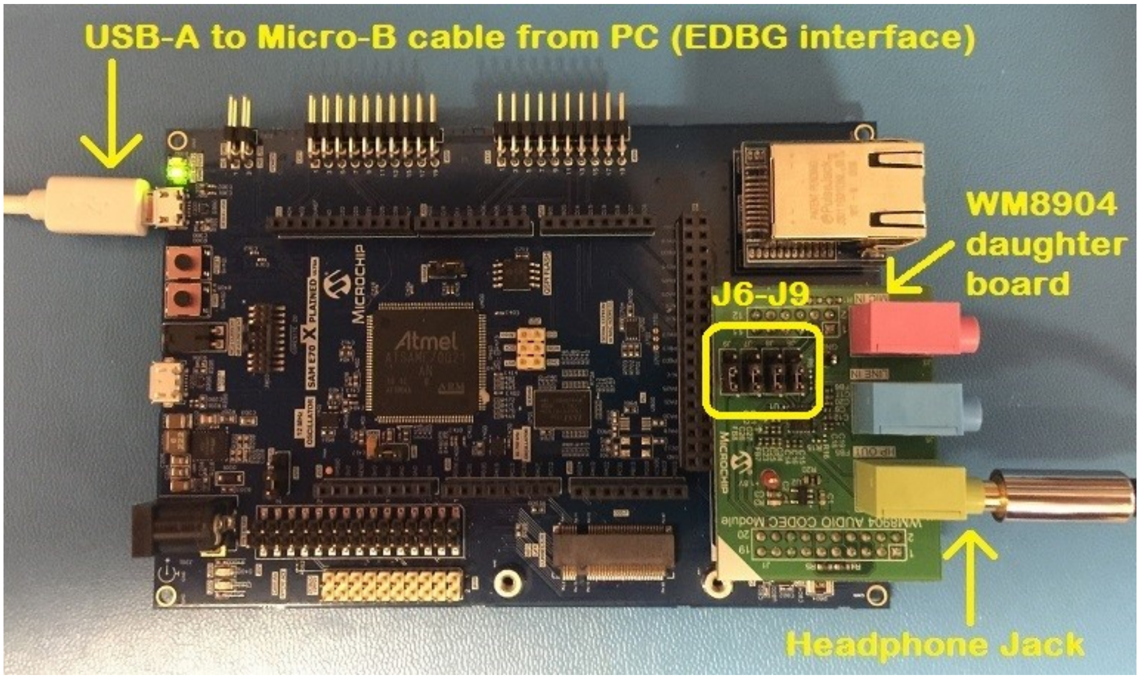
Before beginning this tutorial, ensure that the MPLAB X IDE is installed and necessary language tools as described in Getting Started With MPLAB Harmony. In addition, ensure that the MPLAB Harmony dependent repos are installed on a local hard drive and that the correct version of MHC plug-in is installed within the MPLAB X IDE.

Follow the tutorial found in MPLAB Harmony Core Library Help > Creating Your First Project to first learn about the basics of software development using MPLAB Harmony. Successfully completing the tutorial can serve as a confirmation of the software setup.

Configuring the Hardware

SAM E70 Xplained Ultra board

The board is programmed via an Embedded Debugger (EDBG) interface via a USB cable, connected on one end to your computer using a USB-A plug, and to the E70 XULT board using a Micro-B plug as shown below:



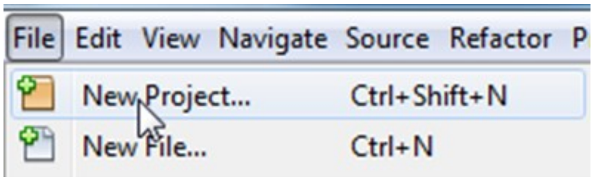
In order to hear the audio, a codec is required. In this example, we are using a Wolfson WM8904 codec, mounted on a daughter board that plugs into the XC32 header of the E70 XULT board. A headphone is then plugged into the headphone jack of the daughter board.

Make sure the jumpers on headers J6-J9 are positioned as shown (towards the front of the board).

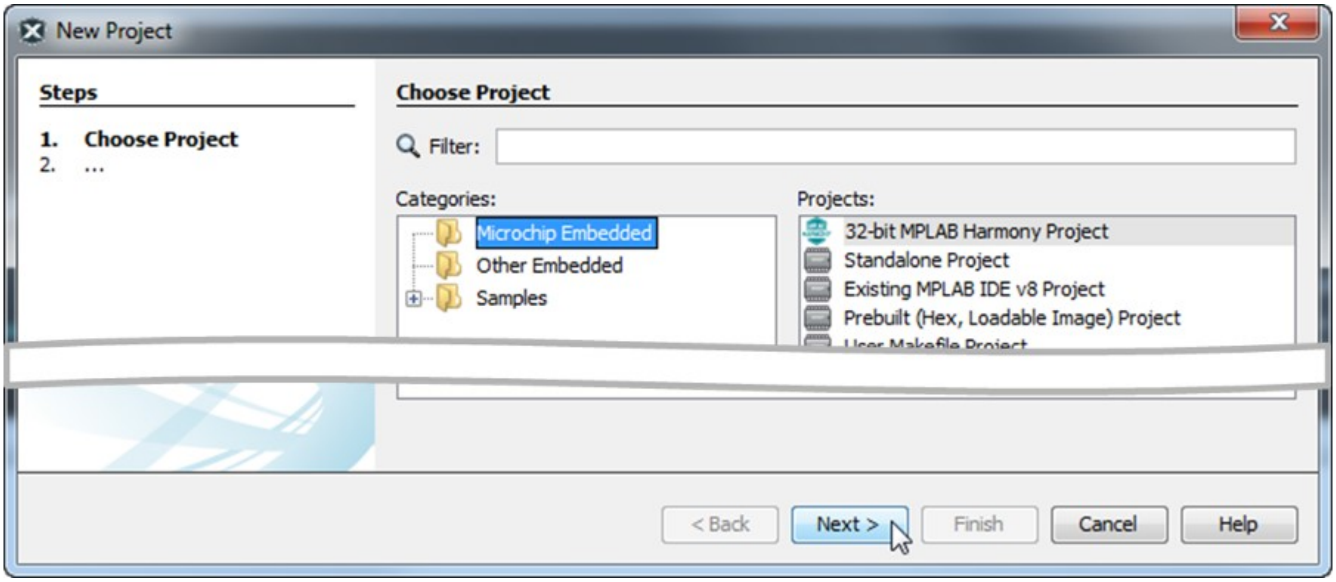
Tutorial Steps

Creating a New Project

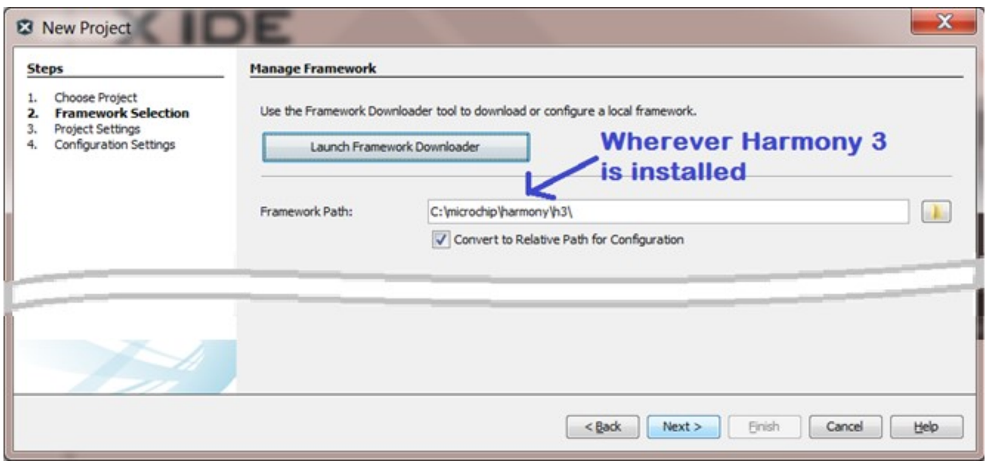
1. Launch the MPLAB X IDE. From the File pull-down menu, select New Project. (This will bring up the New Project dialog window.)



2. On the New Project dialog window, be sure the project type is 32-bit MPLAB Harmony Project, then hit the Next > button.

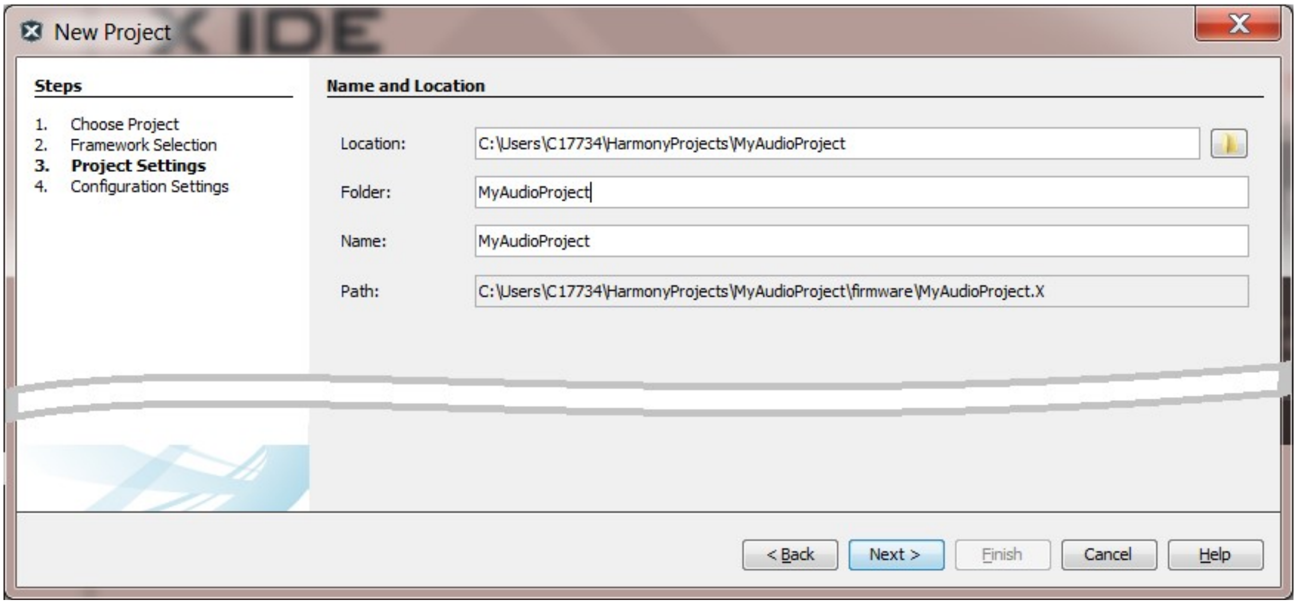


3. In the New Project dialog window, be sure the Harmony Path is pointing to the MPLAB Harmony framework installation. The New Project dialog should show the following:



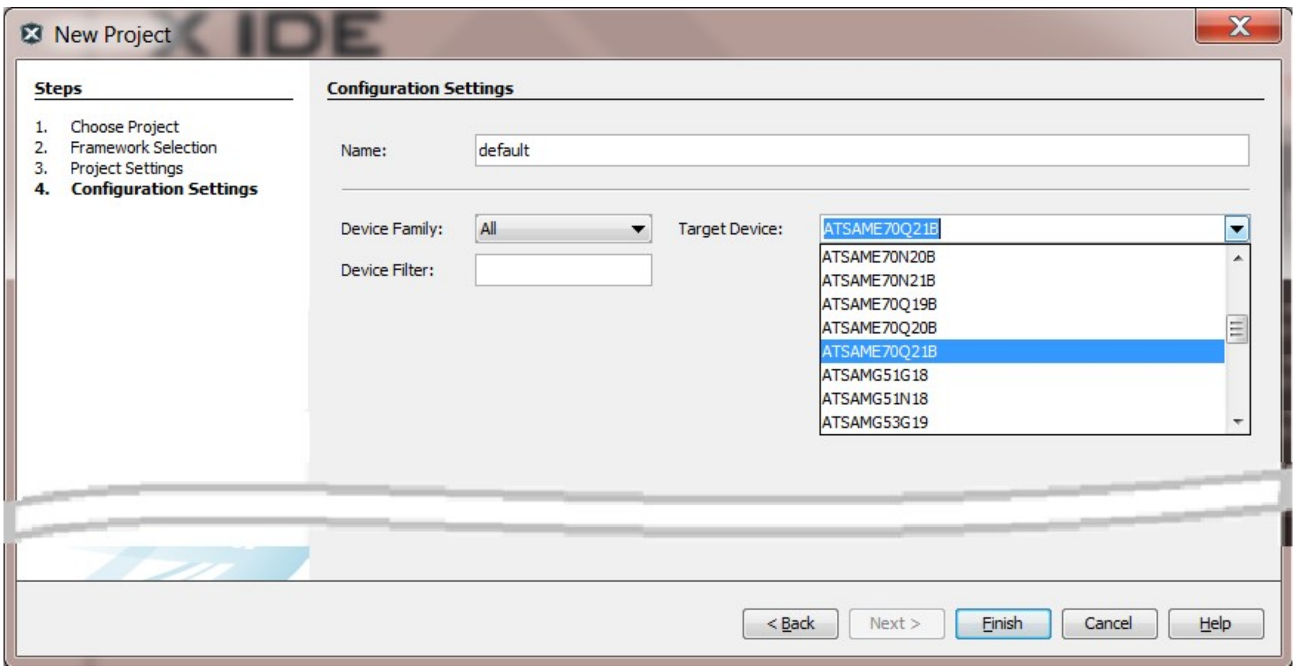
Hit the **Next >** button

4. For the project name, type in MyAudioProject or some other name.



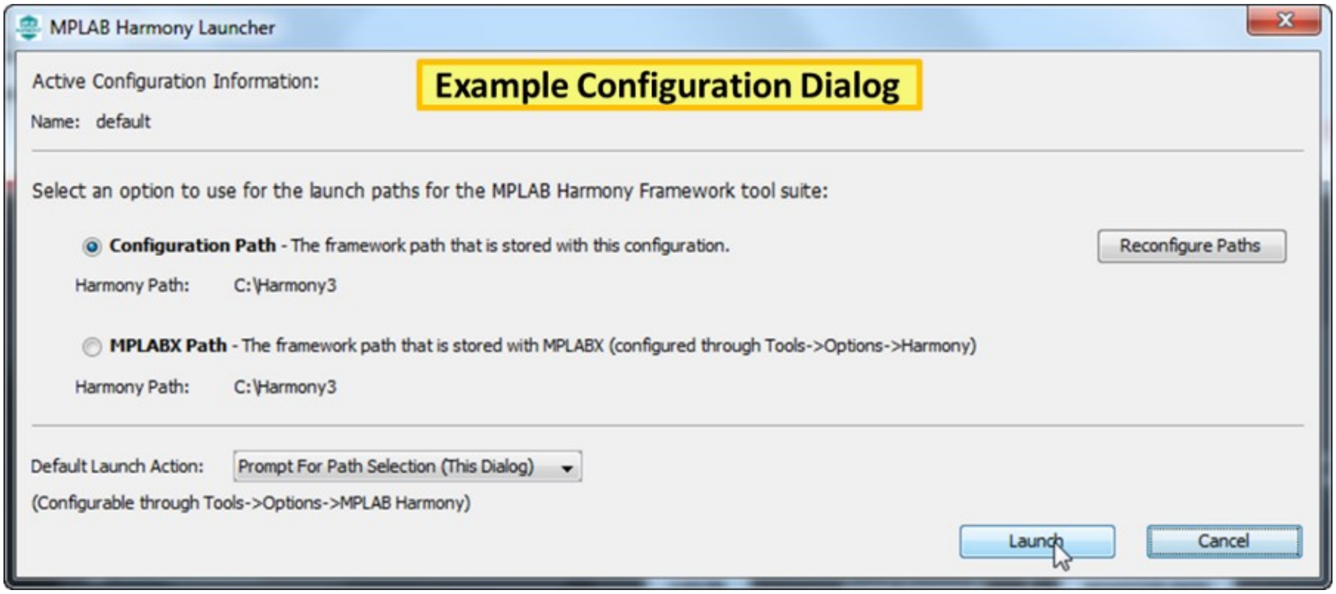
Hit the **Next >** button

5. For the SAM E70 Xplained Ultra board, select the ATSAME70Q21B as the target device.



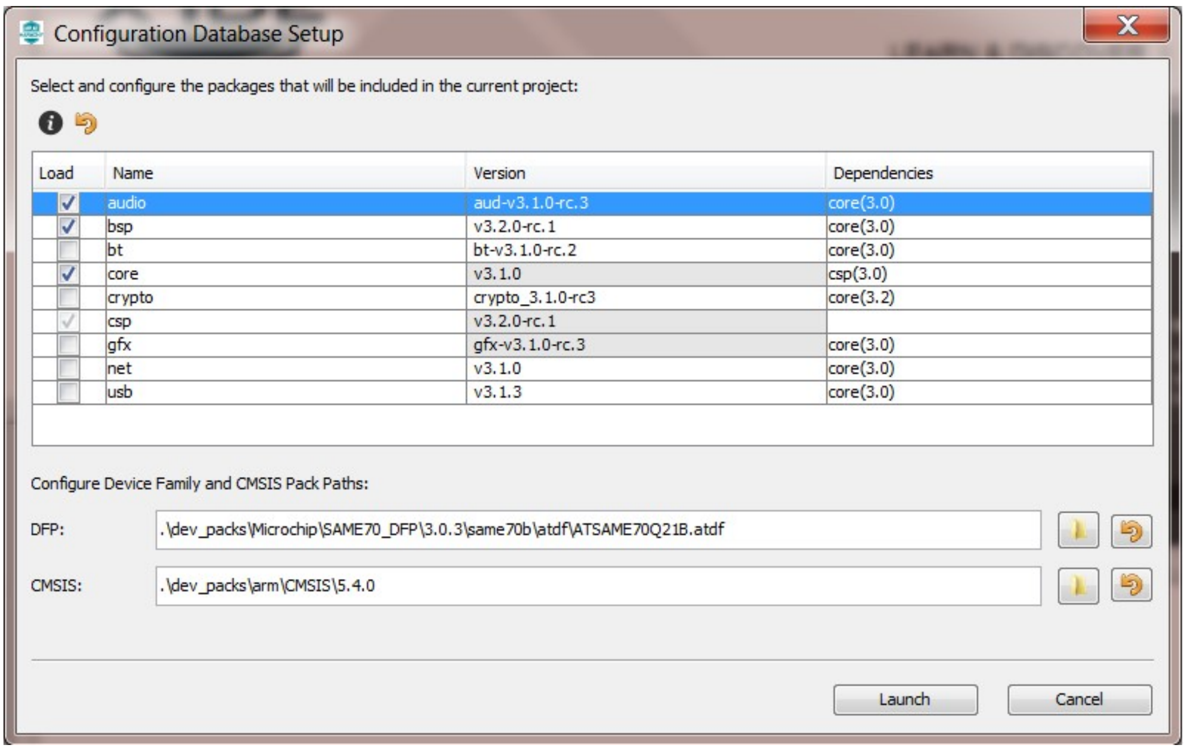
Select **Finish** when done.

6. In the MPLAB Harmony Launcher dialog window select the Launch Path that corresponds to the Harmony installation to be used. In the example below, both MPLAB X IDE's default configuration and the project's path point to the same MPLAB Harmony installation:

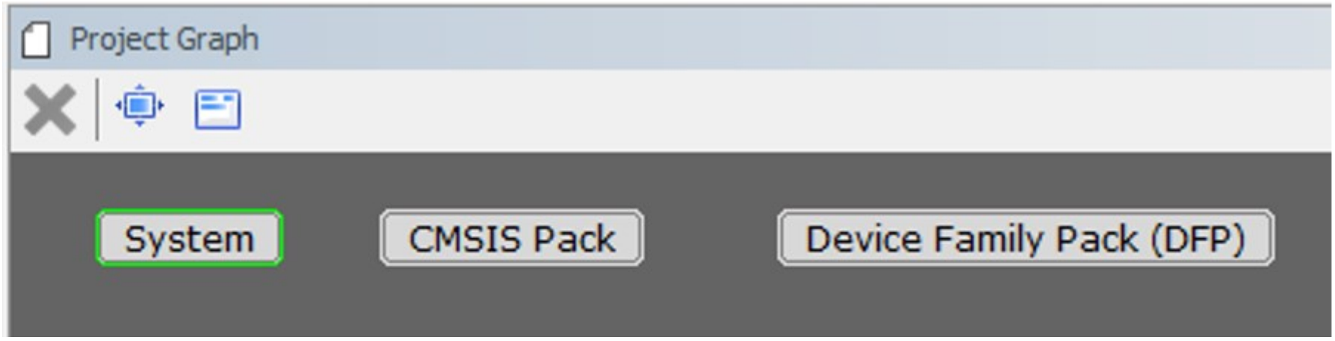


then select the **Launch** button.

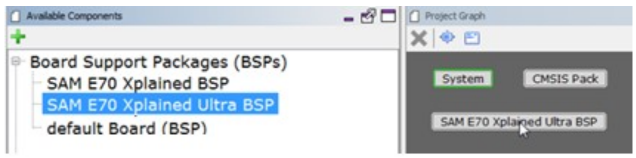
7. When the “Configuration Database Setup” dialog window appears, make sure **audio**, **bsp**, **core** and **csp** are selected:



then select the **Launch** button. MHC’s component graph should appear:



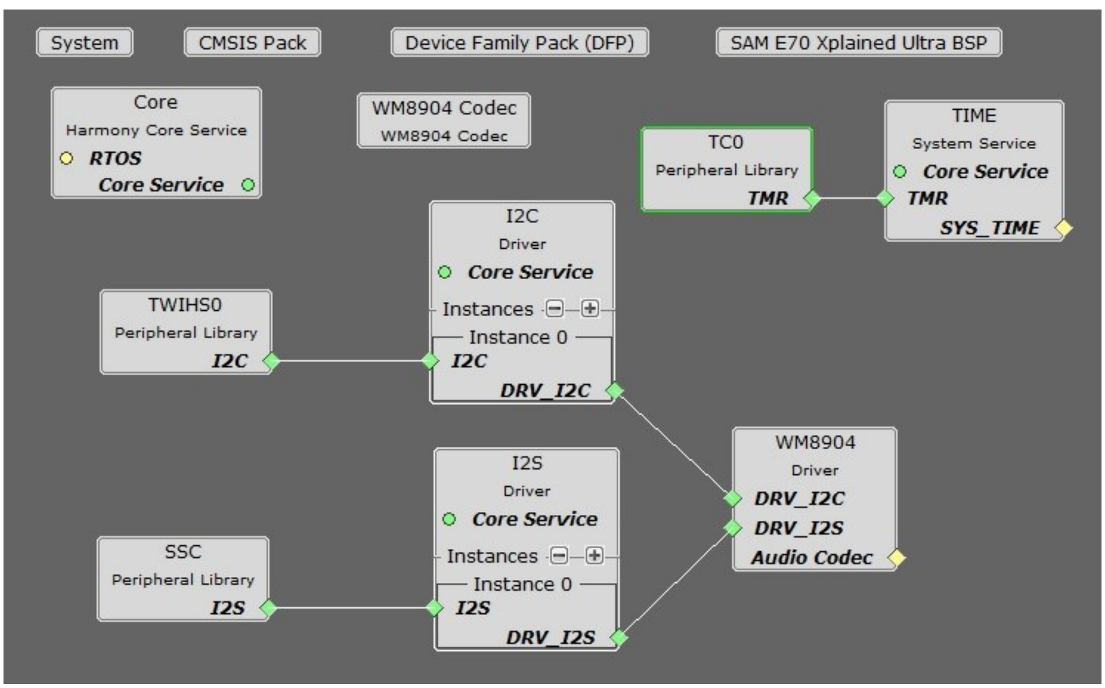
8. In the MHC's Available Components panel drag and drop the SAM E70 Xplained Ultra BSP onto the project's Component Graph



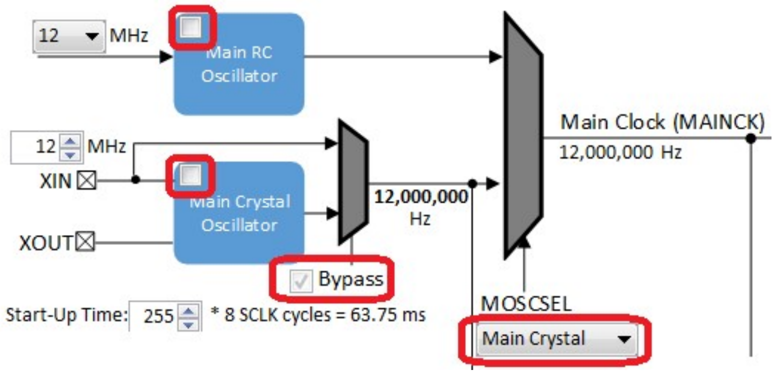
9. In Available Components, under Audio>Templates, double-click on the desired codec, e.g. WM8904. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one.

Besides instantiating all the necessary components (drivers and PLIBs), picking a base BSP plus an audio template also configures all the pins for that board needed by the application.

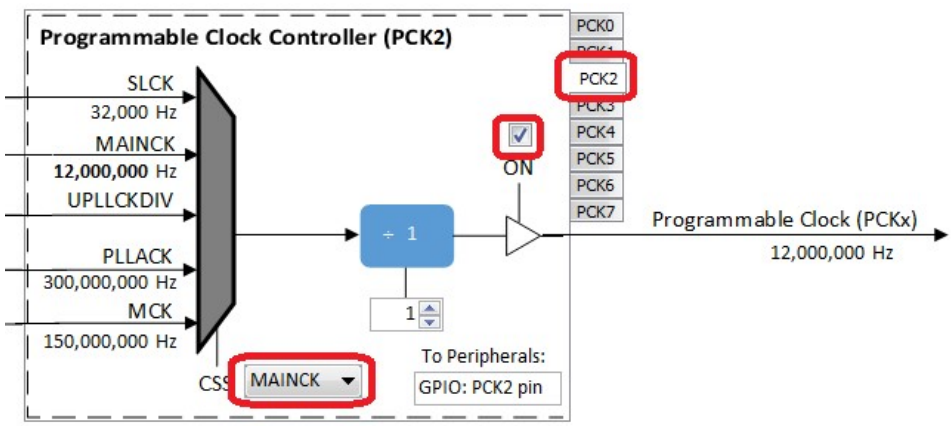
The result will be a project graph that looks like this, after rearranging the boxes,



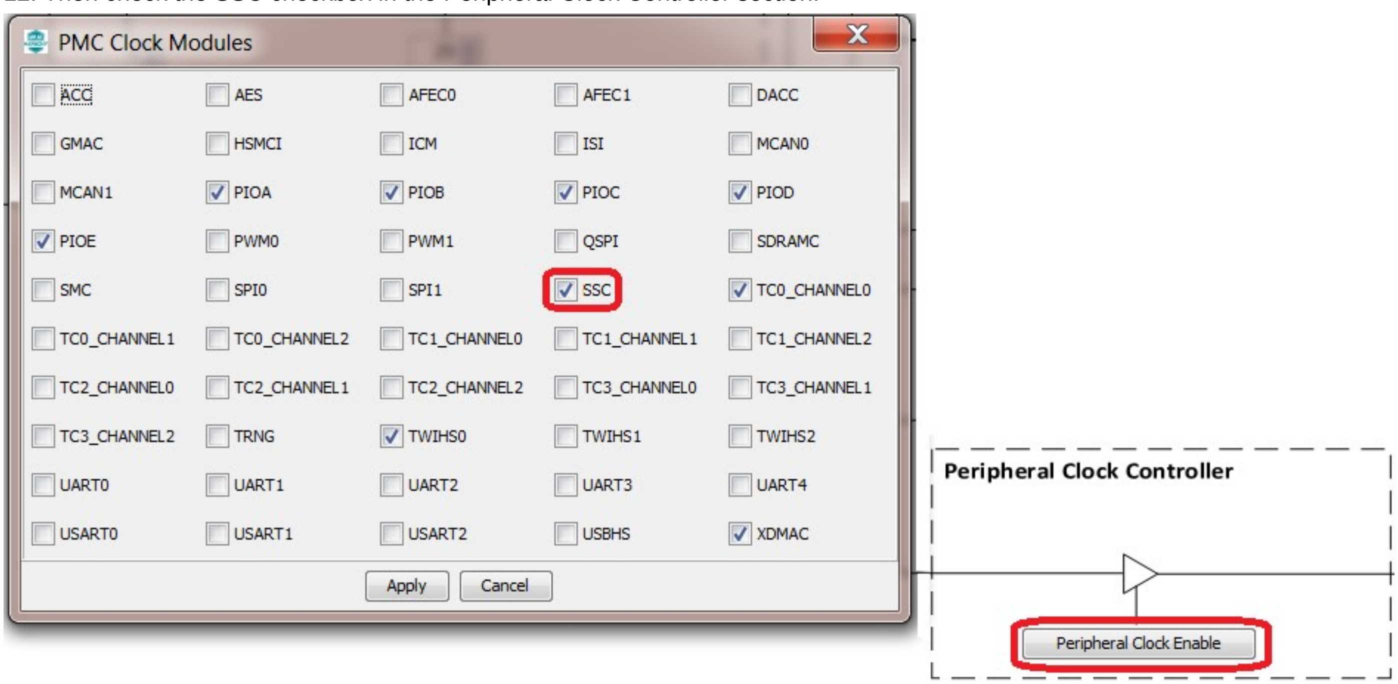
10. In the menu bar, select Tools>Clock Configuration. In the Clock Diagram that pops up, set MOSCEL to Main Crystal, check the Bypass checkbox, and uncheck the RC Crystal Oscillator and Main Crystal Oscillator boxes, to make use the 12 MHz resonator on the board:



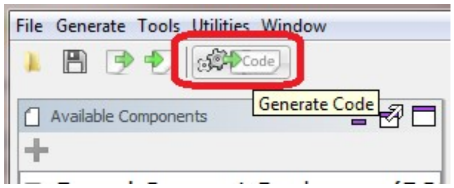
11. Also in the Clock Diagram, in the PCK2 tab of the Programmable Clock Controller section, check the On checkbox, and set CSS to MAINCLK (12 MHz).



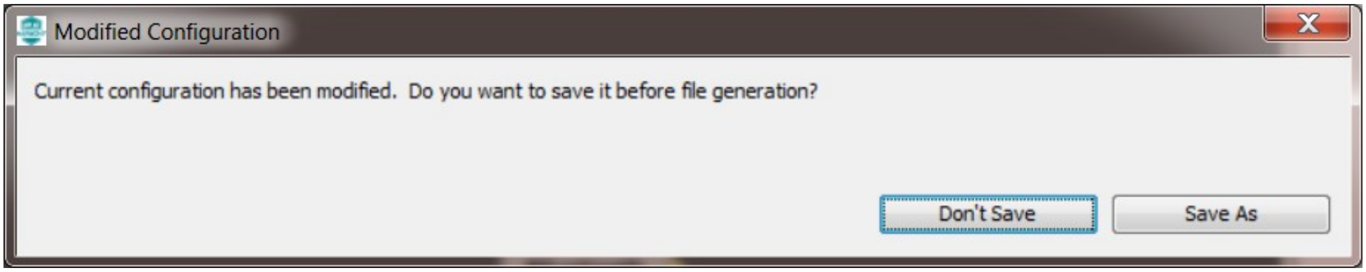
12. Then check the SSC checkbox in the Peripheral Clock Controller section.



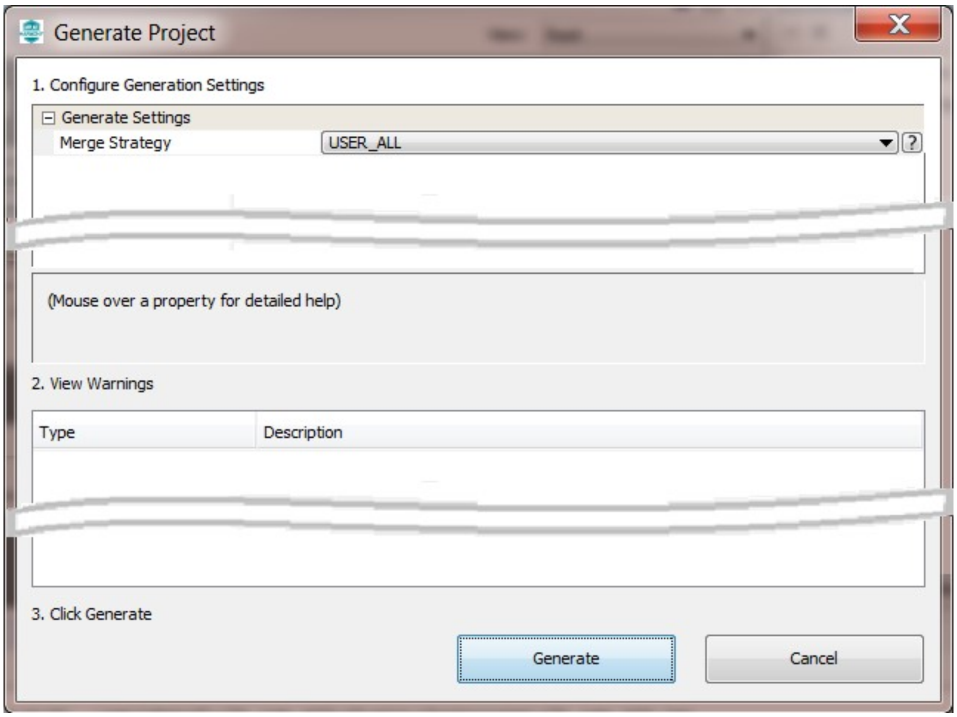
13. Click on the Code button.



On the next screen, click **Don't Save**.

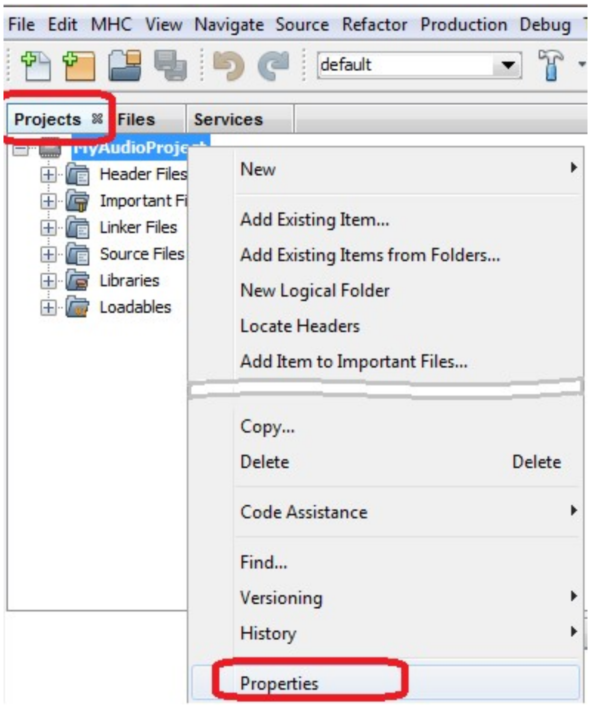


On the Generate Project Screen, click **Generate**:

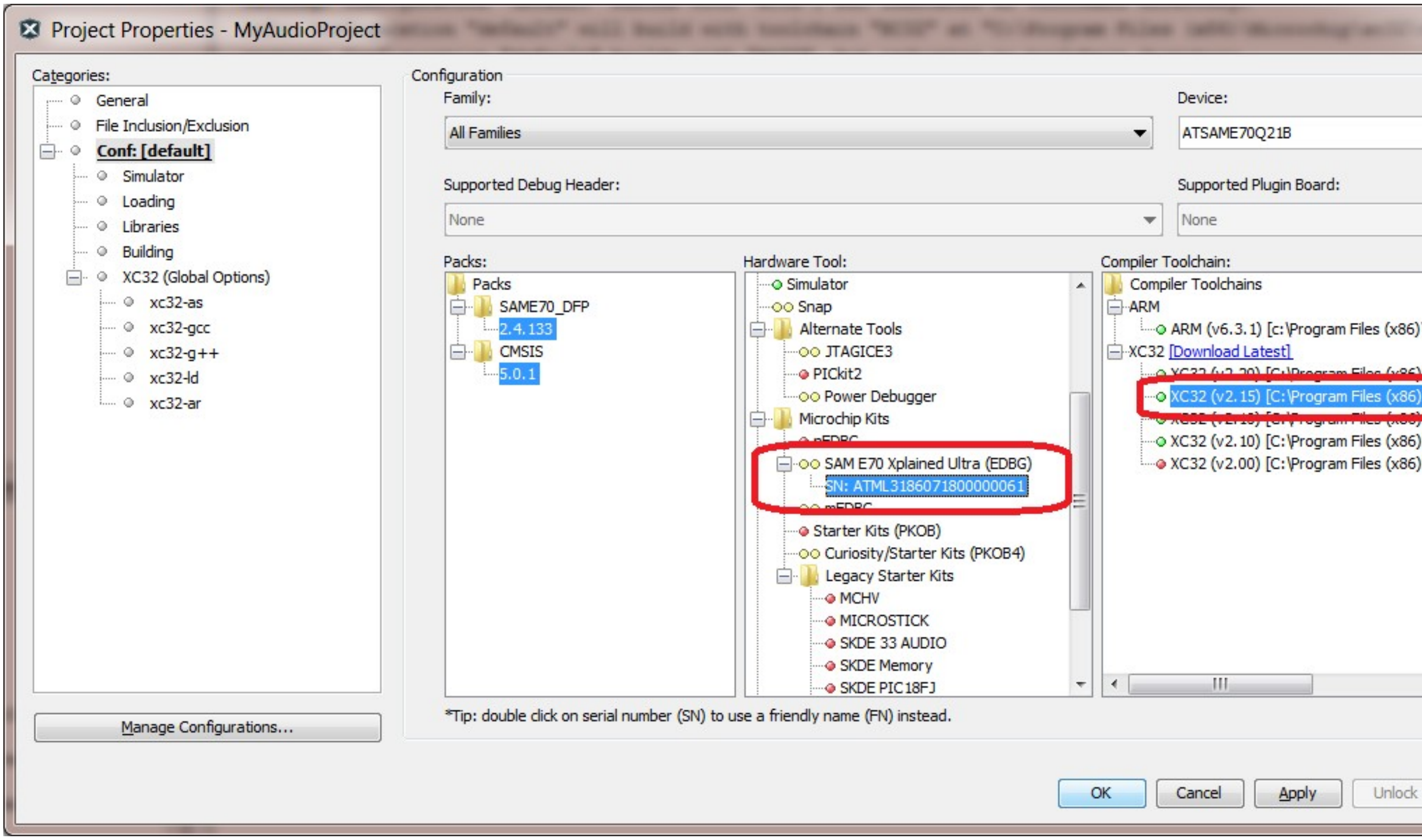


This completes the MHC configuration. The final step is to create the application code. Return to the MPLAB X GUI screen.

14. Right-click on the Project Name and select Properties.



On the next screen, select a compiler and debugger.



15. Click on the Clean and Build Project button.
You should get a BUILD SUCCESSFUL message (if not, go back and click on the Code button again).

Adding code

Under MyAudioProject, expand Header Files, and open app.h. Under Included Files, at around line 35, add:

```
#include "definitions.h"
```

In the Type Definitions section, around line 51, add the following lines:

```
#define MAX_AUDIO_NUM_SAMPLES 9600 // number of samples in buffer
#define FREQUENCY 1000 // 1 kHz

typedef struct // data structure to contain one left-right sample
{
// Left channel data
int16_t leftData;

// Right channel data
int16_t rightData;

} DRV_I2S_DATA16;
```

Under Application States, around line 75, changes the APP_STATES enum to:

```
typedef enum
{
/* Application's state machine's initial state. */
APP_STATE_CODEC_OPEN,
APP_STATE_CODEC_SET_BUFFER_HANDLER,
APP_STATE_CODEC_ADD_BUFFER,
APP_STATE_CODEC_WAIT_FOR_BUFFER_COMPLETE,
} APP_STATES;
```

Under application data, around line 97, add the following struct:

```
typedef struct
{
DRV_HANDLE handle;
DRV_CODEC_BUFFER_HANDLE writeBufHandle1, writeBufHandle2;
DRV_CODEC_BUFFER_EVENT_HANDLER bufferHandler;
uintptr_t context;
uint8_t *txbufferObject1;
uint8_t *txbufferObject2;
size_t bufferSize1, bufferSize2;
} AUDIO_CODEC_DATA;
```

And changes the APP_DATA struct to:

```
typedef struct
{
/* The application's current state */
APP_STATES state;
```

```
uint8_t pingPong;
uint32_t sampleRate;
uint32_t frequency;
AUDIO_CODEC_DATA codecData;
} APP_DATA;
```

Under MyAudioProject, expand Source Files, and open app.c. In the Included Files section, around line 31, add the following line:

```
#include <math.h> // for sin & M_PI
```

In the Global Data Definitions section, around line 56, add the lines:

```
// audio buffers
DRV_I2S_DATA16 __attribute__ ((aligned (32)))
App_Tone_Lookup_Table_tone1[MAX_AUDIO_NUM_SAMPLES];
DRV_I2S_DATA16 __attribute__ ((aligned (32)))
App_Tone_Lookup_Table_tone2[MAX_AUDIO_NUM_SAMPLES];
```

which define the two audio buffers which we will use in a ping-pong fashion, back and forth, each containing up to 9600 samples. In the Application Callback Functions, around line 70, add the lines:

```
void _codecTxBufferComplete()
{
//Next State -- after the buffer complete interrupt.
appData.state = APP_STATE_CODEC_ADD_BUFFER;
}
```

In the Application Local Functions section, around line 86, add the following function:

```
uint32_t _sineTableInit(DRV_I2S_DATA16* buffer, uint32_t frequency,
uint32_t sampleRate)
{
uint32_t i,j,k, maxNumCycles;
uint32_t numSamplesPerCycle;

// generate sine table
// # of samples for one cycle, e.g. 48 for 1 kHz @ 48,000 samples/sec
numSamplesPerCycle = sampleRate / frequency;

// max # of cycles we can fit in buffer
maxNumCycles = MAX_AUDIO_NUM_SAMPLES / numSamplesPerCycle;

k = 0;
for (j=0; j < maxNumCycles; j++)
{
for (i=0; i < numSamplesPerCycle; i++)
{
// radians = degrees × pi / 180°
double radians = (M_PI*(double)(360.0/
(double)numSamplesPerCycle)*(double)i)/180.0;
```

```
buffer[i+k].leftData = (int16_t)(0x7FFF*sin(radians));
buffer[i+k].rightData = buffer[i+k].leftData;
}
k += numSamplesPerCycle;
}

return sizeof(DRV_I2S_DATA16)*k; // return size of filled-in buffer
}
```

In the Application Initialization and State Machine Functions, around line 123, add the following function:

```
void _audioCodecInitialize (AUDIO_CODEC_DATA* codecData)
{
codecData->handle = DRV_HANDLE_INVALID;
codecData->context = 0;
codecData->bufferHandler =
(DRV_CODEC_BUFFER_EVENT_HANDLER) _codecTxBufferComplete;
codecData->txbufferObject1 = (uint8_t *) App_Tone_Lookup_Table_tone1;
codecData->txbufferObject2 = (uint8_t *) App_Tone_Lookup_Table_tone2;
codecData->bufferSize1 = 0;
codecData->bufferSize2 = 0;
}
```

At around line 143, change the APP_Initialize function to:

```
void APP_Initialize ( void )
{
/* TODO: Initialize your application's state machine and other
* parameters.
*/
appData.state = APP_STATE_CODEC_OPEN;

_audioCodecInitialize (&appData.codecData);

appData.frequency = FREQUENCY; // e.g. 1 kHz

appData.pingPong = 1;
}
```

At around line 163, replace the APP_Tasks function with the following:

```
void APP_Tasks ( void )
{
// Check the application's current state
switch ( appData.state )
{
// Application's initial state
case APP_STATE_CODEC_OPEN:
{
// See if codec is done initializing
SYS_STATUS status = DRV_CODEC_Status(sysObjdrvCodec0);
if (SYS_STATUS_READY == status)
```

```
{
// This means the driver can now be opened.
// Open the driver object to get a handle
appData.codecData.handle = DRV_CODEC_Open(DRV_CODEC_INDEX_0,
DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.codecData.handle != DRV_HANDLE_INVALID)
{
// Get sampling rate set up in MHC (e.g. 48000)
appData.sampleRate =
DRV_CODEC_SamplingRateGet(appData.codecData.handle);

appData.state = APP_STATE_CODEC_SET_BUFFER_HANDLER;
}
}
}
break;

// Set a handler for the audio buffer completion event
case APP_STATE_CODEC_SET_BUFFER_HANDLER:
{
DRV_CODEC_BufferEventHandlerSet(appData.codecData.handle,
appData.codecData.bufferHandler,
appData.codecData.context);

// Initialize the first buffer with sine wave data
appData.codecData.bufferSize1 = _sineTableInit(
(DRV_I2S_DATA16*)appData.codecData.txbufferObject1,
appData.frequency, appData.sampleRate);

appData.pingPong = 1; // indicate buffer 1 to be used first

appData.state = APP_STATE_CODEC_ADD_BUFFER;
}
break;

case APP_STATE_CODEC_ADD_BUFFER:
{
// Will ping-pong back and forth between buffer 1 and 2
if (appData.pingPong==1)
{
// hand off first buffer to DMA
DRV_CODEC_BufferAddWrite(appData.codecData.handle,
&appData.codecData.writeBufHandle1,
appData.codecData.txbufferObject1,
appData.codecData.bufferSize1);

if (appData.codecData.writeBufHandle1 !=
DRV_CODEC_BUFFER_HANDLE_INVALID)
{
// just sent buffer 1 to DMA,
// so fill in buffer 2 for next time
appData.codecData.bufferSize2 = _sineTableInit(
```

```
(DRV_I2S_DATA16*)appData.codecData.txbufferObject2,
appData.frequency, appData.sampleRate);

appData.pingPong = 2; // switch to second buffer
appData.state = APP_STATE_CODEC_WAIT_FOR_BUFFER_COMPLETE;
}
}
else
{
// hand off second buffer to DMA
DRV_CODEC_BufferAddWrite(appData.codecData.handle,
&appData.codecData.writeBufHandle2,
appData.codecData.txbufferObject2,
appData.codecData.bufferSize2);

if (appData.codecData.writeBufHandle2 !=
DRV_CODEC_BUFFER_HANDLE_INVALID)
{
// just sent buffer 2 to DMA,
// so fill in buffer 1 for next time
appData.codecData.bufferSize1 = _sineTableInit(
(DRV_I2S_DATA16*)appData.codecData.txbufferObject1,
appData.frequency, appData.sampleRate);

appData.pingPong = 1; // back to first buffer
appData.state = APP_STATE_CODEC_WAIT_FOR_BUFFER_COMPLETE;
}
}
}
break;

// Audio data Transmission under process, wait for callback
case APP_STATE_CODEC_WAIT_FOR_BUFFER_COMPLETE:
{
}
break;

default:
{
/* TODO: Handle error in application's state machine. */
}
break;
}
}
```

Discussion

We want to output a 1 kHz sine wave audio tone at 48,000 samples/sec, using 16-bit quantization, or 48 samples per millisecond. A struct `DRV_I2S_DATA16` is created with two 16-bit members, one for each channel (left/right) of the audio stream. Two tables are then declared, `App_Tone_Lookup_Table_tone1` and `2` each with 9600 of these structs, enough to hold 200 ms of audio.

These two buffers are then used, back and forth (ping-pong fashion) to output the tone. The first buffer is filled initially, and then while that is being output via DMA (without the CPU being directly involved), the processor can fill up the second one.

The application starts off by calling `SYS_Initialize` in the file initialization.c, to initialize the various peripherals that were selected in MHC (clocks, timer, I2C, I2S, audio codec, etc.). Near the end of that routine, the application initialization routine `APP_Initialize` is called, where a number of variables are initialized in the structures `appData` and `appData.codecData`.

After initialization, control passes into the routine `SYS_Tasks` in tasks.c, which is just a tight while loop calling either the task routine for the code (e.g. `DRV_WM8904_Tasks`) or the one for the application (`APP_Tasks`). Both are implemented as state machines.

The initial state in `APP_Tasks` is `APP_STATE_CODEC_OPEN`, where it waits until the codec reports that its initialization is complete. Once the driver is ready, it is opened via a call to `DRV_CODEC_Open` with an intention to write (`DRV_IO_INTENT_WRITE`). If the open call is successful, the application queries the codec driver to get the sampling rate setup in MHC and saves it away.

The application state machine then advances to the state `APP_STATE_CODEC_SET_BUFFER_HANDLER`, where it registers the DMA callback by calling the routine `DRV_CODEC_BufferEventHandlerSet` with the name of the callback function `_codecTxBufferComplete`. The first buffer is then filled by calling the function `_sineTableInit`. It divides the sampling rate (e.g. 48,000) by the frequency (e.g. 1000) to get the number of samples for one cycle (e.g. 48). It then divides this result into the array size `MAX_AUDIO_NUM_SAMPLE` to determine how many cycles can fit into the buffer (e.g. 200).

For each cycle, _sineTableInit calculates the angular displacement in radians for each sample, and then calls the sin function in the C math library to generate a singed value for the buffer. When done, the function returns the size of the filled-in buffer in bytes, since if the frequency was changed, or the sampling rate, the calculated number of cycles might not completely fill the buffer.

The state machine then advances to the state `APP_STATE_CODEC_ADD_BUFFER`, where it makes a call to the routine `DRV_CODEC_BufferAddWrite`, passing the address of the first buffer along with its size, among other parameters. This initiates a DMA function, transferring the contents of the buffer to the I2S peripheral without processor intervention. After a successful return from the `BufferAddWrite` routine, the app calls the function `_sineTableInit` again, this time passing the address of the second buffer, the desired frequency and sampling rate. On return, the next state executed is `APP_STATE_CODEC_WAIT_FOR_BUFFER_COMPLETE`, which is a dummy state (no code) where the state machine waits for the DMA transfer to finish.

When the DMA transfer is finished, the registered callback function `_codecTxBufferComplete` is executed, which tells the state machine it is time to transfer the second buffer to the I2S peripheral. By changing the state back to `APP_STATE_CODEC_ADD_BUFFER`. After initiating DMA transfer of the second buffer, it fills up the first buffer again. This cycle just continues.

Since the same tone is always being output, one could get away with using just one buffer, filling it up initially just once, and keep transferring the same contents over and over via DMA. However with real applications this is seldom the case; rather the audio keeps changing since it will either be coming off of a live source (e.g. microphone), or internal (flash) or external (SD card or USB) storage, in which case during each DMA transfer period the next buffer to be output is being refreshed with new data.

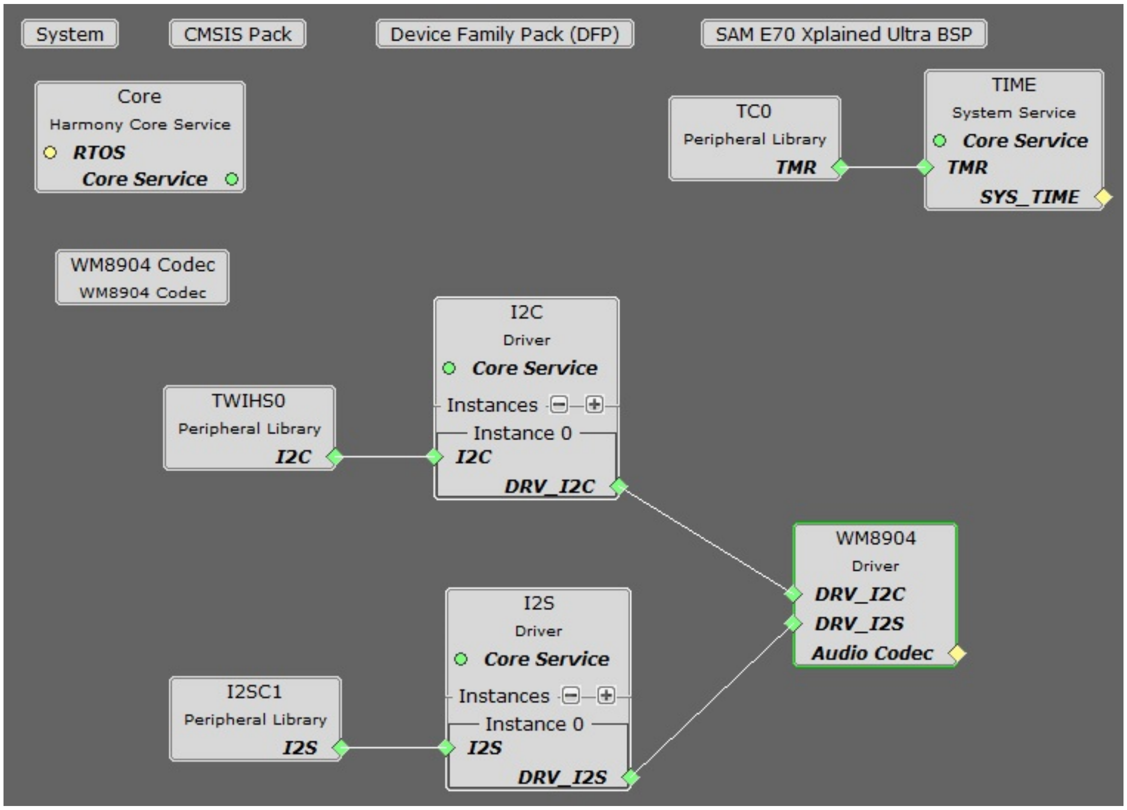
So the ping-pong buffer case is a better model to start with in most cases. If one cannot guarantee how fast audio might be arriving, such as when it comes in via USB from a PC, then sometimes it is necessary to have even more buffers, arranged in a circular fashion, with a pointer to the buffer being filled, and another to the buffer being output.

<u>For projects using the I2SC interface and the codec as a Slave (the MCU generates the I^2S clocks):</u>

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 (e.g. ATSAME70Q21B).

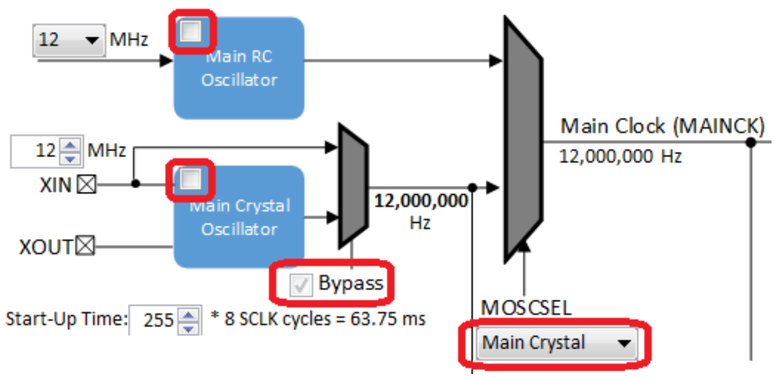
In the MHC, under Available Components select an appropriate BSP (e.g. SAM E70 Xplained Ultra). Under *Audio>Templates*, double-click on the desired codec, e.g. WM8904. Answer Yes to all the questions. Click on the Codec component (*not* the Codec Driver). In the Configuration Options, select I2SC instead of SSC. Answer Yes to all of the questions.

The result will be a project graph that looks like this, after rearranging the boxes:



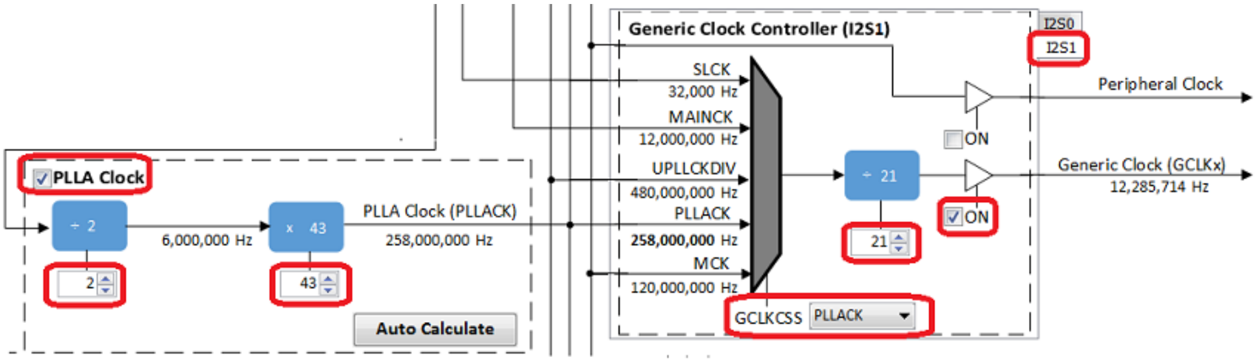
Click on the Codec Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

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 the 12 MHz resonator on the board:



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 Codec Driver and I2SC 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) 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), DMA, Timer and USB library. A FreeRTOS version is 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 screendisplay. 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_speaker Demonstration

The usb_speaker application aacts 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), 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.

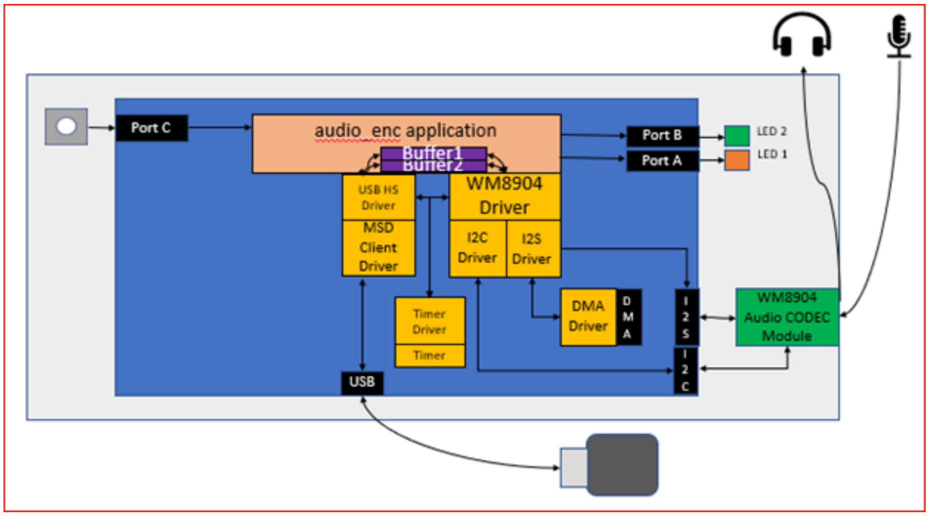
Supported audio files are as represented in the table below.

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

a. Yes to all popups

b. Modify WM8904k Codec from SSC to I2SC

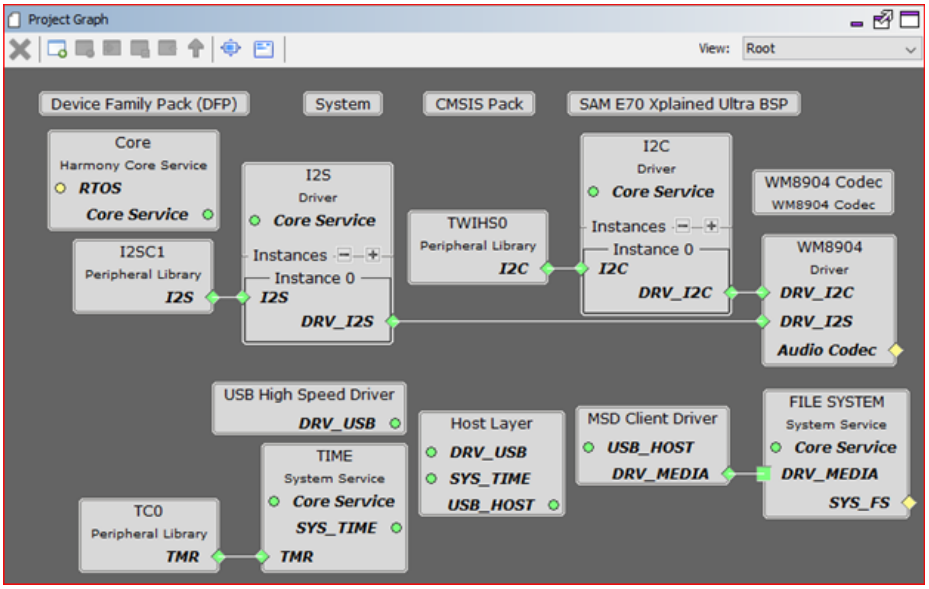
i. Yes to all popups

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

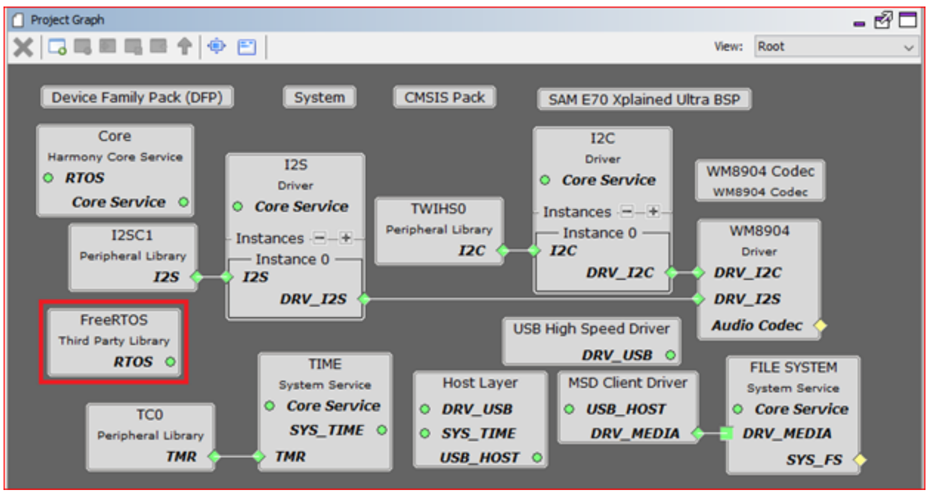
a. Yes to all popups

4. Add Harmony->System Services->FILE SYSTEM
5. Remove FreeRTOS if Bare Metal version
6. 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.

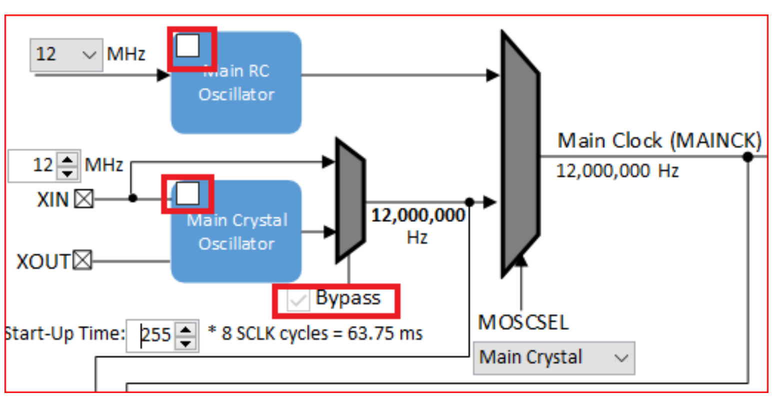


Tools Setup Differences

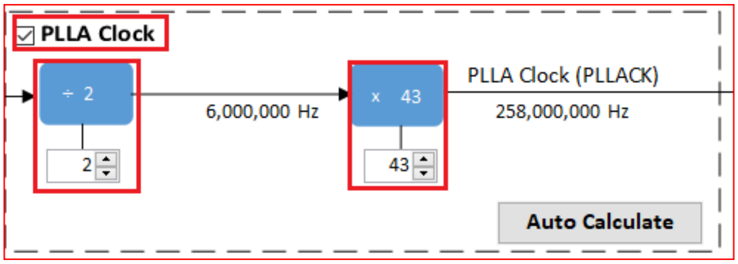
The default configuration should be correct for the majority of the app. There following configurations will need to be changed in order for proper operations.

MPLAB Harmony Configurator: Tools->Clock Configuration

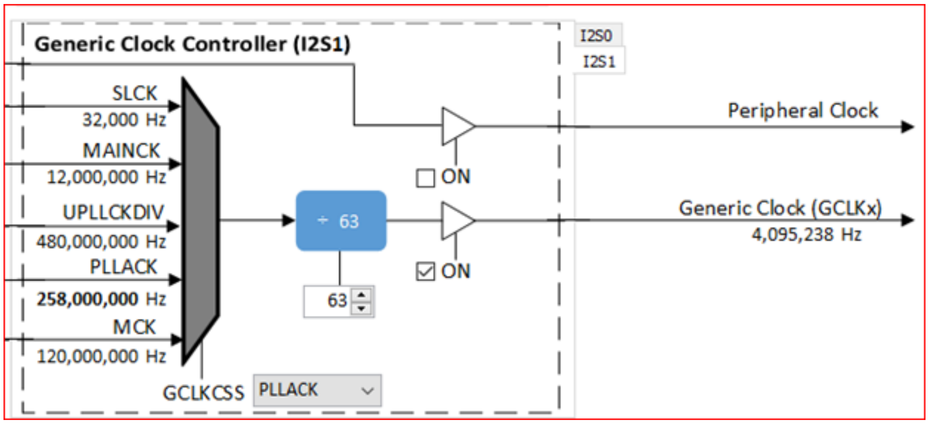
Uncheck the Main RC Oscillator and check the “Bypass” for the Main Crystal Oscillator. When the Bypass is check, it will cause the Main Crystal Oscillator to become unchecked.



Enable the PLLA Clock output.

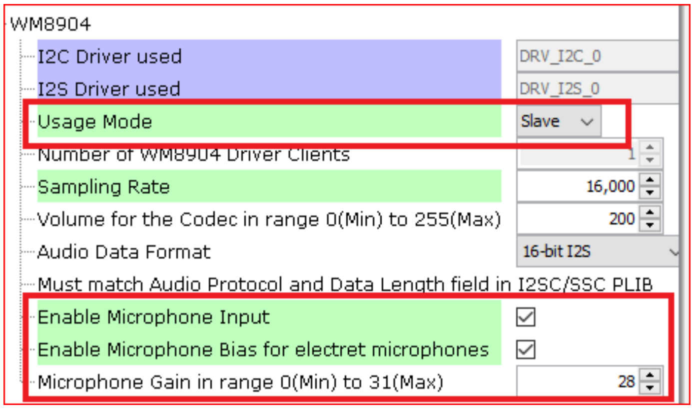


Enable clocking for the I2S1.



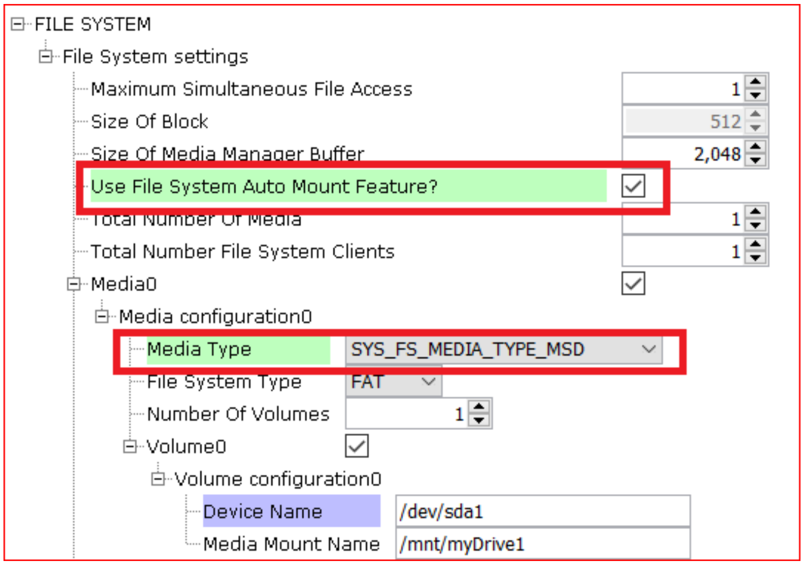
To set the sample rate to a fixed 16KHz, set the PLLA divisor to 2 and the multiplier to 43. Also, set the I2S1 divisor to 63. Please see the two images above.

MPLAB Harmony Configurator: WM8904



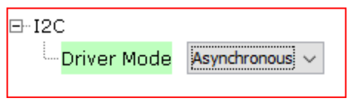
MPLAB Harmony Configurator: File System

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.



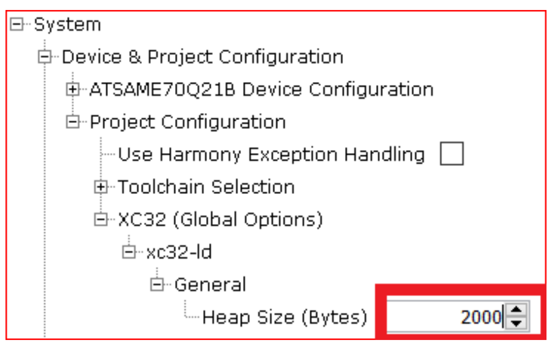
MPLAB Harmony Configurator: I2C Driver

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.



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.

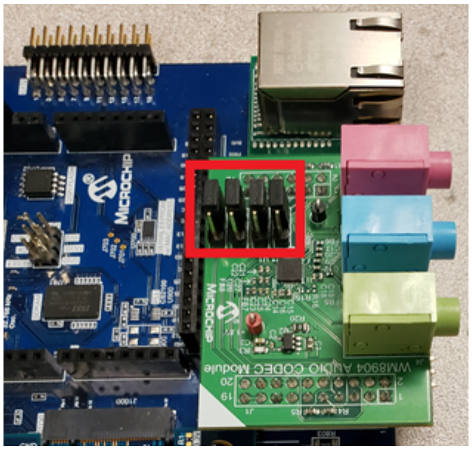
Project Name	BSP Used	Description
audio_enc_i2sc	sam_e70_xult	This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board with the WM8904 Audio Codec Module attached. The project configuration is for encode/playback of an audio file to/from a USB Mass Storage Device. Data 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_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 toward 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 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
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

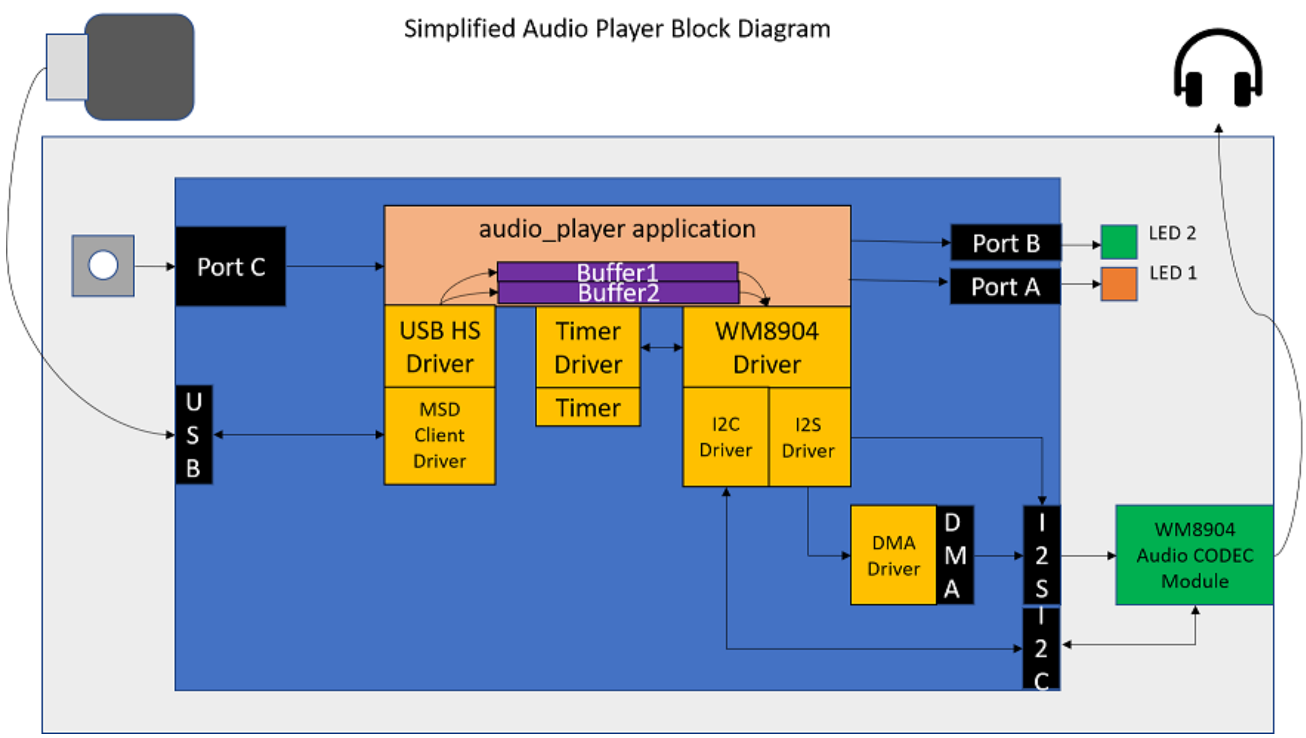
The SAM E70 Xplained Ultra board does not include the WM8904 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC328904.

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

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 USB thumb 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.

Architecture Block Diagram

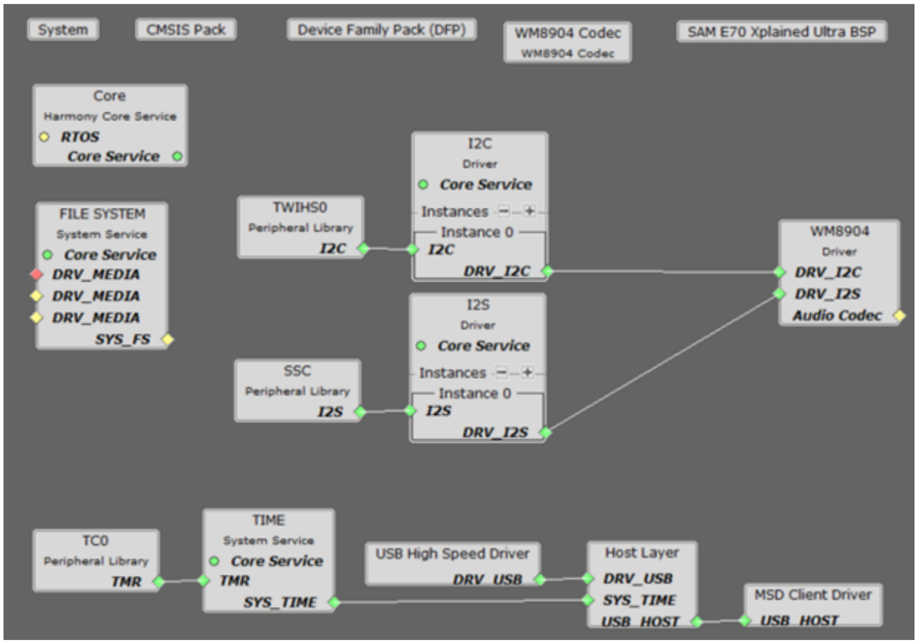


Demonstration Features

- USB MSD Host Client Driver (see USB MSD Host Client Driver Library)
- 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)

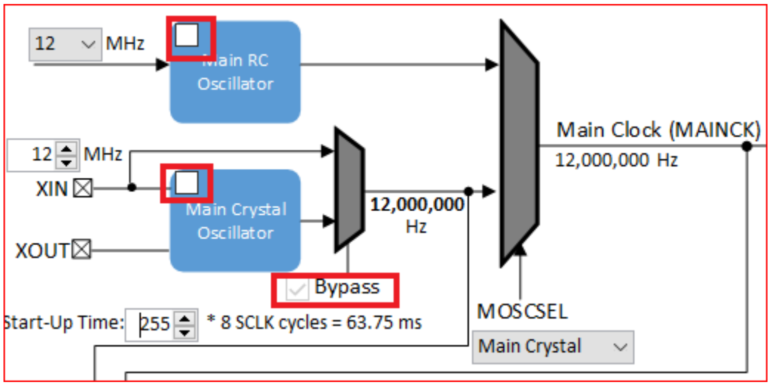
Tools Setup Differences

The project graph below specifies the drivers, services, and libraries being brought into the project to further extend the applications abilities. 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.

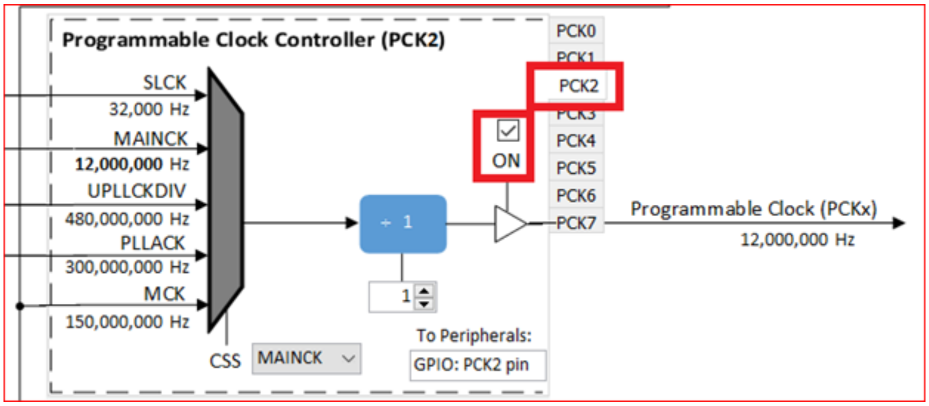


MPLAB Harmony Configurator: *Tools>Clock* Configuration

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 unchecked.

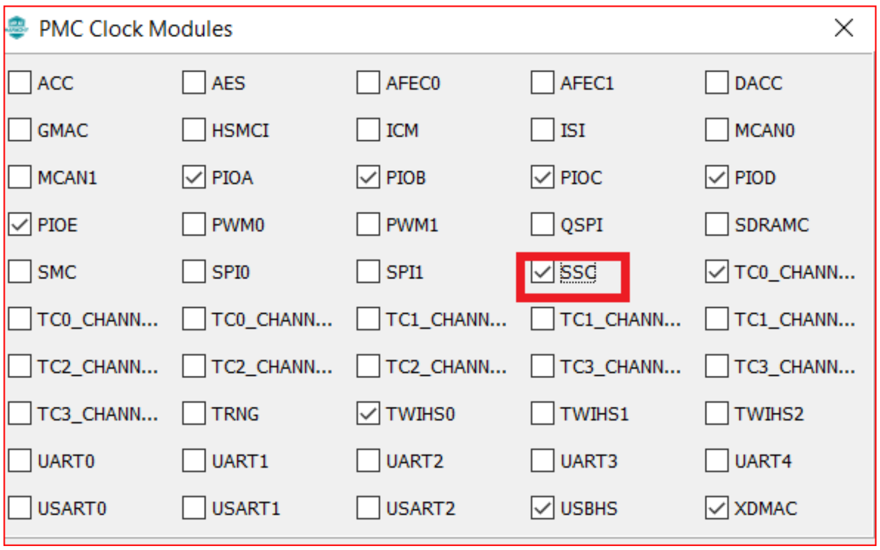


Enable the PCK2 output to enable the WM8904 master clock:



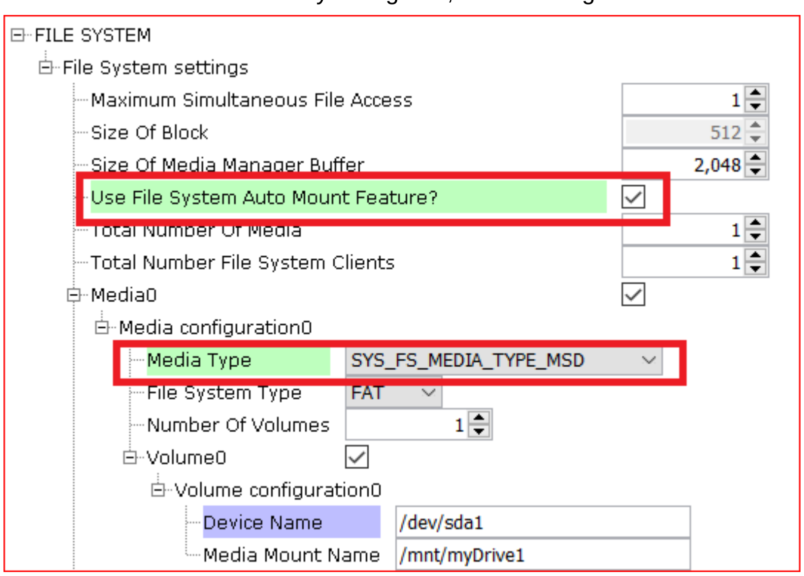
Clock Diagram>Peripheral Clock Enable

Enable clocking for the SSC.



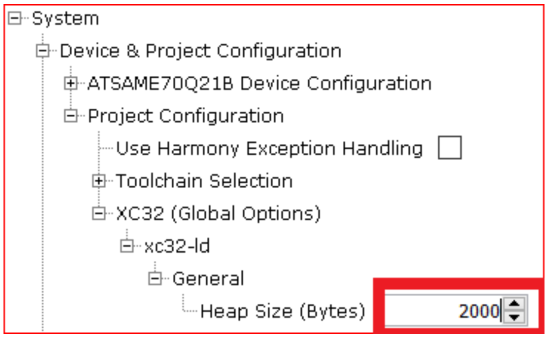
<u>MPLAB Harmony Configurator: *File System*</u>

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.



<u>MPLAB Harmony Configurator: *System*</u>

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



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
audio_player_basic_sam_e70_xult	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 WM8904 codec is configured as the master, and the SSC peripheral as the slave.
audio_player_basic_sam_e70_xult_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 WM8904 codec is configured as the master, and the SSC peripheral as the slave. This demonstration also uses FreeRTOS.

Configuring the Hardware

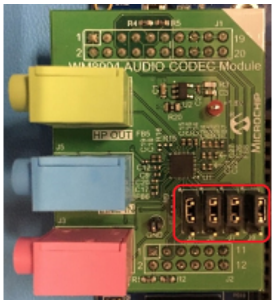
This section describes how to configure the supported hardware.

Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the 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 1**).
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) 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 USB 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.

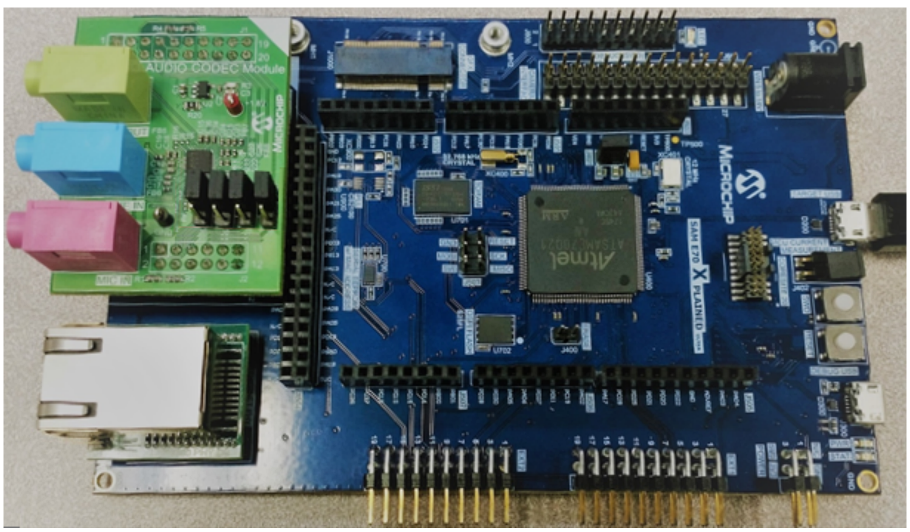


Figure 1: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board

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:

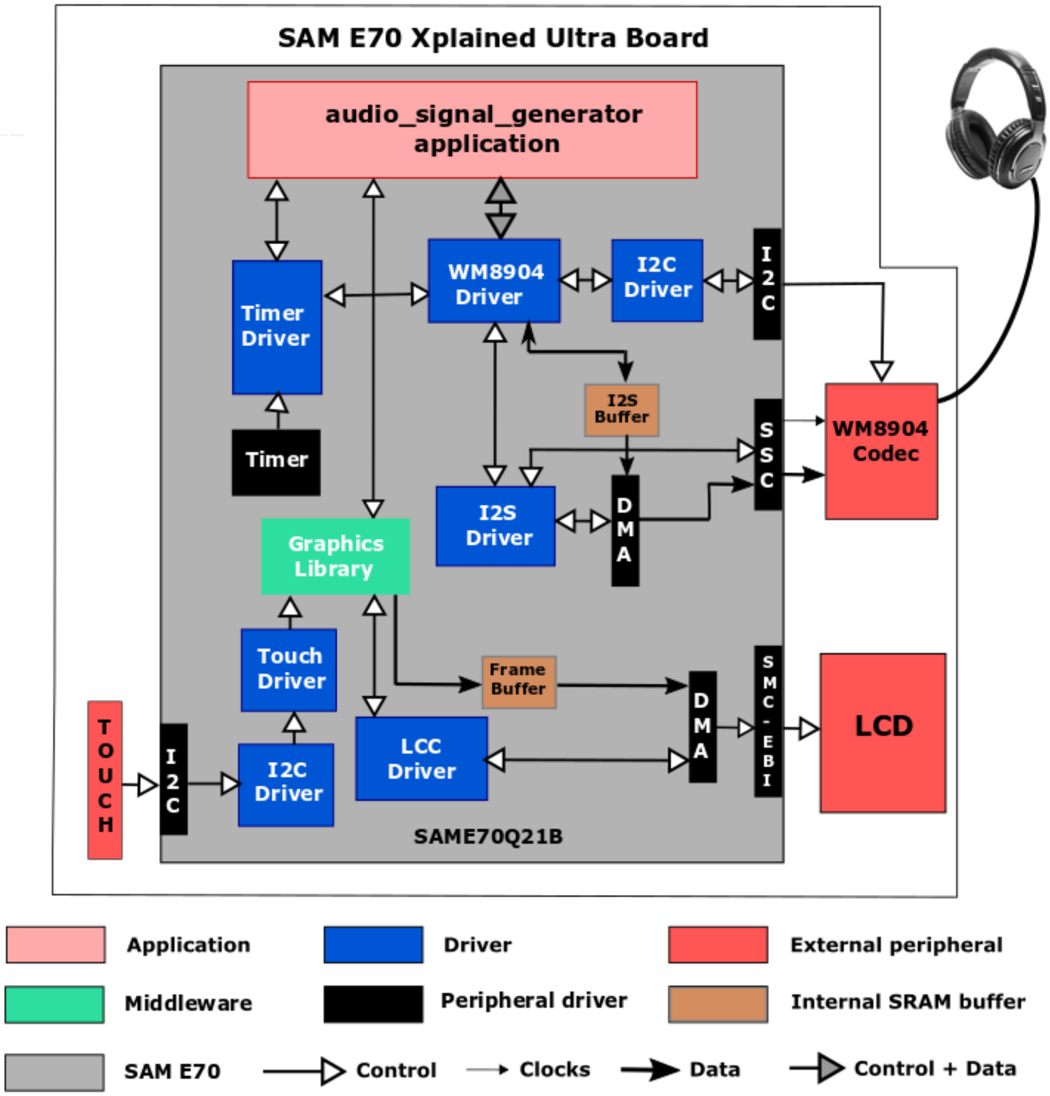
There is one project which runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- 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^{2}S 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^{2}C interface. The audio interface between the SAM E70 and the WM8904 codec use the I^{2}S interface. Audio is configured as 16-bit, 48,000 samples/second, I^{2}S 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^{2}S, 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)

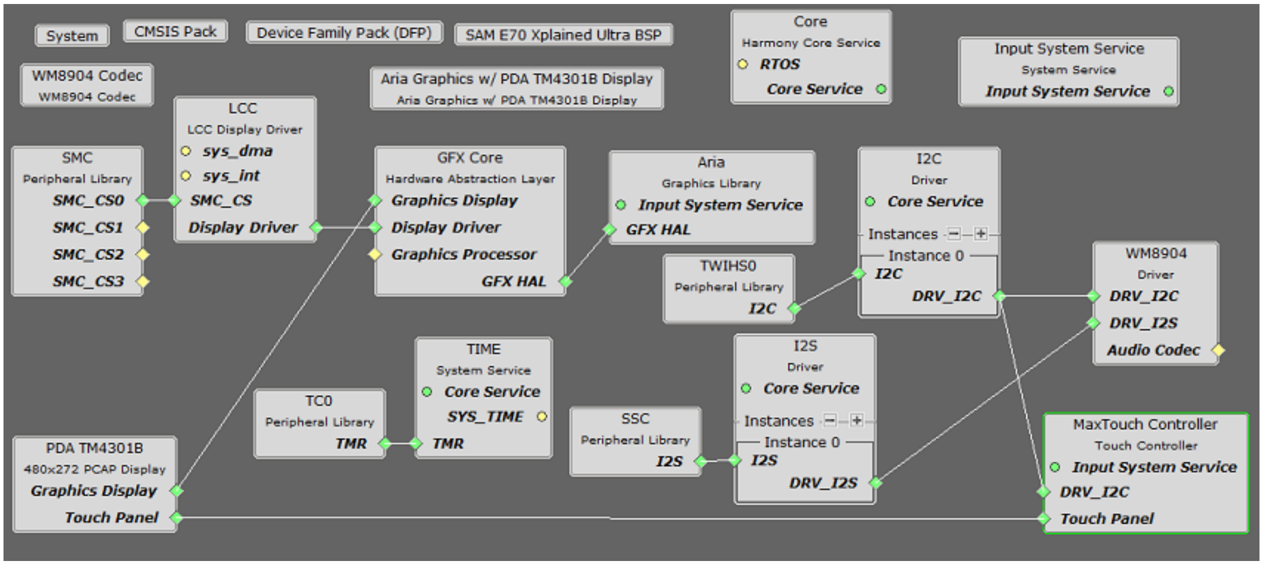
Tools Setup Differences

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.

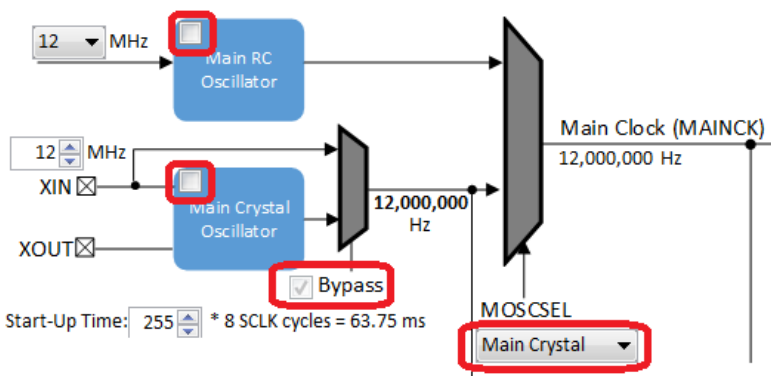
Then under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions.

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.

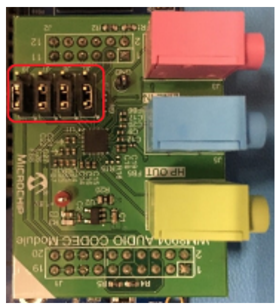
Configuring the Hardware

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.

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 Application for details.

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

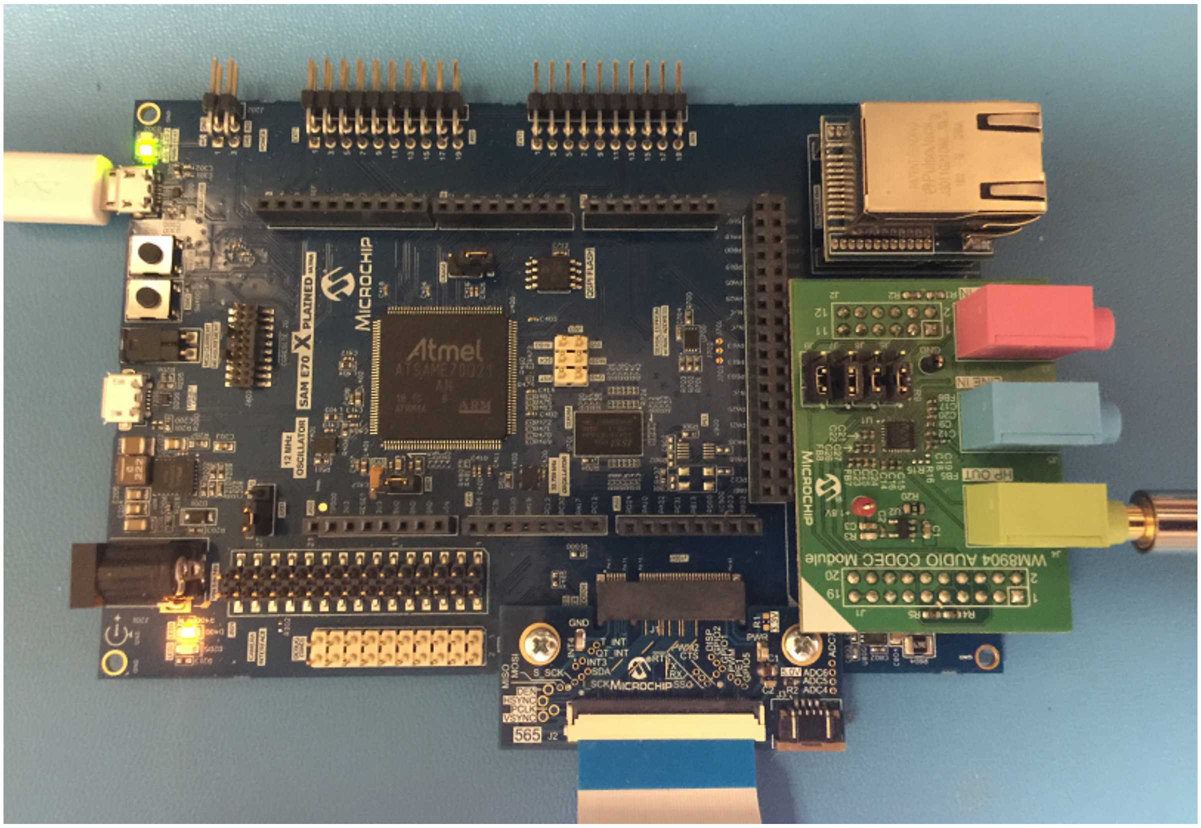
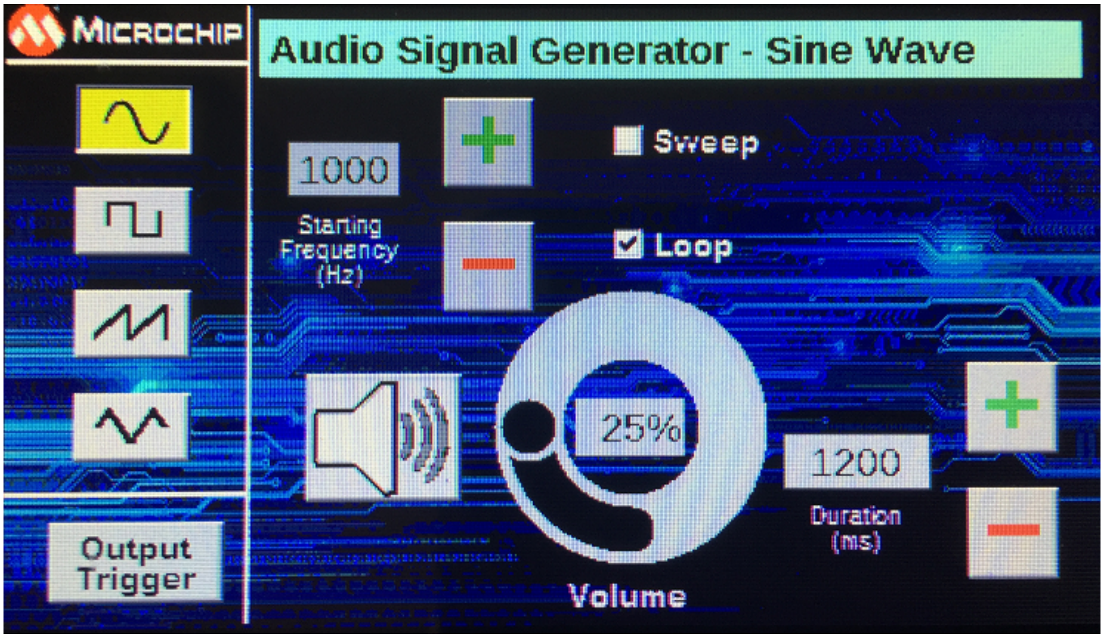
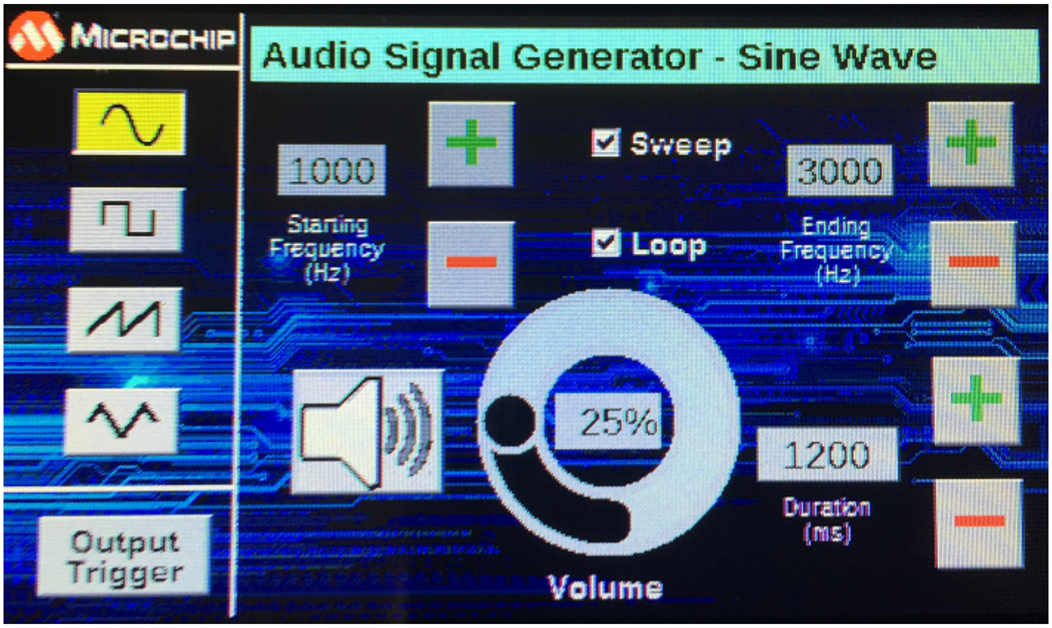


Fig 1. SAM E70 Xplained Ultra Board with WM8904 Codec Daughterboard

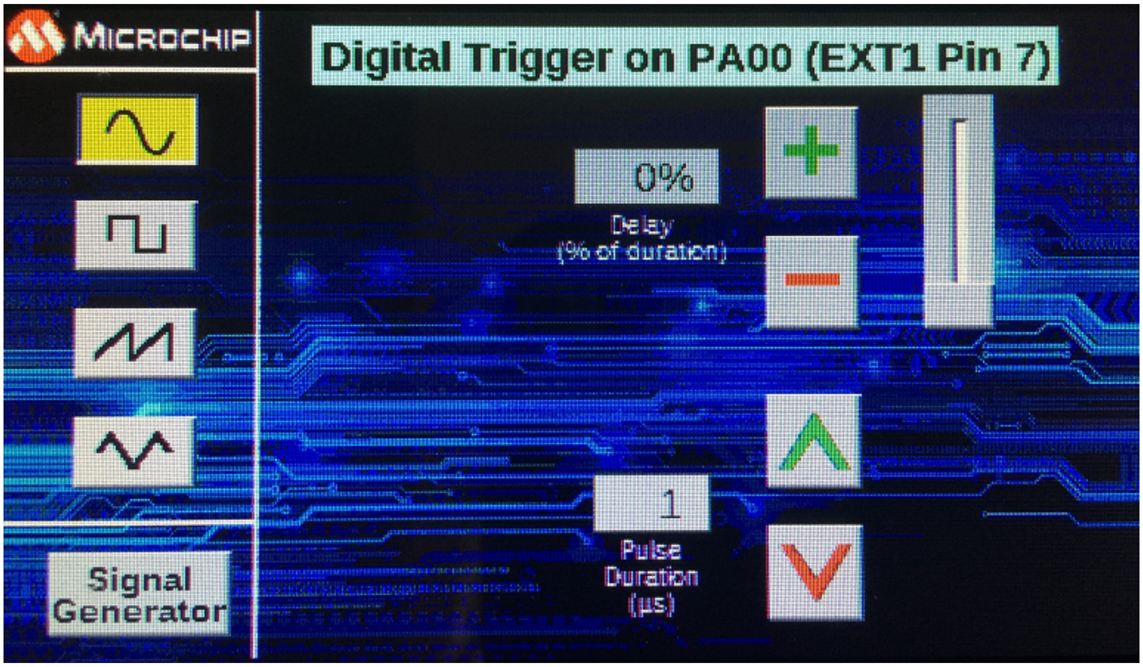
2. Initially the program will be a single-tone continuous sine wave mode, frequency set at 1000 Hz, and a 25% volume level.
3. Turn the tone on and off by tapping on the speaker icon in the center of the screen, to the left of the Volume control. The sound waves will change to a red **X** when enabled.
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 the 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

SAM E70 Xplained Ultra Projects:

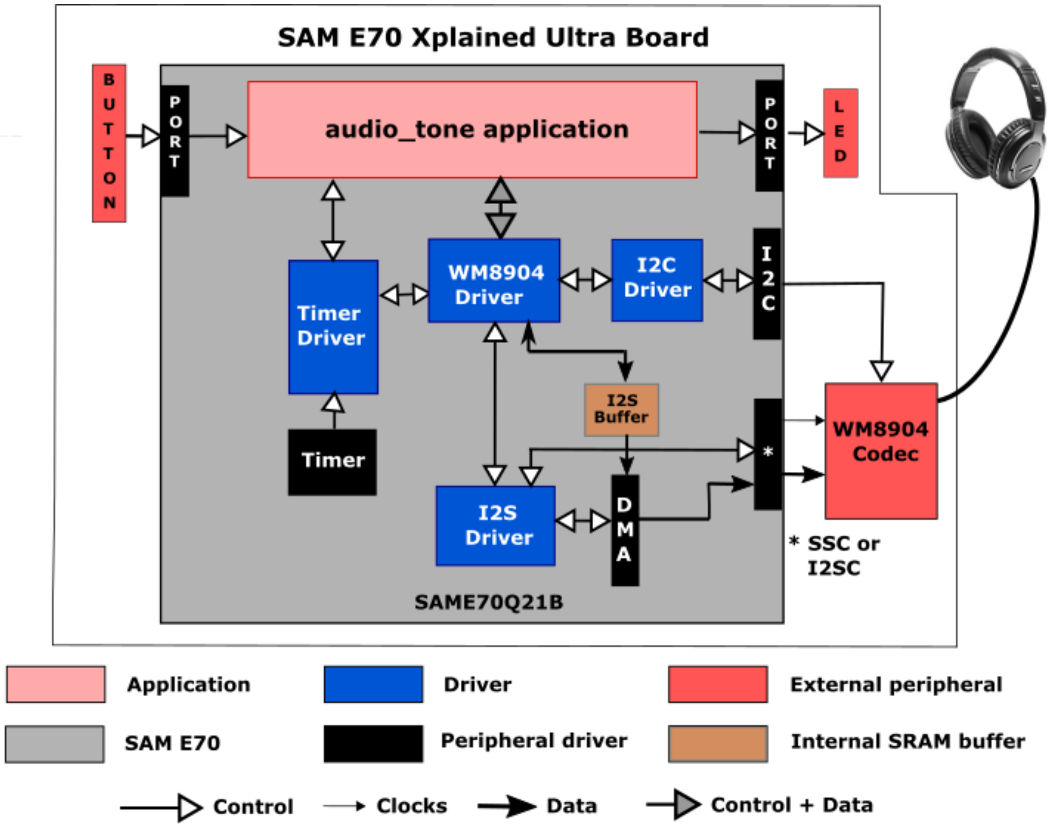
Three projects run on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- 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 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 (RTOS not shown):



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:

- One push button (SW1)
- Two LEDs (LED0 and 1)
- 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 four 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 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 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 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)

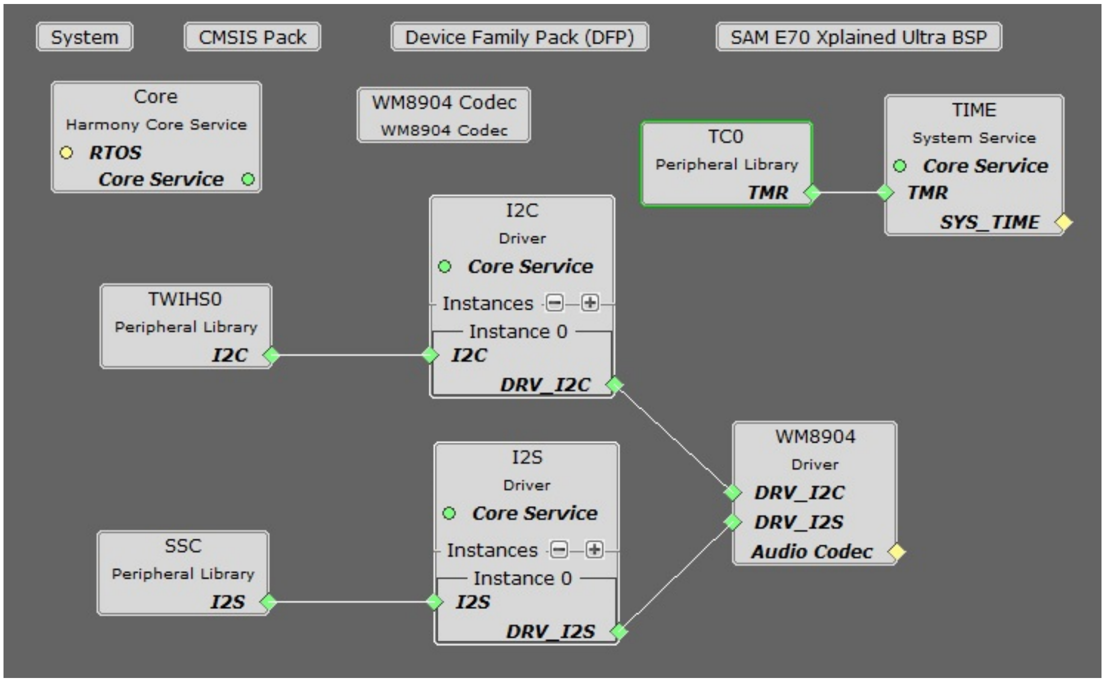
Tools Setup Differences

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.

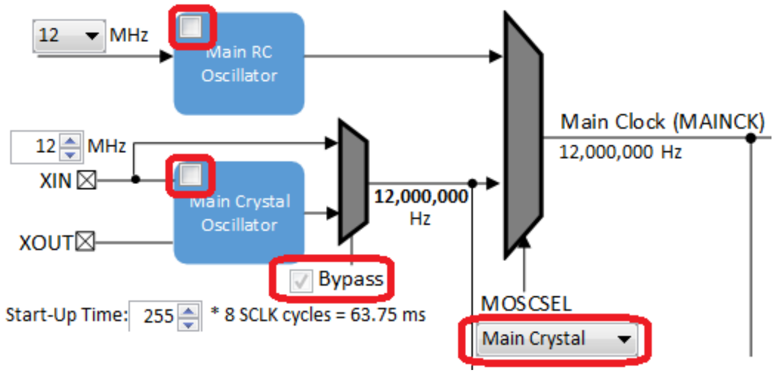
In the MHC, under Available Components select the appropriate BSP (SAM E70 Xplained Ultra or SAM V71 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be 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.

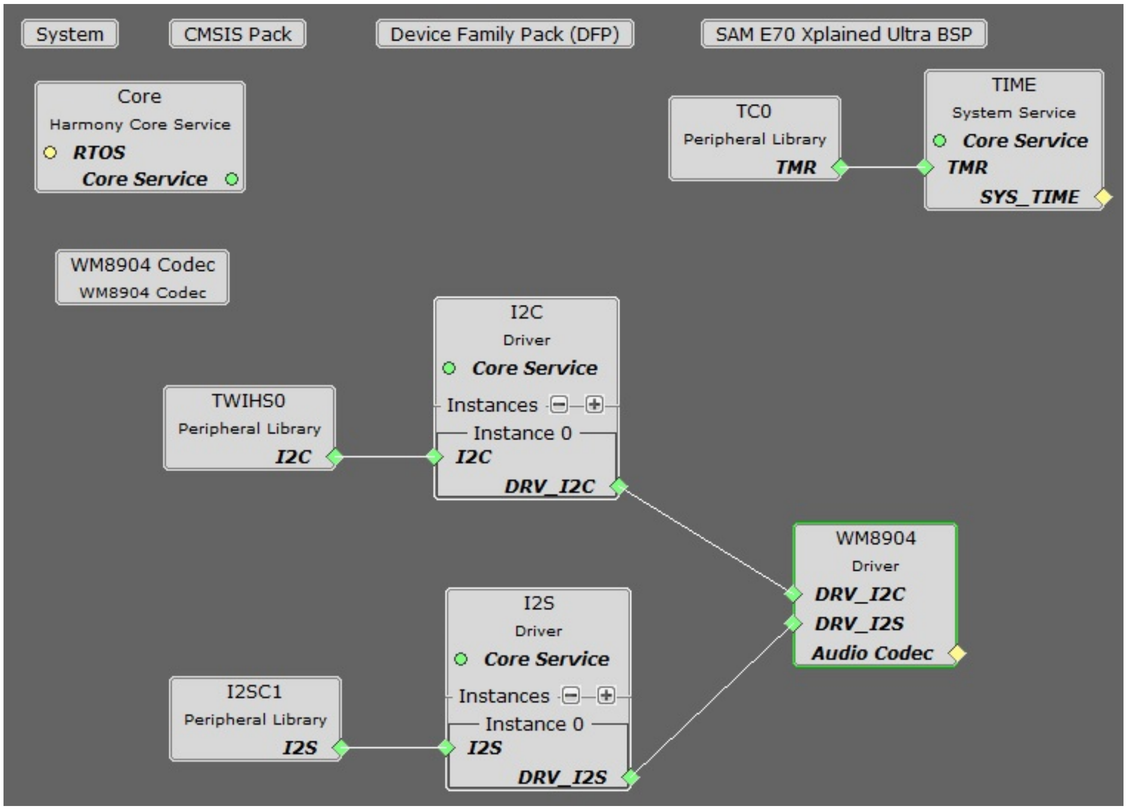
If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the `SYS_Tasks` function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of `_APP_Tasks` (just before the call to `APP_Tasks`).

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.

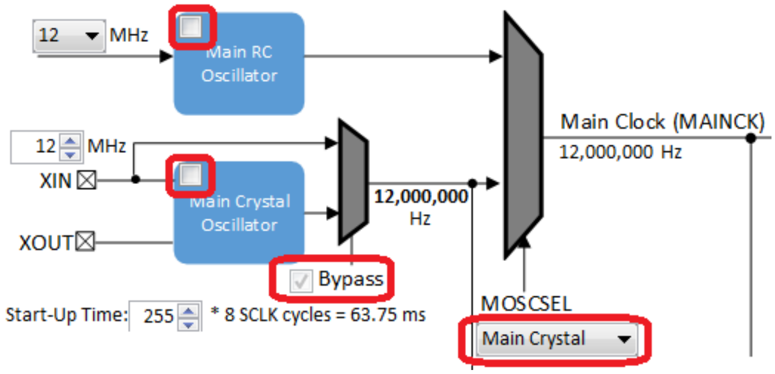
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead of 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:



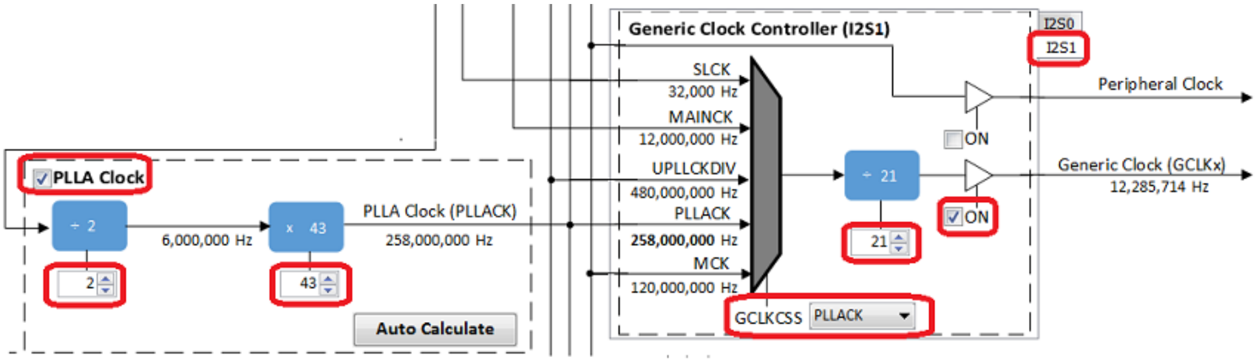
Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

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.

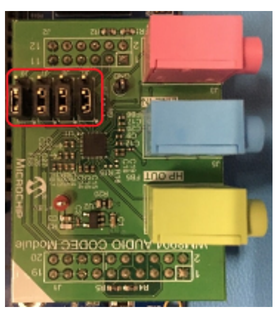
Project Name	BSP Used	Description
audio_tone_sam_e70_xult_wm8904_ssc	sam_e70_xult	This demonstration runs on the ATSAME70Q21B processor on the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^{2}S 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^{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.
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^{2}S 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^{2}S 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.

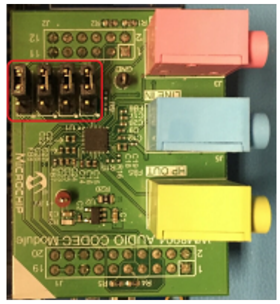
Description

SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using 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.

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.

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 Application for details.

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

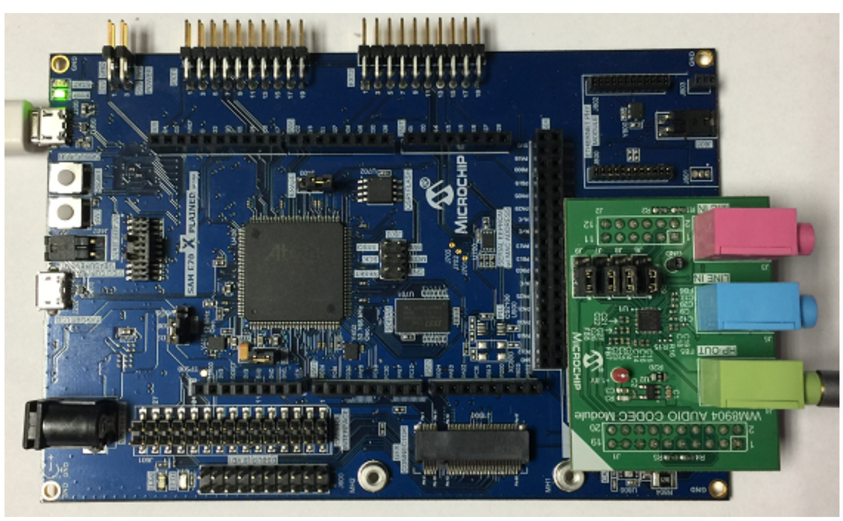


Figure 1: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board

Control Descriptions

Table 1: Button Controls for SAM E70 Xplained Ultra board and SAM V71 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 (> 1 second)	Alternates between modes (LED1 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

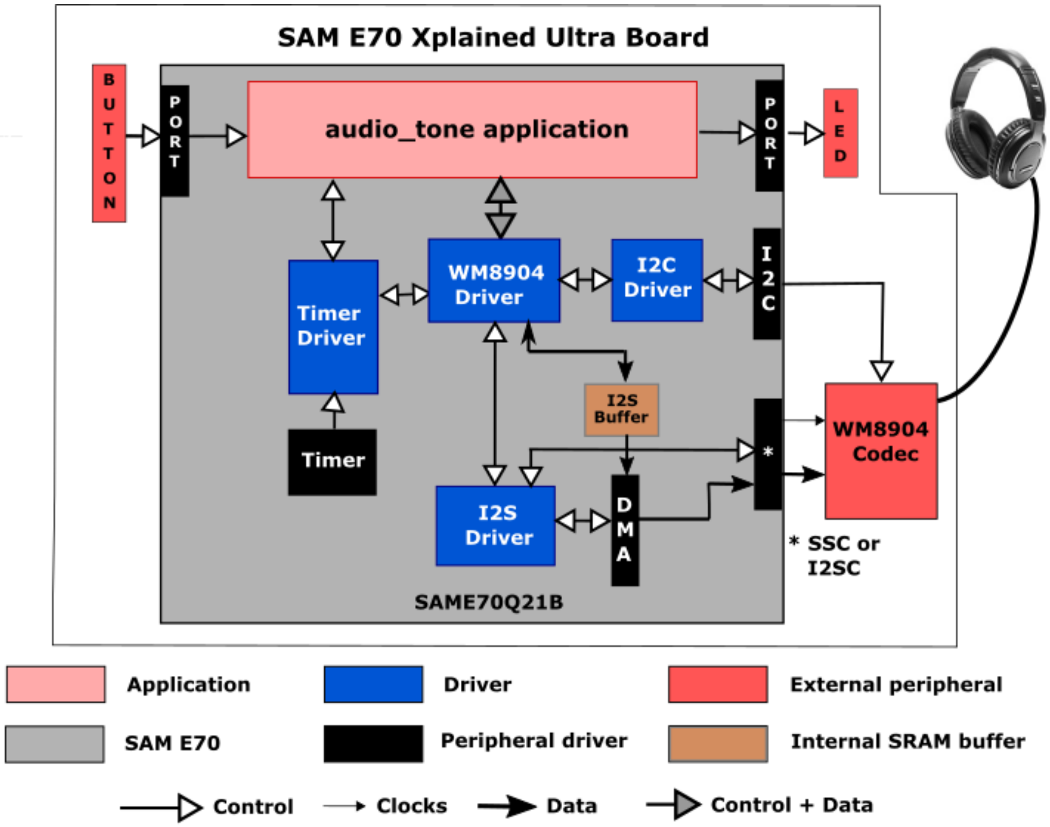
The two projects run on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- 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)

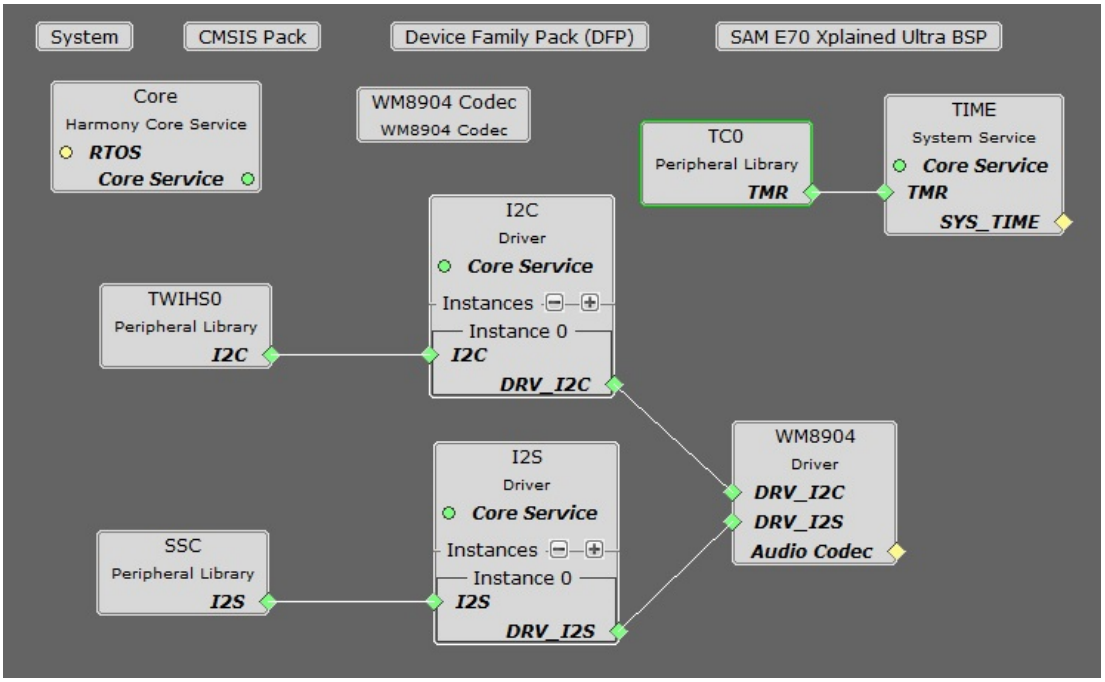
Tools Setup Differences

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.

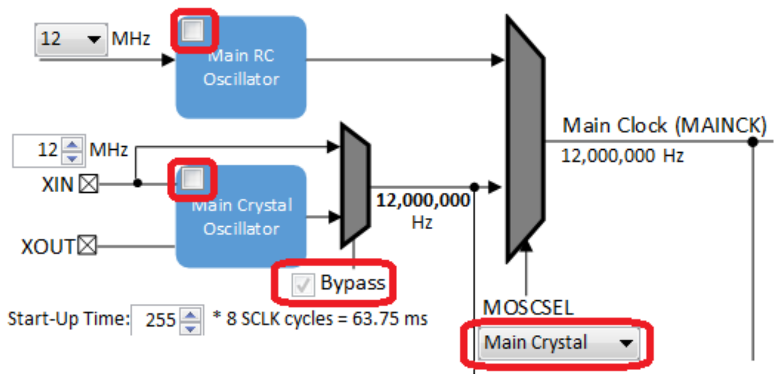
In the MHC, under Available Components select the appropriate BSP (SAM E70 Xplained Ultra or SAM V71 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer Yes or No to that one depending on whether you will be 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.

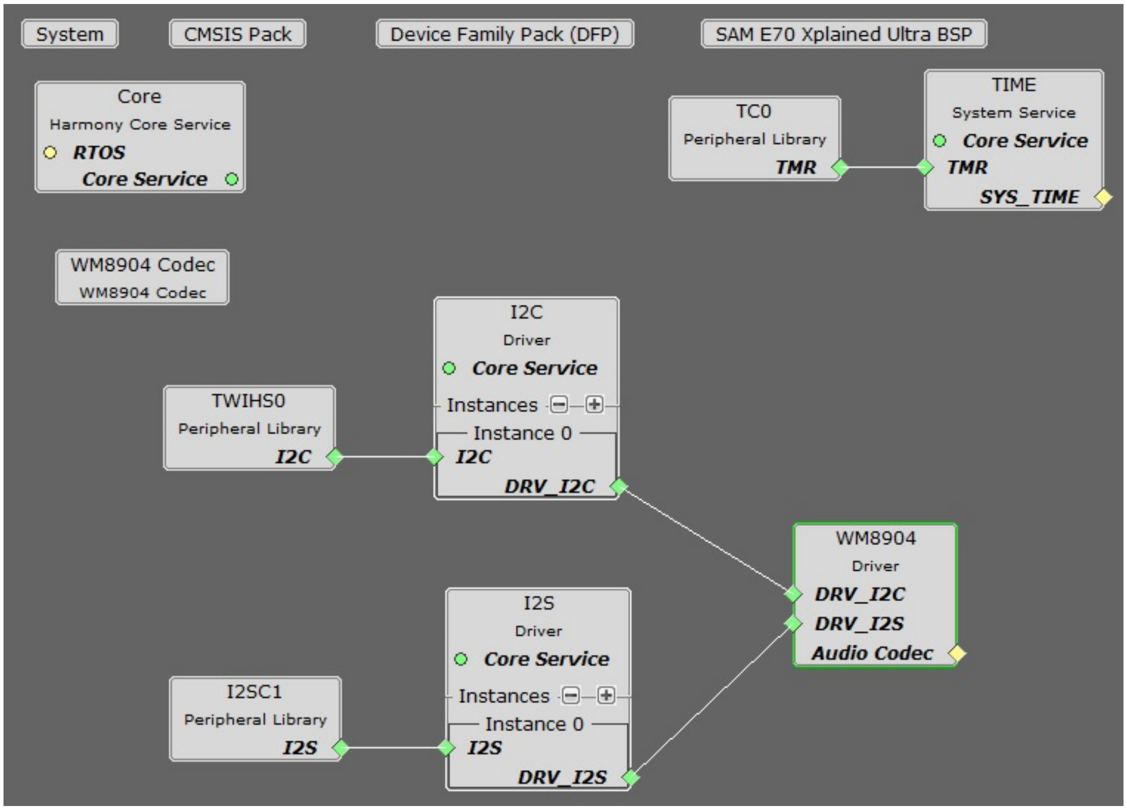
If using FreeRTOS, in the code you will need to move the call to `DRV_WM8904_Tasks(sysObj.drvwm8904Codec0);` from the `SYS_Tasks` function in src/config/<config_name>/`tasks.c` to inside the `while(1)` loop of _`APP_Tasks` (just before the call to `APP_Tasks`).

For projects using the I2SC interface and the WM8904 as a Slave (the SAM E70 generates the I^{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 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.

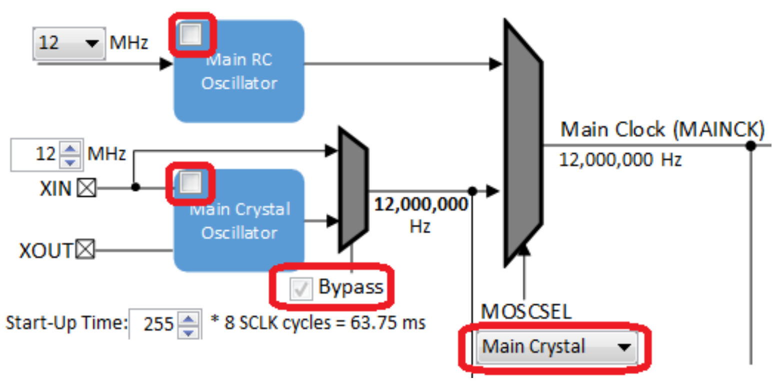
In the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead of 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:



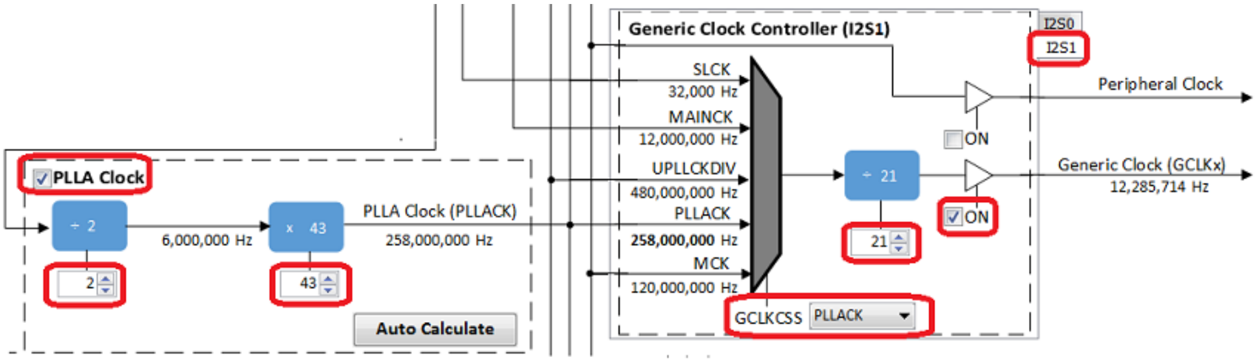
Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate.

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.

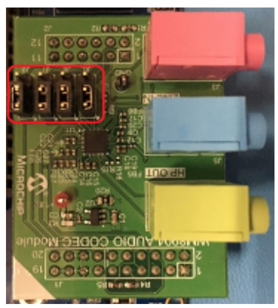
Configuring the Hardware

This section describes how to configure the supported hardware.

Description

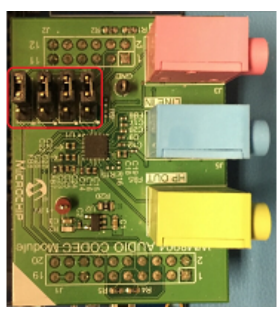
Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, with the SSC PLIB:

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).

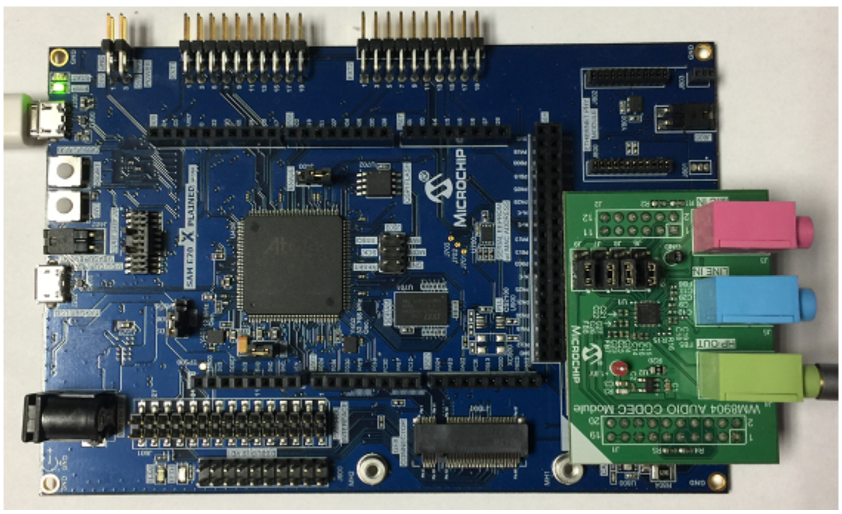


Figure 1: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board

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 (> 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 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 four different projects packaged in this application.

SAM E70 Xplained Ultra Project:

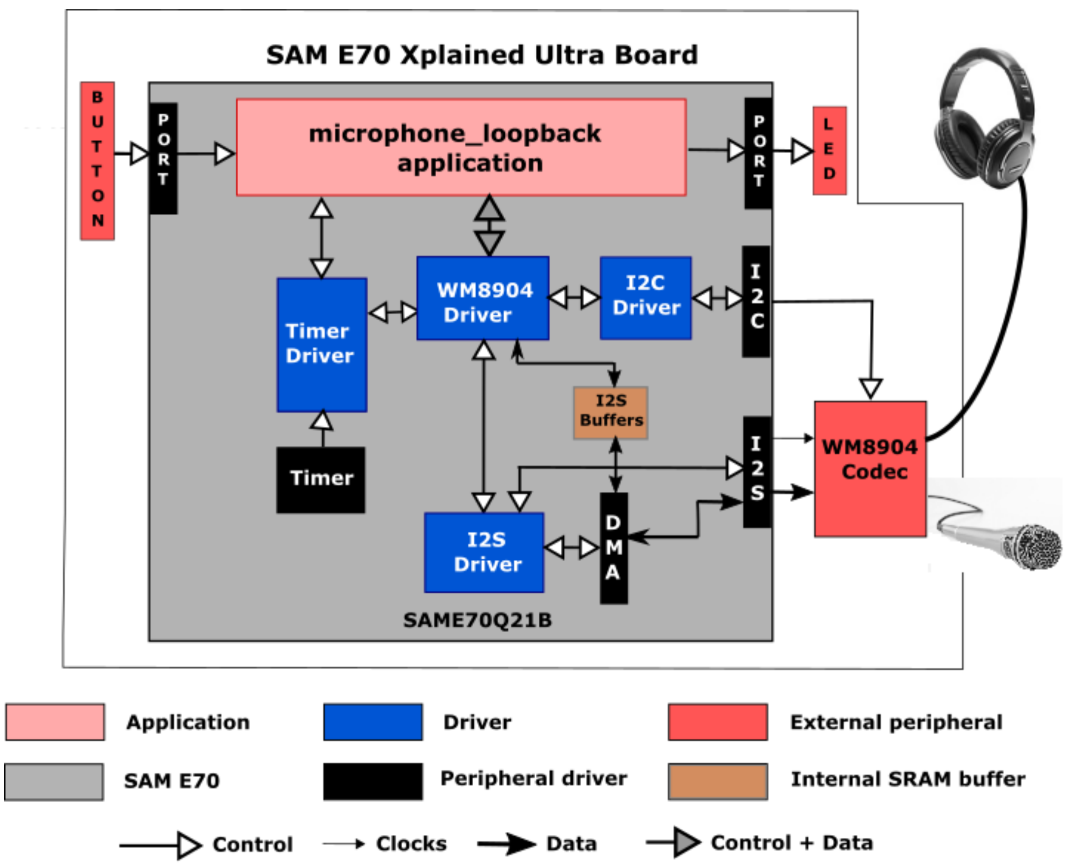
This project runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- One push button
- Two LEDs (only one is used)
- 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% (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 following figure illustrates the application architecture, for the two SAM E70 Xplained Ultra configurations:



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 using FreeRTOS:

This project is identical to the bare-metal version above, except that it uses FreeRTOS. The program takes up less than 1% (18 KB) of the ATSAME70Q21B microcontroller's program space. The 16-bit configuration uses 84% (324 KB) of the RAM. No heap is used.

The application architecture is the same as the previous project, except for the addition of the FreeRTOS block (not shown)

SAM E70 Xplained Ultra Project with LCD display:

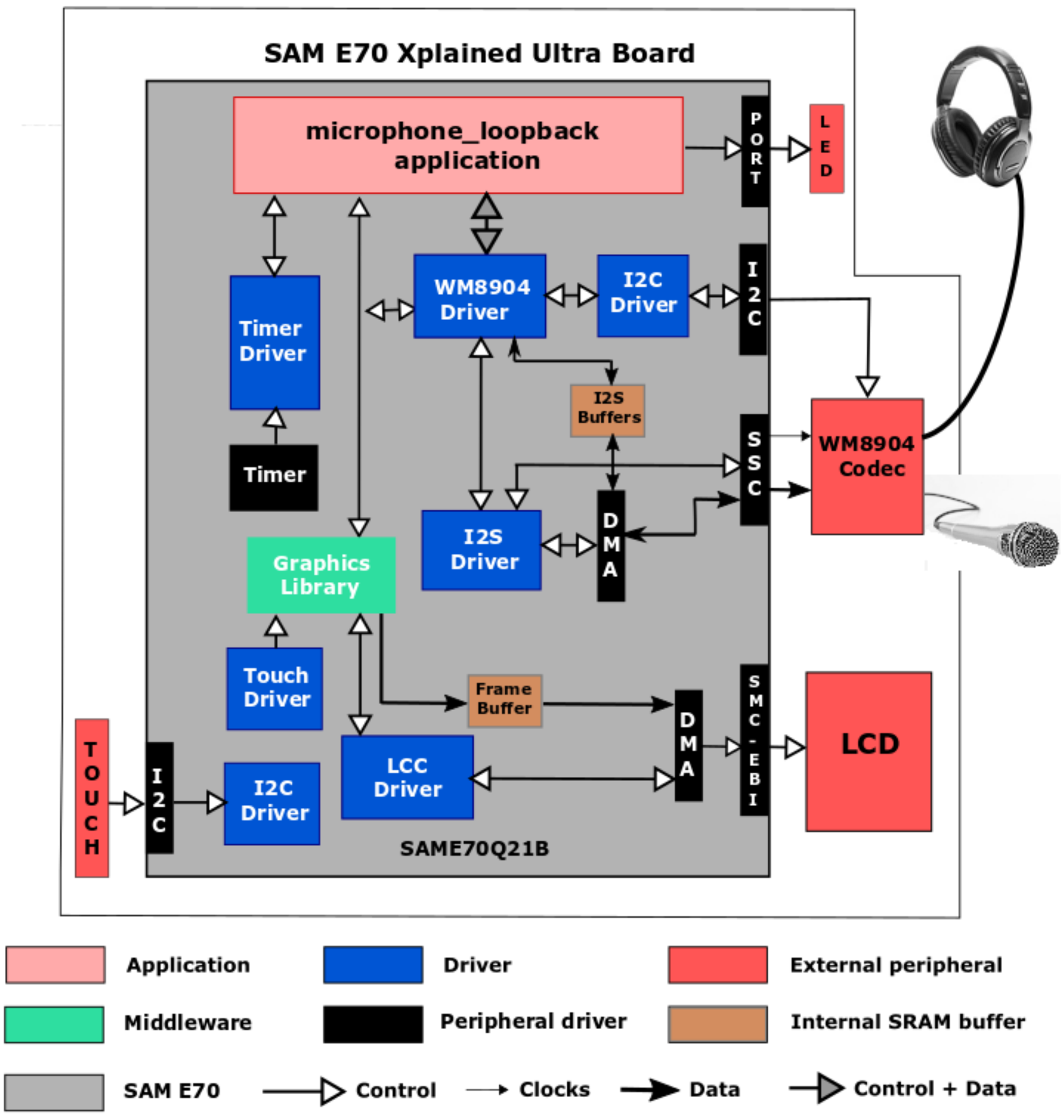
This project runs on the SAM E70 Xplained Ultra Board, which contains a ATSAME70Q21B microcontroller with 2 MB of Flash memory and 384 KB of RAM running at 300 MHz using the following features:

- 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:

- One push button
- Two LEDs (only one 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 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 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 Peripheral Clock section of the SAM E70/V71, and is fixed at 12 MHz.

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), 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 WM8904, and later write them back out 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
- 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)

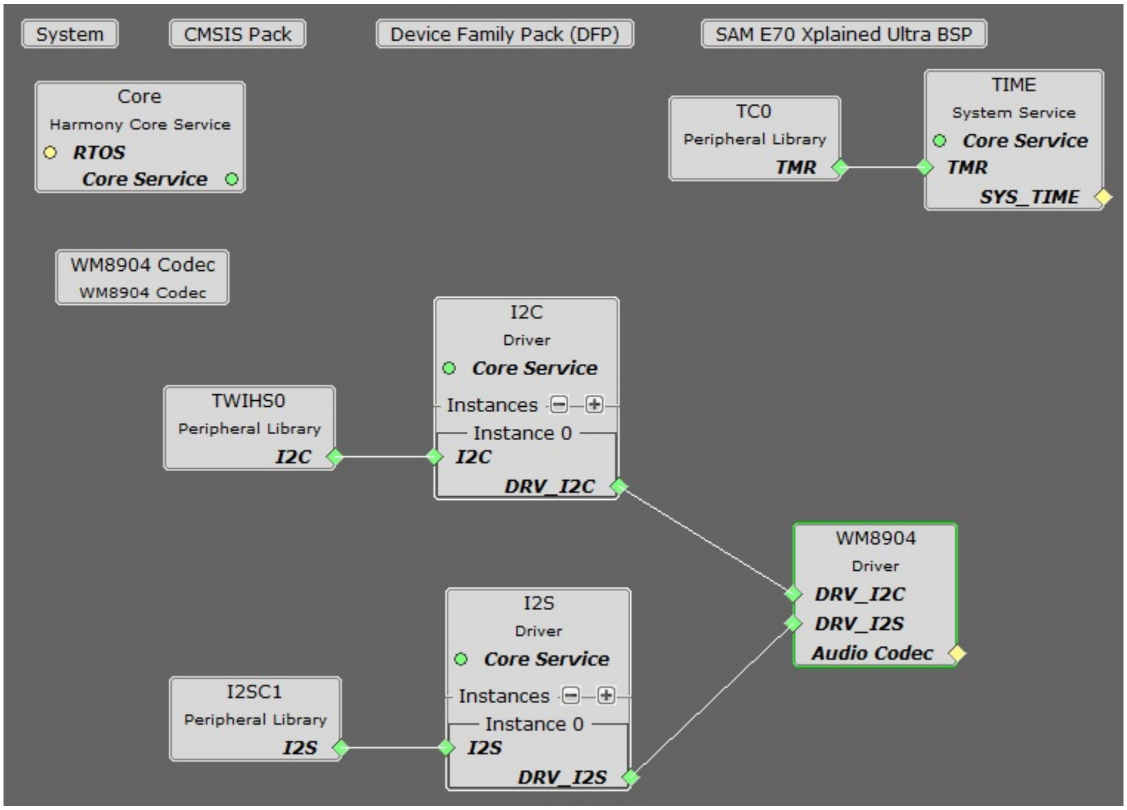
Tools Setup Differences

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.

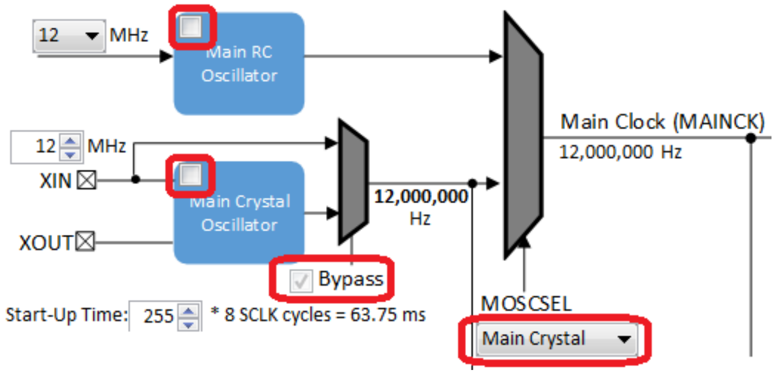
In the the MHC, under Available Components select the BSP SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions. Click on the WM8904 Codec component (*not* the WM8904 Driver). In the Configuration Options, under WM8904 Interface, select I2SC instead 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:



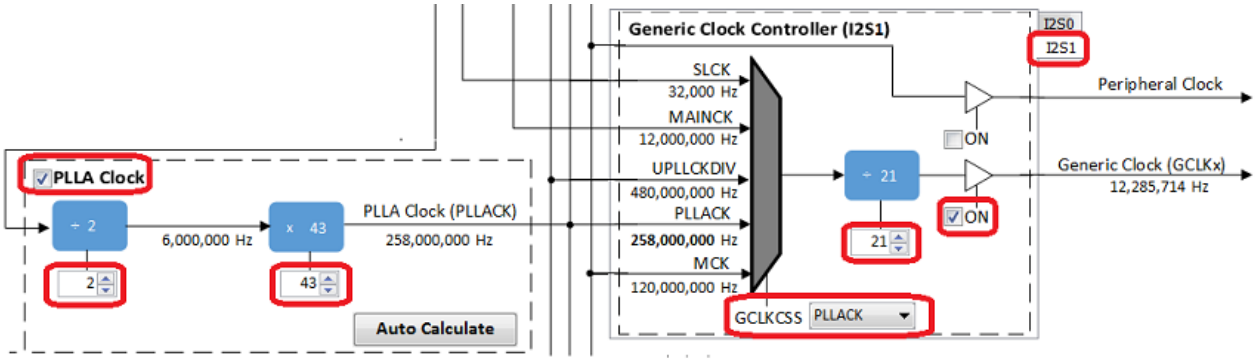
Click on the WM8904 Driver. In the Configurations Options, under Usage Mode, change Master to Slave. Set the desired Sample Rate if different from the default (48,000) under Sampling Rate. Select Enable Microphone Input, and Enable Microphone Bias for elecret microphones if appropriate.

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.

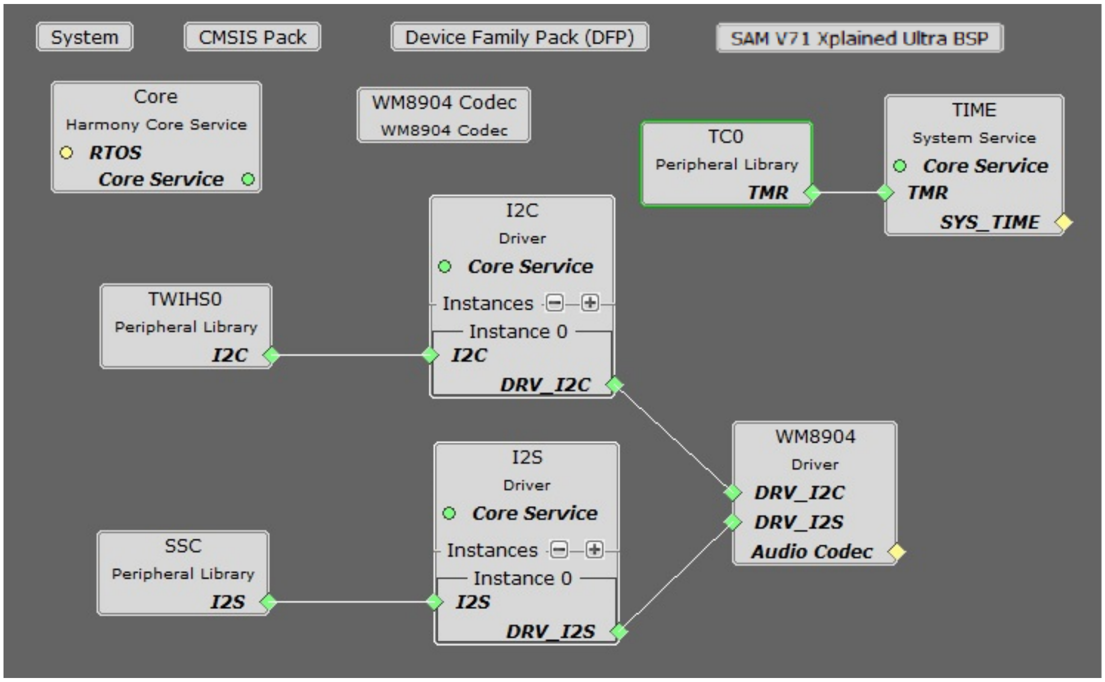
If using FreeRTOS, in the code you will need to move the call to DRV_WM8904_Tasks(sysObj.drvwm8904Codec0); from the SYS_Tasks function in src/config/<config_name>/tasks.c to inside the while(1) loop of _APP_Tasks (just before the call to APP_Tasks).

For projects using the SSC interface and the WM8904 as a Master (the WM8904 codec generates the I^{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 the 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.

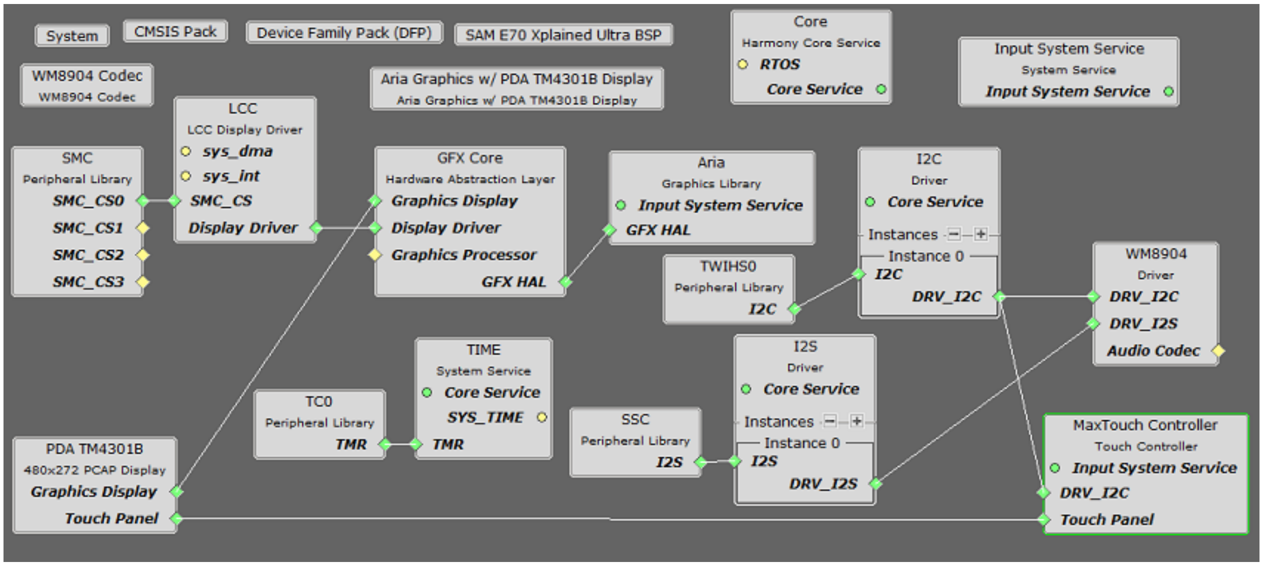
You should end up with a project graph that looks like this, after rearranging the boxes, assuming a non-FreeRTOS project:



If building an application that is goling to use the 480x272 displa, 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.

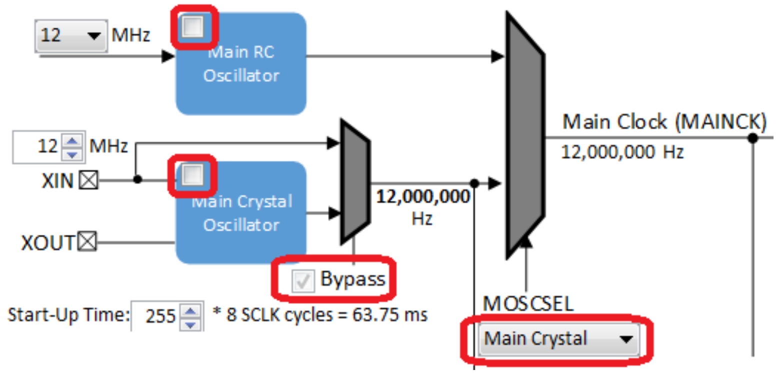
Then under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions.

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_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 sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^2S audio protocol using the I2SC PLIB.
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 sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^2S audio protocol using the I2SC PLIB. This demonstration also uses FreeRTOS.
mic_loopback_sam_e70_xult_wm8904_i2sc_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 sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^2S audio protocol using the I2SC PLIB.

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 sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^2S audio protocol using the SSC PLIB.

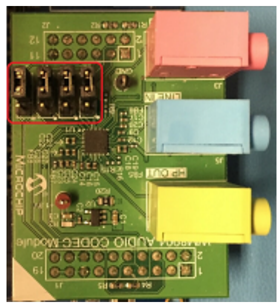
Configuring the Hardware

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

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

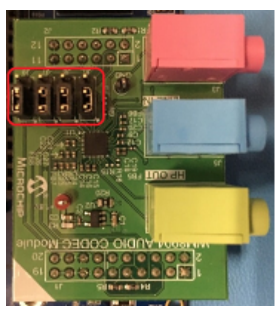
All jumpers on the WM8904 should be toward the **back:**



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.

For the project using 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.

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.

For the project using the SAM V71 Xplained Ultra board with on-board WM8904, using 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:

Four different delays can be generated. **Table 1** 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 WM8904 Audio Codec Daughter Board (see **Figure 1**), or the HEADPHONE jack of the SAM V71 Xplained Ultra board.
2. Connect a microphone to the MIC IN jack of the WM8904 Audio Codec Daughter Board (see **Figure 1**), or the MICROPHONE jack of the SAM V71 Xplained Ultra board.
3. Initially the program will be in delay-setting mode (LED1 on) at a medium volume setting. Pressing SW1 with LED1 on will cycle through four delay settings.
4. Pressing SW1 longer than one second will change to volume-setting mode (LED1 off). Pressing SW1 with LED1 off will cycle through four volume settings (including mute).
5. Pressing SW1 longer than one second again will switch back to delay-setting mode again (LED1 on).

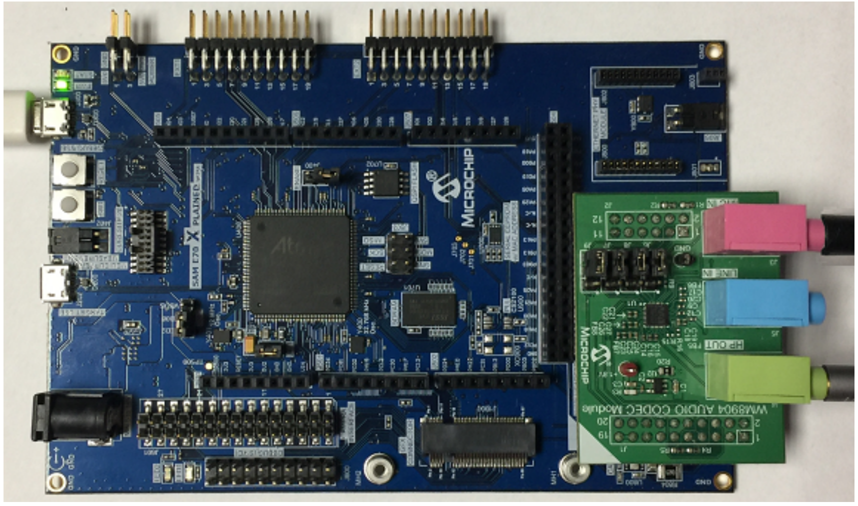


Figure 1: WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board

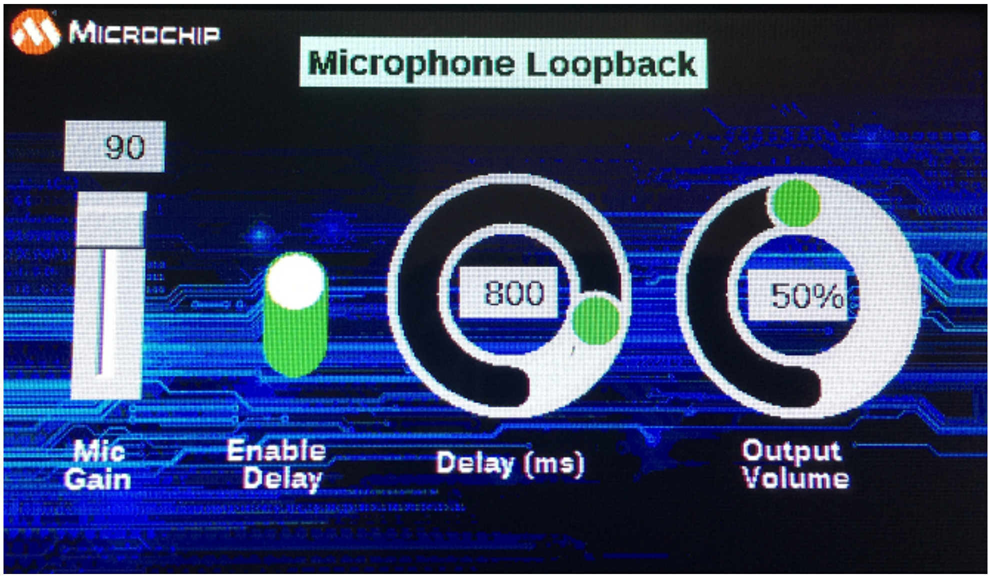
Control Descriptions

Table 1: Button Controls for SAM E70 Xplained Ultra board and SAM V71 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 delays: ¼ second, ½ second, 1 second and 1½ seconds.
SW1 long press (> 1 second)	Alternates between modes (LED1 on or off).

Project using a Graphical User Interface:

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_speaker

This topic provides instructions and information about the MPLAB Harmony 3 USB Speaker Basic 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:

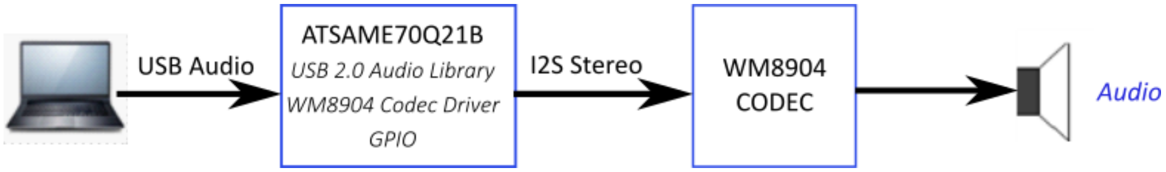
- One push button
- Two LEDs (amber and green). Only the amber LED can be used for a USB Device requiring VBUS sense.
- WM8904 Codec Daughter Board mounted on a X32 socket

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_basic 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
- E70XULT Button processing for volume/mute control
- USB Attach/Detach and mute status using an LED.

The following API calls are made to the USB Library:

- USB_DEVICE_Open - Configures the USB Device
- USB_DEVICE_EventHandlerSet -- Registers APP_USBDeviceEventHandler.
- USB_DEVICE_AUDIO_Read -- Reads a 48Kh/16bit stereo audio buffer.

The following callbacks are registered to the USB Device Library:

- APP_USBDeviceEventHandler - Handles USB interface events
- APP_USBDeviceAudioEventHandler - Handles Read/Write buffer complete events.

Note that all the calls to the USB Library Device Stack library use the form USB_DEVICE_xxx This makes the code more generic, such that another library could be substituted without having to make changes to the application code except for the location of the library public header files.

The codec driver API calls used are:

• DRV_CODEC_Open – Open the driver for write Only.

• DRV_CODEC_BufferEventHandlerSet – Register APP_CODECBufferEventHandler

• DRV_CODEC_SamplingRateSet – Change sampling rate

• DRV_CODEC_BufferAddWrite – add an input buffer to the receive buffer queue

• DRV_CODEC_Mute<On,Off> - Codec mute control

The following callbacks are registered to the Codec driver:

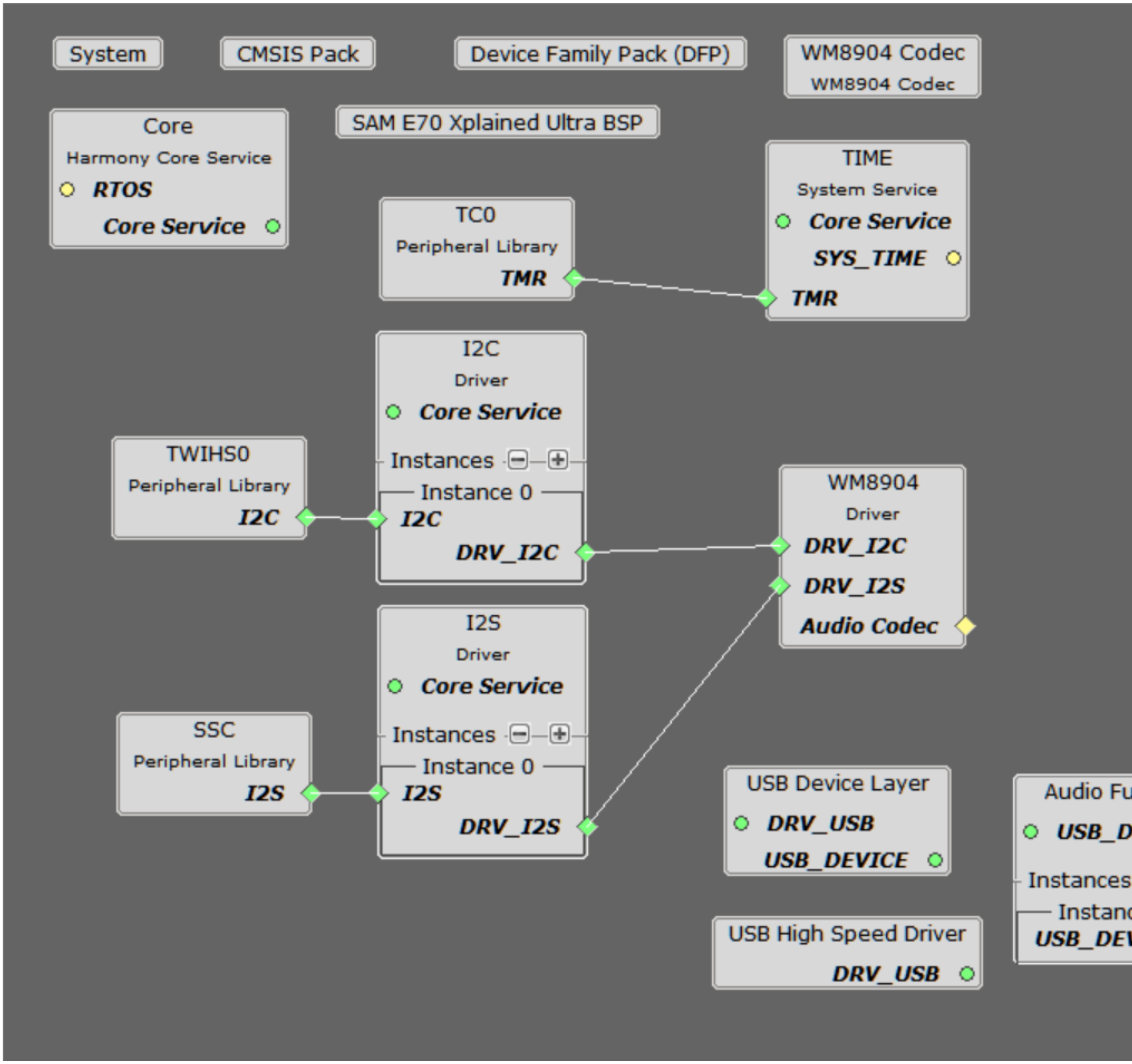
• APP_CODECBufferEventHandler - Handles audio buffer complete events

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).

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

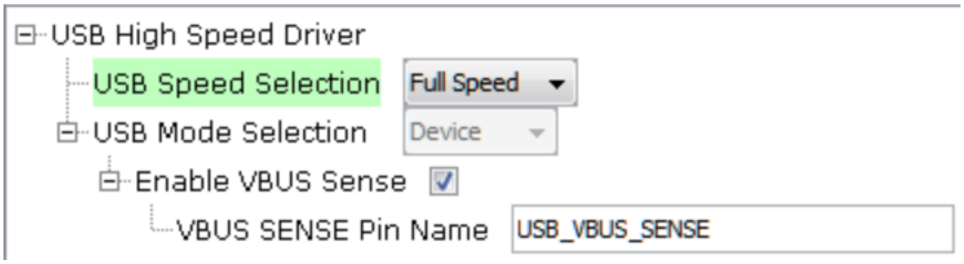
SYS_Tasks, executed from the main polling loop, located in tasks.c.

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

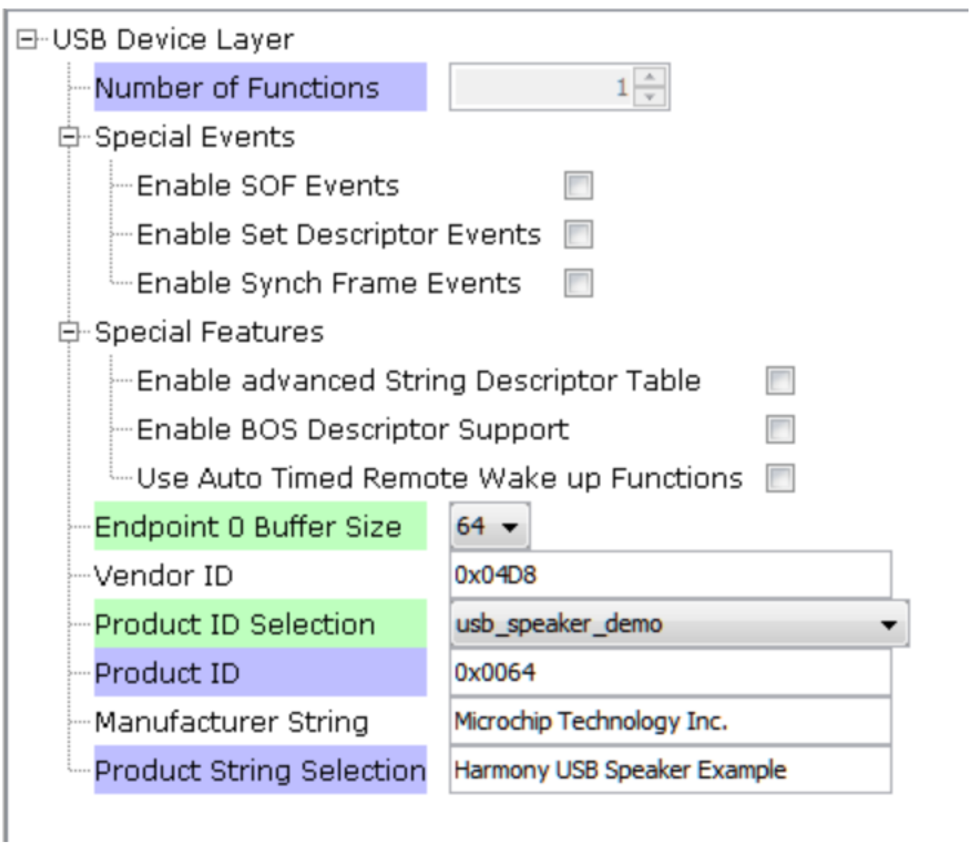
1. Setup the drivers and USB Library interface
2. Respond to USB Host control commands (“Attach”, “Detach”, “Suspend”)
3. Initiate and maintains the bidirectional data audio streaming for the "USB Playback" function.

USB Configuration

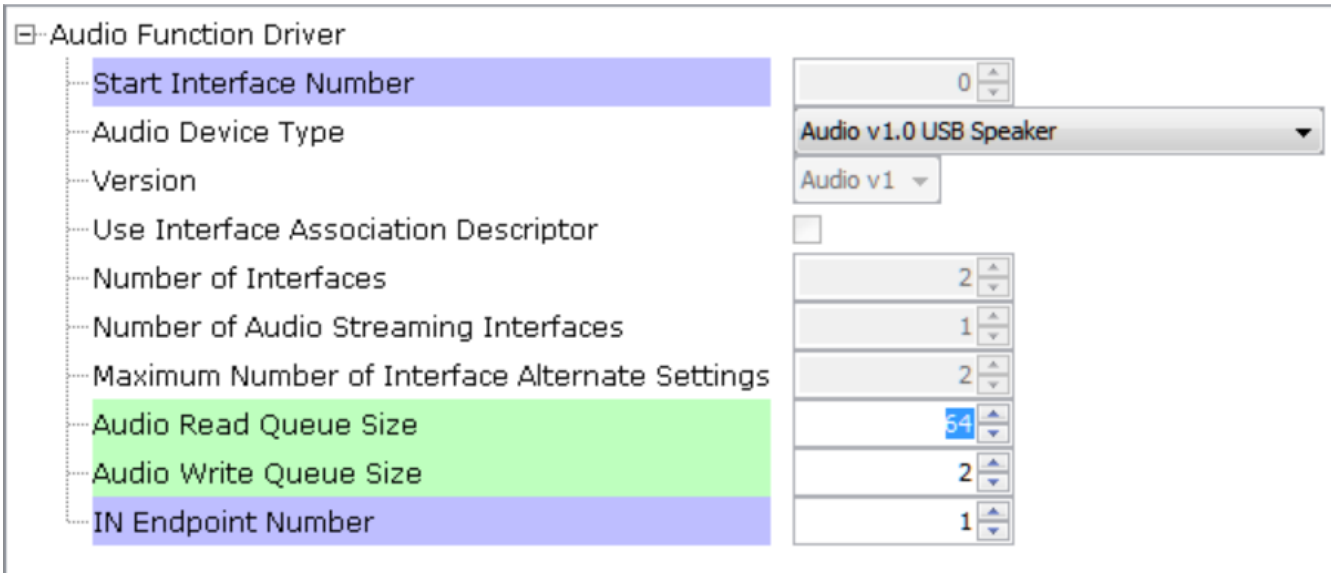
The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is selected to by “Full Speed” (not “High Speed”):



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.

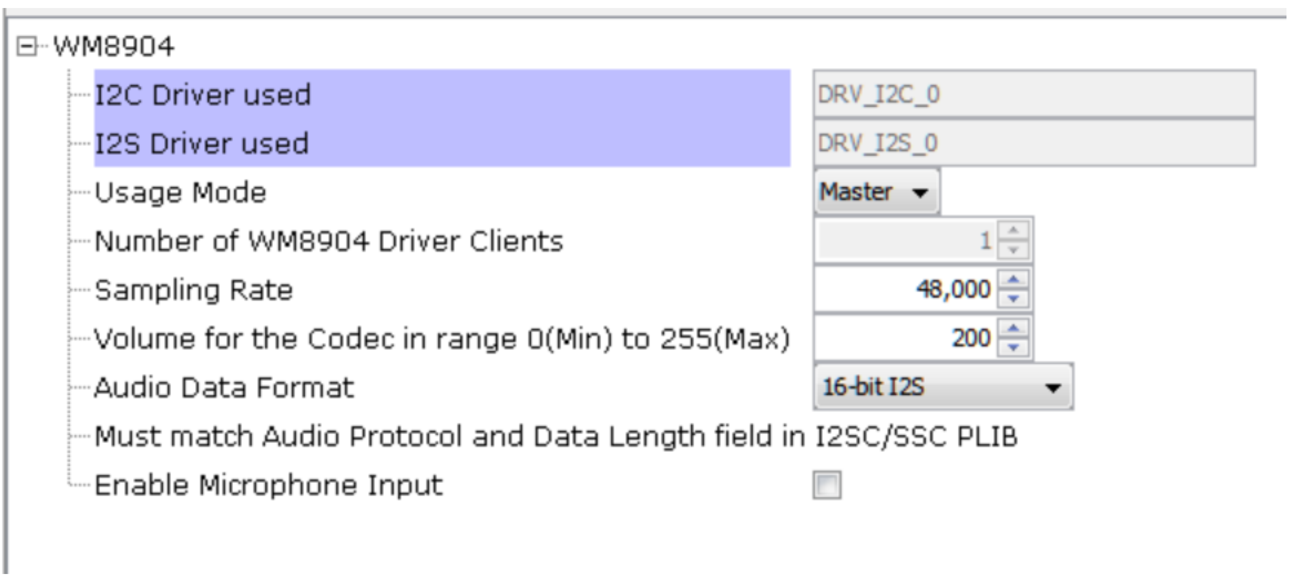


The Audio Function Driver is configured with a Audio Read Queue that matches that of the codec driver write queue for this Audio V1.0 USB Speaker interface.

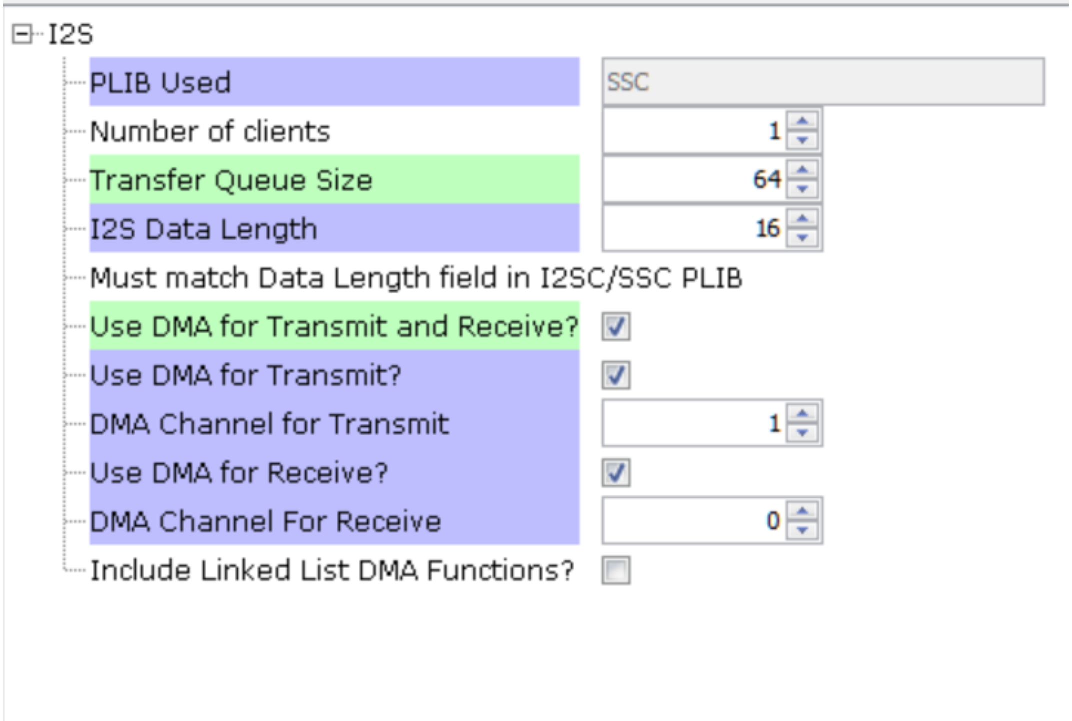


The WM8904 Codec

The WM8904 codec uses a TWIHS (I2C) interface for configuration and control register setting and a SSC (I2S) interface for audio data. 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:

NAME	PORT	Rev 4.0 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 (On Rev4.0 and new Rev2.0) 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

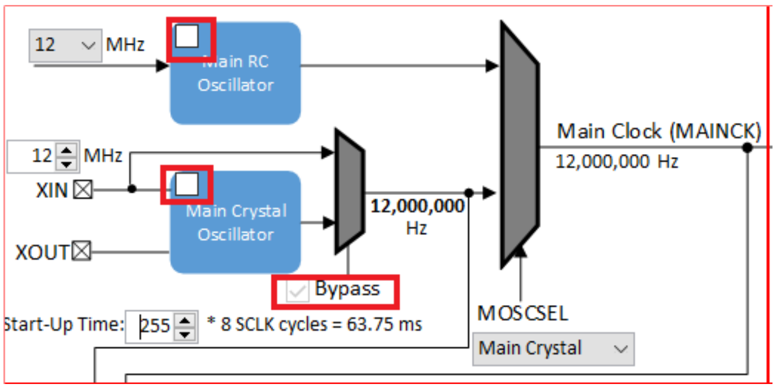
The I2S clocks are setup for 48KHz sampling rate, with stereo 16 bit samples, giving a 32 bit sample frame. The I2S clocks are generated from the WM8904 acting as I2S master using the 12.288 Mhz master clock obtained from Peripheral Clock 2 (PCK2).

The I2S clocks will then be generated, as follows:

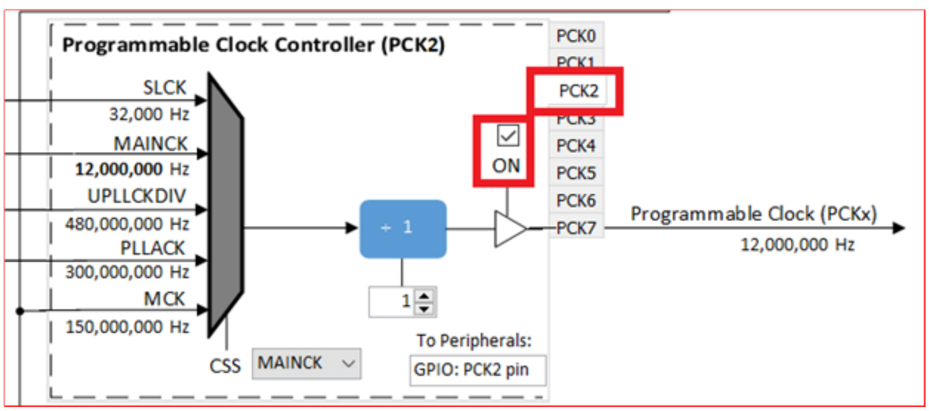
I2S Function	Value	Description
LRCK	48.000000 K	Sample rate clock
BCLK	3072000 Hz	Bit Rate Clock
MCLK	12.288000 MHz	Master Clock

MPLAB Harmony Configurator: Tools>Clock Configuration

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.

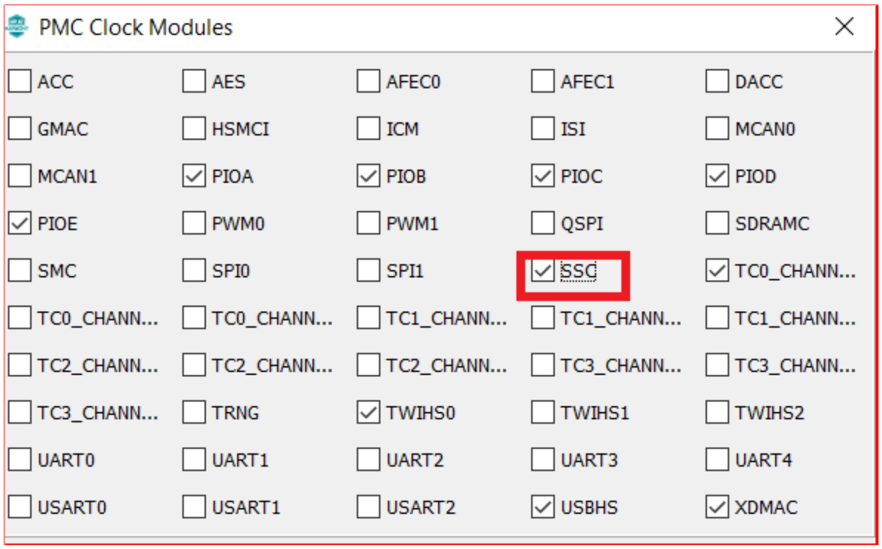


Enable the PCK2 output to enable the WM8904 master clock generation, to enable clocking for the SSC operating as a slave I2S:



Clock Diagram>Peripheral Clock Enable

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



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_basic/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.

Project Name	Location
`usb_speaker_basic_sam_e70_xult_wm8904_usb.X`	`<install-dir>/apps/audio/usb_speaker/firmware`

MPLAB X IDE Project Configurations

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

Project Configuration Name	BSP Used	Description
usb_speaker_basic_sam_e70_xult_wm8904_usb	sam_e70_xult	This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board

Configuring the Hardware

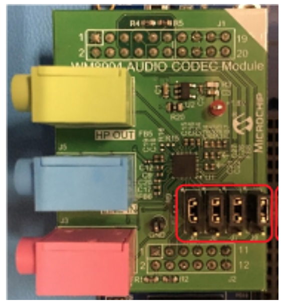
This section describes how to configure the supported hardware.

Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, using the 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. 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 jack of the WM8904 Audio Codec Daughter Board (see Figure 1).

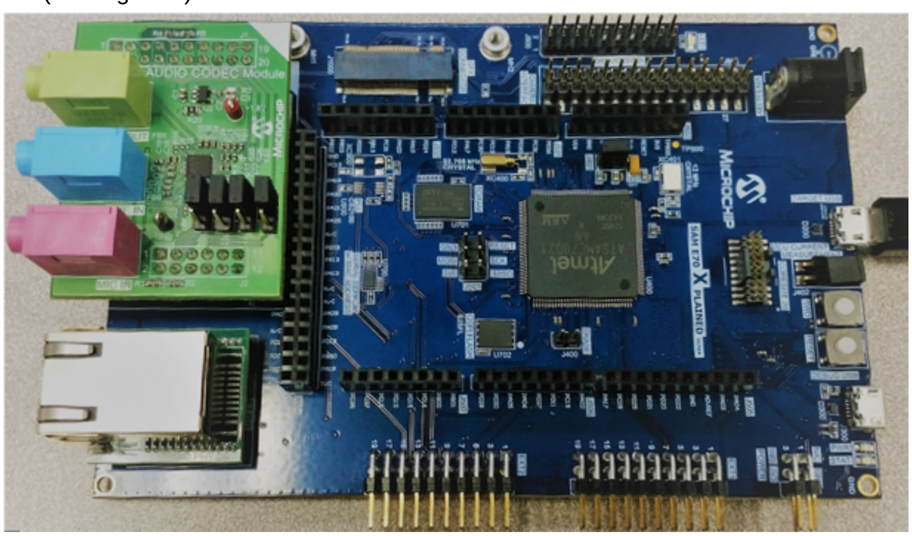


Figure 1. WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green.

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 LED 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.

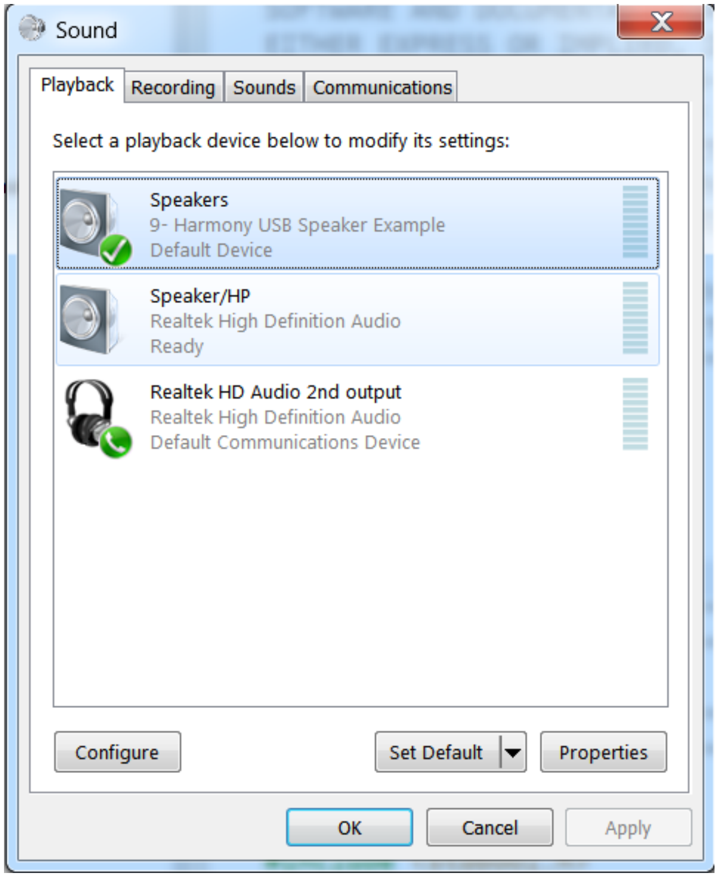


Figure 2. Windows 7 Sound Dialog showing USB Speaker with Sound Level Meter

5. If needed, configure the Host computer to use the usb_speaker outputs as the selected audio device. This may be done in the system configuration or "Control Panel" depending on the operating system. For Windows, this is done in the Playback Devices dialog, which is accessed by right clicking the loudspeaker icon in the taskbar.
6. Open a playback application (such as Window Media Player) and initiate playback through the USB Speaker.
7. Playback of the should demonstrate that the audio is being heard in the USB Speaker headphones.

Control Description

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

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

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

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

LED Indication	Status
OFF	USB cable detached
ON	USB cable attached
Blinking	Playback muted

Audio Driver Libraries Help

This section provides descriptions of the audio driver libraries that are available in MPLAB Harmony.

Description

WM8904 CODEC Driver Library Help

This topic describes the WM8904 Codec Driver Library.

Introduction

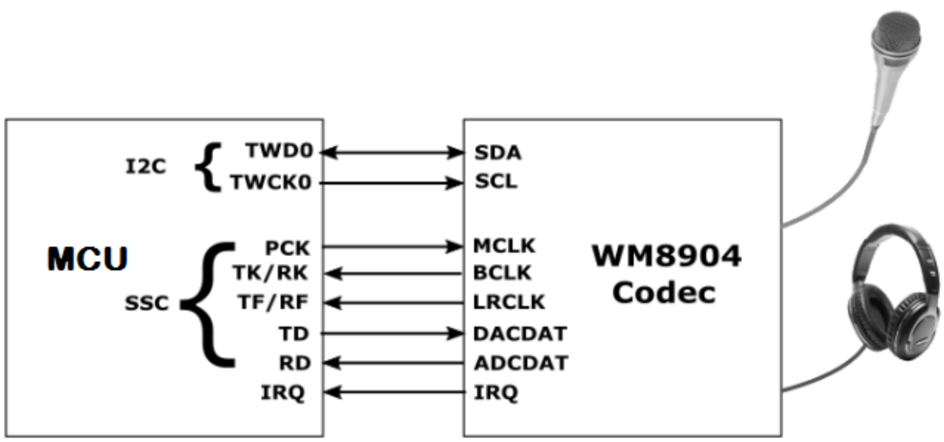
This library provides an Applications Programming Interface (API) to manage the WM8904 Codec that is serially interfaced to the I^2C and I^2S peripherals of a Microchip microcontroller for the purpose of providing audio solutions.

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`

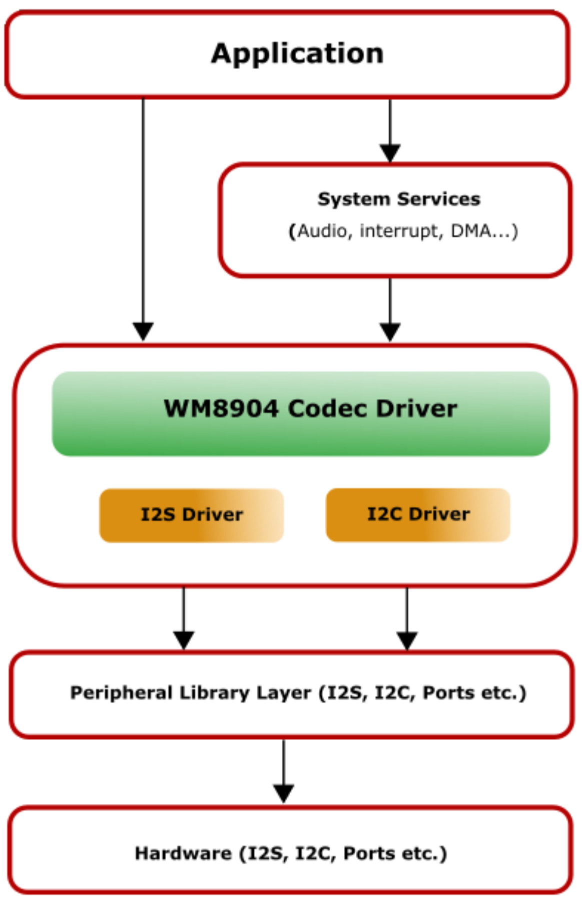
Abstraction Model

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

Configuring MHC

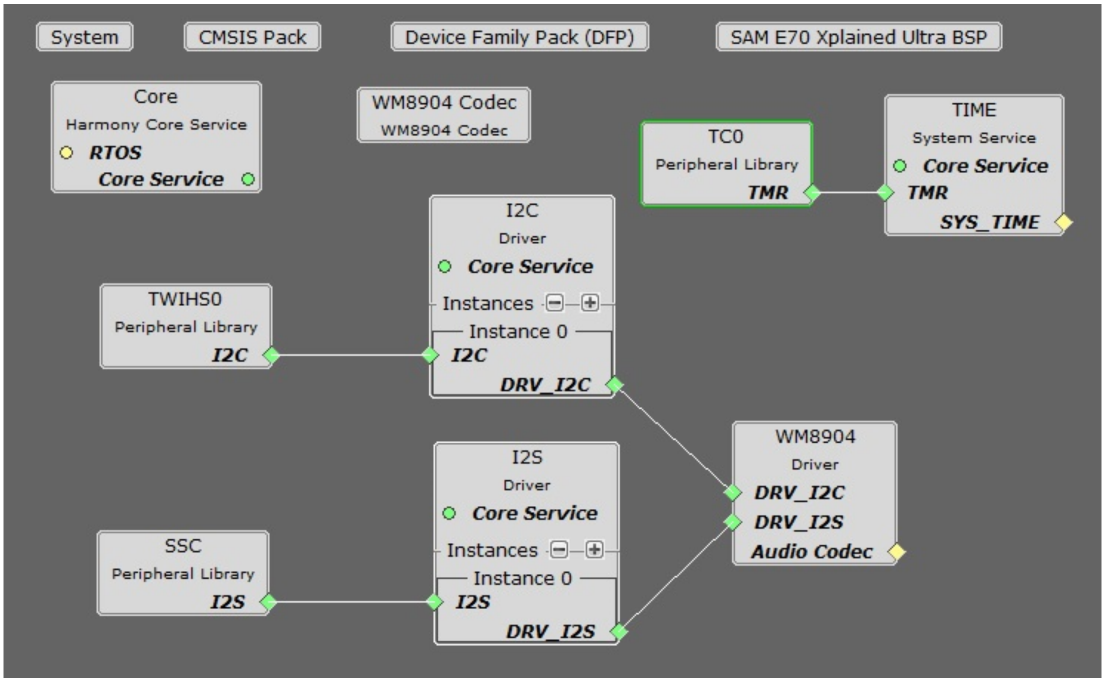
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

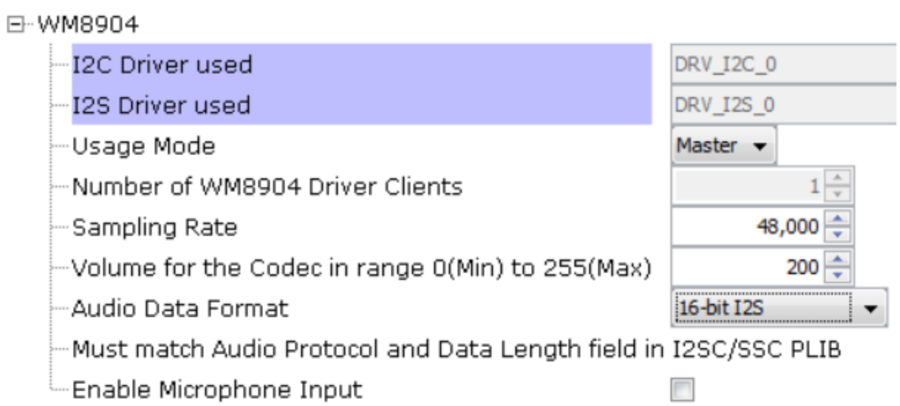
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

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. **Implementation:** Dynamic
	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_I2C_INDEX	This is macro DRV_I2C_INDEX.
	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

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_MuteOn	This function allows WM8904 output for soft mute on.
	DRV_WM8904_MuteOff	This function disables WM8904 output for soft mute.
	DRV_WM8904_SamplingRateGet	This function gets the sampling rate set on the WM8904. **Implementation:** Dynamic
	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.

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

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_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.

Implementation: Dynamic

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_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_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_SamplingRateGet Function

uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle)

Summary

This function gets the sampling rate set on the WM8904.

Implementation: Dynamic

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 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_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 Enumeration

in bits

C

```
typedef enum {
  DATA_LENGTH_16,
  DATA_LENGTH_24,
  DATA_LENGTH_32
} 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;
    uint8_t micGain;
} DRV_WM8904_INIT;
```

DRV_I2C_INDEX Macro

This is macro DRV_I2C_INDEX.

C

```
#define DRV_I2C_INDEX DRV_WM8904_I2C_INSTANCES_NUMBER
```

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
```

Files

Files

Name	Description
drv_wm8904.h	WM8904 Codec Driver Interface header file

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.

Index