MPLAB Harmony Audio Help

MPLAB Harmony Integrated Software Framework

© 2013-2018 Microchip Technology Inc. All rights reserved.

Audio Overview

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

Description

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

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

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

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

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

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

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

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

Legal

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

Digital Audio Basics

This topic covers key concepts in digital audio.

Description

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

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

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

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

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

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

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

Common Audio Formats

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

I2S Audio

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

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

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

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

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

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

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

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

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

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

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

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

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

Digital Audio in Harmony

This topic describes how digital audio is implemented in Harmony.

Description

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

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

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

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

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

Example Audio Projects

Example projects showing Harmony audio in action.

Description

The audio_enc Demonstration

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

The audio_player_basic Demonstration

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

The audio_signal_generator Demonstration

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

The audio_tone Demonstration

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

The audio_tone_linkeddma Demonstration

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

The microphone_loopback Demonstration

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

The universal_audio_decoders Demonstration

The audio_player_basic application scans a mass storage device (e.g. thumb drive) connected to the USB Host port for wave (.wav), ADPCM (.pcm) or MP3 (.mp3) files, and then plays them out one by one through a codec. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA, Timer and USB library. Some version use a touch display to implement the controls.

The usb_microphone Demonstration

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

The usb_speaker Demonstration

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

The usb_speaker_bass_boost Demonstration

The usb_speaker application acts as a USB device, playing streaming audio from a USB host such as a PC. It uses the I2S Driver, I2C Driver, codec driver (e.g. WM8904), I2S peripheral (e.g. SSC or I2SC), DMA and Timer. Audio is processed through a "Bass Boost" filter (using either the CMSIS-DSP or the PIC32 MZ EF DSP library functions) prior to playback.

The usb_speaker_hi_res Demonstration

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

Audio Demonstrations

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

Introduction

MPLAB Harmony Audio Demonstrations Help.

Description

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

Demonstrations

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

audio_enc

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

Description

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

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

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

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

Supported audio files are as represented in the table below.

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

Architecture

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

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

Architecture Block Diagram

Demonstration Features

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

Harmony Configuration

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

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

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

Tools Setup Differences

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

<u>MPLAB Harmony Configurator: Tools->Clock Configuration</u>

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

Enable the PLLA Clock output.

Enable clocking for the I2S1.

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

MPLAB Harmony Configurator: WM8904

MPLAB Harmony Configurator: File System

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

MPLAB Harmony Configurator: I2C Driver

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

MPLAB Harmony Configurator: System

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

MPLAB Harmony Configurator: EFC

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

MPLAB Harmony Configurator: Audio Decoder Libraries

Disable the ADPCM Decoder for this particular app.

Save and generate code for the framework.

Building the Application

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

Description

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

MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

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

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

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

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

Control Description

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

Button control is shown in the table below.

Button Operations

Operation	Function
Long Press (> 1 sec)	Play back the last encoded
Short Press (< 1 sec)	Start/Stop Encoding

Status Indicator Description

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

LED status indication is shown in the table below.

LED Status

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

audio_player_basic

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

Description

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

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

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

Supported audio files are as represented in the table below.

Supported Format

Audio Format	Sample Rate (kHz)	Description
PCM	8 to 96	PCM (Pulse Code Modulation) is an uncompressed format. The digital data is a direct representation of the analog audio waveform. The container for the data will be a WAVE file (.wav) format. It is the native file format used by Microsoft Windows for storing digital audio data.

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

Architecture

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

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

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

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

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

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

Architecture Block Diagram

Demonstration Features

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

Harmony Configuration

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

a. Yes to all popups.

3. Add Audio->Templates->WM8904 Codec

a. Yes to all popups.

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

a. Yes to all popups.

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

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

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

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

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

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

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

Tools Setup Differences

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

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

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

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

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

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

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

Enable the PCK2 output to enable the WM8904 master clock:

Clock Diagram>Peripheral Clock Enable

Enable clocking for the SSC.

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

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

In the MPLAB Harmony Configurator: *System* dialog:

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

Building the Application

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

Description

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

MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

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

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

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

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

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

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

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

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

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

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

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

3. For the E70 board:

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

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

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

Control Description

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

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

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

Button control is shown in the table below.

Button Operations

Long Press (> 1 sec)	Short Press (< 1 sec)
Volume Control	Low (-66 dB)
	Medium (-48 dB)

	High (0 dB)
	Mute/Pause
Linear Track Change	Next sequential track

Status Indicator Description

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

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

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

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

LED status indication is shown in the table below.

LED Status

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

audio_signal_generator

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

Description

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

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

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

Architecture

SAM E70 Xplained Ultra Projects:

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

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

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

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

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

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

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

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

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

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

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

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

Two buffers are used for generating the various waveformse in a ping-pong fashion. Initially values for the first buffer are calculated, and then the buffer is handed off to the DMA using a `DRV_CODEC_BufferAddWrite`. While the DMA is transferring data to the SSC peripheral, causing audio to be sent to the codec over I^2S, the program calculates the values for the next cycle.

(In the current version of the program, this is always the same unless the frequency is changed manually.) Then when the DMA callback `Audio_Codec_BufferEventHandler` is called, the second buffer is handed off and the first buffer re-initialized, back and forth.

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

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

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

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

Then the sample value is calculated using the sine function:

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

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

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

Demonstration Features

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

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, plus a PDA TM4301B 480x272 (WQVGA) display. The project configuration is for a sine tone with 16-bit data width, 48000 Hz sampling frequency, and I^2S audio protocol using the SSC PLIB. The WM8904 codec is configured as the master, and the SSC peripheral as the slave.

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

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

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

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

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

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

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

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

audio_tone

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

Description

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

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

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

Architecture

There are ten different projects packaged in this application.

PIC32_MX/MZ Bluetooth Audio Development Kit Projects:

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

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

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

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

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

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

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

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

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

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

PIC32 MZ EF Curiosity 2.0 Project:

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

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

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

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

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

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

SAM E54 Curiosity Ultra Projects:

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

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

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

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

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

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

SAM E70 Xplained Ultra Projects:

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

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

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

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

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

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

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

SAMV71 Xplained Ultra Project:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then the sample value is calculated using the sine function:

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

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

Demonstration Features

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

Tools Setup Differences

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

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

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

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

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

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

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

You should end up with a clock diagram like this:

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

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

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

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

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

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

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

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

You should end up with a clock diagram like this:

<u>For projects using the E54, the I2S interface and the WM8904 as a Master (the WM8904 codec generates the I^{2}S clocks):</u>

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

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

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

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

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

In the Clock Diagram:

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

You should end up with a clock diagram like this:

<u>For projects using the E70, the SSC interface and the AK4954 as a Slave (the SAM E70 generates the I^{2}S clocks):</u>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Building the Application

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

Description

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

MPLAB X IDE Project Configurations

The following table lists and describes supported project configurations.

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

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

All configurations:

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

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

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

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

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

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

Control Descriptions

Table 1: Button Controls for Bluetooth Audio Development Kit

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

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

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

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

Table 3: Button Controls for SAM V71 Xplained Ultra board

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

audio_tone_linkeddma

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

Description

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

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

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

Architecture

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then the sample value is calculated using the sine function:

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

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

Demonstration Features

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

Tools Setup Differences

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

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

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

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

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

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

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

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

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

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

<u>For projects using the I2SC interface and the WM8904 as a Slave (the SAM E70 generates the I^{2}S clocks):</u>

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

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

Architecture

There are eight different projects packaged in this application.

PIC32 MZ Bluetooth Audio Development Kit Project:

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

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

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

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

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

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

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

SAM E54 Curiosity Ultra Projects:

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

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

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

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

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

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

SAM E70 Xplained Ultra Projects (no display):

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

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

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

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

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

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

SAM E70 Xplained Ultra Project with LCD display:

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

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

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

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

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

SAM V71 Xplained Ultra Project:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Demonstration Features

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

Tools Setup Differences

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

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

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

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

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

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

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

You should end up with a clock diagram like this:

<u>For projects using the E54, the I2S interface and the WM8904 as a Master (the WM8904 codec generates the I^{2}S clocks):</u>

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

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

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

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

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

In the Clock Diagram:

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

You should end up with a clock diagram like this:

<u>For projects using the E70, the SSC interface and the AK4954 as a Slave (the SAM E70 generates the I^{2}S clocks):</u>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Building the Application

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

Description

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

MPLAB X IDE Project Configurations

The following table lists and describes supported configurations.

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

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

Configuring the Hardware

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

Description

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

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

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

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

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

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

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

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

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

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, and the SSC PLIB and the 480x272 display.

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

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

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

Projects using a Pushbutton LED User interface:

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

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

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

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

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

Control Descriptions

Table 1: Button Controls for Bluetooth Audio Development Kit

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

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

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

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

Table 3: Button Controls for SAM V71 Xplained Ultra board

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

Project using a Graphical User Interface:

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

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

universal_audio_decoders

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

Description

The Universal Audio Decoders application uses the MPLAB Harmony Decoder Library to decode music files on a USB thumb drive.

The application configures the development board to be in USB Host mode, and supports the FAT32 file system. When a mass storage device is connected to the development board, the application begins to scan the root directory. Once the scan is complete, the first track in the list will be opened and played. The fill buffer routine will read a chunk of data as an input to the decoder of the supported file format. The decoder will decode the packet and send the first frame of audio data to the codec driver through I2S Driver and the output will be audible through the speakers. The following block diagram depicts a representation of the application.

Button or touch controls provide typical VCR-like control over the playback, such as play/pause, rewind, and next track.

The application currently supports ADPCM, WAV and MP3 files:

Audio Format	Decoder Location	Sampling Rates (kHz)	Description
ADPCM	audio/decoder	8, 16	Adaptive Delta Pulse Code Modulation (ADPCM) is a sub-class of the Microsoft waveform (`.wav`) file format. In this demonstration, it only decodes ADPCM audio which has a WAV header. The extension name for this format is pcm.
WAVE	audio/decoder	8 through 96	The WAVE file format is the native file format used by Microsoft Windows for storing digital audio data.

MP3	helix_mp3	8 through 48	The MPEG 2.5 Layer 3 is a lossy digital compression formats for audio data. The current version of the MP3 Library is an open-source based on the Helix MP3 decoder used in RealPlayer. The version in MPLAB Harmony is v1.0.

Architecture

There are four different projects packaged in this application.

PIC32 MZ EF Curiosity 2.0 Project:

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

- Four push buttons (SW1-SW4, only SW1 is used)
- Four LEDs (LED1-LED4, only LED1 is used)
- AK4954 Codec Daughter Board mounted on X32 HEADER 2 socket
- Optional PDA TM4301B 480x272 (WQVGA) display

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

The following figure illustrates the application architecture for the PIC32 MZ EF Curiosity 2.0 project without a display. The program takes up to approximately 9% (188 KB) of the PIC32MZ2048EFM144 microcontroller's program space, and 14% (71 KB) of the RAM. No heap is used.

This figure illustrates the application architecture for the PIC32 MZ EF Curiosity 2.0 project with a display. The program takes up to approximately 32% (656 KB) of the PIC32MZ2048EFM144 microcontroller's program space, and 66% (340 KB) of the RAM, including a 14 KB heap.

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

SAM E70 Xplained Ultra Projects:

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

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

The following figure illustrates the application architecture for the SAM E70 Xplained Ultra project without a display. The program takes up to approximately 5% (110 KB) of the ATSAME70Q21B microcontroller's program space, and 19% (71 KB) of the RAM. No heap is used.

This figure illustrates the application architecture for the SAM E70 Xplained Ultra project with a display. The program takes up to approximately 14% (277 KB) of the ATSAME70Q21B microcontroller's program space, and 21% (77 KB) of the RAM, including a 14 KB heap.

The SSC (Synchronous Serial Controller) is used with the WM8904 codec, selected by a strapping option on the WM8904 daughterboard. 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.

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
- AK4954 or 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)
- For the projects using the display, use of the touch screen as a UI

Tools Setup Differences

For the projects using the PIC32MZ, the I2S interface and the AK4954 as a Slave (the I2S peripheral generates the I^2S clocks), with or without a display:

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File* > *New*

Project. Chose the Configuration name based on the BSP. Select the appropriate processor (PIC32MZ2048EFM144 for the PIC32 MZ EF Curiosity 2.0). Click Finish.

In the MHC, under Available Components select the BSP PIC32 MZ EF Curiosity 2.0. Under *Audio>Templates*, double-click on AK4954 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one. Then under *Libraries>USB>Host Stack*, double-click on MSD Client Driver. Finally, under *Harmony>System Services*, double-click on FILE SYSTEM.

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

Connect the MSD Client Driver block to the FILE SYSTEM block. Click on the FILE SYSTEM block.

Under File System Settings, check the checkbox for Use File System Auto Mount Feature. Under Media0>Media Configuration0, change the Media Type to SYS_FS_MEDIA_TYPE_MSD.

Click on the I2S Peripheral. In the Configurations Options, under 32/16-Bit Communication Select bits, select (AUDEN=1) 16-bit data, 16-bit FIFO, 32-bit Channel/64-bit Frame. Change Master Clock the Baud Rate to 44,100. Set the Frame Select Pin to C2. The configuration should look like this:

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

- Change the Primary Oscillator frequency to 12,000,000 Hz, and select EC under POSCMOD.
- Change FPLLICLK to POSC, FPLLIDIV to DIV_2, and FPLLMULT to MUL_66 for an SPLL value of 198 MHz. Change the UPLLFSEL to FREQ_12MHZ.
- In the Reference Clock #1 section, check the ON and OE checkboxes,, and set ROTRIM1 to 393 and RODIV1 to 8. This should give a REFCLKO1 output of 11,291,601 Hz, which is 256 times the 44,100 Hz sample rate.

You should end up with a clock diagram like this:

For the project using the PIC32 MZ EF Curiosity 2.0 board with the TM4301B WQVGA display, under *Graphics>Templates*, you will also want to double-click on Aria Graphics w/ PDA TM4301B Display. You will end up with a project graph that looks like this instead.

Follow the instructions above for the non-display version re the File System, I2S and clock settings.

For projects using the E70, the SSC interface and the WM8904 as a Master (the WM8904 codec generates the I^2S clocks), with or without a display:

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

In the MHC, under Available Components select the appropriate BSP (SAM E70 Xplained Ultra). Under *Audio>Templates*, double-click on WM8904 Codec. Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one. Then under *Libraries>USB>Host Stack*, double-click on MSD Client Driver. Finally, under *Harmony>System Services*, double-click on FILE SYSTEM.

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

Connect the MSD Client Driver block to the FILE SYSTEM block. Click on the FILE SYSTEM block.

Under File System Settings, check the checkbox for Use File System Auto Mount Feature. Under Media0>Media Configuration0, change the Media Type to SYS_FS_MEDIA_TYPE_MSD.

Click on the WM8904 Driver. In the Configurations Options, set the Sample Rate to 44,100.

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.

For the project using the SAM E70 Xplained Ultra board with the TM4301B WQVGA display and SSD1963 Controller, under *Graphics>Templates*, you will also want to double-click on Aria Graphics w/ PDA TM4301B Display. Then click on the LCC block, and click on the red X in the toolbar remove it. Also delete the TC2 block left behind.

Under *Graphics>Driver,* double-click on SSD1963. Under *Graphics>Interface,* double-click on Parallel (SMC). Connect the SMC_CS node of the Parallel (SMC) block to SMC_CS0 of the SMC block, and the Display Interface node to the SSD1963 block. Connect the Display Driver node of the SSD1963 block to the GFX Core block. After rearranging the boxes. you should end up with a project graph that looks like this instead.

Follow the instructions above for the non-display version re the File System and clock settings.

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/universal_audio_decoders. To build this project, you must open the audio/apps/universal_audio_decoders/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
uad_pic32mz_ef_c2_ ak4954	pic32mz_ef_curiosity_v2	This demonstration runs on the PIC32MZ2048EFM144 processor on the PIC32 MZ EF Curiosity 2.0 board and the AK4954 Audio Codec Daughter Board. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the I2S PLIB.
uad_pic32mz_ef_c2_ ak4954_wqvga	pic32mz_ef_curiosity_v2	This demonstration runs on the PIC32MZ2048EFM144 processor on the PIC32 MZ EF Curiosity 2.0 board and the AK4954 Audio Codec Daughter Board, plus a PDA TM4301B 480x272 (WQVGA) display. The AK4954 codec is configured as the slave, and the I2S peripheral as the master. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the I2S PLIB.
uad_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 WM8904 codec is configured as the master, and the SSC peripheral as the slave. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the SSC PLIB.
uad_sam_e70_xult_ wm8904_ssc_ssd1963	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 and SSD1963 LCD controller. The WM8904 codec is configured as the master, and the SSC peripheral as the slave. The project configuration is for reading data from a file on a USB Mass Storage Device. The data is sent to the headphone output of the codec daughter card via I2S protocol using the SSC PLIB.

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

For all configurations, a pair of headphones should be connected to the HP Out jack of the audio codec. A FAT32 file system USB Flash drive will be needed to inserted into the USB TARGET connector of the board. You will probably need a USB-A Female to Micro-B Male adapter cable to do so.

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

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

Using the PIC32 MZ EF Curiosity 2.0 board and the AK4954 Audio Codec Daughter Board, I2S PLIB, and PDA TM4301B 480x272 (WQVGA) display. The AK4954 daughter board is plugged into the set of X32 connectors on the right side of the board (X32 HEADER 2).

Attach the flat cable of the PDA TM4301B 480x272 (WQVGA) display to the 565 daughterboard connected to the PIC32 MZ EF Curiosity 2.0 GFX CONNECTOR.

Note: The SAM E70 Xplained Ultra board does not include the either the AK4954 Audio Codec daughterboard or the PDA TM4301B 480x272 (WQVGA) display. They are sold each sold separately on microchipDIRECT as part numbers AC324954 and AC320005-4 respectively.

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

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

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board, SSC PLIB, PDA TM4301B 480x272 (WQVGA) display, and SSD1963 graphics controller. All jumpers on the WM8904 should be toward the **front:**

Attach the flat cable of the PDA TM4301B 480x272 (WQVGA) display to the SSD1963 Graphics Board connected to the SAM E70 Xplained Ultra board GFX CONNECTOR.

Note: The SAM E70 Xplained Ultra board does not include the either the WM8904 Audio Codec daughterboard, the PDA TM4301B 480x272 (WQVGA) display, or the SSD1963 Graphics Board. They are sold each sold separately on microchipDIRECT as part numbers AC328904, AC320005-4, and AC320214 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.

For projects non using a display:

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 Audio Codec Daughter Board.
2. Connect power to the board if necessary. 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 board. You will probably need a USB-A Female to Micro-B Male adapter cable to do so.
4. When the device is connected the system will scan for audio files. Once the scanning is complete and at least one file is found (green LED2 on steady), listen to the audio output on headphones connected to the board. Use Switch SW1 as described under Control Description to change the volume or advance to the next track.

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 supported files with the appropriate extension.

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

If any supported 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

For projects using a display:

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 Audio Codec Daughter Board.
2. Connect power to the board if necessary. The system will be in a wait state for USB to be connected.
3. Connect a USB mass storage device (thumb drive) that contains songs of supported audio format to the USB TARGET connector of the board. You will probably need a USB-A Female to Micro-B Male adapter cable to do so.
4. When the device is connected the system will scan for audio files. Once the scanning is complete and at least one file is found, listen to the audio output on headphones connected to the board. Information about the song will be shown on the display as shown:

Filename displays the FAT32 filename of the song on the USB stick, in 8.3 format.

Sample Rate displays the samples/second, typically 44100 or 48000 for music, or 8000 or 160000 for ADPCM voice files.

Channels displays the number of audio tracks, usually 2 (stereo).

Bit Depth displays the bit depth of the material, usually 16.

For MP3 songs only:

Artist displays the name of the artist, if stored with the song.

Track displays the name of the track, if stored with the song.

Album displays the album title, if stored with the song.

Control Description

The Volume control adjust the volume of the output. The Headphone Enable, when turned off, turns red and mutes the output without affecting either the Volume control setting, or pausing the playback.

The three VCR-like green control buttons in the lower right of the screen are rewind, play, and next track. Rewind goes back to the beginning of the current track. When playing a song, the play button turns into a pause button:

usb_microphone

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

Description

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

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

Architecture

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

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

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

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

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

The following figure shows the basic architecture for the demonstration.

USB Microphone Application Block Diagram

Demonstration Features

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

Harmony Configuration and Code Generation

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

USB Microphone Application MPLAB-X Harmony Configurator Project Graph

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

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

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

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

The usb_microphone_basic custom application code code is located in the firmware/src directory of the application folder. The

app.c and app.h files are initially generated as stub files in this folder. The middleware and library APIs located in definitions.h (as generated by MHC), for the given project graph.

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

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

a. WM8904 Codec Driver

b. Timer

c. USB Audio

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

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

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

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

Tools Setup Differences

USB Configuration

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

USB High Speed Driver Configuration

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

USB Device Layer Configuration

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

USB Audio Function Configuration

The WM8904 Codec Configuration

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

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

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

I2SC1 Peripheral Configuration

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

USB Microphone I2S Configuration

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

USB Microphone I2S Configuration

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

Pin Manager Configuration

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

When the I2SC1 is used the following pins are used:

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

Clock Manager

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

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

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

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

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

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

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

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

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

MPLAB Harmony Configurator: Clock Diagram Configuration

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

USB Microphone MHC Clock Diagram

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

USB Microphone Main Clock

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

USB Microphone PLLA Clock

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

USB Microphone I2S1 GCLK and PCK2 Configuration

Timer Driver

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

Building the Application

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

Description

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

MPLAB X IDE Project

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

MPLAB X IDE Project Configurations

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

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

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

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

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

Running the Demonstration

This section demonstrates how to run the demonstration.

Description

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

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

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

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

Do the following to run the demonstration:

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

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

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

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

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

Figure 3. Audacity Recording Session using USB Microphone

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

Control Description

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

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

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

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

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

usb_speaker

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

Description

This demonstration application configures the development board to implement a USB Speaker device configured to run at 48 kHz sampling rate at 16 bit per sample.

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

Architecture

The application runs on two target boards, as follows:

1. SAM E70 Xplained Ultra Board

- Processor runs @ 300 MHz
- One push button (SW)
- Two LEDs (amber LED1 and green LED2). Only LED 1 can be used for a USB Device application, due to requiring VBUS sense to be selected rather than LED2.
- WM8904 Codec Daughter Board mounted on a X32 socket (configured as I2SC1 slave)
- USB Device interface

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

2. PIC32 MZ EF Curiosity 2

- Processor runs @ 200 MHz
- 4 push buttons (SW1-SW4)
- 3 LEDs (all red LED1-LED3) and RGB LED (LED4).
- AK4954 Codec Daughter Board mounted on X32 socket
- USB Device interface

Note: The PIC32 MZ EF Curiosity 2 does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

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.

Figure 1. Architecture Block Diagram

Demonstration Features

- Audio playback using an WM8904 codec daughterboard on the SAM E70 xPlained Ultra (E70XULT) board.
- Audio playback using a AK4954 codec daughterboard on the PIC32 MZ EF Curiousity 2 (MZEFC2)
- USB connection to a host system using the USB Library Device Stack for a USB Speaker device using E70XULT and MZEFC2 boards
- E70XULT and MZEFC2 board button processing for volume/mute control
- USB Attach/Detach and mute status using an LED on E70XULT and MZEFC2 boards
- Utilization of the of SAM E70 I2S peripherals, E70 SSC (as slave) and I2SC (as master), on E70XULT board
- Utilization of the PIC32 I2S peripheral (as master) on the MZEFC2 board

Note that all the calls to the WM8904 and AK4954 codec drivers use the form DRV_CODEC_xxx rather than DRV_WM8904_xxx. or DRV_AK4954_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 usb_speaker demonstration uses multiples projects for multiple hardware configurations. MPLAB-X Harmony 3 projects only have one associated configuration. When each Harmony 3 project is created the MPLAB-X Harmony Configurator (MHC) code generation is guided by the processor that is selected.

Start MHC for the project in order to add the application code components. Both projects of the usb_speaker application use the USB Audio stack components. These can be added by selecting the Libraries/USB/Device Stack/Audio Function Driver component template under the MHC Available Components list, as shown below.

Answer yes to all questions. This loads the USB High Speed Driver, the USB Device Layer and Audio Function Driver components.

The Codec component to be used depends on the hardware and peripheral configuration. This is described in the following two sections.

The MHC Project Graph for usb_speaker_basic is shown below for the E70XULT board.

Figure 2. USB Speaker Basic MHC Project Graph for the SAM E70 Xplained Ultra Board

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

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

PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board Configuration

The MZEFC2 board configuration in similar to the prior E70XULT board configuration, but replaces the WM8904 with an AK4954 Codec, which uses a different I2S peripheral driver.

In the MHC, under Available Components select the BSP PIC32 MZ EF Curiosity 2.0. Under Audio>Templates, double-click on

AK4954 Codec Template (shown below). Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one. This loads the AK4954 Codec component along with associated I2C, I2S and timer driver components.

The project configuration should now look something like this:

Figure 3. USB Speaker Project Graph for the PIC32 MZ EF Curiosity 2 Board

Harmony Code Configuration Options

Each block in the MHC Project Graph may need to be configured through parameters for the specific application. These parameters are accessed by selecting the block with the mouse, and appear in the Configuration Options window, where they can be edited. The next section describes the configuration of the USB, Math, and Codec component blocks for the USB Speaker with “Bass Boost” application.

USB Configuration

The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is set to “Full Speed”, V1.0 interface (not “High Speed”, V2.0 Interface):

The USB Device Layer is configured by selecting Product ID Selection as “usb_speaker_demo” with an endpoint buffer size of 64 (bytes). The Product String Selection is changed to “Harmony USB Speaker w/Bass Boost Example”. This information will be used to generate the fullSpeedConfigurationDescriptor array variable structure (located in initialization.c under the config folder) that defines the enumeration of device functions with the USB Host. This structure defines the connection to the host at 48 kHz with 16 bit stereo channel data.

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

A playback packet queue of length 64 is set with the Audio Read Queue Size configuration, which should match the Audio Write Queue Size of the WM8904/I2S. The Audio Write Queue Size is unimportant. The 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 WM8904 Codec

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board:

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

When the I2SC1 driver is used for DRV_I2S_0 the usage mode changes to “Slave”

The I2S configuration uses a Transfer queue size of 64, matching that that of the USB Read Queue:

The I2SC1 automatically replaces the “PLIB Used” when it is connected to the I2S block in place of the SSC.

Pin Manager

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

NAME	PORT	E70 PIN	Notes
SSC_TK	PB01	20	I2S BCLK
SSC_TF	PB00	21	I2S LRCK
PMC_PCK2	PA18	24	I2S MCLK (Master Clock)
SSC_TD	PD26	53	I2S Data
SWITCH	PA11	66	Push Button
LED	PA05	73	
TWIHS0_TWCK0	PA04	77	I2C
TWIHS0_TWD0	PA03	91	I2C
STDBY	PD11	98	
LED2/VBUS DETECT(J204)	PB08	141	J204 set to VBUS DETECT for USB Device

When the I2SC1 is used the following pins are used:

NAME	PORT	E70 PIN	Notes
I2SC1_WS	PE00	4	I2S LRCK (Word Select)
I2SC1_DO0	PE01	6	I2S DO (Data Out)
I2SC1_DI0	PE02	7	I2S DI (Data In)
I2SC1_CK	PA20	22	I2S BCLK (Bit Clock)
I2SC1_GCLK	PA19	23	I2S MCLK (Bit Clock as used by the E70 I2SC1, I2S Master)
PMC_PCK2	PA18	24	I2S MCLK (Master Clock as used by the WM8904 Codec I2S Slave)
SWITCH	PA11	66	Push Button
LED1	PA05	73	

TWIHS0_TWCK0	PA04	77	I2C
TWIHS0_TWD0	PA03	91	I2C
STDBY	PD11	98	
LED2/VBUS DETECT(J204)	PB08	141	J204 set to VBUS DETECT for USB Device

Clock Manager

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

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

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

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

MPLAB Harmony Configurator: Tools>Clock Configuration

SSC Clock Configuration

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

Figure 4. Main Clock Configuration

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

Figure 5. Clock Diagram>Peripheral Clock Enable

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

Figure 6. Enable SSC Clock

I2SC1 Clock Configuration

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

Figure 7. PLLA Clock Generation for I2SC Peripheral

Figure 8. Generic Clock (GCLK) Generation for the I2SC1

Harmony Code Generation

All the needed drivers, middleware, libraries and application framework code can be generated from the MHC blocks (MHC components) placed in the MHC Project Graph,

The generated framework code is placed under the firmware/src/config directory under the name of the configuration used for the Harmony 3 project. The initial 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 located in the config directory.

All Harmony applications first execute the SYS_Initialize function, located in initialization.c. This is executed from the main function 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 executed last in the generated SYS_Initialize routine after the system initializations have completed.

The SYS_Tasks function (located in tasks.c) is used to update the USB subsystems, WM8904 driver, timers etc., as well as the application state machine (APP_tasks routine in app.c). This function is executed from the main polling loop. The polling loop either execute SYS_Tasks repeatably in the infinite loop to perform the updates, or it the updates occur as separate processes executed at fixed time intervals using an RTOS schedular.

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.
4. Sense Volume control button pushes and set LED status lights

Building the Application

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

Description

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

MPLAB X IDE Project

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

MPLAB X IDE Project Configurations

This table lists and describes the supported configurations of the demonstration, which are located within

`./firmware/src/config`.

Project Name	BSP Used	Description
us_basic_pic32mz_ef_c2_ ak4954	pic32_mz_ef_c2	This demonstration runs on the PIC32 MZ EF Curiosity 2 board with the AK4954 Codec.
us_basic_sam_e70_xult_ wm8904_i2sc_usb	sam_e70_xult	This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the I2SC PLIB.
us_basic_sam_e70_xult_ wm8904_ssc_usb	sam_e70_xult	This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the SSC PLIB.
us_basic_sam_e70_xult_ wm8904_ssc_usb_freertos	sam_e70_xult	This demonstration runs on the SAM E70 Xplained Ultra board with the WM8904 daughter board using the I2SC PLIB. It also uses FreeRTOS.

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board configured for the I2SC1 PLIB:

Jumper J203, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS (not LED2).

To connect to the I2SC1, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented toward the pink, mic in, connector.

Attach the WM8904 Daughter Board to the X32 connector and connect headphones to the green HP OUT jack, as shown in the figure below.

WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green. Microphone In Jack is pink. Note: the brown wire is a jumper between EXT1-13 to J601(Camera)-14 and is not used for this demo.

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.

As shown, the SAM E70 will be programmed via MPLAB-X through the USB cable connected to the EDBG micro-mini connector. Program debug can also be performed over USB this connection, if required.

Using the PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board:

The PIC32 MZ EF Curiosity board and the AK4954 Audio Codec Daughter Board only requires the AK4954 Codec Daughterboard to be connected to X32 Header 2 as shown below. No jumper settings are required.

AK4954 Audio Codec Daughter Board on PIC32 MZ EF Curiosity 2 board. Headphone Out Jack is green. Microphone In Jack is pink.

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

Connect headphones to the green HP OUT jack of the Codec Daughter Board (as shown in the Figure above).

As shown, the SAM E70 will be programmed through the USB cable connected to the PKOB4 micro-mini connector. Program debug can also be performed over USB this connection.

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. Configure the hardware, as described in the previous section, for the selected MPLAB-X project.

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

2. Connect power to the board, compile the application and program the target device. Run the device. The system will be in a waiting for USB to be connected (LED1 off).
3. Connect to the USB Host via the micro-mini connector (see Configuring the Hardware).
4. Allow the Host computer to acknowledge, install drivers (if needed), and enumerate the device. No special software is necessary on the Host side. LED1 will blink after enumeration. It will continue to blink until the Harmony USB Speaker is selected as the speaker device, and the audio stream is started. It will then turn a solid color.
5. If needed, configure the Host computer to use the usb_speaker as the selected audio recording device. For Windows, this is done in the "Recording" tab in the "Sound" dialog for Windows 7 or by selecting the “Open Sound Setting”. The sound dialog

selection for Both Windows 10 and Windows 7 sound dialog is accessed by right clicking the loudspeaker icon in the taskbar, which will bring up a menu for the selection. This is shown in the Figure below for Windows 10. Windows 7 works in a similar fashion.

Figure 12. Windows 10 - Sound Dialog Showing Harmony USB Speaker Selection.

Note: The device "Harmony USB Speaker Example" should be available along with a sound level meter indication audio when playing. If no sound level is registering or the name of the Harmony speaker is incorrect, uninstall the driver using the Windows Device Manager, since it may have incorrect configuration set by a similar connection to one of the other MPLAB-X Harmony Audio Demos.

6. Open a playback application (such as Window Media Player) and initiate playback through the USB Speaker. LED 1 should be a solid color.

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.
8. Volume can be change by repeated press of SW1. Volume will increase to maximum. It will then mute the speaker (LED1 will fast blink during mute) and then start increasing from minimum volume to maximum after more presses of SW1. NOTE: Volume is also controlled using the PC volume control and the Windows Media Player volume control.,

Control Description

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

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

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

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

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

usb_speaker_bass_boost

This topic provides instructions and information about the MPLAB Harmony 3 USB Speaker with “Bass Boost” audio enhancement demonstration application which is included in the MPLAB Harmony Library distribution.

Description

This demonstration application configures the development board to implement a USB Speaker device configured to run at 48 Khz sampling rate at 16 bit per sample and process the USB audio input through a "Bass Boost" filter (using either the CMSIS-DSP or the PIC32 MZ EF DSP library functions) prior to playback.

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 a "Bass Boost" filter prior to the audio output for playback through codec hardware.

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 a "Bass Boost" filter prior to the audio output for playback through codec hardware.

Architecture

The application runs on two target boards, as follows:

1. SAM E70 Xplained Ultra Board

- Processor runs @ 300 MHz
- One push button (SW)
- Two LEDs (amber LED1 and green LED2). Only LED 1 can be used for a USB Device application, due to requiring VBUS sense to be selected rather than LED2.
- WM8904 Codec Daughter Board mounted on a X32 socket (configured as I2SC1 slave)
- USB Device interface

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

2. PIC32 MZ EF Curiosity 2

- Processor runs @ 200 MHz
- 4 push buttons (SW1-SW4)
- 3 LEDs (all red LED1-LED3) and RGB LED (LED4).
- AK4954 Codec Daughter Board mounted on X32 socket
- USB Device interface

Note: The PIC32 MZ EF Curiosity 2 does not include the AK4954 Audio Codec daughterboard, which is sold separately on microchipDIRECT as part number AC324954.

The usb_speaker_bass_boost 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 Codec Module.

The USB Host system that can provide streaming audio (like a PC) is connected to the micro-mini USB device connector. The application then detects the cable connection, which can also supply device power; recognizes the type of connection (Full Speed); enumerates the devices functions it can provide to the host, and then selects the isochronous audio streaming playback. 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).

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).

Bass Boost Filter

The “Bass Boost” audio enhancement is the 16 bit fixed-point implementation of a "Peak Shelving Filter", described by:

$$K = \tan\left(\frac{\pi Fc}{Fs}\right)$$

$$V0 = 10^{\frac{G}{20}}$$

$$root2 = \frac{1}{Q}$$

$$b0 = \frac{K^2V_0 + K\sqrt{V_0}root_2 + 1}{K^2 + Kroot_2 + 1}$$

$$b1 = \frac{2K^2V_0 - 2}{K^2 + Kroot_2 + 1}$$

$$b2 = \frac{K^2V_0 - K\sqrt{V_0}root_2 + 1}{K^2 + Kroot_2 + 1}$$

$$a1 = \frac{2K^2 - 2}{K^2 + Kroot_2 + 1}$$

$$a2 = \frac{K^2 - Kroot_2 + 1}{K^2 + Kroot_2 + 1}$$

NOTES:

1. The parameter "K" sets the transition frequency, Fc, above where the boost goes away.
2. The parameter "V0" sets the boost or cut value in dB
3. The parameter “root2” adjusts the slope of the transition (replaces square root of 2)

Giving the coefficients of 2nd-order difference equation:

$$a0 \cdot y(k) = b_0x(k) + b_1x(k-1) + b_2x(k-2) - a_1y(k-1) - a_2y(k-2)$$

The Direct-Form 1 implementation (without scaling) is shown below:

Figure 1. Biquad Filter - Direct Form 1

The a0 scaling parameter can assumed to be 1, but for fixed point coefficient multiplies the right hand side coefficients should be scaled to fit into the required “Q” format. See LIBQ or CMIS-DSP library documentation for the defined Q formats for fixed point values required by the library.

The response of the filter for a bass boost of 6dB and a cutoff of 200Hz is shown below

Figure 2. Peak Shelving Filter Frequency Response, Fc=200 Hz, Gboost = 6 dB

The gain is boosted below ~350 Hz and attenuated by -6dB above 800Hz, with a transition in between. the -6dB attenuation of the upper frequencies give headroom such that the boost does not overly saturate the audio output.

The following figure shows the basic architecture for the demonstration.

Figure 3. USB Speaker w/Bass Boost - Block Diagram

Demonstration Features

- Playback using an WM8904 codec daughterboard on the SAM E70 xPlained Ultra (E70XULT) board.
- Playback using an AK4953 codec daughterboard on the PIC32 MZ EF Curiousity 2 (MZEFC2) board
- USB connection to a host system using the USB Library Device Stack for a USB Speaker device using E70XULT or MZEFC2

- Button processing for volume/mute/Bass Boost control using E70XULT or MZEFC2
- USB Attach/Detach and mute status using an LED using E70XULT or MZEFC2
- Utilization of the of I2S peripheral, I2SC (as master), for the E70XULT
- Utilization of the I2S peripheral for the MZEFC2
- Utilization of the DSP Math libraries (CMSIS-DSP and LIBQ/DSP) for both the ARM and MIPS processors to implement "Bass Boost" audio enhancement filters

Note that all the calls to the WM8904 and AK4954 codec drivers use the form DRV_CODEC_xxx rather than DRV_WM8904_xxx. or DRV_AK4954_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 usb_speaker_bass_boost demonstration uses multiples projects for multiple hardware configurations. MPLAB-X Harmony 3 projects only have one associated configurations. When each Harmony 3 project is created the MPLAB-X Harmony Configurator (MHC) code generation is guided by the processor that is selected.

Start MHC for the project in order to add the application code components. Both projects of the usb_speaker_bass_boost application use the USB Audio stack components. These can be added by selecting the Libraries/USB/Device Stack/Audio Function Driver component template under the MHC Available Components list, as shown below.

Answer yes to all questions. This loads the USB High Speed Driver, the USB Device Layer and Audio Function Driver components.

Additional MHC components for Math and Codec depend on the hardware configuration and are described in the next two sections.

The MHC Project Graph for usb_speaker_bass_boost is shown below.

PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board Configuration

In the MHC, under Available Components select the BSP PIC32 MZ EF Curiosity 2.0. Under Audio>Templates, double-click on AK4954 Codec Template (shown below). Answer Yes to all questions except for the one regarding FreeRTOS; answer No to that one. This loads the AK4954 Codec component along with associated I2C, I2S and timer driver components.

Select the Audio/Math/Math Libraries component to load the libq_c, libq/dsp libraries, as shown below

The project configuration should now look something like this:

Figure 4. USB Speaker w/Bass Boost MHC Project Graph showing the PIC32 Math Libraries Component Block.

SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board Configuration

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

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

Select the Libraries/USB/Device Stack/Audio Function Driver component template, as shown below. Answer yes to all questions. This loads the USB High Speed Driver, the USB Device Layer and Audio Function Driver components.

To add the CMSIS-DSP Library to the project, select the CMSIS packs component.

The project configuration should look something like this:

Figure 5. The USB Speaker w/Bass Boost Project MHC Project Graph Showing the CMSIS-DSP Library Block (CMSIS Pack)

Harmony Code Configuration Options

Each block in the MHC Project Graph may need to be configured through parameters for the specific application. These parameters are accessed by selecting the block with the mouse, and appear in the Configuration Options window, where they can be edited. The next section describes the configuration of the USB, Math, and Codec component blocks for the USB Speaker with “Bass Boost” application

USB Configuration

The application uses USB Library as a "Device" stack, which will connect to a "Host". The USB High Speed Driver is set to “Full Speed”, V1.0 interface (not “High Speed”, V2.0 Interface):

The USB Device Layer is configured by selecting Product ID Selection as “usb_speaker_demo” with an endpoint buffer size of 64 (bytes). The Product String Selection is changed to “Harmony USB Speaker w/Bass Boost Example”. This information will be used to generate the fullSpeedConfigurationDescriptor array variable structure (located in initialization.c under the config folder) that defines the enumeration of device functions with the USB Host. This structure defines the connection to the host at 48 Khz with 16 bit stereo channel data.

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

A playback packet queue of length 64 is set with the Audio Read Queue Size configuration, which should match the Audio Write Queue Size of the WM8904/I2S. The Audio Write Queue Size is unimportant. The 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 WM8904 Codec

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board:

The WM8904 codec uses a TWIHS (I2C) interface for configuration and control register setting and the I2SC1 peripheral interface. This application uses the I2SC1 I2S peripheral interface in Master mode, as shown below:

When the I2SC1 driver is used for DRV_I2S_0 the usage mode changes to “Slave”

The I2S configuration uses a Transfer queue Size of 64 and I2S Data Length of 16, matching that that of the USB Read Queue. Bidirectional DMA is set (using 2 channels) for data transfer.

Pin Manager

The following pins are used for Clocks, I2SC1, I2C, Reset, LEDs, Switches, and VBUS Detect:

The I2SC1 peripheral uses the following pins:

NAME	PORT	E70 PIN	Notes
I2SC1_WS	PE00	4	I2S LRCK (Word Select)
I2SC1_DO0	PE01	6	I2S DO (Data Out)
I2SC1_DI0	PE02	7	I2S DI (Data In)
I2SC1_CK	PA20	22	I2S BCLK (Bit Clock)
I2SC1_GCLK	PA19	23	I2S MCLK (Bit Clock as used by the E70 I2SC1, I2S Master)
PMC_PCK2	PA18	24	I2S MCLK (Master Clock as used by the WM8904 Codec I2S Slave)
SWITCH	PA11	66	Push Button
LED1	PA05	73	

TWIHS0_TWCK0	PA04	77	I2C
TWIHS0_TWD0	PA03	91	I2C
STDBY	PD11	98	
LED2/VBUS DETECT(J204)	PB08	141	J204 set to VBUS DETECT for USB Device

Clock Manager

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

Clock	Value	Description
HCLK	240 MHz	Processor Clock
PCK2	12 MHz / 12.288 MHz	Peripheral Clock 2 SSC/I2SC1
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.

I2SC1 Clock Configuration

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

USB Speaker PLLA Clock

USB Speaker I2S1 Bit Clock for 48Mhz operation

Figure 6. I2SC1 Clock Configuration

AK4954 Codec

Using the PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board:

Click on the I2S2 Peripheral. Set the Frame Select Pin to C2. This is used to sense the start of the Left/Right bit frame clock.

Select the I2S driver and change the Transfer Queue Size to 64, as shown below

Select the I2C1 component and select the “Include Force Write I2C …”, This is required for the case where the AK4954 is in reset and does not respond with NACK when the I2C operation completes.

Math Library

Using the PIC32 MZ EF Curiosity 2 with the AK4965 Daughter Board:

The LIBQ/DSP are used with MIPS MCU with microAptiv core require The default configuration of the audio/math/math libraries component is used, as shown below:

The DSP Library requires the LIBQ library to be selected.

Using the SAM E70 Xplained Ultra board and the WM8904 Audio Codec Daughter Board:

The CMSIS-DSP Library is optimized for use with the ARM M0-M7 MCU cores. It is selected using the CMSIS-Pack component (which is loaded automatically with the processor core component), as shown below:

Harmony Code Generation

All the needed drivers, middleware, libraries and application framework code can be generated from the MHC blocks (MHC components) placed in the MHC Project Graph,

The generated framework code is placed under the firmware/src/config directory under the name of the configuration used for the Harmony 3 project. The initial 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 located in the config directory.

All Harmony applications first execute the SYS_Initialize function, located in initialization.c. This is executed from the main function

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 executed last in the generated SYS_Initialize routine after the system initializations have completed.

The SYS_Tasks function (located in tasks.c) is used to update the USB subsystems, WM8904 driver, timers etc., as well as the application state machine (APP_tasks routine in app.c). This function is executed from the main polling loop. The polling loop either execute SYS_Tasks repeatably in the infinite loop to perform the updates, or it the updates occur as separate processes executed at fixed time intervals using an RTOS schedular.

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.

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_bass_boost. To build this project, you must open the audio/apps/usb_speaker_bass_boost/firmware/*.X project file in MPLAB X IDE that corresponds to your hardware configuration.

MPLAB X IDE Project Configurations

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

Project Name	BSP Used	Description
us_bb_pic32_mz_ef_c2_ak4954	pic32_mz_ef_c2	This demonstration runs on the PIC32 MZ EF Curiosity 2 with the AK4954 Codec
us_bb_sam_e70_xult_wm8904_i2sc_usb	sam_e70_xult	This demonstration runs on the SAM E70 xPlained Ultra board with the WM8904 Codec

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 configured for the I2SC1 PLIB:

Jumper J203, which is next to the SAM E70 Xplained Ultra logo, should be jumpered for VBUS (not LED2).

To connect to the I2SC1, the jumpers (J6, J7, J8, and J9) on the WM8904 Codec Daughterboard must be oriented toward the pink, mic in, connector.

Attach the WM8904 Daughter Board to the X32 connector and connect headphones to the green HP OUT jack, as shown in the figure below.

WM8904 Audio Codec Daughter Board on SAM E70 Xplained Ultra board. Headphone Out Jack is green. Microphone In Jack is pink. Note: the brown wire is a jumper between EXT1-13 to J601(Camera)-14 and is not used for this demo.

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.

As shown, the SAM E70 will be programmed via MPLAB-X through the USB cable connected to the EDBG micro-mini connector. Program debug can also be performed over USB this connection, if required.

Using the PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board:

The PIC32 MZ EF Curiosity board and the AK4954 Audio Codec Daughter Board only requires the AK4954 Codec Daughterboard to be connected to X32 Header 2 as shown below. No jumper settings are required.

AK4954 Audio Codec Daughter Board on PIC32 MZ EF Curiosity 2 board. Headphone Out Jack is green. Microphone In Jack is pink.

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

Connect headphones to the green HP OUT jack of the Codec Daughter Board (as shown in the Figure above).

As shown, the SAM E70 will be programmed through the USB cable connected to the PKOB4 micro-mini connector. Program debug can also be performed over USB this connection.

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.

Configure the board as described in the previous section. Attach either the EDBG or PKOB4 USB cable from the PC Host to the board, as described in the prior section.

To compile, select the appropriate MPLAB X ® IDE project configuration (see Table above). Compile and program the target device. Refer to Building the Application for details

Do the following to run the demonstration:

1. After connecting a standard USB cable from the Host PC to either the EDBG or PKOB4 of the respective board, compile the application and program the target device. Run the device. The system will be in a waiting for the USB to be connected (LED1 off). The RGB LED on the Curiosity 2 board will run red, indicating no USB connection.
2. Connect a standard USB cable to the USB Host via the micro-mini connector located, either:
   - Above the push-button switches on the right side of the SAM E70 Xplained board, or
   - Above and to the left of the Reset switch on the PIC32 MZ EF Curiosity 2 board
3. Allow the Host computer to acknowledge, install drivers (if needed), and enumerate the device. No special software is necessary on the Host side.

LED1 will turn on as the USB function is enumerated (it may blink if audio is not streaming over the connection). The RGB LED on the Curiosity board will turn from red to green.

If needed, configure the Host computer to use the usb_speaker as the selected audio recording device. For Windows 7, this is done in the "Playback" tab in the "Sound" dialog accessed by right clicking the loudspeaker icon on the taskbar. For Windows 10, this is accessed by right clicking the loudspeaker icon on the taskbar and clicking “Open Sound Settings”. Select the “Speaker(Harmony USB Speaker w/Bass Boost Example)” as the default device. Now all sounds should be heard through the headphones

Windows 10 Sound Setting showing USB Speaker w/Bass Boost Example selection.

Note: The "Harmony USB Speaker w/Bass Boost Example" should be available along with a sound level meter, indicating audio when playing. If no sound level is registering when audio is playing, uninstall the driver, since it may have incorrect configuration set by a similar connection to one of the other MPLAB-X Harmony USB Audio Demos.

4. Open a playback application (such as Window Media Player) and initiate playback, which should be heard throught the headphones.
5. Four volume levels are available, when pressing SWITCH 1. Initially, it is set to the lowest playback level. Pressing 2 more times will change the volume from the lowest to the highest playback level. Pressing 1 more time will mute audio and LED 1 will blink slowly. Pressing the button again will go back to the initial setting, from which the volume sequence began.
6. “Bass Boost” is demonstrated as follows:

Using the SAM E70 Xplained board and the WM8904 Daughter Board

A “Long Press” of SWITCH 1 will turn on “Bass Boost”. This will enhance (boost) the lower audio frequencies heard through the headphones. You can cycle from boosted audio to normal audio by repeating the “Long Press” of SWITCH1.

Using the PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board

Pressing SWITCH 2 will turn on LED 2 and the RGB LED will turn blue (indicating boost). The audio “Bass Boost” filter will be at its lowest level. Pressing SWITCH 2 repeatedly will cycle through 4 levels of “Bass Boost” (4, 6, 8, and 10 dB). A “Long” press of SWITCH 2 will turn off the “Bass Boost”, LED 2 will turn off and the RGB LED will turn green. Pressing again will turn on “Bass Boost” at the previous level.

Control Description

Using the SAM E70 Xplained board and the WM8904 Daughter Board:

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

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

Mute	Mute

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

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

LED Status	Status
OFF	USB cable detached
ON	USB cable attached
Blinking - Fast	USB cable attached (no audio)
Blinking - Slow	Playback muted

Using the PIC32 MZ EF Curiosity 2 and the AK4965 Daughter Board:

Button control uses SWITCH 1 in the push button function sequence given in the table below:.

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

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

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

LED1 Status	Status
OFF	USB cable detached or (USB attached & invalid enumeration)
ON	USB cable attached
Blinking - Fast	USB cable attached (no audio)
Blinking - Slow	Playback muted

LED2/RGB LED Status	Status
OFF/Red	USB cable detached or (USB attached & invalid enumeration)
ON/Green	USB cable attached and audio enumerated (normal audio)
ON/Blue	USB cable attached and audio enumerated (“Bass Boost” enabled)
Blinking – Fast/Red	USB cable attached (no audio)
Blinking – Slow/Green or Blue	Audio Playback muted in either normal (green) or boosted (blue) modes

usb_speaker_hi_res

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

Description

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

The USB Device driver in Full Speed mode will interface to a USB Host (such as a personal computer) via the USB Device Stack

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

Architecture

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

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

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

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

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

The following figure shows the basic architecture for the demonstration.

Demonstration Features

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

Tools Setup Differences

Harmony Configuration

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

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

Harmony Code Generation

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

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

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

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

USB Configuration

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

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

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

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

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

The WM8904 Codec

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

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

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

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

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

Pin Manager

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

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

Clock Manager

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

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

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

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

MPLAB Harmony Configurator: Tools>Clock Configuration

I2SC1 Clock Configuration

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

Clock Diagram>Peripheral Clock Enable

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

I2SC1 Configuration of PLLA Clock (PLLACK)

I2SC1 GCLK Configuration

MPLAB Harmony Configurator: Timer Driver (TC0)

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

Building the Application

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

Description

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

MPLAB X IDE Project

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

MPLAB X IDE Project Configurations

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

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

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

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

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

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

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

Running the Application

This section demonstrates how to run the demonstration.

Description

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

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

Application for details.

Do the following to run the demonstration:

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

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

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

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

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

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

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

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

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

Control Description

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

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

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

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

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

Driver Libraries Help

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

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

Codec Libraries Help

AK4953 CODEC Driver Library Help

This topic describes the AK4953 Codec Driver Library.

Introduction

This topic describes the basic architecture of the AK4953 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: drv_AK4953.h

The interface to the AK4953 Codec Driver library is defined in the audio/driver/codec/AK4953/drv_AK4953.h header file. Any C language source (.c) file that uses the AK4953 Codec Driver library should include this header.

Library Source Files:

The AK4953 Codec Driver library source files are provided in the audio/driver/codec/AK4953/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.

Using the Library

Abstraction Model

This library provides a low-level abstraction of the AK4954 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4953 Codec Driver is positioned in the MPLAB Harmony framework. The AK4953 Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the AK4953 module.

AK4953 Driver Abstraction Model

Library Overview

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

The AK4953 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4953 Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4953 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 AK4953 Codec Driver Library.

Note: All functions and constants in this section are named with the format DRV_ AK4953_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_ AK4953_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

Description

Setup (Initialization)

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

Description

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 AK4953 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4953_INIT or by using Initialization Overrides) that are supported by the specific AK4953 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_AK4953_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_ AK4953_Deinitialize, DRV_ AK4953_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_AK4953_Open function. The DRV_AK4953_Open function provides a driver handle to the AK4953 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4953_Deinitialize, the application must call the DRV_AK4953_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 AK4953 Codec Driver can be known by calling DRV_ AK4953_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4953Status;
ak4953Status Status = DRV_AK4953_Status(sysObjects.ak4953Status DevObject);
if (SYS_STATUS_READY == ak4953Status)
{
// The driver can now be opened.
```

```
appData.ak4953Client.handle = DRV_AK4953_Open
(DRV_AK4953_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.ak4953Client.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_AK4953_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* AK4953 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 AK4953 Codec.

The following AK4953 Codec specific control command functions are provided:

- DRV_AK4953_SamplingRateSet
- DRV_AK4953_SamplingRateGet
- DRV_AK4953_VolumeSet
- DRV_AK4953_VolumeGet
- DRV_AK4953_MuteOn
- DRV_AK4953_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4953 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_AK4953_BufferAddWrite, DRV_AK4953_BufferAddRead, and DRV_AK4953_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_AK4953_BUFFER_EVENT_COMPLETE, DRV_AK4953_BUFFER_EVENT_ERROR, or DRV_AK4953_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.

In the MHC, under Available Components select the appropriate BSP. Under *Audio>Templates*, double-click on a codec template such as AK4953. Answer Yes to all questions.

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

Click on the AK4953 Driver component (not AK4953 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 AK4953 is a Master (supplies I^2S clocks) or a Slave (MCU supplies I^2S clocks).
- **Number of AK4953 Clients** indicates the maximum number of clients that can be connected to the AK4953 Driver.

- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either
  - 24-bit Left Justified (ADC), 24-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 16-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 24-bit Left-justified(DAC)
  - 24/16-bit I2S

It must match the audio protocol and data length set up in either the SSC or I2S PLIB.

- **Microphone/Line Input** selects which microphone or line input is selected, either:
  - Internal Mic (mounted on the AK4953 daughterboard)
  - External Mic Input
  - Line Input

If External Mic input or Line Input is selected, then the following option is provided:

- **Ext Mic Gain in dB range 0(min) to 31(max)**

If External Mic input is selected, then the following option is provided:

- **Enable Microphone Bias** should be checked if using an electret microphone.

You can also bring in the AK4953 Driver by itself, by double clicking AK4953 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 AK4953 requires the TCx Peripheral Library and TIME System Service in order to perform some of its internal timing sequences.

Bulding the Library

This section lists the files that are available in the AK4953 Codec Driver Library.

Description

This section lists the files that are available in the src folder of the AK4953 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/AK4953.

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_ak4953.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_ak4953.c	This file contains implementation of the AK4953 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 AK4953 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

Client Setup Functions

	Name	Description
	DRV_AK4953_Close	Closes an opened-instance of the AK4953 driver
	DRV_AK4953_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.
	DRV_AK4953_Open	Opens the specified AK4953 driver instance and returns a handle to it.
	DRV_AK4953_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.

Data Transfer Functions

	Name	Description
	DRV_AK4953_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4953_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4953_BufferAddWriteRead	Schedule a non-blocking driver write-read operation.
	DRV_AK4953_ReadQueuePurge	Removes all buffer requests from the read queue.
	DRV_AK4953_WriteQueuePurge	Removes all buffer requests from the write queue.

Data Types and Constants

	Name	Description
	DRV_AK4953_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4953_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4953_BUFFER_EVENT_HANDLER	Pointer to a AK4953 Driver Buffer Event handler function
	DRV_AK4953_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4953_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4953_COMMAND_EVENT_HANDLER	Pointer to a AK4953 Driver Command Event Handler Function
	DRV_AK4953_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4953_INIT	Defines the data required to initialize or reinitialize the AK4953 driver
	DRV_AK4953_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4953_MIC	This is type DRV_AK4953_MIC.
	DRV_AK4953_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Other Functions

	Name	Description
	DRV_AK4953_GetI2SDriver	Get the handle to the I2S driver for this codec instance.
	DRV_AK4953_LRCLK_Sync	Synchronize to the start of the I2S LRCLK (left/right clock) signal
	DRV_AK4953_VersionGet	This function returns the version of AK4953 driver.
	DRV_AK4953_VersionStrGet	This function returns the version of AK4953 driver in string format.

Settings Functions

	Name	Description
	DRV_AK4953_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.

	DRV_AK4953_MicGainGet	This function gets the microphone gain for the AK4953 Codec.
	DRV_AK4953_MicGainSet	This function sets the microphone gain for the AK4953 CODEC.
	DRV_AK4953_MicMuteOff	Umutes th AK4953's microphone input.
	DRV_AK4953_MicMuteOn	Mutes the AK4953's microphone input
	DRV_AK4953_MicSet	This function sets up the codec for the internal or the AK4953 Mic1 or Mic2 input.
	DRV_AK4953_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4953_MuteOff	This function disables AK4953 output for soft mute.
	DRV_AK4953_MuteOn	This function allows AK4953 output for soft mute on.
	DRV_AK4953_SamplingRateGet	This function gets the sampling rate set on the DAC AK4953.
	DRV_AK4953_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4953_VolumeGet	This function gets the volume for AK4953 Codec.
	DRV_AK4953_VolumeSet	This function sets the volume for AK4953 Codec.

System Interaction Functions

	Name	Description
	DRV_AK4953_Deinitialize	Deinitializes the specified instance of the AK4953 driver module.
	DRV_AK4953_EnableInitialization	Enable delayed initialization of the driver.
	DRV_AK4953_Initialize	Initializes hardware and data for the instance of the AK4953 Codec module.
	DRV_AK4953_IsInitializationDelayed	Checks if delayed initialization of the driver has been requested.
	DRV_AK4953_Status	Gets the current status of the AK4953 driver module.
	DRV_AK4953_Tasks	Maintains the driver's control and data interface state machine.

Description

System Interaction Functions

DRV_AK4953_Deinitialize Function

void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object)

Summary

Deinitializes the specified instance of the AK4953 driver module.

Description

Deinitializes the specified instance of the AK4953 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_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_AK4953_Initialize
SYS_STATUS          status;


DRV_AK4953_Deinitialize(object);

status = DRV_AK4953_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

C

```
void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object);
```

DRV_AK4953_EnableInitialization Function

void DRV_AK4953_EnableInitialization(SYS_MODULE_OBJ object);

Summary

Enable delayed initialization of the driver.

Description

If the AK4953 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. Once this has been accomplished, this function should be called to kick-start the codec driver initialization.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4953_Initialize)

Returns

None.

Remarks

This is not needed for audio-only applications without a Bluetooth module.

C

```
void DRV_AK4953_EnableInitialization(SYS_MODULE_OBJ object);
```

DRV_AK4953_Initialize Function

SYS_MODULE_OBJ DRV_AK4953_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

Summary

Initializes hardware and data for the instance of the AK4953 Codec module.

Description

This routine initializes the AK4953 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 AK4953 routine is called.

This routine should only be called once during system initialization unless DRV_AK4953_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Example

```
DRV_AK4953_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = ak4953Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = ak4953Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_AK4953_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_AK4953_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4953_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4953Init->volume;
}
init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_AK4953_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4953_Initialize(DRV_AK4953_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

C

```
SYS_MODULE_OBJ DRV_AK4953_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

DRV_AK4953_IsInitializationDelayed Function

bool DRV_AK4953_IsInitializationDelayed(SYS_MODULE_OBJ object);

Summary

Checks if delayed initialization of the driver has been requested.

Description

If the AK4953 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. This function returns true if that option has been selected in MHC in the checkbox: "Delay driver initialization (due to shared RESET pin)"

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4953_Initialize)

Returns

true if the delayed initialization option has been enabled

Remarks

This is not needed for audio-only applications without a Bluetooth module.

C

```
bool DRV_AK4953_IsInitializationDelayed(SYS_MODULE_OBJ object);
```

DRV_AK4953_Status Function

SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object)

Summary

Gets the current status of the AK4953 driver module.

Description

This routine provides the current status of the AK4953 driver module.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_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_AK4953_Initialize
SYS_STATUS          AK4953Status;

AK4953Status = DRV_AK4953_Status(object);
if (SYS_STATUS_READY == AK4953Status)
{
    // This means the driver can be opened using the
    // DRV_AK4953_Open() function.
}
```

C

```
SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object);
```

DRV_AK4953_Tasks Function

void DRV_AK4953_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_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4953_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_AK4953_Initialize

while (true)
{
    DRV_AK4953_Tasks (object);

    // Do other tasks
}
```

C

```
void DRV_AK4953_Tasks(SYS_MODULE_OBJ object);
```

Client Setup Functions

DRV_AK4953_Close Function

void DRV_AK4953_Close(DRV_Handle handle)

Summary

Closes an opened-instance of the AK4953 driver

Description

This routine closes an opened-instance of the AK4953 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_AK4953_Open before the caller may use the driver again

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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_AK4953_Open

DRV_AK4953_Close(handle);
```

C

```
void DRV_AK4953_Close(const DRV_HANDLE handle);
```

DRV_AK4953_CommandEventHandlerSet Function

void DRV_AK4953_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4953_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4953_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 "AK4953 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_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_CommandEventHandlerSet(myAK4953Handle,
                APP_AK4953CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_DeEmphasisFilterSet(myAK4953Handle, DRV_AK4953_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4953CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

C

```
void DRV_AK4953_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4953_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

DRV_AK4953_Open Function

DRV_HANDLE DRV_AK4953_Open

(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
)

Summary

Opens the specified AK4953 driver instance and returns a handle to it.

Description

This routine opens the specified AK4953 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.

AK4953 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_AK4953_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_AK4953_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_AK4953_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_AK4953_Open(DRV_AK4953_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_AK4953_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

DRV_AK4953_BufferEventHandlerSet Function

```
void DRV_AK4953_BufferEventHandlerSet
(
DRV_HANDLE handle,
const        DRV_AK4953_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_AK4953_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddRead(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);
```

```
if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4953_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4953_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Data Transfer Functions

DRV_AK4953_BufferAddRead Function

void DRV_AK4953_BufferAddRead
(
const DRV_HANDLE handle,
    DRV_AK4953_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_AK4953_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_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4953_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_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_AK4953_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 AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

C

```
void DRV_AK4953_BufferAddRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

DRV_AK4953_BufferAddWrite Function

void DRV_AK4953_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4953_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_AK4953_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_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4953_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_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_AK4953_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 AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddWrite(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
```

```
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4953_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

DRV_AK4953_BufferAddWriteRead Function

```
void DRV_AK4953_BufferAddWriteRead
(
const DRV_HANDLE handle,
        DRV_AK4953_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)
```

Summary

Schedule a non-blocking driver write-read operation.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_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_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4953_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as returned by the DRV_AK4953_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_AK4953_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 AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4953 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_AK4953_BUFFER_HANDLE bufferHandle;

// myak4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myak4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddWriteRead(myak4953handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4953_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

DRV_AK4953_ReadQueuePurge Function

bool DRV_AK4953_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_AK4953_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_AK4953_Open function.
// Use DRV_AK4953_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_AK4953_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_AK4953_ReadQueuePurge(const DRV_HANDLE handle);
```

DRV_AK4953_WriteQueuePurge Function

bool DRV_AK4953_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_AK4953_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_AK4953_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_AK4953_Open function.
// Use DRV_AK4953_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_AK4953_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_AK4953_WriteQueuePurge(const DRV_HANDLE handle);
```

Settings Functions

DRV_AK4953_IntExtMicSet Function

void DRV_AK4953_IntExtMicSet

Summary

This function sets up the codec for the X32 DB internal or the external microphone use.

Description

This function sets up the codec for the internal or the external microphone use.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Returns

None

Remarks

None.

C

```
void DRV_AK4953_IntExtMicSet(DRV_HANDLE handle, DRV_AK4953_INT_EXT_MIC micInput);
```

DRV_AK4953_MicGainGet Function

uint8_t DRV_AK4953_MicGainGet(DRV_HANDLE handle)

Summary

This function gets the microphone gain for the AK4953 Codec.

Description

This functions gets the current microphone gain programmed to the Codec AK4953.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

Microphone gain, in range 0-31.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t gain;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

  gain = DRV_AK4953_MicGainGet(myAK4953Handle);
```

C

```
uint8_t DRV_AK4953_MicGainGet(DRV_HANDLE handle);
```

DRV_AK4953_MicGainSet Function

void DRV_AK4953_MicGainSet(DRV_HANDLE handle, uint8_t gain)

Summary

This function sets the microphone gain for the AK4953 CODEC.

Description

This functions sets the microphone gain value from 0-31 which can range from -1.5 to 28.3 dB

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
gain	Gain value, in range 0-31

Returns

None.

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_MicGainSet(myAK4953Handle, 15);  //AK4953 mic gain set to 15
```

Remarks

None.

C

```
void DRV_AK4953_MicGainSet(DRV_HANDLE handle, uint8_t gain);
```

DRV_AK4953_MicMuteOff Function

void DRV_AK4953_MicMuteOff(DRV_HANDLE handle)

Summary

Umutes th AK4953's microphone input.

Description

This function unmutes the AK4953's microphone input.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_MicMuteOff(myAK4953Handle);  //AK4953 microphone unmuted
```

C

```
void DRV_AK4953_MicMuteOff(DRV_HANDLE handle);
```

DRV_AK4953_MicMuteOn Function

void DRV_AK4953_MicMuteOn(DRV_HANDLE handle);

Summary

Mutes the AK4953's microphone input

Description

This function mutes the AK4953's microphone input

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_MicMuteOn(myAK4953Handle);   //AK4953 microphone muted
```

C

```
void DRV_AK4953_MicMuteOn(DRV_HANDLE handle);
```

DRV_AK4953_MicSet Function

void DRV_AK4953_IntMic12Set

Summary

This function sets up the codec for the internal or the AK4953 Mic1 or Mic2 input.

Description

This function sets up the codec.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Returns

None

Remarks

None.

C

```
void DRV_AK4953_MicSet(DRV_HANDLE handle, DRV_AK4953_MIC micInput);
```

DRV_AK4953_MonoStereoMicSet Function

void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle);

Summary

This function sets up the codec for the Mono or Stereo microphone mode.

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None

Remarks

None.

C

```
void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4953_MONO_STEREO_MIC mono_stereo_mic);
```

DRV_AK4953_MuteOff Function

void DRV_AK4953_MuteOff(DRV_HANDLE handle)

Summary

This function disables AK4953 output for soft mute.

Description

This function disables AK4953 output for soft mute.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_MuteOff(myAK4953Handle); //AK4953 output soft mute disabled
```

C

```
void DRV_AK4953_MuteOff(DRV_HANDLE handle);
```

DRV_AK4953_MuteOn Function

void DRV_AK4953_MuteOn(DRV_HANDLE handle);

Summary

This function allows AK4953 output for soft mute on.

Description

This function Enables AK4953 output for soft mute.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_MuteOn(myAK4953Handle);  //AK4953 output soft muted
```

C

```
void DRV_AK4953_MuteOn(DRV_HANDLE handle);
```

DRV_AK4953_SamplingRateGet Function

uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle)

Summary

This function gets the sampling rate set on the DAC AK4953.

Description

This function gets the sampling rate set on the DAC AK4953.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
uint32_t baudRate;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

baudRate = DRV_AK4953_SamplingRateGet(myAK4953Handle);
```

C

```
uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle);
```

DRV_AK4953_SamplingRateSet Function

void DRV_AK4953_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_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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

```
// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_SamplingRateSet(myAK4953Handle, 48000);  //Sets 48000 media sampling rate
```

C

```
void DRV_AK4953_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

DRV_AK4953_VolumeGet Function

uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan)

Summary

This function gets the volume for AK4953 Codec.

Description

This functions gets the current volume programmed to the CODEC AK4953.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

  volume = DRV_AK4953_VolumeGet(myAK4953Handle,DRV_AK4953_CHANNEL_LEFT);
```

C

```
uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan);
```

DRV_AK4953_VolumeSet Function

void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);

Summary

This function sets the volume for AK4953 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_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_VolumeSet(myAK4953Handle, DRV_AK4953_CHANNEL_LEFT, 120);
```

C

```
void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);
```

Other Functions

DRV_AK4953_GetI2SDriver Function

DRV_HANDLE DRV_AK4953_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_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_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_AK4953_GetI2SDriver(DRV_HANDLE codecHandle);
```

DRV_AK4953_LRCLK_Sync Function

uint32_t DRV_AK4953_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

```
// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_LRCLK_Sync(myAK4953Handle);
```

C

```
bool DRV_AK4953_LRCLK_Sync(const DRV_HANDLE handle);
```

DRV_AK4953_VersionGet Function

uint32_t DRV_AK4953_VersionGet(void)

Summary

This function returns the version of AK4953 driver.

Description

The version number returned from the DRV_AK4953_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 AK4953 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 AK4953version;
AK4953version = DRV_AK4953_VersionGet();
```

C

```
uint32_t DRV_AK4953_VersionGet();
```

DRV_AK4953_VersionStrGet Function

int8_t* DRV_AK4953_VersionStrGet(void)

Summary

This function returns the version of AK4953 driver in string format.

Description

The DRV_AK4953_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4953 driver's version number. is the AK4953 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

Preconditions

None.

Returns

returns a string containing the version of AK4953 driver.

Remarks

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *AK4953string;
AK4953string = DRV_AK4953_VersionStrGet();
```

C

```
int8_t* DRV_AK4953_VersionStrGet();
```

Data Types and Constants

DRV_AK4953_AUDIO_DATA_FORMAT Type

Identifies the Serial Audio data interface format.

Description

AK4953 Audio data format

This enumeration identifies Serial Audio data interface format.

C

```
typedef enum DRV_AK4953_AUDIO_DATA_FORMAT@1 DRV_AK4953_AUDIO_DATA_FORMAT;
```

DRV_AK4953_BUFFER_EVENT Type

Identifies the possible events that can result from a buffer add request.

Description

AK4953 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4953_BufferAddWrite() or the DRV_AK4953_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_AK4953_BufferEventHandlerSet function when a buffer transfer request is completed.

C

```
typedef enum DRV_AK4953_BUFFER_EVENT@1 DRV_AK4953_BUFFER_EVENT;
```

DRV_AK4953_BUFFER_EVENT_HANDLER Type

Pointer to a AK4953 Driver Buffer Event handler function

Description

AK4953 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4953 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_AK4953_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4953_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_AK4953_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_AK4953_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_AK4953_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4953_BUFFER_EVENT event,
                               DRV_AK4953_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

C

```
typedef void (* DRV_AK4953_BUFFER_EVENT_HANDLER)(DRV_AK4953_BUFFER_EVENT event,
DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

DRV_AK4953_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

Description

AK4953 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4953_BufferAddWrite() or DRV_AK4953_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_AK4953_BUFFER_HANDLE;
```

DRV_AK4953_CHANNEL Type

Identifies Left/Right Audio channel

Description

AK4953 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

C

```
typedef enum DRV_AK4953_CHANNEL@1 DRV_AK4953_CHANNEL;
```

DRV_AK4953_COMMAND_EVENT_HANDLER Type

Pointer to a AK4953 Driver Command Event Handler Function

Description

AK4953 Driver Command Event Handler Function

This data type defines the required function signature for the AK4953 driver command event handling callback function.

A command is a control instruction to the AK4953 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_AK4953_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_AK4953CommandEventHandler( 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_AK4953_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

DRV_AK4953_DIGITAL_BLOCK_CONTROL Type

Identifies Bass-Boost Control function

Description

AK4953 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

Remarks

None.

C

```
typedef enum DRV_AK4953_DIGITAL_BLOCK_CONTROL@1 DRV_AK4953_DIGITAL_BLOCK_CONTROL;
```

DRV_AK4953_INIT Type

Defines the data required to initialize or reinitialize the AK4953 driver

Description

AK4953 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4953 Codec driver.

Remarks

None.

C

```
typedef struct DRV_AK4953_INIT@1 DRV_AK4953_INIT;
```

DRV_AK4953_INT_EXT_MIC Type

Identifies the Mic input source.

Description

AK4953 Mic Internal / External Input

This enumeration identifies the Mic input source.

C

```
typedef enum DRV_AK4953_INT_EXT_MIC@1 DRV_AK4953_INT_EXT_MIC;
```

DRV_AK4953_MIC Type

This is type DRV_AK4953_MIC.

C

```
typedef enum DRV_AK4953_MIC@1 DRV_AK4953_MIC;
```

DRV_AK4953_MONO_STEREO_MIC Type

Identifies the Mic input as Mono / Stereo.

Description

AK4953 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

C

```
typedef enum DRV_AK4953_MONO_STEREO_MIC@1 DRV_AK4953_MONO_STEREO_MIC;
```

Files

AK4954 CODEC Driver Library Help

This topic describes the AK4954 Codec Driver Library.

Introduction

This topic describes the basic architecture of the AK4954 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_AK4954.h`

The interface to the AK4954 Codec Driver library is defined in the `audio/driver/codec/AK4954/drv_AK4954.h` header file. Any C language source (.c) file that uses the AK4954 Codec Driver library should include this header.

Library Source Files:

The AK4954 Codec Driver library source files are provided in the `audio/driver/codec/AK4954/src` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features and to **Building the Library** for instructions on how to build the library.

Example Applications:

This library is used by the following applications:

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

Using the Library

Abstration Model

This library provides a low-level abstraction of the AK4954 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4954 Codec Driver is positioned in the MPLAB Harmony framework. The AK4954 Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the AK4954 module.

AK4954 Driver Abstraction Model

Library Overview

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

The AK4954 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4954 Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4954 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions, such as Buffer Read and Write.
Settings Functions	Provides driver specific functions for settings, such as volume control and sampling rate.

Other Functions	Miscellaneous functions, such as getting the driver's version number and syncing to the LRCLK signal.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4954 Codec Driver Library.

Note: All functions and constants in this section are named with the format DRV_ AK4954_xxx, where 'xxx' is a function name or constant. These names are redefined in the appropriate configuration's configuration.h file to the format DRV_CODEC_xxx using #defines so that code in the application that references the library can be written as generically as possible (e.g., by writing DRV_CODEC_Open instead of DRV_ AK4954_Open etc.). This allows the codec type to be changed in the MHC without having to modify the application's source code.

How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Setup (Initialization)

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

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization in the system_init.c file, each instance of the AK4954 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4954_INIT or by using Initialization Overrides) that are supported by the specific AK4954 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

The DRV_AK4954_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4954_Deinitialize, DRV_ AK4954_Status and DRV_I2S_Tasks

Client Access

This topic describes driver initialization and provides a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4954_Open function. The DRV_AK4954_Open function provides a driver handle to the AK4954 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4954_Deinitialize, the application must call the DRV_AK4954_Open function again to set up the instance of the driver.

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

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4954 Codec Driver can be known by calling DRV_ AK4954_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4954Status;
ak4954Status Status = DRV_AK4954_Status(sysObjects.ak4954Status DevObject);
if (SYS_STATUS_READY == ak4954Status)
{
// The driver can now be opened.
appData.ak4954Client.handle =  DRV_AK4954_Open
( DRV_AK4954_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.ak4954Client.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_AK4954_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* AK4954 Driver Is not ready */
}
```

Client Operations

This topic provides information on client operations.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4954 Codec.

The following AK4954 Codec specific control command functions are provided:

- DRV_AK4954_SamplingRateSet
- DRV_AK4954_SamplingRateGet
- DRV_AK4954_VolumeSet
- DRV_AK4954_VolumeGet
- DRV_AK4954_MuteOn
- DRV_AK4954_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4954 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_AK4954_BufferAddWrite, DRV_AK4954_BufferAddRead, and DRV_AK4954_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK4954_BUFFER_EVENT_COMPLETE, DRV_AK4954_BUFFER_EVENT_ERROR, or DRV_AK4954_BUFFER_EVENT_ABORT events.

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

Configuring the Library

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

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

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

System Configuration

Macros

	Name	Description
	DRV_AK4954_AUDIO_DATA_FORMAT_MACRO	Specifies the audio data format for the codec.
	DRV_AK4954_AUDIO_SAMPLING_RATE	Specifies the initial baud rate for the codec.
	DRV_AK4954_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4954_ENABLE_MIC_BIAS	Specifies whether to enable the microphone bias.
	DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx	Specifies the instance number of the I2C interface.
	DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx	Specifies the instance number of the I2S interface.
	DRV_AK4954_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4954_MASTER_MODE	Specifies if codec is in Master or Slave mode.
	DRV_AK4954_MIC_GAIN	Specifies the gain of the microphone (external or line input only)
	DRV_AK4954_VOLUME	Specifies the initial volume level.
	DRV_AK4954_WHICH_MIC_INPUT	Specifies whether to enable the microphone input.

Description

DRV_AK4954_AUDIO_DATA_FORMAT_MACRO Macro

Specifies the audio data format for the codec.

Description

AK4954 Audio Data Format

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

Valid choices are: "DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_LSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_16BIT_LSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_MSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_I2S_16BIT_24BIT"
"DRV_AK4954_AUDIO_DATA_FORMAT_32BIT_MSB_SDTO_32BIT_MSB_SDTI"
"DRV_AK4954_AUDIO_DATA_FORMAT_I2S_32BIT" where SDTO is input line (ADC) and STDI is output line (DAC)

Remarks

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

C

```
#define DRV_AK4954_AUDIO_DATA_FORMAT_MACRO
```

DRV_AK4954_AUDIO_SAMPLING_RATE Macro

Specifies the initial baud rate for the codec.

Description

AK4954 Baud Rate

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

C

```
#define DRV_AK4954_AUDIO_SAMPLING_RATE
```

DRV_AK4954_CLIENTS_NUMBER Macro

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

Description

AK4954 Client Count Configuration

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

C

```
#define DRV_AK4954_CLIENTS_NUMBER
```

DRV_AK4954_ENABLE_MIC_BIAS Macro

Specifies whether to enable the microphone bias.

Description

AK4954 Microphone Enable

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

C

```
#define DRV_AK4954_ENABLE_MIC_BIAS
```

DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2C interface.

Description

AK4954 I2C instance number

Specifies the instance number of the I2C interface being used by the MCU to send commands and receive status to and from the AK4954. enabled.

C

```
#define DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx
```

DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2S interface.

Description

AK4954 I2S instance number

Specifies the instance number of the I2S interface being used by the MCU to send and receive audio data to and from the AK4954. enabled.

C

```
#define DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx
```

DRV_AK4954_INSTANCES_NUMBER Macro

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

Description

AK4954 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4954 Codec modules that are needed by an application, namely one.

C

```
#define DRV_AK4954_INSTANCES_NUMBER
```

DRV_AK4954_MASTER_MODE Macro

Specifies if codec is in Master or Slave mode.

Description

AK4954 Codec Master/Slave Mode

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

C

```
#define DRV_AK4954_MASTER_MODE
```

DRV_AK4954_MIC_GAIN Macro

Specifies the gain of the microphone (external or line input only)

Description

AK4954 Microphone Gain

Specifies the gain of the microphone (external or line input only), on a scale of 0-31

C

```
#define DRV_AK4954_MIC_GAIN
```

DRV_AK4954_VOLUME Macro

Specifies the initial volume level.

Description

AK4954 Volume

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

Remarks

The value is mapped to an internal AK4954 volume level in the range 0-192 using a logarithmic table so the input scale appears linear (128 is half volume).

C

```
#define DRV_AK4954_VOLUME
```

DRV_AK4954_WHICH_MIC_INPUT Macro

Specifies whether to enable the microphone input.

Description

AK4954 Which Microphone

Indicates which microphone (or line input) is chosen

Valid choices are: "MIC1" (Internal Mic on board) "MIC2" (External Mic Input) "MIC3" (Line Input)

C

```
#define DRV_AK4954_WHICH_MIC_INPUT
```

Configuring MHC

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

Description

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

In the MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under *Audio>Templates*, double-click on a codec template such as AK4954. Answer Yes to all questions.

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

Click on the AK4954 Driver component (not AK4954 Codec) and the following menu will be displayed in the Configurations Options:

- **I2C Driver Used** will display the driver instance used for the I^2C interface.
- **I2S Driver Used** will display the driver instance used for the I^2S interface.
- **Usage Mode** indicates whether the AK4954 is a Master (supplies I^2S clocks) or a Slave (MCU supplies I^2S clocks).
- **Number of AK4954 Clients** indicates the maximum number of clients that can be connected to the AK4954 Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either
  - 24-bit Left Justified (ADC), 24-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 16-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 24-bit Left-justified(DAC)
  - 24/16-bit I2S
  - 32-bit Left Justified (ADC), 32-bit Left-justified(DAC)
  - 32-bit I2S

It must match the audio protocol and data length set up in either the SSC or I2S PLIB.

- **Microphone/Line Input** selects which microphone or line input is selected, either:
  - Internal Mic (mounted on the AK4954 daughterboard)
  - External Mic Input
  - Line Input

If External Mic input or Line Input is selected, then the following option is provided:

- **Ext Mic Gain in dB range 0(min) to 31(max)**

If External Mic input is selected, then the following option is provided:

- **Enable Microphone Bias** should be checked if using an electret microphone.

You can also bring in the AK4954 Driver by itself, by double clicking AK4954 under Audio->*Driver*->*Codec* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

Note that the AK4954 requires the TCx Peripheral Library and TIME System Service in order to perform some of its internal timing sequences.

Building the Library

This section lists the files that are available in the AK4954 Codec Driver Library.

Description

This section lists the files that are available in the `src` folder of the AK4954 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `audio/driver/codec/AK4954`.

Interface File(s)

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

Source File Name	Description
`drv_ak4954.h`	Header file that exports the driver API.

Required File(s)

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

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

Source File Name	Description
`/src/drv_ak4954.c`	This file contains implementation of the AK4954 Codec Driver.

Optional File(s)

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

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

Module Dependencies

The AK4954 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

Client Setup Functions

	Name	Description
	DRV_AK4954_Open	Opens the specified AK4954 driver instance and returns a handle to it.
	DRV_AK4954_Close	Closes an opened-instance of the AK4954 driver
	DRV_AK4954_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4954_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Data Transfer Functions

	Name	Description
	DRV_AK4954_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4954_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4954_BufferAddWriteRead	Schedule a non-blocking driver write-read operation.
	DRV_AK4954_ReadQueuePurge	Removes all buffer requests from the read queue.
	DRV_AK4954_WriteQueuePurge	Removes all buffer requests from the write queue.

Data Types and Constants

	Name	Description
	DRV_AK4954_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4954_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4954_BUFFER_EVENT_HANDLER	Pointer to a AK4954 Driver Buffer Event handler function
	DRV_AK4954_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4954_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4954_COMMAND_EVENT_HANDLER	Pointer to a AK4954 Driver Command Event Handler Function

	DRV_AK4954_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4954_INIT	Defines the data required to initialize or reinitialize the AK4954 driver
	DRV_AK4954_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4954_MIC	This is type DRV_AK4954_MIC.
	DRV_AK4954_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.
	SAMPLE_LENGTH	in bits
	DRV_AK4954_AUDIO_DATA_FORMAT_I2S	for compatability with old code
	DRV_AK4954_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4954_COUNT	Number of valid AK4954 driver indices
	DRV_AK4954_INDEX_0	AK4954 driver index definitions
	DRV_AK4954_INDEX_1	This is macro DRV_AK4954_INDEX_1.
	DRV_AK4954_INDEX_2	This is macro DRV_AK4954_INDEX_2.
	DRV_AK4954_INDEX_3	This is macro DRV_AK4954_INDEX_3.
	DRV_AK4954_INDEX_4	This is macro DRV_AK4954_INDEX_4.
	DRV_AK4954_INDEX_5	This is macro DRV_AK4954_INDEX_5.

Other Functions

	Name	Description
	DRV_AK4954_LRCLK_Sync	Synchronize to the start of the I2S LRCLK (left/right clock) signal
	DRV_AK4954_GetI2SDriver	Get the handle to the I2S driver for this codec instance.
	DRV_AK4954_VersionStrGet	This function returns the version of AK4954 driver in string format.
	DRV_AK4954_VersionGet	This function returns the version of AK4954 driver.

Settings Functions

	Name	Description
	DRV_AK4954_MicGainGet	This function gets the microphone gain for the AK4954 Codec.
	DRV_AK4954_MicGainSet	This function sets the microphone gain for the AK4954 CODEC.
	DRV_AK4954_MicMuteOff	Umutes th AK4954's microphone input.
	DRV_AK4954_MicMuteOn	Mutes the AK4954's microphone input
	DRV_AK4954_MicSet	This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.
	DRV_AK4954_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4954_MuteOff	This function disables AK4954 output for soft mute.
	DRV_AK4954_MuteOn	This function allows AK4954 output for soft mute on.
	DRV_AK4954_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.
	DRV_AK4954_SamplingRateGet	This function gets the sampling rate set on the DAC AK4954.
	DRV_AK4954_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4954_VolumeGet	This function gets the volume for AK4954 Codec.
	DRV_AK4954_VolumeSet	This function sets the volume for AK4954 Codec.

System Interaction Functions

	Name	Description
	DRV_AK4954_Initialize	Initializes hardware and data for the instance of the AK4954 Codec module.
	DRV_AK4954_EnableInitialization	Enable delayed initialization of the driver.
	DRV_AK4954_IsInitializationDelayed	Checks if delayed initialization of the driver has been requested.
	DRV_AK4954_Deinitialize	Deinitializes the specified instance of the AK4954 driver module.
	DRV_AK4954_Status	Gets the current status of the AK4954 driver module.
	DRV_AK4954_Tasks	Maintains the driver's control and data interface state machine.

Description

System Interaction Functions

DRV_AK4954_Initialize Function

SYS_MODULE_OBJ DRV_AK4954_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);

Summary

Initializes hardware and data for the instance of the AK4954 Codec module.

Description

This routine initializes the AK4954 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Preconditions

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

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Returns

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

Remarks

This routine must be called before any other AK4954 routine is called.

This routine should only be called once during system initialization unless DRV_AK4954_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Example

```
DRV_AK4954_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = ak4954Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = ak4954Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_AK4954_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_AK4954_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4954_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4954Init->volume;
}
init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_AK4954_COMMAND_EVENT_HANDLER)0;
```

```
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4954_Initialize(DRV_AK4954_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

C

```
SYS_MODULE_OBJ DRV_AK4954_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

DRV_AK4954_EnableInitialization Function

void DRV_AK4954_EnableInitialization(SYS_MODULE_OBJ object);

Summary

Enable delayed initialization of the driver.

Description

If the AK4954 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. Once this has been accomplished, this function should be called to kick-start the codec driver initialization.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4954_Initialize)

Returns

None.

Remarks

This is not needed for audio-only applications without a Bluetooth module.

C

```
void DRV_AK4954_EnableInitialization(SYS_MODULE_OBJ object);
```

DRV_AK4954_IsInitializationDelayed Function

bool DRV_AK4954_IsInitializationDelayed(SYS_MODULE_OBJ object);

Summary

Checks if delayed initialization of the driver has been requested.

Description

If the AK4954 codec is sharing a RESET line with another peripheral, such as a Bluetooth module with its own driver, then the codec driver initialization has to be delayed until after the Bluetooth module has toggled its RESET pin. This function returns true if that option has been selected in MHC in the checkbox: "Delay driver initialization (due to shared RESET pin)"

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4954_Initialize)

Returns

true if the delayed initialization option has been enabled

Remarks

This is not needed for audio-only applications without a Bluetooth module.

C

```
bool DRV_AK4954_IsInitializationDelayed(SYS_MODULE_OBJ object);
```

DRV_AK4954_Deinitialize Function

void DRV_AK4954_Deinitialize(SYS_MODULE_OBJ object)

Summary

Deinitializes the specified instance of the AK4954 driver module.

Description

Deinitializes the specified instance of the AK4954 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4954_Initialize routine

Returns

None.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4954_Initialize
SYS_STATUS          status;


DRV_AK4954_Deinitialize(object);

status = DRV_AK4954_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

C

```
void DRV_AK4954_Deinitialize(SYS_MODULE_OBJ object);
```

DRV_AK4954_Status Function

SYS_STATUS DRV_AK4954_Status(SYS_MODULE_OBJ object)

Summary

Gets the current status of the AK4954 driver module.

Description

This routine provides the current status of the AK4954 driver module.

Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4954_Initialize routine

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize
SYS_STATUS          AK4954Status;

AK4954Status = DRV_AK4954_Status(object);
if (SYS_STATUS_READY == AK4954Status)
{
    // This means the driver can be opened using the
    // DRV_AK4954_Open() function.
}
```

C

```
SYS_STATUS DRV_AK4954_Status(SYS_MODULE_OBJ object);
```

DRV_AK4954_Tasks Function

void DRV_AK4954_Tasks(SYS_MODULE_OBJ object);

Summary

Maintains the driver's control and data interface state machine.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4954_Initialize)

Returns

None.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize

while (true)
{
    DRV_AK4954_Tasks (object);

    // Do other tasks
}
```

C

```
void DRV_AK4954_Tasks(SYS_MODULE_OBJ object);
```

Client Setup Functions

DRV_AK4954_Open Function

DRV_HANDLE DRV_AK4954_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

Summary

Opens the specified AK4954 driver instance and returns a handle to it.

Description

This routine opens the specified AK4954 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4954 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Preconditions

Function DRV_AK4954_Initialize must have been called before calling this function.

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4954_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Remarks

The handle returned is valid until the DRV_AK4954_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4954_Open(DRV_AK4954_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

C

```
DRV_HANDLE DRV_AK4954_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

DRV_AK4954_Close Function

void DRV_AK4954_Close(DRV_Handle handle)

Summary

Closes an opened-instance of the AK4954 driver

Description

This routine closes an opened-instance of the AK4954 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4954_Open before the caller may use the driver again

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4954_Open

DRV_AK4954_Close(handle);
```

C

```c
void DRV_AK4954_Close(const DRV_HANDLE handle);
```

DRV_AK4954_BufferEventHandlerSet Function

void DRV_AK4954_BufferEventHandlerSet
(
DRV_HANDLE handle,
const DRV_AK4954_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)

Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4954_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddRead(myAK4954handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4954_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4954_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

DRV_AK4954_CommandEventHandlerSet Function

void DRV_AK4954_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4954_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted

command have finished.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4954_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4954 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_CommandEventHandlerSet(myAK4954Handle,
                APP_AK4954CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_DeEmphasisFilterSet(myAK4954Handle, DRV_AK4954_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4954CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

C

```
void DRV_AK4954_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4954_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Data Transfer Functions

DRV_AK4954_BufferAddRead Function

void DRV_AK4954_BufferAddRead
(
const DRV_HANDLE handle,
    DRV_AK4954_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

Summary

Schedule a non-blocking driver read operation.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4954_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as return by the DRV_AK4954_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not

otherwise be called directly in an ISR.

C

```
void DRV_AK4954_BufferAddRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

DRV_AK4954_BufferAddWrite Function

void DRV_AK4954_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

Summary

Schedule a non-blocking driver write operation.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4954_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as return by the DRV_AK4954_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
```

```
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWrite(myAK4954handle, &bufferHandle
                                  myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4954_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

DRV_AK4954_BufferAddWriteRead Function

void DRV_AK4954_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

Summary

Schedule a non-blocking driver write-read operation.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4954_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as returned by the DRV_AK4954_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4954 write. The transmit and receive size must be same.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myak4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myak4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWriteRead(myak4954handle, &bufferHandle,
                                  mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
```

```
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_AK4954_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

DRV_AK4954_ReadQueuePurge Function

bool DRV_AK4954_ReadQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the read queue.

Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_AK4954_Open function.

Returns

True - Read queue purge is successful. False - Read queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_AK4954_Open function.
// Use DRV_AK4954_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
```

```
void APP_TimeOut(void)
{
    if(false == DRV_AK4954_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_AK4954_ReadQueuePurge(const DRV_HANDLE handle);
```

DRV_AK4954_WriteQueuePurge Function

bool DRV_AK4954_WriteQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the write queue.

Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_AK4954_Open function.

Returns

True - Write queue purge is successful. False - Write queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_AK4954_Open function.
// Use DRV_AK4954_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_AK4954_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_AK4954_WriteQueuePurge(const DRV_HANDLE handle);
```

Settings Functions

DRV_AK4954_MicGainGet Function

uint8_t DRV_AK4954_MicGainGet(DRV_HANDLE handle)

Summary

This function gets the microphone gain for the AK4954 Codec.

Description

This functions gets the current microphone gain programmed to the Codec AK4954.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

Microphone gain, in range 0-31.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t gain;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

  gain = DRV_AK4954_MicGainGet(myAK4954Handle);
```

C

```
uint8_t DRV_AK4954_MicGainGet(DRV_HANDLE handle);
```

DRV_AK4954_MicGainSet Function

void DRV_AK4954_MicGainSet(DRV_HANDLE handle, uint8_t gain)

Summary

This function sets the microphone gain for the AK4954 CODEC.

Description

This functions sets the microphone gain value from 0-31 which can range from -1.5 to 28.3 dB

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
gain	Gain value, in range 0-31

Returns

None.

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MicGainSet(myAK4954Handle, 15);  //AK4954 mic gain set to 15
```

Remarks

None.

C

```
void DRV_AK4954_MicGainSet(DRV_HANDLE handle, uint8_t gain);
```

DRV_AK4954_MicMuteOff Function

void DRV_AK4954_MicMuteOff(DRV_HANDLE handle)

Summary

Umutes th AK4954's microphone input.

Description

This function unmutes the AK4954's microphone input.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MicMuteOff(myAK4954Handle);  //AK4954 microphone unmuted
```

C

```
void DRV_AK4954_MicMuteOff(DRV_HANDLE handle);
```

DRV_AK4954_MicMuteOn Function

void DRV_AK4954_MicMuteOn(DRV_HANDLE handle);

Summary

Mutes the AK4954's microphone input

Description

This function mutes the AK4954's microphone input

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_MicMuteOn(myAK4954Handle);   //AK4954 microphone muted
```

C

```
void DRV_AK4954_MicMuteOn(DRV_HANDLE handle);
```

DRV_AK4954_MicSet Function

void DRV_AK4954_IntMic12Set

Summary

This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.

Description

This function sets up the codec.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Returns

None

Remarks

None.

C

```
void DRV_AK4954_MicSet(DRV_HANDLE handle, DRV_AK4954_MIC micInput);
```

DRV_AK4954_MonoStereoMicSet Function

void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle);

Summary

This function sets up the codec for the Mono or Stereo microphone mode.

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None

Remarks

None.

C

```
void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4954_MONO_STEREO_MIC mono_stereo_mic);
```

DRV_AK4954_MuteOff Function

void DRV_AK4954_MuteOff(DRV_HANDLE handle)

Summary

This function disables AK4954 output for soft mute.

Description

This function disables AK4954 output for soft mute.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MuteOff(myAK4954Handle); //AK4954 output soft mute disabled
```

C

```
void DRV_AK4954_MuteOff(DRV_HANDLE handle);
```

DRV_AK4954_MuteOn Function

void DRV_AK4954_MuteOn(DRV_HANDLE handle);

Summary

This function allows AK4954 output for soft mute on.

Description

This function Enables AK4954 output for soft mute.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_MuteOn(myAK4954Handle);  //AK4954 output soft muted
```

C

```
void DRV_AK4954_MuteOn(DRV_HANDLE handle);
```

DRV_AK4954_IntExtMicSet Function

void DRV_AK4954_IntExtMicSet

Summary

This function sets up the codec for the X32 DB internal or the external microphone use.

Description

This function sets up the codec for the internal or the external microphone use.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Returns

None

Remarks

None.

C

```
void DRV_AK4954_IntExtMicSet(DRV_HANDLE handle, DRV_AK4954_INT_EXT_MIC micInput);
```

DRV_AK4954_SamplingRateGet Function

uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle)

Summary

This function gets the sampling rate set on the DAC AK4954.

Description

This function gets the sampling rate set on the DAC AK4954.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
uint32_t baudRate;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

baudRate = DRV_AK4954_SamplingRateGet(myAK4954Handle);
```

C

```
uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle);
```

DRV_AK4954_SamplingRateSet Function

void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

Summary

This function sets the sampling rate of the media stream.

Description

This function sets the media sampling rate for the client handle.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_SamplingRateSet(myAK4954Handle, 48000);  //Sets 48000 media sampling rate
```

C

```
void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

DRV_AK4954_VolumeGet Function

uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan)

Summary

This function gets the volume for AK4954 Codec.

Description

This functions gets the current volume programmed to the CODEC AK4954.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

  volume = DRV_AK4954_VolumeGet(myAK4954Handle,DRV_AK4954_CHANNEL_LEFT);
```

C

```
uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan);
```

DRV_AK4954_VolumeSet Function

void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);

Summary

This function sets the volume for AK4954 Codec.

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_VolumeSet(myAK4954Handle, DRV_AK4954_CHANNEL_LEFT, 120);
```

C

```
void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);
```

Other Functions

DRV_AK4954_LRCLK_Sync Function

uint32_t DRV_AK4954_LRCLK_Sync (const DRV_HANDLE handle);

Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

Preconditions

None.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
    // myAK4954Handle is the handle returned
    // by the DRV_AK4954_Open function.

    DRV_AK4954_LRCLK_Sync(myAK4954Handle);
```

C

```
bool DRV_AK4954_LRCLK_Sync(const DRV_HANDLE handle);
```

DRV_AK4954_GetI2SDriver Function

DRV_HANDLE DRV_AK4954_GetI2SDriver(DRV_HANDLE codecHandle)

Summary

Get the handle to the I2S driver for this codec instance.

Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

- A handle to the I2S driver for this codec instance

Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

C

```
DRV_HANDLE DRV_AK4954_GetI2SDriver(DRV_HANDLE codecHandle);
```

DRV_AK4954_VersionStrGet Function

int8_t* DRV_AK4954_VersionStrGet(void)

Summary

This function returns the version of AK4954 driver in string format.

Description

The DRV_AK4954_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4954 driver's version number. is the AK4954 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

Preconditions

None.

Returns

returns a string containing the version of AK4954 driver.

Remarks

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *AK4954string;
AK4954string = DRV_AK4954_VersionStrGet();
```

C

```
int8_t* DRV_AK4954_VersionStrGet();
```

DRV_AK4954_VersionGet Function

uint32_t DRV_AK4954_VersionGet(void)

Summary

This function returns the version of AK4954 driver.

Description

The version number returned from the DRV_AK4954_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Preconditions

None.

Returns

returns the version of AK4954 driver.

Remarks

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK4954version;
AK4954version = DRV_AK4954_VersionGet();
```

C

```
uint32_t DRV_AK4954_VersionGet();
```

Data Types and Constants

DRV_AK4954_AUDIO_DATA_FORMAT Type

Identifies the Serial Audio data interface format.

Description

AK4954 Audio data format

This enumeration identifies Serial Audio data interface format.

C

```
typedef enum DRV_AK4954_AUDIO_DATA_FORMAT@1 DRV_AK4954_AUDIO_DATA_FORMAT;
```

DRV_AK4954_BUFFER_EVENT Type

Identifies the possible events that can result from a buffer add request.

Description

AK4954 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4954_BufferAddWrite() or the DRV_AK4954_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4954_BufferEventHandlerSet function when a buffer transfer request is completed.

C

```
typedef enum DRV_AK4954_BUFFER_EVENT@1 DRV_AK4954_BUFFER_EVENT;
```

DRV_AK4954_BUFFER_EVENT_HANDLER Type

Pointer to a AK4954 Driver Buffer Event handler function

Description

AK4954 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4954 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

If the event is DRV_AK4954_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4954_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4954_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4954_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4954_BUFFER_EVENT event,
                               DRV_AK4954_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

C

```
typedef void (* DRV_AK4954_BUFFER_EVENT_HANDLER)(DRV_AK4954_BUFFER_EVENT event,
DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

DRV_AK4954_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

Description

AK4954 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4954_BufferAddWrite() or DRV_AK4954_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

C

```
typedef uintptr_t DRV_AK4954_BUFFER_HANDLE;
```

DRV_AK4954_CHANNEL Type

Identifies Left/Right Audio channel

Description

AK4954 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

C

```
typedef enum DRV_AK4954_CHANNEL@1 DRV_AK4954_CHANNEL;
```

DRV_AK4954_COMMAND_EVENT_HANDLER Type

Pointer to a AK4954 Driver Command Event Handler Function

Description

AK4954 Driver Command Event Handler Function

This data type defines the required function signature for the AK4954 driver command event handling callback function.

A command is a control instruction to the AK4954 Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4954CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

C

```
typedef void (* DRV_AK4954_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

DRV_AK4954_DIGITAL_BLOCK_CONTROL Type

Identifies Bass-Boost Control function

Description

AK4954 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

Remarks

None.

C

```
typedef enum DRV_AK4954_DIGITAL_BLOCK_CONTROL@1 DRV_AK4954_DIGITAL_BLOCK_CONTROL;
```

DRV_AK4954_INIT Type

Defines the data required to initialize or reinitialize the AK4954 driver

Description

AK4954 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4954 Codec driver.

Remarks

None.

C

```
typedef struct DRV_AK4954_INIT@1 DRV_AK4954_INIT;
```

DRV_AK4954_INT_EXT_MIC Type

Identifies the Mic input source.

Description

AK4954 Mic Internal / External Input

This enumeration identifies the Mic input source.

C

```
typedef enum DRV_AK4954_INT_EXT_MIC@1 DRV_AK4954_INT_EXT_MIC;
```

DRV_AK4954_MIC Type

This is type DRV_AK4954_MIC.

C

```
typedef enum DRV_AK4954_MIC@1 DRV_AK4954_MIC;
```

DRV_AK4954_MONO_STEREO_MIC Type

Identifies the Mic input as Mono / Stereo.

Description

AK4954 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

C

```
typedef enum DRV_AK4954_MONO_STEREO_MIC@1 DRV_AK4954_MONO_STEREO_MIC;
```

SAMPLE_LENGTH Type

in bits

C

```
typedef enum SAMPLE_LENGTH@2 SAMPLE_LENGTH;
```

DRV_AK4954_AUDIO_DATA_FORMAT_I2S Macro

for compatability with old code

C

```
#define DRV_AK4954_AUDIO_DATA_FORMAT_I2S DRV_AK4954_AUDIO_DATA_FORMAT_I2S_16BIT_24BIT  // for compatability with old code
```

DRV_AK4954_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

Description

AK4954 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4954_BufferAddWrite() and the DRV_AK4954_BufferAddRead() function if the buffer add request was not successful.

Remarks

None

C

```
#define DRV_AK4954_BUFFER_HANDLE_INVALID ((DRV_AK4954_BUFFER_HANDLE)(-1))
```

DRV_AK4954_COUNT Macro

Number of valid AK4954 driver indices

Description

AK4954 Driver Module Count

This constant identifies the maximum number of AK4954 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4954 instances on this microcontroller.

Remarks

This value is part-specific.

C

```
#define DRV_AK4954_COUNT
```

DRV_AK4954_INDEX_0 Macro

AK4954 driver index definitions

Description

Driver AK4954 Module Index

These constants provide AK4954 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the

DRV_AK4954_Initialize and DRV_AK4954_Open routines to identify the driver instance in use.

C

```
#define DRV_AK4954_INDEX_0 0
```

DRV_AK4954_INDEX_1 Macro

This is macro DRV_AK4954_INDEX_1.

C

```
#define DRV_AK4954_INDEX_1 1
```

DRV_AK4954_INDEX_2 Macro

This is macro DRV_AK4954_INDEX_2.

C

```
#define DRV_AK4954_INDEX_2 2
```

DRV_AK4954_INDEX_3 Macro

This is macro DRV_AK4954_INDEX_3.

C

```
#define DRV_AK4954_INDEX_3 3
```

DRV_AK4954_INDEX_4 Macro

This is macro DRV_AK4954_INDEX_4.

C

```
#define DRV_AK4954_INDEX_4 4
```

DRV_AK4954_INDEX_5 Macro

This is macro DRV_AK4954_INDEX_5.

C

```
#define DRV_AK4954_INDEX_5 5
```

Files

Files

Name	Description
drv_ak4954.h	AK4954 Codec Driver Interface header file
drv_ak4954_config_template.h	AK4954 Codec Driver Configuration Template.

Description

drv_ak4954.h

drv_ak4954.h

Summary

AK4954 Codec Driver Interface header file

Description

AK4954 Codec Driver Interface

The AK4954 Codec device driver interface provides a simple interface to manage the AK4954 16/24/32-Bit Codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the AK4954 Codec device driver.

drv_ak4954_config_template.h

drv_ak4954_config_template.h

Summary

AK4954 Codec Driver Configuration Template.

Description

AK4954 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

GENERIC AUDIO DRIVER CODEC Library

This topic describes the Generic Codec Driver Library.

Introduction

Summary

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

Description

Description

This file contains the implementation of the Generic Codec driver, which provides a simple interface to manage a codec that can be interfaced to Microchip microcontroller. The user will need to modify it to match the requirements of their codec. Areas where code needs to be added or changed are marked with TO-DO!!

Note: this module assumes the codec is controlled over an $I^2$2C interface. The I^2C Driver will need to be enabled in the MHC Project Graph. If another type of interface is used, the user will need to modify the code to suit. This module makes use of SYS_TIME. It will need to be enabled in the Project Graph.

This module assumes the an I^2S interface is used for audio data output (to headphones or line-out) and input (from microphone or line-in).

The Generic Codec can be configured as either an I^2S clock slave (receives all clocks from the host), or I^2S clock master (generates I^2S clocks from a master clock input MCLK).

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

Using the Library

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

Description

Interface Header File: `drv_Generic.h`

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

Library Source Files:

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

Example Applications:

This codec is not used directly by any demonstration applications. However the following applications could be looked at, to see how a codec such as the WM8904 or AK4954 is used:

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

Abstraction Model

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

Description

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

Generic Driver Abstraction Model

Library Overview

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

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

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

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

How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Setup (Initialization)

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

Description

System Initialization

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

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

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

Client Access

This topic describes driver initialization and provides a code example.

Description

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

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

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

Example:

```
DRV_HANDLE handle;
SYS_STATUS genericStatus;
genericStatus Status = DRV_Generic_Status(sysObjects.genericStatus DevObject);
if (SYS_STATUS_READY == genericStatus)
{
// The driver can now be opened.
appData.genericClient.handle = DRV_Generic_Open
```

```
(DRV_Generic_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if(appData.genericClient.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_Generic_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* Generic Driver Is not ready */
}
```

Client Operations

This topic provides information on client operations.

Description

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

The following Generic Codec specific control command functions are provided:

- DRV_Generic_SamplingRateSet
- DRV_Generic_SamplingRateGet
- DRV_Generic_VolumeSet
- DRV_Generic_VolumeGet
- DRV_Generic_MuteOn
- DRV_Generic_MuteOff

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

DRV_Generic_BufferAddWrite, DRV_Generic_BufferAddRead, and DRV_Generic_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_Generic_BUFFER_EVENT_COMPLETE, DRV_Generic_BUFFER_EVENT_ERROR, or DRV_Generic_BUFFER_EVENT_ABORT events.

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

Configuring the Library

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

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

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

System Configuration

Macros

	Name	Description
	DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO	Specifies the audio data format for the codec.
	DRV_GENERICCODEC_AUDIO_SAMPLING_RATE	Specifies the initial baud rate for the codec.
	DRV_GENERICCODEC_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx	Specifies the instance number of the I2C interface.
	DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx	Specifies the instance number of the I2S interface.
	DRV_GENERICCODEC_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_GENERICCODEC_VOLUME	Specifies the initial volume level.

Description

DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO Macro

Specifies the audio data format for the codec.

Description

Generic Codec Audio Data Format

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

Valid choices are: "DATA_16_BIT_LEFT_JUSTIFIED", 16-bit Left Justified "DATA_16_BIT_I2S", 16-bit I2S" "DATA_32_BIT_LEFT_JUSTIFIED", 32-bit Left Justified "DATA_32_BIT_I2S", 32-bit I2S"

C

```
#define DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO
```

DRV_GENERICCODEC_AUDIO_SAMPLING_RATE Macro

Specifies the initial baud rate for the codec.

Description

Generic Codec Baud Rate

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

C

```
#define DRV_GENERICCODEC_AUDIO_SAMPLING_RATE
```

DRV_GENERICCODEC_CLIENTS_NUMBER Macro

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

Description

Generic Codec Client Count Configuration

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

C

```
#define DRV_GENERICCODEC_CLIENTS_NUMBER
```

DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2C interface.

Description

Generic Codec I2C instance number

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

C

```
#define DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx
```

DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2S interface.

Description

Generic Codec I2S instance number

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

C

```
#define DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx
```

DRV_GENERICCODEC_INSTANCES_NUMBER Macro

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

Description

Generic Codec driver objects configuration

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

C

```
#define DRV_GENERICCODEC_INSTANCES_NUMBER
```

DRV_GENERICCODEC_VOLUME Macro

Specifies the initial volume level.

Description

Generic Codec Volume

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

Remarks

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

C

```
#define DRV_GENERICCODEC_VOLUME
```

Confuguring MHC

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

Description

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

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

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

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

- **I2C Driver Used** will display the driver instance used for the I^2C interface.
- **I2S Driver Used** will display the driver instance used for the I^2S interface.
- **Number of Generic Clients** indicates the maximum number of clients that can be connected to the Generic Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either
  - 24-bit Left Justified (ADC), 24-bit Right-justified(DAC)
  - 24-bit Left Justified (ADC), 16-bit Right-justified(DAC)

- 24-bit Left Justified (ADC), 24-bit Left-justified(DAC)
- 24/16-bit I2S
- 32-bit Left Justified (ADC), 32-bit Left-justified(DAC)
- 32-bit I2S

It must match the audio protocol and data length set up in either the SSC or I2S PLIB..

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

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

Building the Library

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

Description

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

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

Interface File(s)

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

Source File Name	Description
`drv_genericcodec.h`	Header file that exports the driver API.

Required File(s)

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

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

Source File Name	Description
`/src/drv_genericcodec.c`	This file contains implementation of the Generic Codec Driver.

Optional File(s)

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

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

Module Dependencies

The Generic Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

Client Setup Functions

	Name	Description
	DRV_GENERICCODEC_Open	Opens the specified Generic Codec driver instance and returns a handle to it
	DRV_GENERICCODEC_Close	Closes an opened-instance of the Generic Codec driver
	DRV_GENERICCODEC_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_GENERICCODEC_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Dara Types and Constants

	Name	Description
	DRV_GENERICCODEC_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_GENERICCODEC_BUFFER_EVENT_HANDLER	Pointer to a Generic Codec Driver Buffer Event handler function
	DRV_GENERICCODEC_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_GENERICCODEC_CHANNEL	Identifies Left/Right Audio channel
	DRV_GENERICCODEC_COMMAND_EVENT_HANDLER	Pointer to a Generic Codec Driver Command Event Handler Function
	DRV_GENERICCODEC_INIT	Defines the data required to initialize or reinitialize the Generic Codec driver
	DRV_GENERICCODEC_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_GENERICCODEC_COUNT	Number of valid Generic Codec driver indices
	DRV_GENERICCODEC_INDEX_0	Generic Codec driver index definitions
	DRV_GENERICCODEC_INDEX_1	This is macro DRV_GENERICCODEC_INDEX_1.
	DRV_GENERICCODEC_INDEX_2	This is macro DRV_GENERICCODEC_INDEX_2.
	DRV_GENERICCODEC_INDEX_3	This is macro DRV_GENERICCODEC_INDEX_3.
	DRV_GENERICCODEC_INDEX_4	This is macro DRV_GENERICCODEC_INDEX_4.
	DRV_GENERICCODEC_INDEX_5	This is macro DRV_GENERICCODEC_INDEX_5.
	DRV_GENERICCODEC_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.

Data Transfer Functions

	Name	Description
	DRV_GENERICCODEC_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_GENERICCODEC_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_GENERICCODEC_BufferAddWriteRead	Schedule a non-blocking driver write-read operation.
	DRV_GENERICCODEC_ReadQueuePurge	Removes all buffer requests from the read queue.
	DRV_GENERICCODEC_WriteQueuePurge	Removes all buffer requests from the write queue.

Other Functions

	Name	Description
	DRV_GENERICCODEC_GetI2SDriver	Get the handle to the I2S driver for this codec instance.
	DRV_GENERICCODEC_LRCLK_Sync	Synchronize to the start of the I2S LRCLK (left/right clock) signal

Settings Functions

	Name	Description
	DRV_GENERICCODEC_MicGainGet	This function gets the microphone gain for the Generic Codec.
	DRV_GENERICCODEC_MicGainSet	This function sets the microphone gain for the Generic Codec CODEC.
	DRV_GENERICCODEC_MicMuteOff	Umutes th Generic Codec's microphone input.
	DRV_GENERICCODEC_MicMuteOn	Mutes the Generic Codec's microphone input
	DRV_GENERICCODEC_MuteOff	This function disables Generic Codec output for soft mute.
	DRV_GENERICCODEC_MuteOn	This function allows Generic Codec output for soft mute on.
	DRV_GENERICCODEC_SamplingRateGet	This function gets the sampling rate set on the Generic Codec.
	DRV_GENERICCODEC_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_GENERICCODEC_VersionGet	This function returns the version of Generic Codec driver
	DRV_GENERICCODEC_VersionStrGet	This function returns the version of Generic Codec driver in string format.
	DRV_GENERICCODEC_VolumeGet	This function gets the volume for Generic Codec.
	DRV_GENERICCODEC_VolumeSet	This function sets the volume for Generic Codec.

System Interaction Functions

	Name	Description
	DRV_GENERICCODEC_Initialize	Initializes hardware and data for the instance of the Generic Codec module
	DRV_GENERICCODEC_Deinitialize	Deinitializes the specified instance of the Generic Codec driver module
	DRV_GENERICCODEC_Status	Gets the current status of the Generic Codec driver module.
	DRV_GENERICCODEC_Tasks	Maintains the driver's control and data interface state machine.

Description

System Interaction Functions

DRV_GENERICCODEC_Initialize Function

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

Summary

Initializes hardware and data for the instance of the Generic Codec module

Description

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

Preconditions

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

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Returns

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

Remarks

This routine must be called before any other Generic Codec routine is called.

This routine should only be called once during system initialization unless DRV_GENERICCODEC_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Example

```
DRV_GENERICCODEC_INIT               init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = genercicodecInit->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = genercicodecInit->i2cDriverModuleIndex;
init->samplingRate                    = DRV_GENERICCODEC_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_GENERICCODEC_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_GENERICCODEC_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_GENERICCODEC_Initialize(DRV_GENERICCODEC_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

C

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

DRV_GENERICCODEC_Deinitialize Function

void DRV_GENERICCODEC_Deinitialize(SYS_MODULE_OBJ object)

Summary

Deinitializes the specified instance of the Generic Codec driver module

Description

Deinitializes the specified instance of the Generic Codec driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Preconditions

Function DRV_GENERICCODEC_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_GENERICCODEC_Initialize routine

Returns

None.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_GENERICCODEC_Initialize
SYS_STATUS          status;


DRV_GENERICCODEC_Deinitialize(object);

status = DRV_GENERICCODEC_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

C

```
void DRV_GENERICCODEC_Deinitialize(SYS_MODULE_OBJ object);
```

DRV_GENERICCODEC_Status Function

SYS_STATUS DRV_GENERICCODEC_Status(SYS_MODULE_OBJ object)

Summary

Gets the current status of the Generic Codec driver module.

Description

This routine provides the current status of the Generic Codec driver module.

Preconditions

Function DRV_GENERICCODEC_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_GENERICCODEC_Initialize routine

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_GENERICCODEC_Initialize
```

```
SYS_STATUS          GENERICCODECStatus;

GENERICCODECStatus = DRV_GENERICCODEC_Status(object);
if (SYS_STATUS_READY == GENERICCODECStatus)
{
    // This means the driver can be opened using the
    // DRV_GENERICCODEC_Open() function.
}
```

C

```
SYS_STATUS DRV_GENERICCODEC_Status(SYS_MODULE_OBJ object);
```

DRV_GENERICCODEC_Tasks Function

void DRV_GENERICCODEC_Tasks(SYS_MODULE_OBJ object);

Summary

Maintains the driver's control and data interface state machine.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_GENERICCODEC_Initialize)

Returns

None.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_GENERICCODEC_Initialize

while (true)
{
    DRV_GENERICCODEC_Tasks (object);

    // Do other tasks
}
```

C

```
void DRV_GENERICCODEC_Tasks(SYS_MODULE_OBJ object);
```

Client Setup Functions

DRV_GENERICCODEC_Open Function

DRV_HANDLE DRV_GENERICCODEC_Open

(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
)

Summary

Opens the specified Generic Codec driver instance and returns a handle to it

Description

This routine opens the specified Generic Codec driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

Generic Codec can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Preconditions

Function DRV_GENERICCODEC_Initialize must have been called before calling this function.

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_GENERICCODEC_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Remarks

The handle returned is valid until the DRV_GENERICCODEC_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Example

```
DRV_HANDLE handle;

handle = DRV_GENERICCODEC_Open(DRV_GENERICCODEC_INDEX_0, DRV_IO_INTENT_WRITEREAD |
DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

C

```
DRV_HANDLE DRV_GENERICCODEC_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

DRV_GENERICCODEC_Close Function

void DRV_GENERICCODEC_Close(DRV_Handle handle)

Summary

Closes an opened-instance of the Generic Codec driver

Description

This routine closes an opened-instance of the Generic Codec driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_GENERICCODEC_Open before the caller may use the driver again

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

- None

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Example

```
DRV_HANDLE handle;  // Returned from DRV_GENERICCODEC_Open

DRV_GENERICCODEC_Close(handle);
```

C

```
void DRV_GENERICCODEC_Close(const DRV_HANDLE handle);
```

DRV_GENERICCODEC_BufferEventHandlerSet Function

void DRV_GENERICCODEC_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_GENERICCODEC_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_GENERICCODEC_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWrite(myGENERICCODEChandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;
```

```
            default:
                break;
        }
    }
```

C

```
void DRV_GENERICCODEC_BufferEventHandlerSet(DRV_HANDLE handle, const
DRV_GENERICCODEC_BUFFER_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

DRV_GENERICCODEC_CommandEventHandlerSet Function

void DRV_GENERICCODEC_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_GENERICCODEC_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

The event handler should be set before the client performs any "Generic Codec Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.
```

```
// Client registers an event handler with driver

DRV_GENERICCODEC_CommandEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECCommandEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_DeEmphasisFilterSet(myGENERICCODECHandle,
DRV_GENERICCODEC_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECCommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

C

```
void DRV_GENERICCODEC_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_GENERICCODEC_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Data Transfer Functions

DRV_GENERICCODEC_BufferAddRead Function

void DRV_GENERICCODEC_BufferAddRead

(

const DRV_HANDLE handle,

DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

Summary

Schedule a non-blocking driver read operation.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_GENERICCODEC_Open call.

Parameters

Parameters	Description
handle	Handle of the Generic Codec instance as return by the DRV_GENERICCODEC_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

C

```
void DRV_GENERICCODEC_BufferAddRead(const DRV_HANDLE handle, DRV_GENERICCODEC_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

DRV_GENERICCODEC_BufferAddWrite Function

void DRV_GENERICCODEC_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

Summary

Schedule a non-blocking driver write operation.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_GENERICCODEC_Open call.

Parameters

Parameters	Description
handle	Handle of the Generic Codec instance as return by the DRV_GENERICCODEC_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(myGENERICCODECHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWrite(myGENERICCODEChandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_GENERICCODEC_BufferAddWrite(const DRV_HANDLE handle, DRV_GENERICCODEC_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

DRV_GENERICCODEC_BufferAddWriteRead Function

void DRV_GENERICCODEC_BufferAddWriteRead
(
const DRV_HANDLE handle,
DRV_GENERICCODEC_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)

Summary

Schedule a non-blocking driver write-read operation.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_GENERICCODEC_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec device instance and the DRV_GENERICCODEC_Status must have returned SYS_STATUS_READY.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_GENERICCODEC_Open call.

Parameters

Parameters	Description
handle	Handle of the Generic Codec instance as returned by the DRV_GENERICCODEC_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_GENERICCODEC_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the Generic Codec Driver Buffer Event Handler that

is registered by this client. It should not be called in the event handler associated with another Generic Codec driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every Generic Codec write. The transmit and receive size must be same.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle;

// mygenericcodecHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

// Client registers an event handler with driver

DRV_GENERICCODEC_BufferEventHandlerSet(mygenericcodecHandle,
                DRV_GENERICCODECBufferEventHandler, (uintptr_t)&myAppObj);

DRV_GENERICCODEC_BufferAddWriteRead(mygenericcodecHandle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_GENERICCODEC_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void DRV_GENERICCODECBufferEventHandler(DRV_GENERICCODEC_BUFFER_EVENT event,
        DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_GENERICCODEC_BufferAddWriteRead(const DRV_HANDLE handle,
DRV_GENERICCODEC_BUFFER_HANDLE * bufferHandle, void * transmitBuffer, void * receiveBuffer,
size_t size);
```

DRV_GENERICCODEC_ReadQueuePurge Function

bool DRV_GENERICCODEC_ReadQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the read queue.

Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_GENERICCODEC_Open function.

Returns

True - Read queue purge is successful. False - Read queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_GENERICCODEC_Open function.
// Use DRV_GENERICCODEC_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_GENERICCODEC_ReadQueuePurge(myCodecHandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_GENERICCODEC_ReadQueuePurge(const DRV_HANDLE handle);
```

DRV_GENERICCODEC_WriteQueuePurge Function

bool DRV_GENERICCODEC_WriteQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the write queue.

Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_GENERICCODEC_Open function.

Returns

True - Write queue purge is successful. False - Write queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_GENERICCODEC_Open function.
// Use DRV_GENERICCODEC_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_GENERICCODEC_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_GENERICCODEC_WriteQueuePurge(const DRV_HANDLE handle);
```

Settings Functions

DRV_GENERICCODEC_MicGainGet Function

uint8_t DRV_GENERICCODEC_MicGainGet(DRV_HANDLE handle)

Summary

This function gets the microphone gain for the Generic Codec.

Description

This functions gets the current microphone gain programmed to the Generic Codec.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

Microphone gain, in range 0-31.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
```

```
uint8_t gain;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

  gain = DRV_GENERICCODEC_MicGainGet(myGENERICCODECHandle);
```

C

```
uint8_t DRV_GENERICCODEC_MicGainGet(DRV_HANDLE handle);
```

DRV_GENERICCODEC_MicGainSet Function

void DRV_GENERICCODEC_MicGainSet(DRV_HANDLE handle, uint8_t gain)

Summary

This function sets the microphone gain for the Generic Codec CODEC.

Description

This functions sets the microphone gain value from 0-31

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
gain	Gain value, in range 0-31

Returns

None.

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MicGainSet(myGENERICCODECHandle, 15);  //GENERICCODEC mic gain set to 15
```

Remarks

None.

C

```
void DRV_GENERICCODEC_MicGainSet(DRV_HANDLE handle, uint8_t gain);
```

DRV_GENERICCODEC_MicMuteOff Function

void DRV_GENERICCODEC_MicMuteOff(DRV_HANDLE handle)

Summary

Umutes th Generic Codec's microphone input.

Description

This function unmutes the Generic Codec's microphone input.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MicMuteOff(myGENERICCODECHandle);  //Generic Codec microphone unmuted
```

C

```
void DRV_GENERICCODEC_MicMuteOff(DRV_HANDLE handle);
```

DRV_GENERICCODEC_MicMuteOn Function

void DRV_GENERICCODEC_MicMuteOn(DRV_HANDLE handle);

Summary

Mutes the Generic Codec's microphone input

Description

This function mutes the Generic Codec's microphone input

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_MicMuteOn(myGENERICCODECHandle);   //Generic Codec microphone muted
```

C

```
void DRV_GENERICCODEC_MicMuteOn(DRV_HANDLE handle);
```

DRV_GENERICCODEC_MuteOff Function

void DRV_GENERICCODEC_MuteOff(DRV_HANDLE handle)

Summary

This function disables Generic Codec output for soft mute.

Description

This function disables Generic Codec output for soft mute.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_MuteOff(myGENERICCODECHandle); //Generic Codec output soft mute disabled
```

C

```
void DRV_GENERICCODEC_MuteOff(DRV_HANDLE handle);
```

DRV_GENERICCODEC_MuteOn Function

void DRV_GENERICCODEC_MuteOn(DRV_HANDLE handle);

Summary

This function allows Generic Codec output for soft mute on.

Description

This function Enables Generic Codec output for soft mute.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_MuteOn(myGENERICCODECHandle);  //GENERICCODEC output soft muted
```

C

```
void DRV_GENERICCODEC_MuteOn(DRV_HANDLE handle);
```

DRV_GENERICCODEC_SamplingRateGet Function

uint32_t DRV_GENERICCODEC_SamplingRateGet(DRV_HANDLE handle)

Summary

This function gets the sampling rate set on the Generic Codec.

Description

This function gets the sampling rate set on the DAC Generic Codec.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Remarks

None.

Example

```
uint32_t baudRate;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

baudRate = DRV_GENERICCODEC_SamplingRateGet(myGENERICCODECHandle);
```

C

```
uint32_t DRV_GENERICCODEC_SamplingRateGet(DRV_HANDLE handle);
```

DRV_GENERICCODEC_SamplingRateSet Function

void DRV_GENERICCODEC_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

Summary

This function sets the sampling rate of the media stream.

Description

This function sets the media sampling rate for the client handle.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Returns

None.

Remarks

None.

Example

```
// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_SamplingRateSet(myGENERICCODECHandle, 48000);  //Sets 48000 media sampling rate
```

C

```
void DRV_GENERICCODEC_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

DRV_GENERICCODEC_VersionGet Function

uint32_t DRV_GENERICCODEC_VersionGet(void)

Summary

This function returns the version of Generic Codec driver

Description

The version number returned from the DRV_GENERICCODEC_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Preconditions

None.

Returns

returns the version of Generic Codec driver.

Remarks

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t GENERICCODECversion;
```

```
GENERICCODECversion = DRV_GENERICCODEC_VersionGet();
```

C

```
uint32_t DRV_GENERICCODEC_VersionGet();
```

DRV_GENERICCODEC_VersionStrGet Function

int8_t* DRV_GENERICCODEC_VersionStrGet(void)

Summary

This function returns the version of Generic Codec driver in string format.

Description

The DRV_GENERICCODEC_VersionStrGet function returns a string in the format: ".[.][]" Where: is the Generic Codec driver's version number. is the Generic Codec driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Preconditions

None.

Returns

returns a string containing the version of Generic Codec driver.

Remarks

None

Example

```
int8_t *GENERICCODECstring;
GENERICCODECstring = DRV_GENERICCODEC_VersionStrGet();
```

C

```
int8_t* DRV_GENERICCODEC_VersionStrGet();
```

DRV_GENERICCODEC_VolumeGet Function

uint8_t DRV_GENERICCODEC_VolumeGet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel)

Summary

This function gets the volume for Generic Codec.

Description

This functions gets the current volume programmed to the Generic Codec.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

  volume = DRV_GENERICCODEC_VolumeGet(myGENERICCODECHandle, DRV_GENERICCODEC_CHANNEL_LEFT);
```

C

```
uint8_t DRV_GENERICCODEC_VolumeGet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel);
```

DRV_GENERICCODEC_VolumeSet Function

void DRV_GENERICCODEC_VolumeSet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel, uint8_t volume);

Summary

This function sets the volume for Generic Codec.

Description

This functions sets the volume value from 0-255.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Returns

None

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

    DRV_GENERICCODEC_VolumeSet(myGENERICCODECHandle,DRV_GENERICCODEC_CHANNEL_LEFT, 120);
```

C

```
void DRV_GENERICCODEC_VolumeSet(DRV_HANDLE handle, DRV_GENERICCODEC_CHANNEL channel, uint8_t volume);
```

Other Functions

DRV_GENERICCODEC_GetI2SDriver Function

DRV_HANDLE DRV_GENERICCODEC_GetI2SDriver(DRV_HANDLE codecHandle)

Summary

Get the handle to the I2S driver for this codec instance.

Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

Preconditions

The DRV_GENERICCODEC_Initialize routine must have been called for the specified Generic Codec driver instance.

DRV_GENERICCODEC_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

- A handle to the I2S driver for this codec instance

Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

C

```
DRV_HANDLE DRV_GENERICCODEC_GetI2SDriver(DRV_HANDLE codecHandle);
```

DRV_GENERICCODEC_LRCLK_Sync Function

uint32_t DRV_GENERICCODEC_LRCLK_Sync (const DRV_HANDLE handle);

Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

Preconditions

None.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
// myGENERICCODECHandle is the handle returned
// by the DRV_GENERICCODEC_Open function.

DRV_GENERICCODEC_LRCLK_Sync(myGENERICCODECHandle);
```

C

```
bool DRV_GENERICCODEC_LRCLK_Sync(const DRV_HANDLE handle);
```

Dara Types and Constants

DRV_GENERICCODEC_BUFFER_EVENT Type

Identifies the possible events that can result from a buffer add request.

Description

Generic Codec Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_GENERICCODEC_BufferAddWrite() or the DRV_GENERICCODEC_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_GENERICCODEC_BufferEventHandlerSet function when a buffer transfer request is completed.

C

```
typedef enum DRV_GENERICCODEC_BUFFER_EVENT@1 DRV_GENERICCODEC_BUFFER_EVENT;
```

DRV_GENERICCODEC_BUFFER_EVENT_HANDLER Type

Pointer to a Generic Codec Driver Buffer Event handler function

Description

Generic Codec Driver Buffer Event Handler Function

This data type defines the required function signature for the Generic Codec driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

If the event is DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_GENERICCODEC_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_GENERICCODEC_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_GENERICCODEC_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_GENERICCODEC_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void DRV_GENERICCODECBufferEventHandler( DRV_GENERICCODEC_BUFFER_EVENT event,
                                         DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle,
                                         uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_GENERICCODEC_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_GENERICCODEC_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

C

```
typedef void (* DRV_GENERICCODEC_BUFFER_EVENT_HANDLER)(DRV_GENERICCODEC_BUFFER_EVENT event,
DRV_GENERICCODEC_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

DRV_GENERICCODEC_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

Description

Generic Codec Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_GENERICCODEC_BufferAddWrite() or DRV_GENERICCODEC_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

C

```
typedef uintptr_t DRV_GENERICCODEC_BUFFER_HANDLE;
```

DRV_GENERICCODEC_CHANNEL Type

Identifies Left/Right Audio channel

Description

Generic Codec Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

C

```
typedef enum DRV_GENERICCODEC_CHANNEL@1 DRV_GENERICCODEC_CHANNEL;
```

DRV_GENERICCODEC_COMMAND_EVENT_HANDLER Type

Pointer to a Generic Codec Driver Command Event Handler Function

Description

Generic Codec Driver Command Event Handler Function

This data type defines the required function signature for the Generic Codec driver command event handling callback function.

A command is a control instruction to the Generic Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_GENERICCODEC_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void DRV_GENERICCODECCommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;
```

```
    // Last Submitted command is completed.
    // Perform further processing here
}
```

C

```
typedef void (* DRV_GENERICCODEC_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

DRV_GENERICCODEC_INIT Type

Defines the data required to initialize or reinitialize the Generic Codec driver

Description

Generic Codec Driver Initialization Data

This data type defines the data required to initialize or reinitialize the Generic Codec driver.

Remarks

None.

C

```
typedef struct DRV_GENERICCODEC_INIT@1 DRV_GENERICCODEC_INIT;
```

DRV_GENERICCODEC_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

Description

Generic Codec Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_GENERICCODEC_BufferAddWrite() and the DRV_GENERICCODEC_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

C

```
#define DRV_GENERICCODEC_BUFFER_HANDLE_INVALID ((DRV_GENERICCODEC_BUFFER_HANDLE)(-1))
```

DRV_GENERICCODEC_COUNT Macro

Number of valid Generic Codec driver indices

Description

Generic Codec Driver Module Count

This constant identifies the maximum number of Generic Codec Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of Generic Codec instances on this microcontroller.

Remarks

This value is part-specific.

C

```
#define DRV_GENERICCODEC_COUNT
```

DRV_GENERICCODEC_INDEX_0 Macro

Generic Codec driver index definitions

Description

Driver Generic Codec Module Index

These constants provide Generic Codec driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_GENERICCODEC_Initialize and DRV_GENERICCODEC_Open routines to identify the driver instance in use.

C

```
#define DRV_GENERICCODEC_INDEX_0 0
```

DRV_GENERICCODEC_INDEX_1 Macro

This is macro DRV_GENERICCODEC_INDEX_1.

C

```
#define DRV_GENERICCODEC_INDEX_1 1
```

DRV_GENERICCODEC_INDEX_2 Macro

This is macro DRV_GENERICCODEC_INDEX_2.

C

```
#define DRV_GENERICCODEC_INDEX_2 2
```

DRV_GENERICCODEC_INDEX_3 Macro

This is macro DRV_GENERICCODEC_INDEX_3.

C

```
#define DRV_GENERICCODEC_INDEX_3 3
```

DRV_GENERICCODEC_INDEX_4 Macro

This is macro DRV_GENERICCODEC_INDEX_4.

C

```
#define DRV_GENERICCODEC_INDEX_4 4
```

DRV_GENERICCODEC_INDEX_5 Macro

This is macro DRV_GENERICCODEC_INDEX_5.

C

```
#define DRV_GENERICCODEC_INDEX_5 5
```

DRV_GENERICCODEC_AUDIO_DATA_FORMAT Type

Identifies the Serial Audio data interface format.

Description

Generic Codec Audio data format

This enumeration identifies Serial Audio data interface format.

C

```
typedef enum DRV_GENERICCODEC_AUDIO_DATA_FORMAT@1 DRV_GENERICCODEC_AUDIO_DATA_FORMAT;
```

Files

Files

Name	Description
drv_genericcodec.h	Generic Codec Driver Interface header file
drv_genericcodec_config_template.h	Generic Codec Driver Configuration Template.

Description

drv_genericcodec.h

drv_genericcodec.h

Summary

Generic Codec Driver Interface header file

Description

Generic Codec Driver Interface

The Generic Codec device driver interface provides a simple interface to manage a codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the Generic Codec device driver.

drv_genericcodec_config_template.h

drv_genericcodec_config_template.h

Summary

Generic Codec Driver Configuration Template.

Description

Generic Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

WM8904 CODEC Driver Library Help

This topic describes the WM8904 Codec Driver Library.

Introduction

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

Description

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

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

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

The WM8904 Codec supports the following features:

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

Using the Library

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

Description

Interface Header File: `drv_WM8904.h`

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

Library Source Files:

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

Example Applications:

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

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

- `audio/apps/microphone_loopback`

Abstraction Model

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

Description

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

WM8904 Driver Abstraction Model

Library Overview

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

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

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions, such as Buffer Read and Write.

Settings Functions	Provides driver specific functions for settings, such as volume control and sampling rate.
Other Functions	Miscellaneous functions, such as getting the driver's version number and syncing to the LRCLK signal.
Data Types and Constants	These data types and constants are required while interacting and setting up the WM8904 Codec Driver Library.

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

How the Library Works

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Setup (Initialization)

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

Description

System Initialization

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

- Device requested power state: one of the System Module Power States. For specific details please refer to Data Types and Constants in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

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

Client Access

This topic describes driver initialization and provides a code example.

Description

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

set up the instance of the driver.

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

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

Example:

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

Client Operations

This topic provides information on client operations.

Description

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

The following WM8904 Codec specific control command functions are provided:

- DRV_WM8904_SamplingRateSet
- DRV_WM8904_SamplingRateGet
- DRV_WM8904_VolumeSet
- DRV_WM8904_VolumeGet
- DRV_WM8904_MuteOn
- DRV_WM8904_MuteOff

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

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

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

Configuring the Library

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

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

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

System Configuration

Macros

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

Description

DRV_WM8904_AUDIO_DATA_FORMAT_MACRO Macro

Specifies the audio data format for the codec.

Description

WM8904 Audio Data Format

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

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

Remarks

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

C

```
#define DRV_WM8904_AUDIO_DATA_FORMAT_MACRO
```

DRV_WM8904_AUDIO_SAMPLING_RATE Macro

Specifies the initial baud rate for the codec.

Description

WM8904 Baud Rate

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

C

```
#define DRV_WM8904_AUDIO_SAMPLING_RATE
```

DRV_WM8904_CLIENTS_NUMBER Macro

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

Description

WM8904 Client Count Configuration

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

C

```
#define DRV_WM8904_CLIENTS_NUMBER
```

DRV_WM8904_ENABLE_MIC_BIAS Macro

Specifies whether to enable the microphone bias.

Description

WM8904 Microphone Enable

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

C

```
#define DRV_WM8904_ENABLE_MIC_BIAS
```

DRV_WM8904_ENABLE_MIC_INPUT Macro

Specifies whether to enable the microphone input.

Description

WM8904 Microphone Enable

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

C

```
#define DRV_WM8904_ENABLE_MIC_INPUT
```

DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2C interface.

Description

WM8904 I2C instance number

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

C

```
#define DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx
```

DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx Macro

Specifies the instance number of the I2S interface.

Description

WM8904 I2S instance number

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

C

```
#define DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx
```

DRV_WM8904_INSTANCES_NUMBER Macro

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

Description

WM8904 driver objects configuration

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

C

```
#define DRV_WM8904_INSTANCES_NUMBER
```

DRV_WM8904_MASTER_MODE Macro

Specifies if codec is in Master or Slave mode.

Description

WM8904 Codec Master/Slave Mode

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

C

```
#define DRV_WM8904_MASTER_MODE
```

DRV_WM8904_VOLUME Macro

Specifies the initial volume level.

Description

WM8904 Volume

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

Remarks

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

C

```
#define DRV_WM8904_VOLUME
```

Configuring MHC

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

Description

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

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

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

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

- **I2C Driver Used** will display the driver instance used for the I^2C interface.
- **I2S Driver Used** will display the driver instance used for the I^2S interface.
- **Usage Mode** indicates whether the WM8904 is a Master (supplies I^2S clocks) or a Slave (MCU supplies I^2S clocks).
- **Number of WM8904 Clients** indicates the maximum number of clients that can be connected to the WM8904 Driver.
- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Volume** indicates the volume in a linear scale from 0-255.
- **Audio Data Format** is either 16-bit Left Justified, 16-bit I2S, 32-bit Left Justified, or 32-bit I2S. It must match the audio protocol

and data length set up in either the SSC or I2S PLIB.

- **Sampling Rate** indicates the number of samples per second per channel, 8000 to 96,000.
- **Enable Microphone Input** should be checked a microphone is being used. If checked, another option,
- **Enable Microphone Bias** should be checked if using an electret microphone.

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

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

Building the Library

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

Description

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

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

Interface File(s)

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

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

Required File(s)

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

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

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

Optional File(s)

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

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

Module Dependencies

The WM8904 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

Client Setup Functions

	Name	Description
	DRV_WM8904_Open	Opens the specified WM8904 driver instance and returns a handle to it

	DRV_WM8904_Close	Closes an opened-instance of the WM8904 driver
	DRV_WM8904_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_WM8904_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Data Transfer Functions

	Name	Description
	DRV_WM8904_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_WM8904_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_WM8904_BufferAddWriteRead	Schedule a non-blocking driver write-read operation.
	DRV_WM8904_ReadQueuePurge	Removes all buffer requests from the read queue.
	DRV_WM8904_WriteQueuePurge	Removes all buffer requests from the write queue.

Data Types and Constants

	Name	Description
	DATA_LENGTH	in bits
	DRV_WM8904_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_WM8904_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_WM8904_BUFFER_EVENT_HANDLER	Pointer to a WM8904 Driver Buffer Event handler function
	DRV_WM8904_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_WM8904_CHANNEL	Identifies Left/Right Audio channel
	DRV_WM8904_COMMAND_EVENT_HANDLER	Pointer to a WM8904 Driver Command Event Handler Function
	DRV_WM8904_INIT	Defines the data required to initialize or reinitialize the WM8904 driver
	DRV_WM8904_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_WM8904_COUNT	Number of valid WM8904 driver indices
	DRV_WM8904_INDEX_0	WM8904 driver index definitions
	DRV_I2C_INDEX	This is macro DRV_I2C_INDEX.

Other Functions

	Name	Description
	DRV_WM8904_GetI2SDriver	Get the handle to the I2S driver for this codec instance.
	DRV_WM8904_VersionGet	This function returns the version of WM8904 driver
	DRV_WM8904_VersionStrGet	This function returns the version of WM8904 driver in string format.
	DRV_WM8904_LRCLK_Sync	Synchronize to the start of the I2S LRCLK (left/right clock) signal

Settings Functions

	Name	Description
	DRV_WM8904_MuteOff	This function disables WM8904 output for soft mute.
	DRV_WM8904_MuteOn	This function allows WM8904 output for soft mute on.
	DRV_WM8904_SamplingRateGet	This function gets the sampling rate set on the WM8904.
	DRV_WM8904_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_WM8904_VolumeGet	This function gets the volume for WM8904 Codec.
	DRV_WM8904_VolumeSet	This function sets the volume for WM8904 Codec.

System Interaction Functions

	Name	Description
	DRV_WM8904_Initialize	Initializes hardware and data for the instance of the WM8904 DAC module
	DRV_WM8904_Deinitialize	Deinitializes the specified instance of the WM8904 driver module

	DRV_WM8904_Status	Gets the current status of the WM8904 driver module.
	DRV_WM8904_Tasks	Maintains the driver's control and data interface state machine.

System Interaction Functions

DRV_WM8904_Initialize Function

```
SYS_MODULE_OBJ  DRV_WM8904_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);
```

Summary

Initializes hardware and data for the instance of the WM8904 DAC module

Description

This routine initializes the WM8904 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this Codec driver. DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this Codec driver.

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Remarks

This routine must be called before any other WM8904 routine is called.

This routine should only be called once during system initialization unless DRV_WM8904_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Example

```
DRV_WM8904_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = wm8904Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = wm8904Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_WM8904_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_WM8904_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_WM8904_COMMAND_EVENT_HANDLER)0;
```

```
init->commandContextData = 0;
init->mclk_multiplier = DRV_WM8904_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_WM8904_Initialize(DRV_WM8904_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

C

```
SYS_MODULE_OBJ DRV_WM8904_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

DRV_WM8904_Deinitialize Function

void DRV_WM8904_Deinitialize(SYS_MODULE_OBJ object)

Summary

Deinitializes the specified instance of the WM8904 driver module

Description

Deinitializes the specified instance of the WM8904 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_WM8904_Initialize routine

Returns

None.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_WM8904_Initialize
SYS_STATUS          status;


DRV_WM8904_Deinitialize(object);

status = DRV_WM8904_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

C

```
void DRV_WM8904_Deinitialize(SYS_MODULE_OBJ object);
```

DRV_WM8904_Status Function

SYS_STATUS DRV_WM8904_Status(SYS_MODULE_OBJ object)

Summary

Gets the current status of the WM8904 driver module.

Description

This routine provides the current status of the WM8904 driver module.

Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_WM8904_Initialize routine

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize
SYS_STATUS          WM8904Status;

WM8904Status = DRV_WM8904_Status(object);
if (SYS_STATUS_READY == WM8904Status)
{
    // This means the driver can be opened using the
    // DRV_WM8904_Open() function.
}
```

C

```
SYS_STATUS DRV_WM8904_Status(SYS_MODULE_OBJ object);
```

DRV_WM8904_Tasks Function

void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);

Summary

Maintains the driver's control and data interface state machine.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_WM8904_Initialize)

Returns

None.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize

while (true)
{
    DRV_WM8904_Tasks (object);

    // Do other tasks
}
```

C

```
void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);
```

Client Setup Functions

DRV_WM8904_Open Function

DRV_HANDLE DRV_WM8904_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

Summary

Opens the specified WM8904 driver instance and returns a handle to it

Description

This routine opens the specified WM8904 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

The WM8904 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Preconditions

Function DRV_WM8904_Initialize must have been called before calling this function.

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur:

- if the number of client objects allocated via DRV_WM8904_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Remarks

The handle returned is valid until the DRV_WM8904_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Example

```
DRV_HANDLE handle;

handle = DRV_WM8904_Open(DRV_WM8904_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

C

```
DRV_HANDLE DRV_WM8904_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

DRV_WM8904_Close Function

void DRV_WM8904_Close(DRV_Handle handle)

Summary

Closes an opened-instance of the WM8904 driver

Description

This routine closes an opened-instance of the WM8904 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_WM8904_Open before the caller may use the driver again

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

- None

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Example

```
DRV_HANDLE handle;  // Returned from DRV_WM8904_Open

DRV_WM8904_Close(handle);
```

C

```c
void DRV_WM8904_Close(const DRV_HANDLE handle);
```

DRV_WM8904_BufferEventHandlerSet Function

void DRV_WM8904_BufferEventHandlerSet
(
DRV_HANDLE handle,
const DRV_WM8904_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)

Summary

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Description

When a client calls DRV_WM8904_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_WM8904_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_WM8904_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

DRV_WM8904_CommandEventHandlerSet Function

void DRV_WM8904_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_WM8904_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

Summary

This function allows a client to identify a command event handling function for the driver to call back when the last submitted

command have finished.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

The event handler should be set before the client performs any "WM8904 Codec Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_CommandEventHandlerSet(myWM8904Handle,
                APP_WM8904CommandEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_DeEmphasisFilterSet(myWM8904Handle, DRV_WM8904_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_WM8904CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

C

```
void DRV_WM8904_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_WM8904_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Data Transfer Functions

DRV_WM8904_BufferAddRead Function

```
void DRV_WM8904_BufferAddRead
(
const DRV_HANDLE handle,
    DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)
```

Summary

Schedule a non-blocking driver read operation.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_WM8904_Open call.

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as return by the DRV_WM8904_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

C

```
void DRV_WM8904_BufferAddRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE * bufferHandle,
```

```
void * buffer, size_t size);
```

DRV_WM8904_BufferAddWrite Function

void DRV_WM8904_BufferAddWrite
(
const DRV_HANDLE handle,
    DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

Summary

Schedule a non-blocking driver write operation.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_WM8904_Open call.

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as return by the DRV_WM8904_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
```

```
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_WM8904_BufferAddWrite(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

DRV_WM8904_BufferAddWriteRead Function

void DRV_WM8904_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_WM8904_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

Summary

Schedule a non-blocking driver write-read operation.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and

returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_WM8904_Open call.

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as returned by the DRV_WM8904_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every WM8904 write. The transmit and receive size must be same.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// mywm8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(mywm8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWriteRead(mywm8904handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.
```

```
void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_WM8904_BufferAddWriteRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

DRV_WM8904_ReadQueuePurge Function

bool DRV_WM8904_ReadQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the read queue.

Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_WM8904_Open function.

Returns

True - Read queue purge is successful. False - Read queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_WM8904_Open function.
// Use DRV_WM8904_BufferAddRead to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_WM8904_ReadQueuePurge(myCodecHandle))
    {
```

```
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_WM8904_ReadQueuePurge(const DRV_HANDLE handle);
```

DRV_WM8904_WriteQueuePurge Function

bool DRV_WM8904_WriteQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the write queue.

Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_WM8904_Open function.

Returns

True - Write queue purge is successful. False - Write queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myCodecHandle is the handle returned by the DRV_WM8904_Open function.
// Use DRV_WM8904_BufferAddWrite to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_WM8904_WriteQueuePurge(myCodecHandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_WM8904_WriteQueuePurge(const DRV_HANDLE handle);
```

Settings Functions

DRV_WM8904_MuteOff Function

void DRV_WM8904_MuteOff(DRV_HANDLE handle)

Summary

This function disables WM8904 output for soft mute.

Description

This function disables WM8904 output for soft mute.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

    DRV_WM8904_MuteOff(myWM8904Handle); //WM8904 output soft mute disabled
```

C

```
void DRV_WM8904_MuteOff(DRV_HANDLE handle);
```

DRV_WM8904_MuteOn Function

void DRV_WM8904_MuteOn(DRV_HANDLE handle);

Summary

This function allows WM8904 output for soft mute on.

Description

This function Enables WM8904 output for soft mute.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_MuteOn(myWM8904Handle);  //WM8904 output soft muted
```

C

```
void DRV_WM8904_MuteOn(DRV_HANDLE handle);
```

DRV_WM8904_SamplingRateGet Function

uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle)

Summary

This function gets the sampling rate set on the WM8904.

Description

This function gets the sampling rate set on the DAC WM8904.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Remarks

None.

Example

```
uint32_t baudRate;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

baudRate = DRV_WM8904_SamplingRateGet(myWM8904Handle);
```

C

```
uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle);
```

DRV_WM8904_SamplingRateSet Function

void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

Summary

This function sets the sampling rate of the media stream.

Description

This function sets the media sampling rate for the client handle.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Returns

None.

Remarks

None.

Example

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_SamplingRateSet(myWM8904Handle, 48000);  //Sets 48000 media sampling rate
```

C

```
void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

DRV_WM8904_VolumeGet Function

uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel)

Summary

This function gets the volume for WM8904 Codec.

Description

This functions gets the current volume programmed to the Codec WM8904.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Returns

None.

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

  volume = DRV_WM8904_VolumeGet(myWM8904Handle, DRV_WM8904_CHANNEL_LEFT);
```

C

```
uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel);
```

DRV_WM8904_VolumeSet Function

void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);

Summary

This function sets the volume for WM8904 Codec.

Description

This function sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Returns

None

Remarks

None.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

    DRV_WM8904_VolumeSet(myWM8904Handle,DRV_WM8904_CHANNEL_LEFT, 120);
```

C

```
void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);
```

Other Functions

DRV_WM8904_GetI2SDriver Function

DRV_HANDLE DRV_WM8904_GetI2SDriver(DRV_HANDLE codecHandle)

Summary

Get the handle to the I2S driver for this codec instance.

Description

Returns the appropriate handle to the I2S based on the ioIent member of the codec object.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

A handle to the I2S driver for this codec instance

Remarks

This allows the caller to directly access portions of the I2S driver that might not be available via the codec API.

C

```
DRV_HANDLE DRV_WM8904_GetI2SDriver(DRV_HANDLE codecHandle);
```

DRV_WM8904_VersionGet Function

uint32_t DRV_WM8904_VersionGet(void)

Summary

This function returns the version of WM8904 driver

Description

The version number returned from the DRV_WM8904_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Preconditions

None.

Returns

returns the version of WM8904 driver.

Remarks

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t WM8904version;
WM8904version = DRV_WM8904_VersionGet();
```

C

```
uint32_t DRV_WM8904_VersionGet();
```

DRV_WM8904_VersionStrGet Function

int8_t* DRV_WM8904_VersionStrGet(void)

Summary

This function returns the version of WM8904 driver in string format.

Description

The DRV_WM8904_VersionStrGet function returns a string in the format: ".[.][]" Where: is the WM8904 driver's version number. is the WM8904 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Preconditions

None.

Returns

returns a string containing the version of WM8904 driver.

Remarks

None

Example

```
int8_t *WM8904string;
WM8904string = DRV_WM8904_VersionStrGet();
```

C

```
int8_t* DRV_WM8904_VersionStrGet();
```

DRV_WM8904_LRCLK_Sync Function

uint32_t DRV_WM8904_LRCLK_Sync (const DRV_HANDLE handle);

Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

Description

This function waits until low-to high transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Justified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

Preconditions

None.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_LRCLK_Sync(myWM8904Handle);
```

C

```
bool DRV_WM8904_LRCLK_Sync(const DRV_HANDLE handle);
```

Data Types and Constants

DATA_LENGTH Type

in bits

C

```
typedef enum DATA_LENGTH@4 DATA_LENGTH;
```

DRV_WM8904_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

Description

WM8904 Audio data format

This enumeration identifies Serial Audio data interface format.

C

```
typedef enum {
  DATA_16_BIT_LEFT_JUSTIFIED,
  DATA_16_BIT_I2S,
  DATA_32_BIT_LEFT_JUSTIFIED,
  DATA_32_BIT_I2S
} DRV_WM8904_AUDIO_DATA_FORMAT;
```

DRV_WM8904_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

Description

WM8904 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_WM8904_BufferAddWrite() or the DRV_WM8904_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_WM8904_BufferEventHandlerSet function when a buffer transfer request is completed.

C

```
typedef enum {
  DRV_WM8904_BUFFER_EVENT_COMPLETE,
  DRV_WM8904_BUFFER_EVENT_ERROR,
  DRV_WM8904_BUFFER_EVENT_ABORT
} DRV_WM8904_BUFFER_EVENT;
```

DRV_WM8904_BUFFER_EVENT_HANDLER Type

Pointer to a WM8904 Driver Buffer Event handler function

Description

WM8904 Driver Buffer Event Handler Function

This data type defines the required function signature for the WM8904 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

If the event is DRV_WM8904_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_WM8904_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_WM8904_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_WM8904_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_WM8904_BUFFER_EVENT event,
                               DRV_WM8904_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
```

```
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

C

```
typedef void (* DRV_WM8904_BUFFER_EVENT_HANDLER)(DRV_WM8904_BUFFER_EVENT event,
DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

DRV_WM8904_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

Description

WM8904 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_WM8904_BufferAddWrite() or DRV_WM8904_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

C

```
typedef uintptr_t DRV_WM8904_BUFFER_HANDLE;
```

DRV_WM8904_CHANNEL Enumeration

Identifies Left/Right Audio channel

Description

WM8904 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

C

```
typedef enum {
  DRV_WM8904_CHANNEL_LEFT,
  DRV_WM8904_CHANNEL_RIGHT,
  DRV_WM8904_CHANNEL_LEFT_RIGHT,
  DRV_WM8904_NUMBER_OF_CHANNELS
} DRV_WM8904_CHANNEL;
```

DRV_WM8904_COMMAND_EVENT_HANDLER Type

Pointer to a WM8904 Driver Command Event Handler Function

Description

WM8904 Driver Command Event Handler Function

This data type defines the required function signature for the WM8904 driver command event handling callback function.

A command is a control instruction to the WM8904 Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_WM8904CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

C

```
typedef void (* DRV_WM8904_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

DRV_WM8904_INIT Structure

Defines the data required to initialize or reinitialize the WM8904 driver

Description

WM8904 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the WM8904 Codec driver.

Remarks

None.

C

```
typedef struct {
```

```
    SYS_MODULE_INIT moduleInit;
    SYS_MODULE_INDEX i2sDriverModuleIndex;
    SYS_MODULE_INDEX i2cDriverModuleIndex;
    bool masterMode;
    uint32_t samplingRate;
    uint8_t volume;
    DRV_WM8904_AUDIO_DATA_FORMAT audioDataFormat;
    bool enableMicInput;
    bool enableMicBias;
  } DRV_WM8904_INIT;
```

DRV_WM8904_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

Description

WM8904 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_WM8904_BufferAddWrite() and the DRV_WM8904_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

C

```
#define DRV_WM8904_BUFFER_HANDLE_INVALID ((DRV_WM8904_BUFFER_HANDLE)(-1))
```

DRV_WM8904_COUNT Macro

Number of valid WM8904 driver indices

Description

WM8904 Driver Module Count

This constant identifies the maximum number of WM8904 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of WM8904 instances on this microcontroller.

Remarks

This value is part-specific.

C

```
#define DRV_WM8904_COUNT
```

DRV_WM8904_INDEX_0 Macro

WM8904 driver index definitions

Description

Driver WM8904 Module Index

These constants provide WM8904 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_WM8904_Initialize and DRV_WM8904_Open routines to identify the driver instance in use.

C

```
#define DRV_WM8904_INDEX_0 0
```

DRV_I2C_INDEX Macro

This is macro DRV_I2C_INDEX.

C

```
#define DRV_I2C_INDEX DRV_WM8904_I2C_INSTANCES_NUMBER
```

Files

Files

Name	Description
drv_wm8904.h	WM8904 Codec Driver Interface header file
drv_wm8904_config_template.h	WM8904 Codec Driver Configuration Template.

Description

This section will list only the library's interface header file(s).

drv_wm8904.h

drv_wm8904.h

Summary

WM8904 Codec Driver Interface header file

Description

WM8904 Codec Driver Interface

The WM8904 Codec device driver interface provides a simple interface to manage the WM8904 16/24/32-Bit Codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the WM8904 Codec device driver.

drv_wm8904_config_template.h

drv_wm8904_config_template.h

Summary

WM8904 Codec Driver Configuration Template.

Description

WM8904 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

I2S Driver Library Help

This section describes the I2S Driver Library.

Introduction

This library provides an interface to manage the I2S Audio Protocol Interface Modes.

Description

The I2S Driver is connected to a hardware module that provides the actual I2S stream, on some MCUs this is a Serial Peripheral Interface (SPI), on others it may be an I2S Controller (I2SC), or Serial Synchronous Controller (SSC).

The I2S hardware peripheral is then interfaced to various devices such as codecs and Bluetooth modules to provide microcontroller-based audio solutions.

Using the Library

This topic describes the basic architecture of the I2S Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_i2s.h`

The interface to the I2S Driver Library is defined in the `drv_i2s.h` header file. Any C language source (`.c`) file that uses the I2S Driver Library should include `drv_i2s.h`.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

Abstraction Model

The I2S Driver provides a high level abstraction of the lower level (SPI/I2SC/SSC) I2S modules with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2S Driver Library interface.

Description

Different types of I2S capable PLIBs are available on various Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The I2S Driver Library abstracts out these differences and provides a unified model for audio data transfer across different types of I2S modules.

Both the transmitter and receiver provide a buffer in the driver, which transmits and receives data to/from the hardware. The I2S Driver Library provides a set of interfaces to perform the read and the write. The following diagrams illustrate the abstraction model used by the I2S Driver Library. The I2SC Peripheral is used as an example of an I2S-capable PLIB.

I2S Driver Abstraction Model

The PLIBs currently provided, such as SSC and I2SC, only support an interrupt/DMA mode of operation. Polled mode of operation is not supported.

Library Overview

Refer to the Driver Library Overwiew section for information on how the driver operates in a system.

The I2S driver library provides an API interface to transfer/receive digital audio data using supported Audio protocols. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2S Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides device initialization and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions.
Miscellaneous Functions	Provides driver miscellaneous functions such as get error functions, L/R clock sync, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the I2S Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

System Access

This section provides information on system access.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2S module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_I2S_INIT or by using Initialization Overrides) that are supported by the specific I2S device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, DMA TX Channel, DMA RX Channel and Error Interrupt

The DRV_I2S_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_I2S_Deinitialize, DRV_I2S_Status, DRV_I2S_Tasks, and DRV_I2S_TasksError.

Notes:

1. The system initialization setting only effect the instance of the peripheral that is being initialized.
2. Configuration of the dynamic driver for DMA mode(uses DMA channel for data transfer) or Non DMA mode can be performed by appropriately setting the 'dmaChannelTransmit' and 'dmaChannelReceive' variables of the DRV_I2S_INIT structure. For example the TX will be in DMA mode when 'dmaChannelTransmit' is initialized to a valid supported channel number from the enum DMA_CHANNEL. TX will be in Non DMA mode when 'dmaChannelTransmit' is initialized to 'DMA_CHANNEL_NONE'.

Example:

```
DRV_I2S_INIT                  init;
SYS_MODULE_OBJ              objectHandle;

/* I2S Driver Initialization Data */
DRV_I2S_INIT drvI2S0InitData =
{
    .i2sPlib = &drvI2S0PlibAPI,
    .interruptI2S = DRV_I2S_INT_SRC_IDX0,
    .numClients = DRV_I2S_CLIENTS_NUMBER_IDX0,
    .queueSize = DRV_I2S_QUEUE_SIZE_IDX0,
    .dmaChannelTransmit = DRV_I2S_XMIT_DMA_CH_IDX0,
    .dmaChannelReceive  = DRV_I2S_RCV_DMA_CH_IDX0,
    .i2sTransmitAddress = (void *)&(SSC_REGS->SSC_THR),
    .i2sReceiveAddress = (void *)&(SSC_REGS->SSC_RHR),
    .interruptDMA = XDMAC_IRQn,
    .dmaDataLength = DRV_I2S_DATA_LENGTH_IDX0,
};

sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);
```

Task Routine

There is no task routine, since polled mode is not currently supported.

Client Access

This section provides information on general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_I2S_Open function. This provides the settings required to open the I2S instance for operation.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Example:

```
DRV_HANDLE handle;
handle = DRV_I2S_Open(drvObj->i2sDriverModuleIndex,
    (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_NONBLOCKING));
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Client Operations - Buffered

This section provides information on buffered client operations.

Description

Client Operations - Buffered

Client buffered operations provide a the typical audio interface. The functions DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead are the buffered data operation functions. The buffered functions schedules non-blocking operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_I2S_BUFFER_EVENT_COMPLETE, DRV_I2S_BUFFER_EVENT_ERROR or DRV_I2S_BUFFER_EVENT_ABORT events.

The buffer add requests are processed from the I2S channel ISR in interrupt mode.

The following diagram illustrates the buffered data operations

Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. If DMA mode is desired, the DMA should be initialized by calling SYS_DMA_Initialize.
3. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).
4. The driver object should have been initialized by calling DRV_I2S_Initialize. If DMA mode is desired, related attributes in the init structure must be set.
5. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
6. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
7. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
8. The Register and event handler for the client handle should be set up by calling DRV_I2S_BufferEventHandlerSet.
9. Add a buffer to initiate the data transfer by calling DRV_I2S_BufferAddWrite/DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWriteRead.
10. When the DMA Channel has finished, the callback function registered in step 8 will be called.
11. Repeat step 9 through step 10 to handle multiple buffer transmission and reception.
12. When the client is done it can use DRV_I2S_Close to close the client handle.

Example:

```
// The following is an example for interrupt mode buffered transmit

#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj; //I2S module object
DRV_HANDLE handle;  //Client handle
uint32_t i2sClock;   //BCLK frequency
uint32_t baudrate;   //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state; //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);

                /* Add a buffer to write*/
                DRV_I2S_WriteBufferAdd(handle, myAudioBuffer, BUFFER_SIZE,
                    &bufferHandle);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
```

```
                    // Error handling here
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:

                state = APP_STATE_DONE;
            break;


            case  APP_STATE_DONE:
            {
                // Close done
                DRV_I2S_Close(handle);
            }
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
        // Take Action as needed
    }
    else
    {
        // Do nothing
    }
}


void SYS_Tasks ( void )
{
    /* Call the application's tasks routine */
    APP_Tasks ( );
}
```

Configuring the Library

The configuration of the I2S Driver Library is based on the file `configurations.h`.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver

Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

System Configuration

Configurations for driver instances, polled/interrupt mode, etc.

Configuring MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Chose the Configuration name the based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under Audio->*Templates*, double-click on a codec template such as WM8904. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:

Click on the I2S Driver component, Instance 0, and the following menu will be displayed in the Configurations Options:

PLIB Used will display the hardware peripheral instance connected to the I2S Driver, such as SPI0, SSC, or I2SC1.

Number of Clients indicates the maximum number of clients that can be connected to the I2S Driver.

Transfer Queue Size indicates number of buffers, of each transfer queue (transmit/receive).

I2S Data Length is the number of bits for one channel of audio (left or right). It must match the size of the PLIB.

Use DMA for Transmit and Receive should always be checked if using DMA, which is currently the only supported mode.

Use DMA for Transmit should be checked if sending data to a codec or Bluetooth module.

DMA Channel for Transmit indicates the DMA channel # assigned (done automatically when you connect the PLIB).

Use DMA for Receive should be checked if receiving data from a codec or Bluetooth module.However if you are only writing to the I2S stream, leaving this checked won't harm anything.

DMA Channel for Receive indicates the DMA channel # assigned (done automatically when you connect the PLIB).

Included Linked List DMA Functions should be checked if using the Linked DMA feature of some MCUs.

You can also bring in the I2S Driver by itself, by double clicking I2S under Harmony->*Drivers* in the Available Components list. You will then need to add any additional needed components manually and connect them together.

Building the Library

This section lists the files that are available in the I2S Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is core`/driver/i2s`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`drv_i2s.h`	This file provides the interface definitions of the I2S driver (generated via template `audio/driver/i2s/templates/drv_i2s.h.ftl`)

Required File(s)

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_i2s.c`	This file contains the core implementation of the I2S driver with DMA support (generated via template `audio/driver/i2s/templates/drv_i2s.c.ftl`)

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`N/A`	

Module Dependencies

The I2S Driver Library depends on the following modules:

- I2S Peripheral Library, *or*
- SSC Peripheral Library
- I2SC Peripheral Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_I2S_Initialize	Initializes the I2S instance for the specified driver index.
	DRV_I2S_Status	Gets the current status of the I2S driver module.

b) Client Setup Functions

	Name	Description
	DRV_I2S_Open	Opens the specified I2S driver instance and returns a handle to it.
	DRV_I2S_Close	Closes an opened-instance of the I2S driver.

c) Data Transfer Functions

	Name	Description
	DRV_I2S_ReadBufferAdd	Queues a read operation.
	DRV_I2S_WriteBufferAdd	Queues a write operation.
	DRV_I2S_WriteReadBufferAdd	Queues a write/read operation.
	DRV_I2S_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_I2S_ReadQueuePurge	Removes all buffer requests from the read queue.
	DRV_I2S_WriteQueuePurge	Removes all buffer requests from the write queue.
	DRV_I2S_BufferStatusGet	Returns the transmit/receive request status.
	DRV_I2S_BufferCompletedBytesGet	Returns the number of bytes that have been processed for the specified buffer request.

d) Miscellaneous Functions

	Name	Description
	DRV_I2S_ErrorGet	Gets the I2S hardware errors associated with the client.
	DRV_I2S_LRCLK_Sync	Synchronize to the start of the I2S LRCLK (left/right clock) signal
	DRV_I2S_SerialSetup	Sets the I2S serial communication settings dynamically.
	DRV_I2S_ClockGenerationSet	Set the clock(PLLA and I2SC GCLK clock) generation values
	DRV_I2S_ProgrammableClockSet	Set the Programmable Clock

e) Data Types and Constants

	Name	Description
	DRV_I2S_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2S_BUFFER_EVENT_HANDLER	Pointer to a I2S Driver Buffer Event handler function
	DRV_I2S_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_I2S_SERIAL_SETUP	Defines the data required to dynamically set the serial settings.
	DRV_I2S_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.

Description

This section describes the Application Programming Interface (API) functions of the I2S Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_I2S_Initialize Function

SYS_MODULE_OBJ DRV_I2S_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
)

Summary

Initializes the I2S instance for the specified driver index.

Description

This routine initializes the I2S driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the I2S module ID. For example, driver instance 0 can be assigned to I2S2.

Preconditions

None.

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to the init data structure containing any data necessary to initialize the driver.

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Remarks

This routine must be called before any other I2S routine is called.

This routine should only be called once during system initialization. This routine will NEVER block for hardware access.

Example

```
// The following code snippet shows an example I2S driver initialization.

SYS_MODULE_OBJ              objectHandle;

I2S_PLIB_API drvUsart0PlibAPI = {
    {
        .readCallbackRegister = I2S1_ReadCallbackRegister,
        .read = I2S1_Read,
        .readIsBusy = I2S1_ReadIsBusy,
        .readCountGet = I2S1_ReadCountGet,
        .writeCallbackRegister = I2S1_WriteCallbackRegister,
        .write = I2S1_Write,
        .writeIsBusy = I2S1_WriteIsBusy,
        .writeCountGet = I2S1_WriteCountGet,
        .errorGet = I2S1_ErrorGet
    }
};

DRV_I2S_INIT drvUsart0InitData =
{
    .i2sPlib = &drvUsart0PlibAPI,
    .interruptI2S = I2S1_IRQn,
    .queueSizeTransmit = DRV_I2S_XMIT_QUEUE_SIZE_IDX0,
    .queueSizeReceive = DRV_I2S_RCV_QUEUE_SIZE_IDX0,
```

```
    .dmaChannelTransmit = SYS_DMA_CHANNEL_NONE,
    .dmaChannelReceive = SYS_DMA_CHANNEL_NONE,
    .i2sTransmitAddress = I2S1_TRANSMIT_ADDRESS,
    .i2sReceiveAddress = I2S1_RECEIVE_ADDRESS,
    .interruptDMA = XDMAC_IRQn
};

objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1,
    (SYS_MODULE_INIT*)&drvUsart0InitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

C

```
SYS_MODULE_OBJ DRV_I2S_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

DRV_I2S_Status Function

SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object)

Summary

Gets the current status of the I2S driver module.

Description

This routine provides the current status of the I2S driver module.

Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2S_Initialize routine

Returns

SYS_STATUS_READY - Initialization have succeeded and the I2S is ready for additional operations

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize
SYS_STATUS          i2sStatus;

i2sStatus = DRV_I2S _Status(object);
if (SYS_STATUS_READY == i2sStatus)
{
    // This means the driver can be opened using the
    // DRV_I2S_Open() function.
}
```

C

```
SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object);
```

b) Client Setup Functions

DRV_I2S_Open Function

DRV_HANDLE DRV_I2S_Open
(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
)

Summary

Opens the specified I2S driver instance and returns a handle to it.

Description

This routine opens the specified I2S driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Preconditions

Function DRV_I2S_Initialize must have been called before calling this function.

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended useof the driver. See function description for details.

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_I2S_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver peripheral instance being opened is not initialized or is invalid.
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.

Remarks

The handle returned is valid until the DRV_I2S_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Example

```
DRV_HANDLE handle;

handle = DRV_I2S_Open(DRV_I2S_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

C

```
DRV_HANDLE DRV_I2S_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

DRV_I2S_Close Function

void DRV_I2S_Close(DRV_Handle handle)

Summary

Closes an opened-instance of the I2S driver.

Description

This routine closes an opened-instance of the I2S driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. A new handle must be obtained by calling DRV_I2S_Open before the caller may use the driver again.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Remarks

None.

Example

```
// 'handle', returned from the DRV_I2S_Open

DRV_I2S_Close(handle);
```

C

```
void DRV_I2S_Close(const DRV_HANDLE handle);
```

c) Data Transfer Functions

DRV_I2S_ReadBufferAdd Function

void DRV_I2S_ReadBufferAdd
(
const DRV_HANDLE handle,
void * buffer,
const size_t size,
    DRV_I2S_BUFFER_HANDLE * bufferHandle
)

Summary

Queues a read operation.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue

and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_I2S_Open function.
buffer	Buffer where the received data will be stored.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The buffer handle is returned in the bufferHandle argument. This is DRV_I2S_BUFFER_HANDLE_INVALID if the request was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by the client. It should not be called in the event handler associated with another I2S driver instance. It should not be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

DRV_I2S_ReadBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.
```

C

```
void DRV_I2S_ReadBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_I2S_BUFFER_HANDLE * const bufferHandle);
```

DRV_I2S_WriteBufferAdd Function

void DRV_I2S_WriteBufferAdd

(

const DRV_HANDLE handle,

void * buffer,
size_t size,
DRV_I2S_BUFFER_HANDLE * bufferHandle
);

Summary

Queues a write operation.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the driver instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. On returning, the bufferHandle parameter may be DRV_I2S_BUFFER_HANDLE_INVALID for the following reasons:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the transmit queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or a DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_I2S_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

DRV_I2S_WriteBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.
```

C

```
void DRV_I2S_WriteBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_I2S_BUFFER_HANDLE * bufferHandle);
```

DRV_I2S_WriteReadBufferAdd Function

void DRV_I2S_BufferAddWriteRead(const DRV_HANDLE handle,

void *transmitBuffer, void *receiveBuffer,

size_t size, DRV_I2S_BUFFER_HANDLE *bufferHandle)

Summary

Queues a write/read operation.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_I2S_Open function.
transmitBuffer	Data to be transmitted.
receiveBuffer	Will hold data that is received.
size	Buffer size in bytes (same for both buffers)
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every I2S write. The transmit and receive size must be same.

C

```
void DRV_I2S_WriteReadBufferAdd(const DRV_HANDLE handle, void * transmitBuffer, void *
receiveBuffer, size_t size, DRV_I2S_BUFFER_HANDLE * bufferHandle);
```

DRV_I2S_BufferEventHandlerSet Function

void DRV_I2S_BufferEventHandlerSet
(
const DRV_HANDLE handle,
const DRV_I2S_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t context
)

Summary

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Description

This function allows a client to register a buffer event handling function with the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine.
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Returns

None.

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandler,
                               (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd(myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    &bufferHandle);
```

```
if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

C

```
void DRV_I2S_BufferEventHandlerSet(const DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

DRV_I2S_ReadQueuePurge Function

bool DRV_I2S_ReadQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the read queue.

Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_I2S_Open function.

Returns

True - Read queue purge is successful. False - Read queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myI2SHandle is the handle returned by the DRV_I2S_Open function.
// Use DRV_I2S_ReadBufferAdd to queue read requests
```

```
// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_I2S_ReadQueuePurge(myI2Shandle))
    {
        //Couldn't purge the read queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_I2S_ReadQueuePurge(const DRV_HANDLE handle);
```

DRV_I2S_WriteQueuePurge Function

bool DRV_I2S_WriteQueuePurge(const DRV_HANDLE handle)

Summary

Removes all buffer requests from the write queue.

Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_I2S_Open function.

Returns

True - Write queue purge is successful. False - Write queue purge has failed.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Example

```
// myI2SHandle is the handle returned by the DRV_I2S_Open function.
// Use DRV_I2S_WriteBufferAdd to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_I2S_WriteQueuePurge(myI2Shandle))
    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}
```

C

```
bool DRV_I2S_WriteQueuePurge(const DRV_HANDLE handle);
```

DRV_I2S_BufferStatusGet Function

```
DRV_I2S_BUFFER_EVENT DRV_I2S_BufferStatusGet
(
const    DRV_I2S_BUFFER_HANDLE bufferHandle
)
```

Summary

Returns the transmit/receive request status.

Description

This function can be used to poll the status of the queued buffer request if the application doesn't prefer to use the event handler (callback) function to get notified.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function must have been called and a valid buffer handle returned.

Parameters

Parameters	Description
bufferhandle	Handle for the buffer of which the processed number of bytes to be obtained.

Returns

The success or error event of the buffer.

Remarks

This function returns error event if the buffer handle is invalid.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;
DRV_I2S_BUFFER_EVENT event;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                                (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd( myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

//Check the status of the buffer
//This call can be used to wait until the buffer is processed.
event  = DRV_I2S_BufferStatusGet(bufferHandle);
```

C

```
DRV_I2S_BUFFER_EVENT DRV_I2S_BufferStatusGet(const DRV_I2S_BUFFER_HANDLE bufferHandle);
```

DRV_I2S_BufferCompletedBytesGet Function

size_t DRV_I2S_BufferCompletedBytesGet
(
    DRV_I2S_BUFFER_HANDLE bufferHandle
);

Summary

Returns the number of bytes that have been processed for the specified buffer request.

Description

The client can use this function, in a case where the buffer is terminated due to an error, to obtain the number of bytes that have been processed. Or in any other use case. This function can be used for non-DMA buffer transfers only. It cannot be used when the I2S driver is configured to use DMA.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd function must have been called and a valid buffer handle returned.

Parameters

Parameters	Description
bufferhandle	Handle for the buffer of which the processed number of bytes to be obtained.

Returns

Returns the number of bytes that have been processed for this buffer.

Returns DRV_I2S_BUFFER_HANDLE_INVALID for an invalid or an expired buffer handle.

Remarks

This function is expected to work in non-DMA mode only. This function is thread safe when used in a RTOS application. If called from the callback, it must not call an OSAL mutex or critical section.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                               (uintptr_t)&myAppObj );

DRV_I2S_ReadBufferAdd( myI2Shandle, myBuffer, MY_BUFFER_SIZE,
    bufferHandle);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}
```

```
// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
            // Error handling here.
            // We can find out how many bytes have been processed in this
            // buffer request prior to the error.
            processedBytes= DRV_I2S_BufferCompletedBytesGet(bufferHandle);
            break;

        default:
            break;
    }
}
```

C

```
size_t DRV_I2S_BufferCompletedBytesGet(DRV_I2S_BUFFER_HANDLE bufferHandle);
```

d) Miscellaneous Functions

DRV_I2S_ErrorGet Function

DRV_I2S_ERROR DRV_I2S_ErrorGet(const DRV_HANDLE handle)

Summary

Gets the I2S hardware errors associated with the client.

Description

This function returns the errors associated with the given client. The call to this function also clears all the associated error flags.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Returns

Errors occurred as listed by DRV_I2S_ERROR. This function reports multiple I2S errors if occurred.

Remarks

I2S errors are normally associated with the receiver. The driver clears all the errors internally and only returns the occurred error information for the client.

Example

```
// 'handle', returned from the DRV_I2S_Open

if (DRV_I2S_ERROR_OVERRUN & DRV_I2S_ErrorGet(handle))
{
    //Errors are cleared by the driver, take respective action
    //for the overrun error case.
}
```

C

```
DRV_I2S_ERROR DRV_I2S_ErrorGet(const DRV_HANDLE handle);
```

DRV_I2S_LRCLK_Sync Function

uint32_t DRV_I2S_LRCLK_Sync (const DRV_HANDLE handle, const uint32_t sample_rate);

Summary

Synchronize to the start of the I2S LRCLK (left/right clock) signal

Description

This function waits until low-tohigh transition of the I2S LRCLK (left/right clock) signal (high-low if Left-Jusitified format, this is determined by the PLIB). In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize calls to the DMA with the LRCLK signal so the left/right channel association is valid.

Preconditions

None.

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
    // myI2SHandle is the handle returned
    // by the DRV_I2S_Open function.

    DRV_I2S_LRCLK_Sync(myI2SHandle, 48000);
```

C

```
bool DRV_I2S_LRCLK_Sync(const DRV_HANDLE handle, const uint32_t sample_rate);
```

DRV_I2S_SerialSetup Function

bool DRV_I2S_SerialSetup(const DRV_HANDLE handle,
    DRV_I2S_SERIAL_SETUP * setup)

Summary

Sets the I2S serial communication settings dynamically.

Description

This function sets the I2S serial communication settings dynamically.

Preconditions

DRV_I2S_Open must have been called to obtain a valid opened device handle. The I2S transmit or receive transfer status should

not be busy.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
setup	Pointer to the structure containing the serial setup.

Returns

true - Serial setup was updated successfully. false - Failure while updating serial setup.

Remarks

None.

Example

```
// 'handle', returned from the DRV_I2S_Open

DRV_I2S_SERIAL_SETUP setup = {
        115200,
        DRV_I2S_DATA_8_BIT,
        DRV_I2S_PARITY_ODD,
        DRV_I2S_STOP_1_BIT
    };

DRV_I2S_SerialSetup(handle, &setup);
```

C

```
bool DRV_I2S_SerialSetup(const DRV_HANDLE handle, DRV_I2S_SERIAL_SETUP * setup);
```

DRV_I2S_ClockGenerationSet Function

bool DRV_I2S_ClockGenerationSet(DRV_HANDLE handle, uint8_t div, uint8_t mul, uint8_t div2);

Summary

Set the clock(PLLA and I2SC GCLK clock) generation values

Description

Set the clock(PLLA and I2SC GCLK clock) generation values

Preconditions

None.

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
    // myI2SHandle is the handle returned
    // by the DRV_I2S_Open function.
    DRV_I2S_ClockGenerationSet(myI2SHandle, 2, 40, 5);
```

C

```
bool DRV_I2S_ClockGenerationSet(DRV_HANDLE handle, uint8_t div, uint8_t mul, uint8_t div2);
```

DRV_I2S_ProgrammableClockSet Function

bool DRV_I2S_ProgrammableClockSet(DRV_HANDLE handle, uint8_t pClkNum, uint8_t div2);

Summary

Set the Programmable Clock

Description

Set the Programmable Clock ignoring glitch control

Preconditions

None.

Returns

true if the function was successful, false if a timeout occurred (no transitions seen)

Remarks

None.

Example

```
// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.
DRV_I2S_ProgrammableClockSet(myI2SHandle, 2, 7);
```

C

```
bool DRV_I2S_ProgrammableClockSet(DRV_HANDLE handle, uint8_t pClkNum, uint8_t div2);
```

e) Data Types and Constants

DRV_I2S_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

Description

I2S Driver Buffer Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd functions.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_I2S_BufferEventHandlerSet function when a buffer transfer request is completed.

C

```
typedef enum {
  DRV_I2S_BUFFER_EVENT_COMPLETE,
  DRV_I2S_BUFFER_EVENT_ERROR,
  DRV_I2S_BUFFER_EVENT_ABORT
} DRV_I2S_BUFFER_EVENT;
```

DRV_I2S_BUFFER_EVENT_HANDLER Type

Pointer to a I2S Driver Buffer Event handler function

Description

I2S Driver Buffer Event Handler Function Pointer

This data type defines the required function signature for the I2S driver buffer event handling callback function. A client must register a pointer using the buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the vent relates
context	Value identifying the context of the application that registered the event handling function.

Returns

None.

Remarks

If the event is DRV_I2S_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_I2S_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully. The DRV_I2S_BufferCompletedBytesGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event. And bufferHandle will be valid while the buffer request is in the queue and during callback, unless an error occurred. After callback returns, the driver will retire the buffer handle.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_I2S_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The DRV_I2S_ReadBufferAdd and DRV_I2S_WriteBufferAdd functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running. For example, I2S2 driver buffers cannot be added in I2S1 driver event handler.

Example

```
void APP_MyBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
                               DRV_I2S_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
```

```
    }
}
```

C

```
typedef void (* DRV_I2S_BUFFER_EVENT_HANDLER)(DRV_I2S_BUFFER_EVENT event, DRV_I2S_BUFFER_HANDLE
bufferHandle, uintptr_t context);
```

DRV_I2S_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

Description

I2S Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_I2S_ReadBufferAdd or DRV_I2S_WriteBufferAdd functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

C

```
typedef uintptr_t DRV_I2S_BUFFER_HANDLE;
```

DRV_I2S_SERIAL_SETUP Type

Defines the data required to dynamically set the serial settings.

Description

I2S Driver Serial Setup Data

This data type defines the data required to dynamically set the serial settings for the specific I2S driver instance.

Remarks

This structure is implementation specific. It is fully defined in drv_i2s_definitions.h.

C

```
typedef struct _DRV_I2S_SERIAL_SETUP DRV_I2S_SERIAL_SETUP;
```

DRV_I2S_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

Description

I2S Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_I2S_ReadBufferAdd and DRV_I2S_WriteBufferAdd functions if the buffer add request was not successful.

Remarks

None

C

```
#define DRV_I2S_BUFFER_HANDLE_INVALID
```

Files

Files

Name	Description
drv_i2s.h	I2S Driver Interface Header File
drv_i2s_config_template.h	I2S Driver Configuration Template.

Description

drv_i2s.h

drv_i2s.h

Summary

I2S Driver Interface Header File

Description

I2S Driver Interface Header File

The I2S device driver provides a simple interface to manage the I2S or SSC modules on Microchip PIC32 microcontrollers. This file provides the interface definition for the I2S driver.

drv_i2s_config_template.h

drv_i2s_config_template.h

Summary

I2S Driver Configuration Template.

Description

I2S Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

Peripheral Libraries Help

This topic provides help for the peripheral libraries that are available in the audio repo.

For additional information on Harmony 3 peripheral libraries (PLIBs), refer to the documentation in the csp repository.

Description

The MPLAB Harmony Configurator (MHC) is a Graphical User Interface (GUI) plug-in tool for MPLAB X IDE to configures a rich set of peripherals and functions specific to your application and generate the corresponding peripheral library code. The generated code directly accesses the Peripheral registers without any abstraction layers and it is easy to understand.

I2S Peripheral Library Help

This section provides an interface to use the I2S peripheral.

Introduction

This library provides a brief overview of the I2S peripheral.

Description

The I2S module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2S module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **SDI:** Serial Data Input for receiving sample digital audio data (ADCDAT as output from the codec)
- **SDO:** Serial Data Output for transmitting digital audio data (DACDAT as input to the codec)
- **SCKn:** Serial Clock, also known as bit clock (BCLK)
- **FSn:** Frame Select, also known as Word Select or Left/Right Channel Clock (LRCK)

In addition, there is a fifth line, called **MCKn** (Master Clock), which can be used to drive the codec.

There are two clock units in the I2S module, so **n**=1 or 2 depending on which is used. However there are only two data lines (SDO/SDI), one for each serializer (transmit/receive).

The SCK provides the clock required to drive the data out or into the module, while FS provides the synchronization of the frame based on the protocol mode selected. In I2S mode, the leading edge of audio data is driven out one SCK period of starting the frame.

In Master mode, the module generates both the SCK and FS.

In Slave mode, the peripheral generates the BCLK and LRCLK signals, driving the SCK and FS pins of the I2S module. When in Slave mode, the I2S cannot generate a master clock (MCK), so a generic clock (GCLKn) must be used.

Master Mode

Master Generating its Own Clock – Output BCLK and LRCK

Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out

Configuring the Library

This section describes how to configure the peripheral library using the MHC.

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the I2S peripheral by clicking on the appropriate instance under *Peripherals->I2S* in the Available Components section of MHC. (The I2S component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2S peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options:

Typical values are shown for working in Slave mode, with a codec such as the WM8904 in Master mode. **DMA** and **Interrupt Mode** are always enabled.

<u>For each of the two Clock Units (0 and 1, only the first is shown):</u>

Clock Unit n Enable -- if checked, the clock unit is enabled

Master/Slave Mode can be either Master -- the I2S peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

Master Clock Output Divisor -- the Generic I2S Clock (GCLK_I2S_n) is divided by this number to generate the Master Clock (MCKn)

Master Clock Division Factor -- the Master Clock (MCKn) is divided by this number to generate a serial clock SCKn

Frame Sync Output Invert -- the FSn signal is output with or without inversion

Frame Sync Input Invert -- the FSn signal is used with or without inversion on input

Data Delay from Frame Sync -- either 1 bit for I2S format, or 0 for left/right adjusted format

Slot Size selects the number of bits per slot: 8, 16, 24 or 32

For each of the Serializers (TX/RX):

Stereo/Mono selects stereo or mono (left channel is duplicated to the right)

Data Word Formatting Adjust selects left or right adjustment of data samples within the word

Data Word Size selects the number of bits per sample: 8, 16, 18, 20, 24 or 32 (or 8-bit compact stereo or 16-bit compact stereo)

Data Slot Formatting Adjust selects left or right adjustment of data samples within the slot

Clock Unit Selection selects which clock unit to use for this serializer, 0 or 1

Frame Select Pin specifies which pin is used as the FSn output as selected in the Pin Diagram

Invert selects whether the output of the I2S_LRCLK_Get function returns the true value or inverted one.

Using the Library

This topic describes the basic architecture of the I2S Peripheral Library and provides information and examples on how to use it.

Description

Abstraction Model

The I2S module sits between the I2S Driver, and the actual hardware.

Interface Header File: `plib_i2sc.h`

The interface to the I2S Peripheral Library is defined in the `plib_i2sc.h` header file. Any C language source (`.c`) file that uses the I2S Peripheral Library should include plib_i2sc.h.

Library Source Files:

The I2S Peripheral Library library source files are provided in the audio`/peripheral/i2s_xxxx/src` or audio`/peripheral/i2s_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

Usage Model

The only usage model for the I2S Peripheral Library is to use a Interrupt/DMA model. Therefore the remaining functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right clock (LRCLK) for the I2S stream.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

b) Transaction Functions

Files

Files

Name	Description
plib_i2s_u2224.h	I2S PLIB Header File for documentation

Description

plib_i2s_u2224.h

plib_i2s_u2224.h

Summary

I2S PLIB Header File for documentation

Description

I2SC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of an Inter-IC Sound Controller (I2S). This file must not be included in any MPLAB Project.

I2SC Peripheral Library Help

This section provides an interface to use the Inter-IC Sound Controller (I2SC) peripheral.

Introduction

This library provides a brief overview of the I2SC peripheral.

Description

The I2SC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2SC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **DI:** Serial Data Input for receiving sample digital audio data (ADCDAT)
- **DO:** Serial Data Output for transmitting digital audio data (DACDAT)
- **CK:** Serial Clock, also known as bit clock (BCLK)
- **WS:** Word Select, also known as Left/Right Channel Clock (LRCK)

The BCLK provides the clock required to drive the data out or into the module, while the LRCK provides the synchronization of the frame based on the protocol mode selected.

In Master mode, the module generates both the BCLK on the CK pin and the LRCK on the WS pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of CK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out DO, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

Master Mode

Master Generating its Own Clock – Output BCLK and LRCK

Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out

Audio Formats

The I2SC Module supports just one audio format, I2S. Left Justified format is not available.

Configuring the Library

This section describes how to configure the peripheral library using the MHC.

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the I2SC peripheral by clicking on the appropriate instance under *Peripherals->I2SC* in the Available Components section of MHC. (The I2SC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2SCx peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options (example shown for instance I2SC1):

Default values are shown. The **DMA** and **Interrupt Mode** are always enabled.

Master/Slave Mode can be either Master -- the I2SC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

Data Word Length can be selected from 8 to 32-bits.

Data Format is currently always I2S.

Receiver Stereo/Mono can be either Stereo or Mono (left channel duplicated to right).

# of DMA Channels for Receiver can either Single or Multiple (1/channel).

Loopback Test Mode can be Normal or Loop mode.

Transmitter Stereo/Mono can be either Stereo or Mono (left channel duplicated to right).

Transmit Data When Underrun can be Transmit 0 or Transmit previous.

Slot Width can be 24 or 32-bits wide.

The following options are only shown if the Master/Slave mode is Master:

- **Selected Clock for IMCK Ratio** is selectable as any integer
- **Master Clock to Sample Rate Ratio** can be set from 32 to 2048
- **Master Clock Mode** can be either Master Clock Generated or not

Using the Library

This topic describes the basic architecture of the I2SC Peripheral Library and provides information and examples on how to use it.

Description

Abstraction Model

The I2SC module sits between the I2S Driver, and the actual hardware.

Interface Header File: `plib_i2sc.h`

The interface to the I2SC Peripheral Library is defined in the `plib_i2sc.h` header file. Any C language source (`.c`) file that uses the I2SC Peripheral Library should include plib_i2sc.h.

Library Source Files:

The I2SC Peripheral Library library source files are provided in the `csp/peripheral/i2sc_xxxx/src` or `csp/peripheral/i2sc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

Usage Model

The only usage model for the I2SC Peripheral Library is to use a Interrupt/DMA model. Therefore the remaining functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right clock (LRCLK) for the I2S stream.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`
- `audio/apps/microphone_loopback`

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	I2SCx_Initialize	Initializes I2SCx module of the device

b) Transaction Functions

	Name	Description
	I2SCx_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal

a) Initialization Function

I2SCx_Initialize Function

void I2SCx_Initialize (void);

Summary

Initializes I2SCx module of the device

Description

This function initializes I2SCx module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Preconditions

MHC GUI should be configured with the right values.

Returns

None.

Remarks

This function must be called only once and before any other I2SC function is called.

Example

```
I2SC0_Initialize();
```

C

```
void I2SCx_Initialize();
```

b) Transaction Functions

I2SCx_LRCLK_Get Function

uint32_t I2SCx_LRCLK_Get(void);

Summary

Get the level of the I2S LRCLK (left/right clock) signal

Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize itself to the LRCLK signal.

Preconditions

None.

Returns

State of the LRCLK pin for the I2SCx module -- 1 if high, 0 if low

Remarks

None.

Example

```
I2SC1_LRCLK_Get();
```

C

```
uint32_t I2SCx_LRCLK_Get();
```

c) Data Types and Constants

Files

Files

Name	Description
plib_i2sc_11241.h	I2SC PLIB Header File for documentation

Description

plib_i2sc_11241.h

plib_i2sc_11241.h

Summary

I2SC PLIB Header File for documentation

Description

I2SC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of an Inter-IC Sound Controller (I2SC). This file must not be included in any MPLAB Project.

SPI-I2S Peripheral Library Help

This section provides an interface to use the Inter-IC Sound (I2S) peripheral.

Introduction

This library provides a brief overview of the I2S peripheral. On PIC32MX/MZ processors, this peripheral is shared with the SPI (Serial Peripheral Interface) hardware.

Description

The I2S module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module.

The I2S module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **SDI:** Serial Data Input for receiving sample digital audio data (ADCDAT as output from the codec)
- **SDO:** Serial Data Output for transmitting digital audio data (DACDAT as input to the codec)
- **SCK:** Serial Clock, also known as bit clock (BCLK)
- **FS:** Frame Select, also known as Word Select or Left/Right Channel Clock (LRCK)

In addition, there is a fifth line, called REFCLKO (Reference CLock Output, or Master Clock), which can be used to drive the codec.

The SCK provides the clock required to drive the data out or into the module, while FS provides the synchronization of the frame based on the protocol mode selected. In I2S mode, the leading edge of audio data is driven out one SCK period of starting the frame.

In Master mode, the module generates both the SCK and FS.

In Slave mode, the peripheral generates the BCLK and LRCLK signals, driving the SCK and FS pins of the I2S module.

Master Mode

Master Generating its Own Clock – Output BCLK and LRCK

Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out

Configuring the Library

This section describes how to configure the peripheral library using the MHC.

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip

Harmony Configurator (MHC).

Choose the I2S peripheral by clicking on the appropriate instance under *Peripherals->I2S* in the Available Components section of MHC. (The I2S component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the I2S peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options:

Note: DMA and Interrupt Mode are always enabled.

Master Mode Enable Bit -- only Master Mode is currently supported.

Clock Polarity Select Bit -- determines whether idle state of clock is high (default) or vice versa.

SPI Clock Edge Select Bit -- selects whether output chamges on transistion from active to idle (default) or vice versa.

32/16 Bit Communication Select Bits -- selects the data, FIFO, channel, and frame sizes.

Master CLock Enable Bit -- always selects REFCLK to be used by the baud rate generator.

Audio Protocol Mode -- I2S, left or right justified, or PCM.

Transmit Audio Data Format -- stereo or mono.

Frame Select Pin -- specifies which pin is used for the frame select (aka L/R clock or word select)

Invert -- indicates whether the output of the LRCLK_Get function is inverted or not

Using the Library

This section explains how to use the peripheral library.

Description

```
Library Source Files:
The I2S Peripheral Library library source files are provided in the
audio/peripheral/i2s_xxxx/src or audio/peripheral/i2s_xxxx/templates directory. This folder may
contain optional files and alternate implementations. Please refer to Configuring the Library
for instructions on how to select optional features.

Usage Model
The only usage model for the I2S Peripheral Library is to use a Interrupt/DMA model. Therefore
the remaining functions normally associated with a PLIB like this (handling write or read
requests, or returning transfer status), will be accomplished by the I2S driver directly
communicating with the appropriate DMA functions, which is why they are not provided here.

The one function provided besides initialization is one for synchronizing with the left/right
clock (LRCLK) for the I2S stream.

Example Applications:
This library is used by the following applications, among others:
  * audio/apps/audio_tone
  * audio/apps/microphone_loopback
```

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	I2Sx_Initialize	Initializes I2S x module of the device

b) Status Functions

	Name	Description
	I2Sx_LRCLK_Get	Get LRCLK state

a) Initialization Function

I2Sx_Initialize Function

void I2Sx_Initialize (void);

Summary

Initializes I2S x module of the device

Description

This function initializes I2S x module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Preconditions

MHC GUI should be configured with the right values.

Returns

None.

Remarks

This function must be called only once and before any other SPI function is called.

Example

```
I2S1_Initialize();
```

C

```c
void I2Sx_Initialize();
```

b) Status Functions

I2Sx_LRCLK_Get Function

uint32_t I2Sx_LRCLK_Get (void);

Summary

Get LRCLK state

Description

This function returns the state of the LRCLK (left/right clock, aka word select line) of the I2S peipheral.

Preconditions

MHC GUI should be configured with the right values.

Returns

None.

Example

```
lrclkState = I2Sx_LRCLK_Get();
```

C

```
uint32_t I2Sx_LRCLK_Get();
```

Files

Files

Name	Description
plib_spi_01329.h	SPI PLIB Header File for documentation

Description

plib_spi_01329.h

plib_spi_01329.h

Summary

SPI PLIB Header File for documentation

Description

SPI-I2S PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Peripheral Interface (SPI) controller used for I2S. This file must not be included in any MPLAB Project.

SSC Peripheral Library Help

This section provides an interface to use the Serial Synchronous Controller (SSC) peripheral.

Introduction

This section provides a brief overview of the SSC peripheral.

Description

The SSC module implements an I2S (Inter-IC Sound) interface, for connection between an MCU and an audio peripheral such as a codec or Bluetooth module. The SSC hardware also provides for other protocols, which can be configured manually.

The SSC module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- **RD:** Serial Data Input for receiving sample digital audio data (ADCDAT)
- **TD:** Serial Data Output for transmitting digital audio data (DACDAT)
- **TK/RK:** Transmit/Receive Serial Clock, also known as bit clock (BCLK)
- **TF/RF:** Transmit/Receive Frame Clock, also known as Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

In Master mode, the module generates both the BCLK on the TK pin and the LRCK on the TF pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of BCLK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out TD, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

Master Mode

Master Generating its Own Clock – Output BCLK and LRCK

Slave Mode

Codec Device as Master Derives MCLK from MCU Reference Clock Out

Audio Formats

The SSC Module supports two audio formats, I2S and Left Justified, which are selectable via MHC.

Configuring the Library

This section describes how to configure the peripheral library using the MHC.

Description

The library is configured for the supported processor when the processor is chosen in MPLAB X IDE, using the Microchip Harmony Configurator (MHC).

Choose the SSC peripheral by clicking on the appropriate instance under *Peripherals->SSC* in the Available Components section of MHC. (The SSC component may also be added automatically as a result of using a BSP Template, such as one for a codec or Bluetooth module).

When the SSC peripheral is clicked on in the Project Graph, the following menu is displayed in the Configurations Options):

Default values are shown. The **DMA** and **Interrupt Mode** are always enabled.

Usage Mode can be either Master -- the SSC peripheral supplies the I2S clocks, or Slave -- the peripheral such as a codec or Bluetooth module supplies the I2S clocks.

Audio Protocol Mode can be either I2S, Left Justified, or Custom. In the latter case, a number of other options are then available, which allows the SSC peripheral to be used for a number of protocols (not currently supported by existing drivers).

Data Length is set to the number of data bits per channel.

Using the Library

This topic describes the basic architecture of the SSC Peripheral Library and provides information and examples on how to use it.

Description

Abstraction Model

The SSC module sits between the I2S Driver, and the actual hardware.

Note: Some devices may have only one instance of an SSC interface (SSC0 only).

Interface Header File: plib_ssc.h

The interface to the SSC Peripheral Library is defined in the plib_ssc.h header file. Any C language source (.c) file that uses the SSC Peripheral Library should include plib_ssc.h.

Library Source Files:

The SSC Peripheral Library library source files are provided in the audio`/peripheral/ssc_xxxx/src` or audio`/peripheral/ssc_xxxx/templates` directory. This folder may contain optional files and alternate implementations. Please refer to **Configuring the Library** for instructions on how to select optional features.

Usage Model

The only usage model for the SSC Peripheral Library is to use a Interrupt/DMA model. Therefore many functions normally associated with a PLIB like this (handling write or read requests, or returning transfer status), will be accomplished by the I2S driver directly communicating with the appropriate DMA functions, which is why they are not provided here.

Example Applications:

This library is used by the following applications, among others:

- `audio/apps/audio_tone`
- `audio/apps/audio_tone_linkeddma`

Library Interface

This section describes the Application Programming Interface (API) functions of the Peripheral Library.

Refer to each section for a detailed description.

a) Initialization Function

	Name	Description
	SSC_Initialize	Initializes SSC module of the device

b) Transaction Functions

	Name	Description
	SSC_BaudSet	Changes the baud rate (samples/second) of the interface.
	SSC_LRCLK_Get	Get the level of the I2S LRCLK (left/right clock) signal

a) Initialization Function

SSC_Initialize Function

void SSC_Initialize (void);

Summary

Initializes SSC module of the device

Description

This function initializes SSC module of the device with the values configured in MHC GUI. Once the peripheral is initialized, transfer APIs can be used to transfer the data.

Preconditions

MHC GUI should be configured with the right values.

Returns

None.

Remarks

This function must be called only once and before any other SSC function is called.

Example

```
SSC_Initialize();
```

C

```
void SSC_Initialize();
```

b) Transaction Functions

SSC_BaudSet Function

void SSC_BaudSet (const uint32_t baud);

Summary

Changes the baud rate (samples/second) of the interface.

Description

This function Cchanges the baud rate, or samples/second of the interface.

Preconditions

None.

Returns

None.

Remarks

None.

Example

```
SSC_BaudSet(44100);
```

C

```
void SSC_BaudSet(const uint32_t baud);
```

SSC_LRCLK_Get Function

uint32_t SSC_LRCLK_Get(void);

Summary

Get the level of the I2S LRCLK (left/right clock) signal

Description

This function returns the state of the I2S LRCLK (left/right clock) signal. In the case where this signal is generated from a codec or other external source, this allows the caller to synchronize itself to the LRCLK signal.

Preconditions

None.

Returns

State of the LRCLK pin for the SSC module -- 1 if high, 0 if low if the audio format is I2S, and 0 if high, 1 if low if the format is Left Jusutifed

Remarks

None.

Example

```
SSC_LRCLK_Get();
```

C

```
uint32_t SSC_LRCLK_Get();
```

Files

Files

Name	Description
plib_ssc_6078.h	SSC PLIB Header File for documentation

Description

plib_ssc_6078.h

plib_ssc_6078.h

Summary

SSC PLIB Header File for documentation

Description

SSC PLIB

This library provides documentation of all the interfaces which can be used to control and interact with an instance of a Serial Synchronous Controller (SSC). This file must not be included in any MPLAB Project.

Audio Libraries Help

Decoder Library Help

This section provides descriptions of the software Decoder Libraries that are available in MPLAB Harmony.

ADPCM Decoder Library

Introduction

Adaptive Differential Pulse Code Modulation (ADPCM) is a royalty-free audio codec. It is a lossy compression standard. It is a variant of Differential Pulse-Code Modulation (DPCM). The file format used to contain an ADPCM files, is the same as WAV. Decoding is fast and requires little processing power.

Typical applications using ADPCM are for speech. This implementation currently supports 16 bit stereo.

Library Interface

Functions

	Name	Description
	ADPCM_Initialize	This is function ADPCM_Initialize.
	ADPCM_Decoder	Decode the ADPCM stream of inSize from input pointer into output pointer.
	ADPCM_GetChannels	This is function ADPCM_GetChannels.
	ADPCM_Decoder_ConfigByteOrder	Configure input ADPCM stream byte order.
	ADPCM_HdrGetFormat	This is function ADPCM_HdrGetFormat.
	ADPCM_HdrGetBitsPerSample	This is function ADPCM_HdrGetBitsPerSample.
	ADPCM_HdrGetDataLen	This is function ADPCM_HdrGetDataLen.
	ADPCM_HdrGetFileSize	This is function ADPCM_HdrGetFileSize.
	ADPCM_HdrGetBlockAlign	This is function ADPCM_HdrGetBlockAlign.
	ADPCM_HdrGetBytesPerSec	This is function ADPCM_HdrGetBytesPerSec.
	ADPCM_HdrGetNumOfChan	This is function ADPCM_HdrGetNumOfChan.
	ADPCM_HdrGetSamplesPerSec	This is function ADPCM_HdrGetSamplesPerSec.

Data Types and Constants

	Name	Description
	ADPCMHEADER	ADPCM WAV File Header
	ADPCM_HEADER_SIZE	WAV Header Size

Description

This section describes the Application Programming Interface (API) functions of the ADPCM Decoder Library.

Functions

ADPCM_Initialize Function

This is function ADPCM_Initialize.

C

```
void ADPCM_Initialize(uint8_t * input);
```

ADPCM_Decoder Function

bool ADPCM_Decoder
(uint8_t *input,
uint16_t inSize,
uint16_t *read,
int16_t *output,
uint16_t *written);

Summary

Decode the ADPCM stream of inSize from input pointer into output pointer.

Description

Decode the ADPCM stream of inSize from input pointer into output pointer.

Parameters

Parameters	Description
inSize	Size of ADPCM input stream in bytes.
read	Size of ADPCM decoder has decoded after this function complete

Returns

Pointer to the decoded data. written - Size of decoded data.

If successful, return true, otherwise, return false.

C

```
bool ADPCM_Decoder(uint8_t * input, uint16_t inSize, uint16_t * read, int16_t * output,
uint16_t * written);
```

ADPCM_GetChannels Function

This is function ADPCM_GetChannels.

C

```
uint8_t ADPCM_GetChannels();
```

ADPCM_Decoder_ConfigByteOrder Function

void ADPCM_Decoder_ConfigByteOrder (bool isLE);

Summary

Configure input ADPCM stream byte order.

Description

Configure input ADPCM stream byte order.

Parameters

Parameters	Description
isLE	True if ADPCM stream is in little endian format, otherwise, false.

Returns

None.

C

```
void ADPCM_Decoder_ConfigByteOrder(bool isLE);
```

ADPCM_HdrGetFormat Function

This is function ADPCM_HdrGetFormat.

C

```
int ADPCM_HdrGetFormat();
```

ADPCM_HdrGetBitsPerSample Function

This is function ADPCM_HdrGetBitsPerSample.

C

```
int ADPCM_HdrGetBitsPerSample();
```

ADPCM_HdrGetDataLen Function

This is function ADPCM_HdrGetDataLen.

C

```
int ADPCM_HdrGetDataLen();
```

ADPCM_HdrGetFileSize Function

This is function ADPCM_HdrGetFileSize.

C

```
unsigned int ADPCM_HdrGetFileSize();
```

ADPCM_HdrGetBlockAlign Function

This is function ADPCM_HdrGetBlockAlign.

C

```
int ADPCM_HdrGetBlockAlign();
```

ADPCM_HdrGetBytesPerSec Function

This is function ADPCM_HdrGetBytesPerSec.

C

```
int ADPCM_HdrGetBytesPerSec();
```

ADPCM_HdrGetNumOfChan Function

This is function ADPCM_HdrGetNumOfChan.

C

```
int ADPCM_HdrGetNumOfChan();
```

ADPCM_HdrGetSamplesPerSec Function

This is function ADPCM_HdrGetSamplesPerSec.

C

```
int ADPCM_HdrGetSamplesPerSec();
```

Data Types and Constants

ADPCMHEADER Type

ADPCM WAV File Header

Description

ADPCM Audio File Header Structure

This structure is header structure of wav container.

C

```
typedef struct ADPCMHEADER@1 ADPCMHEADER;
```

ADPCM_HEADER_SIZE Macro

WAV Header Size

Description

ADPCM WAV File Header Size

WAV Header Size

C

```
#define ADPCM_HEADER_SIZE 44
```

MP3 Decoder Library

Introduction

MP3 is a lossy compression standard. MP3 compression works by reducing (or approximating) the accuracy of certain

components of sound that are considered (by psychoacoustic analysis) to be beyond the hearing capabilities of most humans. This method is commonly referred to as perceptual coding or as psychoacoustic modeling.[8] The remaining audio information is then recorded in a space-efficient manner, using MDCT and FFT algorithms. Compared to CD-quality digital audio, MP3 compression can commonly achieve a 75 to 95% reduction in size.

In Harmony 3, the MP3 decoder is limited to 16-bit audio, typically recorded at either CD quality (44100 samples/second) or DVD quality (48000 samples/second).

In addition to audio, an MP3 file may contain metadata, called ID3 tags, which include the name of the song, name of the artist, name of the album, and other information.

An MP3 file is made up of MP3 frames, which consist of a header and a data block. This sequence of frames is called an elementary stream. Due to the "bit reservoir", frames are not independent items and cannot usually be extracted on arbitrary frame boundaries. The MP3 Data blocks contain the (compressed) audio information in terms of frequencies and amplitudes. The MP3 Header consists of a sync word, which is used to identify the beginning of a valid frame. This is followed by a bit indicating that this is the MPEG standard and two bits that indicate that layer 3 is used; hence MPEG-1 Audio Layer 3 or MP3.

MP3 was originally patented, but the last patent expired on April 16, 2017.

In Harmony 3, an open-source MP3 decoder called Helix MP3 is used. The main porition of the code is located in a separate repository called helix_mp3. The MP3 Decoder Library in the audio repository is just a wrapper that calls functions in the main library (which is brought in automatically when you generate code in MHC, with the MP3 Decoder Library selected).

Bulding the Library

This section lists the files that are available in the MP3 Decoder Library.

Description

This section lists the files that are available in the `src` folder of the MP3 Decoder Library. It lists which files need to be included in the build based on a 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 `src/config/<config_dir>/audio/decoder/`

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
`audio_decoders.h`	Header file that includes the lower-level include files (e.g. mp3_dec.h)
`audio_decoders_config.h`	Header file that contains #defines for each enabled decoder (e.g. MP3_DECODER_ENABLED)

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 header and source files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`audio_decoders/mp3/mp3_dec.c,.h`	This file contains implementation of the API for the MP3 Decoder
`audio_decoders/mp3/id3/id3.c,.h`	This file contains implementation of the API for parsing ID3 tags
`audio_decoders/mp3/id3/utils.c,.h`	These files contain utility functions for parsing ID3 tags
`audio_decoders/mp3/fixpnt/*.*`	Files from the Helix MP3 Decoder repository (see below)

The MP3 Decoder Library files in the apps repository only contain a wrapper around the Helix open-source MP3 Decoder, located in the helix_mp3 repository. Files from the fixpnt folder in that repo are copied into the fixpnt folder listed above when code is generated using MHC.

If using the MP3 Decoder Library from the audio repository, the helix_mp3 repository must also be downloaded.

Library Interface

Functions

	Name	Description
	isMP3decoder_enabled	This is function isMP3decoder_enabled.
	MP3_Decoder	This is function MP3_Decoder.
	MP3_FillReadBuffer	This is function MP3_FillReadBuffer.
	MP3_FindSyncWord	This is function MP3_FindSyncWord.
	MP3_GetAudioSize	This is function MP3_GetAudioSize.
	MP3_GetBitRate	This is function MP3_GetBitRate.
	MP3_GetBitsPerSample	This is function MP3_GetBitsPerSample.
	MP3_GetChannels	This is function MP3_GetChannels.
	MP3_GetDuration	This is function MP3_GetDuration.
	MP3_GetSampleRate	This is function MP3_GetSampleRate.
	MP3_HdrGetBytesPerSec	This is function MP3_HdrGetBytesPerSec.
	MP3_HdrGetDataLen	This is function MP3_HdrGetDataLen.
	MP3_HdrGetFileSize	This is function MP3_HdrGetFileSize.
	MP3_HdrGetSamplesPerSec	This is function MP3_HdrGetSamplesPerSec.
	MP3_Initialize	This is function MP3_Initialize.
	MP3_Initialize_N	This is function MP3_Initialize_N.
	MP3_UpdatePlaytime	This is function MP3_UpdatePlaytime.
	read_ID3Tags	This is function read_ID3Tags.

Description

This section describes the Application Programming Interface (API) functions of the MP3 Decoder Library.

Functions

isMP3decoder_enabled Function

This is function isMP3decoder_enabled.

C

```
bool isMP3decoder_enabled();
```

MP3_Decoder Function

This is function MP3_Decoder.

C

```
bool MP3_Decoder(uint8_t * input, uint16_t inSize, uint16_t * read, int16_t * output, uint16_t
* written);
```

MP3_FillReadBuffer Function

This is function MP3_FillReadBuffer.

C

```
int MP3_FillReadBuffer(unsigned char * readBuf, unsigned char * readPtr, int bufSize, int
bytesLeft, SYS_FS_HANDLE fileHandle);
```

MP3_FindSyncWord Function

This is function MP3_FindSyncWord.

C

```
int32_t MP3_FindSyncWord(HMP3Decoder hMP3Decoder, unsigned char * readPtr, int32_t bytesLeft);
```

MP3_GetAudioSize Function

This is function MP3_GetAudioSize.

C

```
uint32_t MP3_GetAudioSize();
```

MP3_GetBitRate Function

This is function MP3_GetBitRate.

C

```
uint32_t MP3_GetBitRate(SYS_FS_HANDLE fileHandle, uint32_t firstframe, uint8_t * heap);
```

MP3_GetBitsPerSample Function

This is function MP3_GetBitsPerSample.

C

```
int MP3_GetBitsPerSample();
```

MP3_GetChannels Function

This is function MP3_GetChannels.

C

```
uint8_t MP3_GetChannels();
```

MP3_GetDuration Function

This is function MP3_GetDuration.

C

```
uint32_t MP3_GetDuration();
```

MP3_GetSampleRate Function

This is function MP3_GetSampleRate.

C

```
uint32_t MP3_GetSampleRate();
```

MP3_HdrGetBytesPerSec Function

This is function MP3_HdrGetBytesPerSec.

C

```
int MP3_HdrGetBytesPerSec();
```

MP3_HdrGetDataLen Function

This is function MP3_HdrGetDataLen.

C

```
int MP3_HdrGetDataLen();
```

MP3_HdrGetFileSize Function

This is function MP3_HdrGetFileSize.

C

```
unsigned int MP3_HdrGetFileSize();
```

MP3_HdrGetSamplesPerSec Function

This is function MP3_HdrGetSamplesPerSec.

C

```
int MP3_HdrGetSamplesPerSec();
```

MP3_Initialize Function

This is function MP3_Initialize.

C

```
void MP3_Initialize(uint8_t * input);
```

MP3_Initialize_N Function

This is function MP3_Initialize_N.

C

```
void MP3_Initialize_N(uint8_t * input, SYS_FS_HANDLE wavFilehandle);
```

MP3_UpdatePlaytime Function

This is function MP3_UpdatePlaytime.

C

```
uint32_t MP3_UpdatePlaytime();
```

read_ID3Tags Function

This is function read_ID3Tags.

C

```
void read_ID3Tags(SYS_FS_HANDLE fileHandle, uint8_t * heap, ID3_EVENT_HANDLER eventHandler);
```

Data Types and Constants

WAV Decoder Library

Introduction

Library Interface

Functions

	Name	Description
	WAV_Initialize_N	This is function WAV_Initialize_N.
	WAV_Decoder	This is function WAV_Decoder.
	WAV_GetChannels	This is function WAV_GetChannels.
	WAV_UpdatePlaytime	This is function WAV_UpdatePlaytime.
	WAV_GetDuration	This is function WAV_GetDuration.
	WAV_HdrGetDataLen	This is function WAV_HdrGetDataLen.
	WAV_HdrGetFileSize	This is function WAV_HdrGetFileSize.
	WAV_GetSampleRate	This is function WAV_GetSampleRate.
	WAV_GetBitRate	This is function WAV_GetBitRate.
	WAV_HdrGetBytesPerSec	This is function WAV_HdrGetBytesPerSec.
	WAV_HdrGetSamplesPerSec	This is function WAV_HdrGetSamplesPerSec.
	WAV_GetAudioSize	This is function WAV_GetAudioSize.
	isWAVdecoder_enabled	This is function isWAVdecoder_enabled.
	WAV_Initialize	This is function WAV_Initialize.

Data Types and Constants

	Name	Description
	WAV_HEADER_SIZE	This is macro WAV_HEADER_SIZE.

	WAV_HEAP_SIZE	This is macro WAV_HEAP_SIZE.
	WAVE_FORMAT_ALAW	This is macro WAVE_FORMAT_ALAW.
	WAVE_FORMAT_MULAW	This is macro WAVE_FORMAT_MULAW.
	WAVE_FORMAT_PCM	This is macro WAVE_FORMAT_PCM.
	WAV_INPUT_BUFFER_SIZE	This is macro WAV_INPUT_BUFFER_SIZE.
	WAV_OUTPUT_BUFFER_SIZE	This is macro WAV_OUTPUT_BUFFER_SIZE.
	dWAVHEADER	This is type dWAVHEADER.
	WAV_DEC	This is type WAV_DEC.

Description

This section describes the Application Programming Interface (API) functions of the WAV Decoder Library.

Functions

WAV_Initialize_N Function

This is function WAV_Initialize_N.

C

```
void WAV_Initialize_N(uint8_t * input, SYS_FS_HANDLE wavFilehandle);
```

WAV_Decoder Function

This is function WAV_Decoder.

C

```
bool WAV_Decoder(uint8_t * input, uint16_t inSize, uint16_t * read, int16_t * output, uint16_t
* written);
```

WAV_GetChannels Function

This is function WAV_GetChannels.

C

```
uint8_t WAV_GetChannels();
```

WAV_UpdatePlaytime Function

This is function WAV_UpdatePlaytime.

C

```
uint32_t WAV_UpdatePlaytime();
```

WAV_GetDuration Function

This is function WAV_GetDuration.

C

```
uint32_t WAV_GetDuration();
```

WAV_HdrGetDataLen Function

This is function WAV_HdrGetDataLen.

C

```
int WAV_HdrGetDataLen();
```

WAV_HdrGetFileSize Function

This is function WAV_HdrGetFileSize.

C

```
unsigned int WAV_HdrGetFileSize();
```

WAV_GetSampleRate Function

This is function WAV_GetSampleRate.

C

```
uint32_t WAV_GetSampleRate();
```

WAV_GetBitRate Function

This is function WAV_GetBitRate.

C

```
uint32_t WAV_GetBitRate();
```

WAV_HdrGetBytesPerSec Function

This is function WAV_HdrGetBytesPerSec.

C

```
int WAV_HdrGetBytesPerSec();
```

WAV_HdrGetSamplesPerSec Function

This is function WAV_HdrGetSamplesPerSec.

C

```
int WAV_HdrGetSamplesPerSec();
```

WAV_GetAudioSize Function

This is function WAV_GetAudioSize.

C

```
uint32_t WAV_GetAudioSize();
```

isWAVdecoder_enabled Function

This is function isWAVdecoder_enabled.

C

```
bool isWAVdecoder_enabled();
```

WAV_Initialize Function

This is function WAV_Initialize.

C

```
void WAV_Initialize(uint8_t * input);
```

Data Types and Constants

WAV_HEADER_SIZE Macro

This is macro WAV_HEADER_SIZE.

C

```
#define WAV_HEADER_SIZE 44
```

WAV_HEAP_SIZE Macro

This is macro WAV_HEAP_SIZE.

C

```
#define WAV_HEAP_SIZE 1024
```

WAVE_FORMAT_ALAW Macro

This is macro WAVE_FORMAT_ALAW.

C

```
#define WAVE_FORMAT_ALAW 0x0006
```

WAVE_FORMAT_MULAW Macro

This is macro WAVE_FORMAT_MULAW.

C

```
#define WAVE_FORMAT_MULAW 0x0007
```

WAVE_FORMAT_PCM Macro

This is macro WAVE_FORMAT_PCM.

C

```
#define WAVE_FORMAT_PCM 0x0001
```

WAV_INPUT_BUFFER_SIZE Macro

This is macro WAV_INPUT_BUFFER_SIZE.

C

```
#define WAV_INPUT_BUFFER_SIZE
(DECODER_MAX_OUTPUT_BUFFER_SIZE>DECODER_MAX_INPUT_BUFFER_SIZE?DECODER_MAX_INPUT_BUFFER_SIZE:DECO
DER_MAX_OUTPUT_BUFFER_SIZE)
```

WAV_OUTPUT_BUFFER_SIZE Macro

This is macro WAV_OUTPUT_BUFFER_SIZE.

C

```
#define WAV_OUTPUT_BUFFER_SIZE (WAV_INPUT_BUFFER_SIZE)
```

dWAVHEADER Type

This is type dWAVHEADER.

C

```
typedef struct dWAVHEADER@1 dWAVHEADER;
```

WAV_DEC Type

This is type WAV_DEC.

C

```
typedef struct WAV_DEC@1 WAV_DEC;
```

Encoder Library Help

This section provides descriptions of the software Encoder Libraries that are available in MPLAB Harmony.

ADPCM Ecncoder Library

Introduction

Adaptive Differential Pulse Code Modulation (ADPCM) is a royalty-free audio codec. It is a lossy compression standard. It is a variant of Differential Pulse-Code Modulation (DPCM). The file format used to contain an ADPCM files, is the same as WAV.

Its popularity comes because it achieves compression of 4 to 1 but does not require high processing power.

Typical applications using ADPCM are for speech. This implementation currently supports 16 bit stereo.

Using the Library

Library Interface

Functions

	Name	Description
	adpcm_encoder_init	This is function adpcm_encoder_init.
	adpcm_encode_frame	This is function adpcm_encode_frame.
	adpcm_encoder_config_byteorder	This is function adpcm_encoder_config_byteorder.
	adpcm_encoder_free	This is function adpcm_encoder_free.

Description

Functions

adpcm_encoder_init Function

This is function adpcm_encoder_init.

C

```
bool adpcm_encoder_init();
```

adpcm_encode_frame Function

This is function adpcm_encode_frame.

C

```
bool adpcm_encode_frame(void * pin, uint32_t insize, void * pout, uint32_t * outsize);
```

adpcm_encoder_config_byteorder Function

This is function adpcm_encoder_config_byteorder.

C

```
void adpcm_encoder_config_byteorder(bool isLE);
```

adpcm_encoder_free Function

This is function adpcm_encoder_free.

C

```
bool adpcm_encoder_free();
```

Data Types and Constants

Files

Files

Name	Description
adpcm_enc.h	Contains the ADPCM encoder specific defintions and function prototypes.

Description

adpcm_enc.h

adpcm_enc.h

Summary

Contains the ADPCM encoder specific defintions and function prototypes.

Description

ADPCM Decoder Header File

This file contains the ADPCM encoder specific defintions and function prototypes.

PCM Encoder Library

Introduction

The PCM encoder is a royalty-free audio codec. It is a lossless standard and there is no compression.

It can typically be used to share speech; share music; create files to playback by older/slower processors. This implementation currently supports 16 bit stereo. The sample rate can vary widely.

Using the Library

Library Interface

Functions

	Name	Description
	pcm_encoder_init	This is function pcm_encoder_init.

	pcm_encode_frame	@Function int ExampleFunctionName (int param1, int param2) @Summary Brief one-line description of the function. @Description Full description, explaining the purpose and usage of the function. Additional description in consecutive paragraphs separated by HTML paragraph breaks, as necessary. Type "JavaDoc" in the "How Do I?" IDE toolbar for more information on tags. @Precondition List and describe any required preconditions. If there are no preconditions, enter "None." @Parameters @param param1 Describe the first parameter to the function. @param param2 Describe the second parameter to the function. @Returns List (if feasible) and describe the return values of the function. 1 Indicates... more
	pcm_encoder_free	This is function pcm_encoder_free.

Description

Functions

pcm_encoder_init Function

This is function pcm_encoder_init.

C

```
bool pcm_encoder_init(int channel, int inputSampleRate);
```

pcm_encode_frame Function

@Function int ExampleFunctionName (int param1, int param2)

@Summary Brief one-line description of the function.

@Description Full description, explaining the purpose and usage of the function.

Additional description in consecutive paragraphs separated by HTML paragraph breaks, as necessary.

Type "JavaDoc" in the "How Do I?" IDE toolbar for more information on tags.

@Precondition List and describe any required preconditions. If there are no preconditions, enter "None."

@Parameters @param param1 Describe the first parameter to the function.

@param param2 Describe the second parameter to the function.

@Returns List (if feasible) and describe the return values of the function. 1 Indicates an error occurred 0 Indicates an error did not occur @Remarks Describe any special behavior not described above.

Any additional remarks.

@Example @code if(ExampleFunctionName(1, 2) == 0) { return 3;

C

```
bool pcm_encode_frame(void * pin, uint32_t insize, void * pout, uint32_t * outsize);
```

pcm_encoder_free Function

This is function pcm_encoder_free.

C

```
bool pcm_encoder_free();
```

Data Types and Constants

Files

Files

Name	Description
pcm_enc.h	• WAV Encoder Header File @Company Company Name @File Name wav_enc.h @Summary Brief description of the file. @Description Describe the purpose of this file.

Description

pcm_enc.h

- WAV Encoder Header File

@Company Company Name

@File Name wav_enc.h

@Summary Brief description of the file.

@Description Describe the purpose of this file.

Math Libraries Help

Introduction

This section provides descriptions of the Math libraries that are available in MPLAB-X Harmony 3.

Description

The fixed point math libraries use fractional value represented in “Q” format as described below

Integer Representation of Fractions

Fractional representation of a real number is given by:

Qn.m where:

• *n* is the number of data bits to the left of the radix point

• *m* is the number of data bits to the right of the radix point

• a signed bit is implied, and takes one bit of resolution

• Shorthand may eliminate the leading 0, such as in Q0.15, which may be shortened to Q15, and similarly Q0.31, which is shortened to Q31

Qn.m numerical values are used by the library processing data as integers. In this format the n represents the number of integer bits, and the m represents the number of fractional bits. All values assume a sign bit in the most significant bit. Therefore, the range of the numerical value is:

$$-2^{(n-1)} \text{ to } [2^{(n-1)} - 2^{(-m)}]; \text{ with a resolution of } 2^{(-m)}.$$

A Q16 format number (Q15.16) would range from -32768.0 (0x8000 0000) to 32767.99998474 with a precision of 0.000015259 (or 2^{-16}). For example, a numerical representation of the number 3.14159 in Q2.13 notation would be:

$$3.14159 * 2^{13} = 25735.9 \Rightarrow 0x6488$$

And converting from the Q7.8 format with the value 0x1D89 would be:

$$0x1D89 / 2^{8} = 7561 / 256 \Rightarrow 29.5316, \text{ accurate to } 0.00391$$

The majority of the math libraries uses functions with variables in Q15 or Q31 format. This limits the equivalent numerical range to roughly -1.0 to 0.999999999. It is possible to represent other number ranges, but scaling before and after the function call are necessary. All library functions will saturate the output if the value exceeds the maximum or is lower than the minimum allowable value for that resolution. Some prescaling may be necessary to prevent unwanted saturation in functions that may otherwise create calculation errors.

Usually, a Q16, Q31 and version of each function is available. Floating point versions are also available for CMSIS-DSP.

CMSIS-DSP

CMSIS-DSP Library

Introduction

This topic describes the CMSIS-DSP Library.

Description

The Cortex Microcontroller System Interface Standard-DSP (CMSIS-DSP) Library is the ARM® DSP Math Library integrated with MPLAB Harmony. It can only be used for ARM Cortex cores M0 through M7.

The CMSIS-DSP Library contains functions implementing 16-bit (Q15) and 32-bit (Q31) fixed-point math, as well as 32-bit floating point (F32) math.

Functions included in the CMSIS-DSP Library include complex math, vector math, matrix math, digital filters, adaptive filters, control and transforms. In many cases, these functions require specific data structures to operate, which are detailed in the header file.

Using the Library

This topic describes the basic architecture of the CMSIS-DSP Library and provides information and examples on its use.

Description

Interface Header File: `arm_math.h`

The interface to the DSP Fixed-Point library is defined in the `arm_math.h` header file. Any C language source (.c) file that uses the DSP Fixed-Point library must include `arm_math.h`.

Library Source Files

The CMSIS-DSP Library source files are provided in the dev_packs repo under:

`packs\arm\CMSIS\<Version Number>\CMSIS\DSP\Source`

The source is not generated to the Harmony 3 project but will give the implementation details and the ability to rebuild the library.

Library Files

The CMSIS-DSP Library archive file is generated by MHC to the following project directory:

`firmware\src\packs\CMSIS\CMSIS\DSP\Lib\GCC`

The following prebuilt versions of the library are available:

`libarm_cortexM0l_math.a`
`libarm_cortexM3l_math.a`
`libarm_cortexM4l_math.a`
`libarm_cortexM4lf_math.a`
`libarm_cortexM7l_math.a`
`libarm_cortexM7lfdp_math.a`
`libarm_cortexM7lfsp_math.a`

The naming convention is as follows:

`libarm_<ARM,cortex><core><codes>_math.a`

The cores are: M0,M3,M4, or M7,

The codes are as follows:

<l,b> - little or big endian byte order

<dp,sp> – double or single precision

<f> - Floating Point Unit on

Configuring the Library using MHC

Provides information on using MPLAB Harmony Configurator (MHC) to configure the library.

Description

The library can be generated by the MPLAB Harmony Configurator to any MPLAB® Harmony 3 project that has the CMSIS Pack component, i.e. only MPLAB® Harmony 3 projects that target ARM processors. Select the CMSIS Pack component highlighted in red below from Project Graph

The Configuration Options window looks like the following, with the selection for the CMSIS-DSP Library

The speed optimized library for the given processor will be generated to the project.

Library Overview

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the CMSIS-DSP Library.

Library Interface Section	Description
Basic Math Functions	Basic vector math functions.
Fast Math Functions	This set of functions provides a fast approximation to sine, cosine, and square root.
Complex Math Functions	This set of functions operates on complex data vectors. The data in the complex arrays is stored in an interleaved fashion (real, imaginary, real, imaginary, ...). In the API functions, the number of samples in a complex array refers to the number of complex values; the array contains twice this number of real values.
Filters	Implementation of biquad cascade, IIR Direct Form I and II Transposed, convolution, correlation, FIR, IIR Lattice, LMS, and Normalized LMS filter functions
Matrix Functions	This set of functions provides basic matrix math operations. The functions operate on matrix data structures.
Transforms	FFT, DCT Type IV, and Real FFT functions.
Statistical Functions	Maximum, Minimum, Mean, Power, RMS, Standard Deviation and Variance calculations.
Support Functions	Vector Copy, Vector Fill, and conversion between F32, Q15, Q32 and Q7 values.
Interpolation Functions	Linear and Bilinear interpolation functions.
Motor Control Functions	PID Motor Control and Vector Clark, Vector Park, and Vector Inverse

The CMSIS-DSP Library uses fixed-point fractional functions to optimize execution speed. These functions limit the accuracy of the calculations to the bits specified for the function. Due to parallelism in some operations, the 16-bit version of the functions are more efficient than their 32-bit counterparts. In many cases both 16-bit and 32-bit functions are available to give the user the choice of balance between speed and functional resolution.

Floating point versions of the function are also available.

Library Interface

Provides information on the list of available functions and interfaces.

Description

Documentation of the available CMSIS-DSP Library functions and interfaces and their descriptions can be found at: https://arm-software.github.io/CMSIS_5/DSP/html/modules.html

DSP

DSP Fixed-Point Math Libraries

Introduction

This topic describes the DSP Fixed-Point Math Libraries.

Description

The DSP Fixed-Point Library are available for the PIC32MZ family of microcontrollers. This library was created from optimized assembly routines written specifically for devices with microAptiv™ core features that utilize DSP ASE.

The LIBQ library is required to use the DSP library.

The DSP Fixed-Point Library contains building block functions for developing digital signal processing algorithms. The library supports the Q15 and Q31 fractional data formats. The functions are implemented in efficient assembly specifically targeted at the DSP extensions in this core family.

The library makes these functions available in a simple C-callable structure.

Functions included in the DSP Fixed-Point Library include complex math, vector math, matrix math, digital filters, and transforms. In many cases, these functions require specific data structures to operate, which are detailed in the header file and examples.

Using the Library

This topic describes the basic architecture of the DSP Fixed-Point Library and provides information and examples on its use.

Description

Interface Header File: `dsp.h`,

The interface to the DSP Fixed-Point library is defined in the `dsp.h` header file. Any C language source (.c) file that uses the DSP Fixed-Point library must include `dsp.h`. This file is automatically included by definitions`.h` when the library is generated using MHC.

Some functions make special use of the optimized fixed-point math library `libq.h`. For use of those functions, the `libq.h` file must also be included in a project. The libq.h file is also installed with MPLAB Harmony. Specific notes within each function will describe if the function is dependent on the LibQ Fixed-Point Math Library.

Library Files:

The DSP Fixed-Point library archive (.a) files are installed with MPLAB Harmony. Although there are two PIC32 DSP files, the linker will only utilize one of these depending on your device usage.

This library is only available in prebuilt binary form, with prototypes for each function described in the `dsp.h` file. The available library versions are:

- dsp_mips32_mz_ef_nfO3.a
- dsp_mips32_mz_ef_nfOs.a

The naming convention for these libraries is as follows:

<name>_<arch>_<core>_<fpu>_<codes>.a

where, “name” is the name of the library (DSP); “arch” is the 32 MIPS architecture (MIPS32); “core” is the microAptive core with DSP ASE extensions (MZ); and “fpu” indicates that a Floating Point Unit(FPU) is available. The “codes” are as follows:

- <nf, fpu> - FPU disabled or enabled
- <O3,OS> - Fully optimized for speed or optimized for size.

Configuring the Library using MHC

Select the Audio/Math/Math Libraries component to load the libq_c, libq/dsp libraries, as shown below

The project configuration should now contain the Math Libraries block.

The configuration window (when the block is selected) looks like the following:

Either the Speed Optimized (as shown above) or the Space Optimized versions of the library can be selected (only 1) to be generated to the project.

NOTE: LIBQ is also required to be selected.

Library Overview

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the DSP Fixed-Point Math Library.

Library Interface Section	Description
Complex Math Functions	General mathematical operations using a complex structure with the form (a + bi)
Vector Math Functions	Mathematical operations on a array of numbers or vector
Matrix Math Functions	Mathematical operations on a matrix
Digital Filter Functions	FIR and IIR filtering functions with various architectures
Transform Functions	FFT, Windows and related transform elements
Support Functions	Quick support functions for numerical transform

The DSP Fixed-Point Library uses fixed-point fractional functions to optimize execution speed. These functions limit the accuracy of the calculations to the bits specified for the function. Due to parallelism in some operations, the 16-bit version of the functions are more efficient than their 32-bit counterparts. In many cases both 16-bit and 32-bit functions are available to give the user the choice of balance between speed and functional resolution.

A majority of the DSP Fixed-Point Library uses functions with variables in Q15 or Q31 format. Representations of these numbers are given in **Data Types and Constants** in the Library Interface section, and generally are int16_t (for Q15 fractional representation) and int32_t (for Q31 fractional representation). This limits the equivalent numerical range to roughly -1.0 to 0.999999999. It is possible to represent other number ranges, but scaling before and after the function call are necessary.

All library functions will saturate the output if the value exceeds the maximum or is lower than the minimum allowable value for that resolution.

Some prescaling may be necessary to prevent unwanted saturation in functions that may otherwise create calculation errors.

Library Interface

Functions

	Name	Description
	DSP_ComplexAdd32	Calculates the sum of two complex numbers.
	DSP_ComplexConj16	Calculates the complex conjugate of a complex number.
	DSP_ComplexConj32	Calculates the complex conjugate of a complex number.
	DSP_ComplexDotProd32	Calculates the dot product of two complex numbers.
	DSP_ComplexMult32	Multiplies two complex numbers.
	DSP_ComplexScalarMult32	Multiplies a complex number and a scalar number.
	DSP_ComplexSub32	Calculates the difference of two complex numbers.
	DSP_FilterFIR32	Performs a Finite Infinite Response (FIR) filter on a vector.
	DSP_FilterFIRDecim32	Performs a decimating FIR filter on the input array.
	DSP_FilterFIRInterp32	Performs an interpolating FIR filter on the input array.
	DSP_FilterIIR16	Performs a single-sample cascaded biquad Infinite Impulse Response (IIR) filter.
	DSP_FilterIIRBQ16	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_cascade8	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.
	DSP_FilterIIRBQ16_cascade8_fast	Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.
	DSP_FilterIIRBQ16_fast	Performs a single-pass IIR Biquad Filter.
	DSP_FilterIIRBQ16_parallel8	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ16_parallel8_fast	Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.
	DSP_FilterIIRBQ32	Performs a high resolution single-pass IIR Biquad Filter.
	DSP_FilterIIRSetup16	Converts biquad structure to coeffs array to set up IIR filter.
	DSP_FilterLMS16	Performs a single sample Least Mean Squares FIR Filter.
	DSP_MatrixAdd32	Addition of two matrices C = (A + B).
	DSP_MatrixEqual32	Equality of two matrices C = (A).
	DSP_MatrixInit32	Initializes the first N elements of a Matrix to the value num.
	DSP_MatrixMul32	Multiplication of two matrices C = A x B.
	DSP_MatrixScale32	Scales each element of an input buffer (matrix) by a fixed number.
	DSP_MatrixSub32	Subtraction of two matrices C = (A - B).
	DSP_MatrixTranspose32	Transpose of a Matrix C = A (T).
	DSP_TransformFFT16	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT16_setup	Creates FFT coefficients for use in the FFT16 function.
	DSP_TransformFFT32	Creates an Fast Fourier Transform (FFT) from a time domain input.
	DSP_TransformFFT32_setup	Creates FFT coefficients for use in the FFT32 function.
	DSP_TransformIFFT16	Creates an Inverse Fast Fourier Transform (FFT) from a frequency domain input.
	DSP_TransformWindow_Bart16	Perform a Bartlett window on a vector.
	DSP_TransformWindow_Bart32	Perform a Bartlett window on a vector.
	DSP_TransformWindow_Black16	Perform a Blackman window on a vector.
	DSP_TransformWindow_Black32	Perform a Blackman window on a vector.
	DSP_TransformWindow_Cosine16	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Cosine32	Perform a Cosine (Sine) window on a vector.
	DSP_TransformWindow_Hamm16	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hamm32	Perform a Hamming window on a vector.
	DSP_TransformWindow_Hann16	Perform a Hanning window on a vector.

	DSP_TransformWindow_Hann32	Perform a Hanning window on a vector.
	DSP_TransformWindow_Kaiser16	Perform a Kaiser window on a vector.
	DSP_TransformWindow_Kaiser32	Perform a Kaiser window on a vector.
	DSP_TransformWinInit_Bart16	Create a Bartlett window.
	DSP_TransformWinInit_Bart32	Create a Bartlett window.
	DSP_TransformWinInit_Black16	Create a Blackman window.
	DSP_TransformWinInit_Black32	Create a Blackman window.
	DSP_TransformWinInit_Cosine16	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Cosine32	Create a Cosine (Sine) window.
	DSP_TransformWinInit_Hamm16	Create a Hamming window.
	DSP_TransformWinInit_Hamm32	Create a Hamming window.
	DSP_TransformWinInit_Hann16	Create a Hanning window.
	DSP_TransformWinInit_Hann32	Create a Hanning window.
	DSP_TransformWinInit_Kaiser16	Create a Kaiser window.
	DSP_TransformWinInit_Kaiser32	Create a Kaiser window.
	DSP_VectorAbs16	Calculate the absolute value of a vector.
	DSP_VectorAbs32	Calculate the absolute value of a vector.
	DSP_VectorAdd16	Calculate the sum of two vectors.
	DSP_VectorAdd32	Calculate the sum of two vectors.
	DSP_VectorAddc16	Calculate the sum of a vector and a constant.
	DSP_VectorAddc32	Calculate the sum of a vector and a constant.
	DSP_VectorAutocorr16	Computes the Autocorrelation of a Vector.
	DSP_VectorBexp16	Computes the maximum binary exponent of a vector.
	DSP_VectorBexp32	Computes the maximum binary exponent of a vector.
	DSP_VectorChkEqu32	Compares two input vectors, returns an integer '1' if equal, and '0' if not equal.
	DSP_VectorCopy	Copies the elements of one vector to another.
	DSP_VectorCopyReverse32	Reverses the order of elements in one vector and copies them into another.
	DSP_VectorDivC	Divides the first N elements of inVector by a constant divisor, and stores the result in outVector.
	DSP_VectorDotp16	Computes the dot product of two vectors, and scales the output by a binary factor.
	DSP_VectorDotp32	Computes the dot product of two vectors, and scales the output by a binary factor
	DSP_VectorExp	Computes the EXP (e^x) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorFill	Fills an input vector with scalar data.
	DSP_VectorLn	Computes the Natural Log, Ln(x), of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog10	Computes the Log10(x), of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorLog2	Computes the Log2(x) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorMax32	Returns the maximum value of a vector.
	DSP_VectorMaxIndex32	Returns the index of the maximum value of a vector.
	DSP_VectorMean32	Calculates the mean average of an input vector.
	DSP_VectorMin32	Returns the minimum value of a vector.
	DSP_VectorMinIndex32	Returns the index of the minimum value of a vector.
	DSP_VectorMul16	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMul32	Multiplication of a series of numbers in one vector to another vector.
	DSP_VectorMulc16	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorMulc32	Multiplication of a series of numbers in one vector to a scalar value.
	DSP_VectorNegate	Inverses the sign (negates) the elements of a vector.

	DSP_VectorRecip	Computes the reciprocal (1/x) of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorRMS16	Computes the root mean square (RMS) value of a vector.
	DSP_VectorShift	Shifts the data index of an input data vector.
	DSP_VectorSqrt	Computes the square root of the first N elements of inVector, and stores the result in outVector.
	DSP_VectorStdDev16	Computes the Standard Deviation of a Vector.
	DSP_VectorSub16	Calculate the difference of two vectors.
	DSP_VectorSub32	Calculate the difference of two vectors.
	DSP_VectorSumSquares16	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorSumSquares32	Computes the sum of squares of a vector, and scales the output by a binary factor.
	DSP_VectorVari16	Computes the variance of N elements of a Vector.
	DSP_VectorVariance	Computes the variance of N elements of inVector.
	DSP_VectorZeroPad	Fills an input vector with zeros.
	mul16	multiply and shift integer
	mul16r	multiply and shift Q15
	mul32	multiply and shift Q31
	SAT16	saturate both positive and negative Q15
	SAT16N	saturate negative Q15
	SAT16P	saturate positive Q15

Description

Functions

DSP_ComplexAdd32 Function

Calculates the sum of two complex numbers.

Description

Function DSP_ComplexAdd32:

void DSP_ComplexAdd32(int32c *indata1, int32c *indata2, int32c *Output);

Calculates the sum of two complex numbers, indata1 and indata2, and stores the complex result in Output. Complex numbers must be in the structural form that includes real and imaginary components. The function saturates the output values if maximum or minimum are exceeded. All values are in Q31 fractional data format. (a + bi) + (c + di) => (a + c) + (b + d)i

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//                  (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexAdd32(input1, input2, res);

// result = {0x73333333, 0xC0000000} = (0.9 - 0.5i)
```

C

```
void DSP_ComplexAdd32(int32c * indata1, int32c * indata2, int32c * Output);
```

DSP_ComplexConj16 Function

Calculates the complex conjugate of a complex number.

Description

DSP_ComplexConj16:

void DSP_ComplexConj16(int16c *indata, int16c *Output);

CCalculates the complex conjugate of Indata, and stores the result in Outdata. Both numbers must be in the complex number data structure which includes real and imaginary components. Values are in Q15 fractional data format. The function will saturate the output if maximum or minimum values are exceeded. (a + bi) => (a - bi)

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata	pointer to input complex number (int16c)

Returns

pointer to result complex numbers (int16c)

None.

Remarks

None.

Example

```
int16c *res, result;
int16c *input1;
int16c test_complex_1 = {0x4000,0x0CCC};
//                  (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;

DSP_ComplexConj16(input1, res);

// result = {0x4000, 0xF334} = (0.5 - 0.1i)
```

C

```
void DSP_ComplexConj16(int16c * indata, int16c * Output);
```

DSP_ComplexConj32 Function

Calculates the complex conjugate of a complex number.

Description

Function DSP_ComplexConj32:

void DSP_ComplexConj32(int32c *indata, int32c *Output);

Calculates the complex conjugate of indata, and stores the result in Output. Both numbers must be in the complex number data structure, which includes real and imaginary components. Values are in Q31 fractional data format. The function will saturate the output if maximum or minimum values are exceeded. (a + bi) => (a - bi)

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;

DSP_ComplexConj32(input1, res);

// result = {0x40000000, 0xF3333334} = (0.5 - 0.1i)
```

C

```
void DSP_ComplexConj32(int32c * indata, int32c * Output);
```

DSP_ComplexDotProd32 Function

Calculates the dot product of two complex numbers.

Description

Function DSP_ComplexDotProd32:

void DSP_ComplexDotProd32(int32c *indata1, int32c *indata2, int32c *Output);

Calculates the dot product of two complex numbers, indata1 and indata2, and stores the result in Output. All numbers must be in complex structural format that includes real and imaginary components, and the numbers are in fractional Q31 format. The function will saturate the output if it exceeds maximum or minimum ratings. The formula for the dot product is as follows: Output(real) = (Input1.re * Input2.re) + (Input1.im * Input2.im); Output(img) = [(Input1.re * Input2.im) - (Input1.im * Input2.re)]i (a +

bi) dot (c + di) => (a * c + b * d) + (a * d - b * c)i

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//                  (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexDotProd32(input1, input2, res);

// result = {0x31EB851E, 0xCE147AE3} = (0.39 - 0.39i)
```

C

```
void DSP_ComplexDotProd32(int32c * indata1, int32c * indata2, int32c * Output);
```

DSP_ComplexMult32 Function

Multiplies two complex numbers.

Description

Function DSP_ComplexMult32:

void DSP_ComplexMult32(int32c *indata1, int32c *indata2, int32c *Output);

Multiplies two complex numbers, indata1 and indata2, and stores the complex result in Output. All numbers must be in the int32c complex data structure. All data is in Q31 fractional format. The function will saturate if maximum or minimum values are exceeded. Output(real) = (Input1.re * Input2.re) - (Input1.im * Input2.im); Output(img) = [(Input1.re * Input2.im) + (Input1.im * Input2.re)]i (a + bi) x (c + di) => (a * c - b * d) + (a * d + b * c)i

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//                  (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexMult32(input1, input2, res);

// result = {0x4147AE14, 0xE51EB8551} = (0.51 - 0.21i)
```

C

```
void DSP_ComplexMult32(int32c * indata1, int32c * indata2, int32c * Output);
```

DSP_ComplexScalarMult32 Function

Multiplies a complex number and a scalar number.

Description

Function DSP_ComplexScalarMult32:

void DSP_ComplexScalarMult32(int32c *indata, int32_t Scalar, int32c *Output);

Multiplies a complex number, indata, by a scalar number, Scalar, and stores the result in Output. indata and Output must be in int32c structure with real and imaginary components. All data must be in the fractional Q31 format. The function will saturate if maximum or minimum values are exceeded. Output(real) = (Input1.re * Scalar); Output(img) = [(Input1.im * Scalar)]i (a + bi) * C => (a * C + b * Ci)

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata	pointer to input complex number (int32c)
Scalar	fractional scalar input value (int32_t)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1;
int32_t  scalarInput = 0x20000000; // 0.25
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)

res=&result;
input1=&test_complex_1;
```

```
DSP_ComplexScalarMult32(input1, scalarInput, res);

// result = {0x10000000, 0x03333333} = (0.125 + 0.025i)
```

C

```
void DSP_ComplexScalarMult32(int32c * indata, int32_t Scalar, int32c * Output);
```

DSP_ComplexSub32 Function

Calculates the difference of two complex numbers.

Description

Function DSP_ComplexSub32:

void DSP_ComplexSub32(int32c *indata1, int32c *indata2, int32c *Output);

Calculates the difference of two complex numbers, indata1 less indata2, and stores the complex result in Output. Both numbers must be in a complex data structure, which includes real and imaginary components. The function saturates the output values if maximum or minimum are exceeded. Real and imaginary components are in the Q31 fractional data format. (a + bi) - (c + di) => (a - c) + (b - d)i

Preconditions

Complex numbers must be in the int32c format.

Parameters

Parameters	Description
indata1	pointer to input complex number (int32c)
indata2	pointer to input complex number (int32c)

Returns

pointer to result complex numbers (int32c)

None.

Remarks

None.

Example

```
int32c *res, result;
int32c *input1, *input2;
int32c test_complex_1 = {0x40000000,0x0CCCCCCC};
//                  (0.5 + 0.1i)
int32c test_complex_2 = {0x73333333,0xB3333334};
//                  (0.9 - 0.6i)
res=&result;
input1=&test_complex_1;
input2=&test_complex_2;

DSP_ComplexSub32(input1, input2, res);

// result = {0xCCCCCCCD, 0x59999998} = (-0.4 + 0.7i)
```

C

```
void DSP_ComplexSub32(int32c * indata1, int32c * indata2, int32c * Output);
```

DSP_FilterFIR32 Function

Performs a Finite Infinite Response (FIR) filter on a vector.

Description

Function DSP_FilterFIR32:

void DSP_FilterFIR32(int32_t *outdata, int32_t *indata, int32_t *coeffs2x, int32_t *delayline, int N, int K, int scale);

Performs an FIR filter on the vector indata, and stores the output in the vector outdata. The number of samples processed in the array is given by N. The number of filter taps is given by K. The values are scaled upon input by the binary scaling factor (right shift), scale. The array of 2*K coefficients is contained in the array coeffs2x, where the values are in order b0, b1, b2... and repeated. Lastly the delayline is an array of K values that are initialized to zero and represent previous values. All values are in fractional Q31 data format. The function will saturate results if minimum or maximum values are exceeded.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four. K must be greater than 2 and a multiple of 2. delayline must have K elements, and be initialized to zero. coeffs2x must have 2*K elements.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
coeffs2x	pointer to an array of coefficients (int32_t)
delayline	pointer to an array of delay variables (int32_t)
N	number of points in the array to process (int) number of samples (int)
K	number of filter taps
scale	binary scaler divisor (1 / 2^scale) (int)

Returns

None.

Remarks

Filter coefs must be repeated within the array. The array is twice as large as the number of taps, and the values are repeated in order b0, b1, b2,...bn, b0, b1, b2,... bn. The function updates the delayline array, which must be K elements long. The array should be initialized to zero prior to the first processing. It will contain values for processing cascaded filters within a loop.

Example

```
#define TAPS 4
#define numPOINTS 256

int filterN = numPOINTS;
int filterK = TAPS;
int filterScale = 1;  // scale output by 1/2^1 => output * 0.5
int32_t FilterCoefs[TAPS*2] = {0x40000000, 0x20000000, 0x20000000, 0x20000000,
            0x40000000, 0x20000000, 0x20000000, 0x20000000};
// note repeated filter coefs, A B C D A B C D
//             0.5, 0.25, 0.25, 0.25, 0.5, 0.25, 0.25, 0.25

int32_t outFilterData[numPOINTS]={0};
int32_t inFilterData[numPOINTS];
int filterDelayLine[TAPS]={0};

while(true)
{

//   put some data into input array, inFilterData, here //

DSP_FilterFIR32(outFilterData, inFilterData, FilterCoefs, filterDelayLine,
            filterN, filterK, filterScale);
}
```

C

```
void DSP_FilterFIR32(int32_t * outdata, int32_t * indata, int32_t * coeffs2x, int32_t *
```

```
delayline, int N, int K, int scale);
```

DSP_FilterFIRDecim32 Function

Performs a decimating FIR filter on the input array.

Description

Function DSP_FilterFIRDecim32:

void DSP_FilterFIRDecim32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);

Compute a FIR decimation filter on the input vector indata, and store the results to the vector outdata. The total number of output elements is set by N, and therefore the outdata array must be at least N in length. The decimation ratio is given by rate. The input is sampled every integer value of rate, skipping every (rate-1) input samples. The input array must therefore be (rate*N) samples long. The amount of filter taps is specified by K. Coeffs specifies the coefficients array. The delayline array holds delay inputs for calculation, and must be initialized to zero prior to calling the filter. Both coeffs and delayline must be K in length. Scale divides the input by a scaling factor by right shifting the number of bits (1/2^scale). All values of input, output, and coeffs are given in Q31 fractional data format. The function will saturate if the output value exceeds the maximum or minimum value.

Y = b0 * X0 + (b1 * X(-1)) + (b2 * X(-2)

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. delayline must have K elements, and be initialized to zero. coeffs must have K elements. outdata must have N elements indata must have (N*rate) elements

Parameters

Parameters	Description
outdata	pointer to output array of elements (int32_t)
indata	pointer to input array of elements (int32_t)
coeffs	pointer to an array of coefficients (int32_t)
delayline	pointer to an array of delay variables (int32_t)
N	number of output elements to be processed (int)
K	number of filter taps and coeffs (int)
scale	binary scaler divisor (1 / 2^scale) (int)
rate	decimation ratio (int)

Returns

None.

Remarks

Coefs are loaded into the array with corresponding to the least delay first (C0, C(-1), C(-2)). K must be greater than rate. Even while decimating the input stream, every input passes through the delayline. So FIR filters of arbitrary length will give the same output as a non-decimating FIR, just with fewer responses.

Example

```
#define N 8         // number of output samples
#define TAPS 5
#define SKIP 3

int testFilterN = N;        // number of output elements
int testFilterK = TAPS;     // number of taps
int testFilterRate = SKIP;  // decimation rate R
int32_t outFiltDataDec[N]={0};
int32_t *inTestFilter[N*SKIP];
int filtScaleNum = 1;       // scale output (1 /2^n) => Y * 0.5

int32_t filtDelayTest[8]={0};  // always initialize to zero

// get pointer to input buffer here //
inTestFilter = &inputBuffer;
```

```
DSP_FilterFIRDecim32(outFiltDataDec, inTestFilter, inTestCoefs,
        filtDelayTest, testFilterN, testFilterK, filtScaleNum, testFilterRate);
```

C

```
void DSP_FilterFIRDecim32(int32_t * outdata, int32_t * indata, int32_t * coeffs, int32_t *
delayline, int N, int K, int scale, int rate);
```

DSP_FilterFIRInterp32 Function

Performs an interpolating FIR filter on the input array.

Description

Function DSP_FilterFIRInterp32:

void DSP_FilterFIRInterp32(int32_t *outdata, int32_t *indata, int32_t *coeffs, int32_t *delayline, int N, int K, int scale, int rate);

Perform an interpolating FIR filter on the first N samples of indata, and stores the result in outdata. The number of output elements is N*rate. The number of filter taps, K, must be an even multiple of N. The coefficients array, Coeffs, must be K elements long. The delay line array, delayline, must be K/R elements long, and be initialized to zero. All data elements must be in Q31 fractional data format. Scaling is performed via binary shift on the input equivalent to (1/2^shift). The function will saturate the output if it exceeds maximum or minimum values. The function creates R output values for each input value processed. The delayline of previous values is processed with R elements of the coefficient array. Numerically:

Y(1,0) = X(0)*C(0) + X(-1)*C(rate) + X(-2)*C(2*rate) ... Y(1,1) = X(0)*C(1) + X(-1)*C(rate+1) + X(-2)*C(2*rate + 1) ... Y(1,rate) = X(0)*C(N) + X(-1)*C(rate+N) + X(-2)*C(2*rate + N) ...

where output Y corresponds to (input,rate) different outputs, input X has (M/rate) sample delays and C is the coefficient array.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. delayline must have (K/R) elements, and be initialized to zero. K (taps) must be an even multiple of R (rate). outdata must have R*N elements.

Parameters

Parameters	Description
outdata	pointer to output array of elements (int32_t)
indata	pointer to input array of elements (int32_t)
coeffs	pointer to an array of coefficients (int32_t)
delayline	pointer to an array of delay variables (int32_t)
N	number of output elements to be processed (int)
K	number of filter taps and coeffs (int)
scale	binary scaler divisor (1 / 2^scale) (int)
rate	decimation ratio (int)

Returns

None.

Remarks

The function processes each input (rate) times. With each pass, coefficients are offset so that (K/rate) multiply accumulate cycles occur.

Example

```
// interpret evenly 1/3 spaced values
#define N 4         // number of output samples
#define TAPS 6
#define INTERP 3

int ifiltN = N;
int ifiltK = TAPS;  // k must be an even multiple of R
int ifiltR = INTERP;
```

```
int32_t ifiltOut[N*INTERP]={0};
int32_t ifiltDelay[2]={0};  // must be initialized to zero
int ifiltScale = 0;         // no scaling

int32_t ifiltCoefsThirds[TAPS]={0x2AAAAAA9, 0x55555555,0x7FFFFFFE,
                                0x55555555,0x2AAAAAA9,0x00000000};
//                  0.333333, 0.6666667, 0.99999999, 0.6666667, 0.33333333, 0

int32_t ifiltInput[N]={0x0CCCCCCD, 0x19999999, 0x26666666, 0x33333333};
//                  0.1,  0.2,  0.3,  0.4

DSP_FilterFIRInterp32(ifiltOut, ifiltInput, ifiltCoefsThirds, ifiltDelay,
                                ifiltN, ifiltK, ifiltScale, ifiltR);

// ifiltOut = {0x04444444, 0x08888889, 0x0CCCCCCD, 0x11111111, 0x15555555, 0x19999999,
//          0x1DDDDDDD, 0x22222221, 0x26666665, 0x2AAAAAAA,0x2EEEEEEE, 0x33333332}
// = 0.0333, 0.0667, 0.1, 0.1333, 0.1667, 0.2, 0.2333, 0.2667, 0.3, 0.3333, 0.3667, 0.4
```

C

```
void DSP_FilterFIRInterp32(int32_t * outdata, int32_t * indata, int32_t * coeffs, int32_t *
delayline, int N, int K, int scale, int rate);
```

DSP_FilterIIR16 Function

Performs a single-sample cascaded biquad Infinite Impulse Response (IIR) filter.

Description

Function DSP_FilterIIR16:

int16_t DSP_FilterIIR16(int16_t in, int16_t *coeffs, int16_t *delayline, int B, int scale);

Performs a single element cascaded biquad IIR filter on the input, in. The filter contains B number of biquad sections, and cascades the output of one to the input of the next. B must be greater than 2 and a multiple of 2. The int16_t output generated by the function is the computation from the final biquad stage. Delay pipeline array delayline must contain 2*B values and be initialized to zero prior to use. The coefficient array must contain 4*B elements, and must be set up in order of biquad a1, a2, b1, b2. A binary (right shift) factor, scale, will scale the output equivalent to (1/2^scale). All numerical values must be in Q15 fractional data format. The function will saturate values if maximum or minimum values are exceeded.

Y = X0 + (b1 * X(-1)) + (b2 * X(-2) + (a1 * Y(-1)) + (a2 * Y(-2))

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. B must be greater than 2 and a multiple of 2. delayline must have 2*B elements, and be initialized to zero. coeffs must have 4*B elements.

Parameters

Parameters	Description
in	input data element X (int16_t)
coeffs	pointer to an array of coefficients (int16_t)
delayline	pointer to an array of delay variables (int16_t)
B	number of cascaded biquad filter groups to process (int)
scale	binary scaler divisor (1 / 2^scale) (int)

Returns

Sample output Y (int16_t)

Example

```
#define B 8         // use * biquad filters in cascade

int dataSamples = 256;
int i, j;
biquad16 bquad[B];
int16_t coefs[4*B]= {0};
```

```
int16_t delaylines[2*B]= {0};
int16_t Y, X;
int scaleBquad = 1;  // scale output (1 /2^n) => Y * 0.5

// do something to set up coefs, for instance this example //

for (j=0; jRemarks:Filter coefs must be stored within the array as a1, a2, b1, b2, a1, a2, b1,
b2, in order of biquads form input to output. A function to translate the coeffs from biquad
structure to coeffs is available in DSP_FilterIIRSetup16. The function updates the delayline
array, which must be 2*B elements long. The array should be initialized to zero prior to the
first processing. It will contain values for processing cascaded filters within a loop.
```

C

```
int16_t DSP_FilterIIR16(int16_t in, int16_t * coeffs, int16_t * delayline, int B, int scale);
```

DSP_FilterIIRBQ16 Function

Performs a single-pass IIR Biquad Filter.

Description

Function DSP_FilterIIRBQ16:

int16_t DSP_FilterIIRBQ16(int16_t Xin, PARM_EQUAL_FILTER *pFilter);

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, with results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. All values are fractional Q15 and Q31, see data structure for specifics.

Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))

Preconditions

Delay register values should be initialized to zero.

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER *ptrFilterEQ;
PARM_EQUAL_FILTER FilterEQ;
uint16_t DataIn, DataOut;
ptrFilterEQ = &FilterEQ;

// 48KHz sampling; 1 KHz bandpass filter; Q=0.9
//  divide by 2 and convert to Q15
//   b0 = 0.06761171785499065
//   b1 = 0
//   b2 = -0.06761171785499065
//   a1 = -1.848823142275648
//   a2 = 0.8647765642900187

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x0453;       // feed forward b0 coef
ptrFiltEQ32->b[1]=0;            // feed forward b1 coef
```

```
ptrFiltEQ32->b[2]=0xFBAD;       // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x89AD;       // feedback a1 coef
ptrFiltEQ32->a[1]=0x3758;       // feedback a2 coef

for (i=0;i<256;i++)
    {
    // *** get some input data here
       DataIn32 = three_hundred_hz[i];

       DataOut = DSP_FilterIIRBQ16(DataIn, ptrFilterEQ);

     // *** do something with the DataOut here
     }
```

C

```
int16_t DSP_FilterIIRBQ16(int16_t Xin, PARM_EQUAL_FILTER * pFilter);
```

DSP_FilterIIRBQ16_cascade8 Function

Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.

Description

Function DSP_FilterIIRBQ16_cascade8:

int16_t DSP_FilterIIRBQ16_cascade8(int16_t Xin, PARM_EQUAL_FILTER *pFilter_Array);

Calculates a single pass IIR biquad cascade filter on Xin, and delivers the result as a 16-bit output. The cascade of filters is 8 unique biquad filters arranged in series such that the output of one is provided as the input to the next. A unique filter coefficient set is provided to each, and 32 bit delay lines are maintained for each. All math is performed using 32 bit instructions, which results truncated to 16-bits for the output. Global gain values are available on the output. Fracgain is a Q15 fractional gain value and expgain is a binary shift gain value. The combination of the two can be utilized to normalize the output as desired. All values are fractional Q15 and Q31, see data structure for specifics.

Y = Y7 <- Y6 <- Y5 <- Y4 <- Y3 <- Y2 <- Y1 <- Y0 where each Yn filter element represents a unique IIR biquad: Yn = Y(n-1)*b0 + (b1 * Y(n-2)) + (b2 * Y(n-3)) - (a1 * Yn(-1)) - (a2 * Yn(-2)) and: for Y0; Y(n-1) = Xin(0)

Preconditions

Delay register values should be initialized to zero.

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER filtArray[8];
uint16_t dataY, dataX;

// example to use 2 filter blocks as notch filters
// fill entire Filter Array with coefs
for (i=0;i<8;i++)
```

```
{
    filtArray[i].Z[0]=0;
    filtArray[i].Z[1]=0;

    // note all coefs are half value of original design, gain handled in algorithm
    // all pass
    filtArray[i].b[0]=0x4000;
    filtArray[i].b[1]=0;        // feed forward b1 coef
    filtArray[i].b[2]=0;  // feed forward b2 coef

    filtArray[i].a[0]=0;  // feedback a1 coef
    filtArray[i].a[1]=0;  // feedback a2 coef
}
    // Unique filters for example
    // 10KHz notch filter -- divide coefs by 2
    b0 = 0.5883783602332997
    b1 = -0.17124071441396285
    b2 = 0.5883783602332997
    a1 = -0.17124071441396285
    a2 = 0.1767567204665992

    // note all coefs are half value of original design, gain handled in algorithm
    filtArray[3].b[0]=0x25a7;           // feed forward b0 coef
    filtArray[3].b[1]=0xf508;           // feed forward b1 coef
    filtArray[3].b[2]=0x25a7;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArray[3].a[0]=0xf508;           // feedback a1 coef
    filtArray[3].a[1]=0x0b4f;           // feedback a2 coef

    // 1 KHz notch filter -- divide coefs by 2
    b0 = 0.9087554064944908
    b1 = -1.7990948352036205
    b2 = 0.9087554064944908
    a1 = -1.7990948352036205
    a2 = 0.8175108129889816

    // note all coefs are half value of original design, gain handled in algorithm
    filtArray[7].b[0]=0x3a29;           // feed forward b0 coef
    filtArray[7].b[1]=0x8cdc;           // feed forward b1 coef
    filtArray[7].b[2]=0x3a29;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArray[7].a[0]=0x8cdc;           // feedback a1 coef
    filtArray[7].a[1]=0x3452;           // feedback a2 coef

for (i=0;i<256;i++)
    {
     // *** get input data here
     dataX = compound_300_1K_hz16[i];

     dataY = DSP_FilterIIRBQ16_cascade8(dataX, filtArray);

     // *** do something with the DataY here
     }
```

C

```
int16_t DSP_FilterIIRBQ16_cascade8(int16_t Xin, PARM_EQUAL_FILTER * pFilter_Array);
```

DSP_FilterIIRBQ16_cascade8_fast Function

Performs a single-sample IIR Biquad Filter as a cascade of 8 series filters.

Description

Function DSP_FilterIIRBQ16_cascade8_fast:

int16_t DSP_FilterIIRBQ16_cascade8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter_Array);

Calculates a single pass IIR biquad cascade filter on Xin, and delivers the result as a 16-bit output. The cascade of filters is 8 unique biquad filters arranged in series such that the output of one is provided as the input to the next. A unique filter coefficient set is provided to each, and 16 bit delay lines are maintained for each. All math is performed using 16 bit instructions, which results rounded to 16-bits for the output. All values are fractional Q15, see data structure for specifics. The function will saturate the output should it exceed maximum or minimum values.

Y = Y7 <- Y6 <- Y5 <- Y4 <- Y3 <- Y2 <- Y1 <- Y0 where each Yn filter element represents a unique IIR biquad: Yn = Y(n-1)*b0 + (b1 * Y(n-2)) + (b2 * Y(n-3)) - (a1 * Yn(-1)) - (a2 * Yn(-2)) and: for Y0; Y(n-1) = Xin(0)

Preconditions

Delay register values should be initialized to zero.

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER_16 filtArray[8];
uint16_t dataY, dataX;

// example to use 2 filter blocks as notch filters
// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArray[i].Z[0]=0;
    filtArray[i].Z[1]=0;

    // note all coefs are half value of original design, gain handled in algorithm
    // all pass
    filtArray[i].b[0]=0x4000;
    filtArray[i].b[1]=0;        // feed forward b1 coef
    filtArray[i].b[2]=0;  // feed forward b2 coef

    filtArray[i].a[0]=0;  // feedback a1 coef
    filtArray[i].a[1]=0;  // feedback a2 coef
}
    // Unique filters for example
    // 10KHz notch filter -- divide coefs by 2
    b0 = 0.5883783602332997
    b1 = -0.17124071441396285
    b2 = 0.5883783602332997
    a1 = -0.17124071441396285
    a2 = 0.1767567204665992

    // note all coefs are half value of original design, gain handled in algorithm
    filtArray[3].b[0]=0x25a7;           // feed forward b0 coef
    filtArray[3].b[1]=0xf508;           // feed forward b1 coef
    filtArray[3].b[2]=0x25a7;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
```

```
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArray[3].a[0]=0xf508;           // feedback a1 coef
    filtArray[3].a[1]=0x0b4f;           // feedback a2 coef

    // 1 KHz notch filter -- divide coefs by 2
    b0 = 0.9087554064944908
    b1 = -1.7990948352036205
    b2 = 0.9087554064944908
    a1 = -1.7990948352036205
    a2 = 0.8175108129889816

    // note all coefs are half value of original design, gain handled in algorithm
    filtArray[7].b[0]=0x3a29;           // feed forward b0 coef
    filtArray[7].b[1]=0x8cdc;           // feed forward b1 coef
    filtArray[7].b[2]=0x3a29;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArray[7].a[0]=0x8cdc;           // feedback a1 coef
    filtArray[7].a[1]=0x3452;           // feedback a2 coef

for (i=0;i<256;i++)
    {
     // *** get input data here
     dataX = compound_300_1K_hz16[i];

     dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);

     // *** do something with the DataY here
     }
```

C

```
int16_t DSP_FilterIIRBQ16_cascade8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter_Array);
```

DSP_FilterIIRBQ16_fast Function

Performs a single-pass IIR Biquad Filter.

Description

Function DSP_FilterIIRBQ16_fast:

int16_t DSP_FilterIIRBQ16_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 16 bit instructions, with results rounded to 16-bits for the output. The delay register is stored as a 16-bit value for subsequent functions. The function will saturate the results if maximum or minimum fractional values are exceeded. All values are fractional Q15 format.

Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))

Preconditions

Delay register values should be initialized to zero.

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter

designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER_16 *ptrFilterEQ;
PARM_EQUAL_FILTER_16 FilterEQ;
uint16_t DataIn, DataOut;
ptrFilterEQ = &FilterEQ;

// 48KHz sampling; 1 KHz bandpass filter; Q=0.9
//  divide by 2 and convert to Q15
//   b0 = 0.06761171785499065
//   b1 = 0
//   b2 = -0.06761171785499065
//   a1 = -1.848823142275648
//   a2 = 0.8647765642900187

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x0453;       // feed forward b0 coef
ptrFiltEQ32->b[1]=0;            // feed forward b1 coef
ptrFiltEQ32->b[2]=0xFBAD;       // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x89AD;       // feedback a1 coef
ptrFiltEQ32->a[1]=0x3758;       // feedback a2 coef

for (i=0;i<256;i++)
    {
    // *** get some input data here
       DataIn32 = three_hundred_hz[i];

       DataOut = DSP_FilterIIRBQ16_fast(DataIn, ptrFilterEQ);

     // *** do something with the DataOut here
     }
```

C

```
int16_t DSP_FilterIIRBQ16_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter);
```

DSP_FilterIIRBQ16_parallel8 Function

Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.

Description

Function DSP_FilterIIRBQ16_parallel8:

int16_t DSP_FilterIIRBQ16_parallel8(int16_t Xin, PARM_EQUAL_FILTER *pFilter);

Calculates a 8 parallel, single pass IIR biquad filters on Xin, sums the result and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, which results truncated to 16-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation. All values are fractional Q15 and Q31, see data structure for specifics.

Y = Y7/8 + Y6/8 + Y5/8 + Y4/8 + Y3/8 + Y2/8 + Y1/8 + Y0/8 where each Yn filter element represents a unique IIR biquad: Yn = X(0)*b0 + (b1 * X(n-1)) + (b2 * X(n-2)) - (a1 * Yn(-1)) - (a2 * Yn(-2))

Preconditions

Delay register values should be initialized to zero. The sum of all fracgain should be <= 1

Parameters

Parameters	Description
Xin	input data element X (int16_t)

pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER filtArrayPara[8];
uint16_t dataY, dataX;

// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArrayPara[i].Z[0]=0;
    filtArrayPara[i].Z[1]=0;

    filtArrayPara[i].G.fracGain = 0x7FFF;       // gain = 1 default
    filtArrayPara[i].G.expGain = 1;             // == 2^N; gain of 2

    // note all coefs are half value of original design, gain handled in algorithm
    // none pass -- default
    filtArrayPara[i].b[0]=0;        // feed forward b0 coef
    filtArrayPara[i].b[1]=0;        // feed forward b1 coef
    filtArrayPara[i].b[2]=0;        // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[i].a[0]=0;        // feedback a1 coef
    filtArrayPara[i].a[1]=0;        // feedback a2 coef
 }

    // 1K bandpass Q=0.9
    filtArrayPara[7].G.fracGain = 0x4000;   // gain = 0.5 because using 2 outputs
    // note all coefs are half value of original design, gain handled in algorithm
    filtArrayPara[7].b[0]=0x04ad;
    filtArrayPara[7].b[1]=0;                // feed forward b1 coef
    filtArrayPara[7].b[2]=0xfb53;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[7].a[0]=0x8a90;           // feedback a1 coef
    filtArrayPara[7].a[1]=0x36a4;           // feedback a2 coef

    // 300 Hz bandpass    Q=0.9
    filtArrayPara[6].G.fracGain = 0x1000;   // gain = 0.125 as an example
    // note all coefs are half value of original design, gain handled in algorithm
    filtArrayPara[6].b[0]=0x017b;           // feed forward b0 coef
    filtArrayPara[6].b[1]=0;                // feed forward b1 coef
    filtArrayPara[6].b[2]=0xfe85;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[6].a[0]=0x8316;           // feedback a1 coef
    filtArrayPara[6].a[1]=0x3d08;           // feedback a2 coef

for (i=0;i<256;i++)
    {
     // *** get input data here
     dataX = compound_300_1K_hz16[i];

     dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);
```

```
    // *** do something with the DataY here
    }
```

C

```
int16_t DSP_FilterIIRBQ16_parallel8(int16_t Xin, PARM_EQUAL_FILTER * pFilter);
```

DSP_FilterIIRBQ16_parallel8_fast Function

Performs a 8 parallel single-pass IIR Biquad Filters, and sums the result.

Description

Function DSP_FilterIIRBQ16_parallel8_fast:

int16_t DSP_FilterIIRBQ16_parallel8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 *pFilter);

Calculates a 8 parallel, single pass IIR biquad filters on Xin, sums the result and delivers the result as a 16-bit output. All math is performed using 16 bit instructions, which results rounded to 16-bits for the output. The delay register is stored as a 16-bit value for subsequent functions. Output is tuned by 2 multiplier factors. First each parallel section has a fractional gain (attenuation) that enables individual scaling of that section. Second, a global binary (log2N) gain is applied to the result. The combination of gain factors enable both gain and attenuation. All values are fractional Q15. The function will round outputs and saturate if maximum or minimum values are exceeded.

Y = Y7/8 + Y6/8 + Y5/8 + Y4/8 + Y3/8 + Y2/8 + Y1/8 + Y0/8 where each Yn filter element represents a unique IIR biquad: Yn = X(0)*b0 + (b1 * X(n-1)) + (b2 * X(n-2)) - (a1 * Yn(-1)) - (a2 * Yn(-2))

Preconditions

Delay register values should be initialized to zero. The sum of all fracgain should be <= 1

Parameters

Parameters	Description
Xin	input data element X (int16_t)
pFilter	pointer to filter coef and delay structure

Returns

Sample output Y (int16_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER_16 filtArrayPara[8];      // note change in data structure
uint16_t dataY, dataX;

// fill entire Filter Array with coefs
for (i=0;i<8;i++)
{
    filtArrayPara[i].Z[0]=0;
    filtArrayPara[i].Z[1]=0;

    filtArrayPara[i].G.fracGain = 0x7FFF;       // gain = 1 default
    filtArrayPara[i].G.expGain = 1;             // log2N; gain of 2

    // note all coefs are half value of original design, gain handled in algorithm
    // none pass -- default
    filtArrayPara[i].b[0]=0;        // feed forward b0 coef
    filtArrayPara[i].b[1]=0;        // feed forward b1 coef
    filtArrayPara[i].b[2]=0;        // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
```

```
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[i].a[0]=0;        // feedback a1 coef
    filtArrayPara[i].a[1]=0;        // feedback a2 coef
 }

    // 1K bandpass Q=0.9
    filtArrayPara[7].G.fracGain = 0x4000;   // gain = 0.5 because using 2 outputs
    // note all coefs are half value of original design, gain handled in algorithm
    filtArrayPara[7].b[0]=0x04ad;
    filtArrayPara[7].b[1]=0;                // feed forward b1 coef
    filtArrayPara[7].b[2]=0xfb53;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[7].a[0]=0x8a90;           // feedback a1 coef
    filtArrayPara[7].a[1]=0x36a4;           // feedback a2 coef

    // 300 Hz bandpass    Q=0.9
    filtArrayPara[6].G.fracGain = 0x1000;   // gain = 0.125 as an example
    // note all coefs are half value of original design, gain handled in algorithm
    filtArrayPara[6].b[0]=0x017b;           // feed forward b0 coef
    filtArrayPara[6].b[1]=0;                // feed forward b1 coef
    filtArrayPara[6].b[2]=0xfe85;           // feed forward b2 coef

    // note all coefs are half value of original design, gain handled in algorithm
    // note subtract is handled in algorithm, so coefs go in at actual value
    filtArrayPara[6].a[0]=0x8316;           // feedback a1 coef
    filtArrayPara[6].a[1]=0x3d08;           // feedback a2 coef

for (i=0;i<256;i++)
    {
     // *** get input data here
     dataX = compound_300_1K_hz16[i];

     dataY = DSP_FilterIIRBQ16_cascade8_fast(dataX, filtArray);

     // *** do something with the DataY here
     }
```

C

```
int16_t DSP_FilterIIRBQ16_parallel8_fast(int16_t Xin, PARM_EQUAL_FILTER_16 * pFilter);
```

DSP_FilterIIRBQ32 Function

Performs a high resolution single-pass IIR Biquad Filter.

Description

Function DSP_FilterIIRBQ32:

int32_t DSP_FilterIIRBQ32(int32_t Xin, PARM_EQUAL_FILTER_32 *pFilter);

Calculates a single pass IIR biquad filter on Xin, and delivers the result as a 16-bit output. All math is performed using 32 bit instructions, with results truncated to 32-bits for the output. The delay register is stored as a 32-bit value for subsequent functions. All values are fractional Q31, see data structure for specifics.

Y = X(0)*b0 + (b1 * X(-1)) + (b2 * X(-2)) - (a1 * Y(-1)) - (a2 * Y(-2))

Preconditions

Delay register values should be initialized to zero.

Parameters

Parameters	Description
Xin	input data element X (int32_t)
pFilter	pointer to high resolution filter coef and delay structure

Returns

Sample output Y (int32_t)

Remarks

The delay register values should be initialized to zero prior to the first call to the function, they are updated each pass. A gain of 2 has been hard coded into the function. This implies that all coefs should be input at half value. This is purposeful, since many filter designs need a div2 to have each coef between the required -1

Example

```
PARM_EQUAL_FILTER_32 *ptrFiltEQ32;
PARM_EQUAL_FILTER_32 FilterEQ32;
int32_t DataIn32, DataOut32;

ptrFiltEQ32 = &FilterEQ32;

ptrFiltEQ32->Z[0]=0;
ptrFiltEQ32->Z[1]=0;

//  1000 Hz Q= 0.9 BP filter design, 44.1K sampling
//
//    b0 = 0.07311778239751009  forward
//    b1 = 0
//    b2 = -0.07311778239751009
//    a1 = -1.8349811166056893  back
//    a2 = 0.8537644352049799

// note all coefs are half value of original design, gain handled in algorithm
ptrFiltEQ32->b[0]=0x04ADF635;   // feed forward b0 coef
ptrFiltEQ32->b[1]=0;            // feed forward b1 coef
ptrFiltEQ32->b[2]=0xFB5209CB;   // feed forward b2 coef

// note all coefs are half value of original design, gain handled in algorithm
// note subtract is handled in algorithm, so coefs go in at actual value
ptrFiltEQ32->a[0]=0x8A8FAB5D;  // feedback a1 coef
ptrFiltEQ32->a[1]=0x36A41395;  // feedback a2 coef

for (i=0;i<256;i++)
    {
     // *** get input data here
       DataIn32 = three_hundred_hz[i];

       DataOut32 = DSP_FilterIIRBQ32(DataIn32, ptrFiltEQ32);

     // *** do something with the DataOut32 here
     }
```

C

```
int32_t DSP_FilterIIRBQ32(int32_t Xin, PARM_EQUAL_FILTER_32 * pFilter);
```

DSP_FilterIIRSetup16 Function

Converts biquad structure to coeffs array to set up IIR filter.

Description

Function DSP_FilterIIRSetup16:

void DSP_FilterIIRSetup16(int16_t *coeffs, biquad16 *bq, int B);

Converts an array of biquad coefficients, bq, into an linear array of coefficients, coeffs. The output array must be 4*B elements long. The number of biquads in the resulting factor is given by B. All numerical values must be in Q15 fractional data format.

Preconditions

coeffs must have 4*B elements.

Parameters

Parameters	Description
coeffs	pointer to an array of coefficients (int16_t)
bq	pointer to array of biquad structure filter coefs (biquad16)
B	number of cascaded biquad filter groups to process (int)

Returns

None.

Remarks

None.

Example

```
see DSP_FilterIIR16 for example.
```

C

```
void DSP_FilterIIRSetup16(int16_t * coeffs, biquad16 * bq, int B);
```

DSP_FilterLMS16 Function

Performs a single sample Least Mean Squares FIR Filter.

Description

Function DSP_FilterLMS16:

int16_t DSP_FilterLMS16(int16_t in, int16_t ref, int16_t *coeffs, int16_t *delayline, int16_t *error, int K, int16_t mu);

Computes an LMS adaptive filter on the input in. Filter output of the FIR is given as a 16 bit value. The filter target is ref, and the calculation difference between the output and the target is error. The filter adapts its coefficients, coefs, on each pass. The number of coefficients (filter taps) is given by the value K. The delayline array should be initialized to zero prior to calling the filter for the first time, and have K elements. The value mu is the rate at which the filter adapts. All values are Q15 fractional numbers. The function will saturate the output if it exceeds maximum or minimum values. The LMS will adapt its coefs to attempt to drive the output value toward the ref value. The rate of adaption on each pass depends on mu and the error from the previous calculation.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. delayline must have (K) elements, and be initialized to zero. K (taps) must be a multiple of 4, and >= 8. mu must be positive.

Parameters

Parameters	Description
in	input data value (int16_t)
ref	target output value (int16_t)
coeffs	pointer to an array of coefficients (int16_t)
delayline	pointer to an array of delay variables (int16_t)
error	output minus reference (int16_t)
K	number of filter taps and coeffs (int)
mu	adaption rate (int16_t)

Returns

(int16_t) - FIR filter output

Remarks

Filter coefs may start at random or zero value, but convergence is dependent on the amount of update required.

Example

```
#define lmsTAPS 8

int16_t lmsOut;
int lmsTaps = lmsTAPS;

int16_t lmsCoefs[lmsTAPS]={0x5000, 0x4000,0x3000, 0x2000,0x1000, 0x0000,0xF000, 0xE000};
int16_t lmsDelay[lmsTAPS]={0};
int16_t *ptrLMSError;
int16_t lmsError = 0x0200;
int16_t inVal=0;

int16_t refVal = 0x0CCC;  // some target value = 0.1
int16_t lmsAdapt = 0x3000;

ptrLMSError = &lmsError;

for (i=0;i<200;i++)
{
// get some input value here //
 if (i < 100)
 {
     inVal = 0x4233;
 }
 else
 {
     inVal = 0xCF10;
 }

 lmsOut =  DSP_FilterLMS16(inVal, refVal, lmsCoefs, lmsDelay,
       ptrLMSError, lmsTaps, lmsAdapt);
}
```

C

```
int16_t DSP_FilterLMS16(int16_t in, int16_t ref, int16_t * coeffs, int16_t * delayline, int16_t
* error, int K, int16_t mu);
```

DSP_MatrixAdd32 Function

Addition of two matrices C = (A + B).

Description

Function DSP_MatrixAdd32:

void DSP_MatrixAdd32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);

Vector summation of two matrices, where both have 32-bit integer elements. The resulting output will saturate if element addition exceeds MAX32 or MIN32.

Preconditions

Both matrices must be equivalent in rows and columns. Both Matrices must be set into structure (ROWS, COLUMNS, vector_pointer).

Parameters

Parameters	Description
resMat	pointer to new sum Matrix C (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

Returns

None.

Remarks

Execution Time (cycles): 225 cycles + 23 / matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Example

```
#define ROW 2
#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {1,2,3,4};
int32_t matB[ROW*COL] = {2,4,6,8};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

srcMat2->col=COL;
srcMat2->row=ROW;
srcMat2->pMatrix=matB;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixAdd32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] + matB[i] = {3,6,9,0xA}
```

C

```
void DSP_MatrixAdd32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

DSP_MatrixEqual32 Function

Equality of two matrices C = (A).

Description

Function DSP_MatrixEqual32:

void DSP_MatrixEqual32(matrix32 *resMat, matrix32 *srcMat);

Vector copy of all elements from one matrix to another. C is a duplicate of A.

Preconditions

None.

Parameters

Parameters	Description
resMat	pointer to completed new Matrix C (*int32_t)
srcMat	pointer to the Matrix A structure (*int32_t)

Returns

None.

Remarks

Execution Time (cycles): 163 cycles + 12 / matrix_element.

Example

```
#define ROW 2
#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {5,2,-3,8};

matrix32 mat, mat2;
resMat=&mat;
srcMat1=&mat2;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixEqual32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] = {5, 2, -3, 8}
```

C

```
void DSP_MatrixEqual32(matrix32 * resMat, matrix32 * srcMat);
```

DSP_MatrixInit32 Function

Initializes the first N elements of a Matrix to the value num.

Description

Function DSP_MatrixInit32:

void DSP_MatrixInit32(int32_t *data_buffer, int N, int32_t num);

Copy the value num into the first N Matrix elements of data_buffer.

Preconditions

data_buffer must be predefined to be equal to or greater than N elements. N must be a factor of four, or it will truncate to the nearest factor of four.

Parameters

Parameters	Description
data_buffer	pointer to the Matrix to be initialized (int32_t[M*N])
N	number of elements to be initialized (int32_t)
num	value to be initialized into the matrix (int32_t)

Returns

None.

Remarks

None.

Example

```
#define ROW 3
```

```
#define COL 3

int32_t numElements = 4;  // multiple of 4
int valueElements = -1;

int32_t matA[ROW*COL] = {5,2,-3,8,4,2,-6,8,9};

DSP_MatrixInit32(matA, numElements, valueElements);

// matA[i] = {-1,-1,-1,-1,4,2,-6,8,9}
```

C

```
void DSP_MatrixInit32(int32_t * data_buffer, int N, int32_t num);
```

DSP_MatrixMul32 Function

Multiplication of two matrices C = A x B.

Description

Function DSP_MatrixMul32:

void DSP_MatrixMul32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);

Multiplication of two matrices, with inputs and outputs being in fractional Q31 numerical format. The output elements will saturate if the dot product exceeds maximum or minimum fractional values.

Preconditions

Matrices must be aligned such that columns of A = rows of B. resMat must have the format of rows of A, columns of B. All Matrices must be set into structure (ROWS, COLUMNS, vector_pointer).

Parameters

Parameters	Description
resMat	pointer to different Matrix C structure (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

Returns

None.

Remarks

Execution Time (cycles): 319 cycles + 38 / output matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Example

```
#define ROW1 3
#define COL1 2
#define ROW2 2
#define COL2 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW1*COL2];

int32_t test_MatrixA[ROW1*COL1]=
{
    0x40000000,0x20000000,      // 0.5,  0.25
    0xD999999A,0x4CCCCCCC,      // -0.3, 0.6
    0xC0000000,0x0CCCCCCD       // -0.5  0.1
};

int32_t test_MatrixB[ROW2*COL2]=
{
    0x40000000,0x20000000,      // 0.5, 0.25
```

```
    0x0CCCCCCD,0xCCCCCCCD       // 0.1, -0.4
};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW1;
srcMat1->col=COL1;
srcMat1->pMatrix=test_MatrixA;

srcMat2->col=COL2;
srcMat2->row=ROW2;
srcMat2->pMatrix=test_MatrixB;

resMat->row=ROW1;  // note resulting matrix MUST have ROW1 & COL2 format
resMat->col=COL2;
resMat->pMatrix=result;

DSP_MatrixMul32(resMat, srcMat1, srcMat2);

// result[] = matA[] x matB[] =
//    { 0x23333333, 0x03333333   // 0.275, 0.025
//      0xF47AE147, 0xD7AE147B   // -0.9, -0.315
//      0xE147AE14, 0xEAE147AE } // -0.24, -0.165
```

C

```
void DSP_MatrixMul32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

DSP_MatrixScale32 Function

Scales each element of an input buffer (matrix) by a fixed number.

Description

Function DSP_MatrixScale32:

void DSP_MatrixScale32(int32_t *data_buffer, int N, int32_t num);

Multiply the first N elements of an input buffer by a fixed scalar num. The resulting value is stored back into the input buffer. N number total samples of the input buffer are processed. All values are in Q31 fractional integer format. The result of calculations is saturated to the MAX32 or MIN32 value if exceeded.

Preconditions

data_buffer must be predefined to be equal to or greater than N elements. N must be a factor of four, or will truncate to the nearest factor of four.

Parameters

Parameters	Description
data_buffer	pointer to the Matrix to be initialized (int32_t[M*N])
N	number of elements to be initialized (int)
num	value to be initialized into the matrix (int32_t)

Returns

None.

Remarks

Execution time (cycles): 190 + 9 cycles / element, typical.

Example

```
int32_t numScale = 0x40000000;  // 0.5
int valN = 12;
```

```
int32_t inputBufScale[12] = {0x40000000, 0x40000000, 0x20000000, 0x20000000,
            0x19999999, 0xCCCCCCCD, 0xF3333333, 0x80000000,
            0x7FFFFFFF, 0x00000000, 0x40000000, 0x70000000 };
// 0.5, 0.5, 0.25, 0.25, 0.25, 0.2, -0.4, -0.1, -1, 1, 0, 0.5, 0.875

DSP_MatrixScale32(inputBufScale,valN,numScale);

// inputBufScale[i] = {0x20000000, 0x20000000, 0x10000000, 0x10000000,
//          0x0CCCCCCC, 0xE6666666, 0xF9999999, 0xC0000000,
//          0x3FFFFFFF, 0x00000000, 0x20000000, 0x38000000}
//  0.25, 0.25, 0.125, 0.125, 0.1, -0.2, -0.05, -0.5, 0.5, 0, 0.25, 0.4375
```

C

```c
void DSP_MatrixScale32(int32_t * data_buffer, int N, int32_t num);
```

DSP_MatrixSub32 Function

Subtraction of two matrices C = (A - B).

Description

Function DSP_MatrixSub32:

void DSP_MatrixSub32(matrix32 *resMat, matrix32 *srcMat1, matrix32 *srcMat2);

Vector subtraction of two matrices, where both have 32-bit integer elements. The resulting output will saturate if element addition exceeds MAX32 or MIN32.

Preconditions

Both matrices must be equivalent in rows and columns. All Matrices must be set into structure (ROWS, COLUMNS, vector_pointer)

Parameters

Parameters	Description
resMat	pointer to different Matrix C structure (*int32_t)
srcMat1	pointer to the Matrix A structure (*int32_t)
srcMat2	pointer to the Matrix B structure (*int32_t)

Returns

None.

Remarks

Execution Time (cycles): 222 cycles + 21 / matrix_element. The function will saturate the output value if it exceeds maximum limits per element.

Example

```c
#define ROW 2
#define COL 2

matrix32 *resMat, *srcMat1, *srcMat2;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = {5,2,-3,8};
int32_t matB[ROW*COL] = {2,2,2,2};

matrix32 mat, mat2, mat3;
resMat=&mat;
srcMat1=&mat2;
srcMat2=&mat3;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;
```

```
srcMat2->col=COL;
srcMat2->row=ROW;
srcMat2->pMatrix=matB;

resMat->row=ROW;
resMat->col=COL;
resMat->pMatrix=result;

DSP_MatrixSub32(resMat, srcMat1, srcMat2);

// result[i] = matA[i] - matB[i] = {3,0,-5,6}
```

C

```
void DSP_MatrixSub32(matrix32 * resMat, matrix32 * srcMat1, matrix32 * srcMat2);
```

DSP_MatrixTranspose32 Function

Transpose of a Matrix C = A (T).

Description

Function DSP_MatrixTranspose32:

void DSP_MatrixTranspose32(matrix32 *desMat, matrix32 *srcMat);

Transpose of rows and columns of a matrix.

Preconditions

Matrix definitions for ROWS and COLS must be transposed prior to the function call.

Parameters

Parameters	Description
desMat	pointer to transposed new Matrix C (*int32_t)
srcMat	pointer to the Matrix A structure (*int32_t)

Returns

None.

Remarks

Execution Time (cycles): 210 cycles + 10 / matrix_element.

Example

```
#define ROW 3
#define COL 4

matrix32 *resMat, *srcMat1;
int32_t result[ROW*COL];

int32_t matA[ROW*COL] = { 1,  2,  3,  4,
                          5,  6,  7,  8,
                         -1, -3, -5, -7};

matrix32 mat, mat2;
resMat=&mat;
srcMat1=&mat2;

srcMat1->row=ROW;
srcMat1->col=COL;
srcMat1->pMatrix=matA;

resMat->row=COL;   // note the shift in columns and rows
resMat->col=ROW;
resMat->pMatrix=result;
```

```
DSP_MatrixTranspose32(resMat, srcMat1);

// result[] = matA(T)[] = {   1,  5, -1,
//                            2,  6, -3,
//                            3,  7, -5,
//                            4,  8, -7}
```

C

```
void DSP_MatrixTranspose32(matrix32 * desMat, matrix32 * srcMat);
```

DSP_TransformFFT16 Function

Creates an Fast Fourier Transform (FFT) from a time domain input.

Description

Function DSP_TransformFFT16:

void DSP_TransformFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);

Performs an complex FFT on the input, din, and stores the complex result in dout. Performs 2^log2N point calculation, and the working buffer scratch as well as the input and output must be 2^log2N in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of DSP_TransformFFT16_setup. All values are 16 bit (Q15) fractional.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as 2^(log2N) log2N must be >= 3 FFT factors must be calculated in advance, use DSP_TransformFFT16_setup

Parameters

Parameters	Description
dout	pointer to complex output array (int16c)
din	pointer to complex input array (int16c)
twiddles	pointer to an complex array of factors (int16c)
scratch	pointer to a complex scratch pad buffer (int16c)
log2N	binary exponent of number of samples (int)

Returns

None.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. Din is a complex number array, but may be loaded solely with real numbers if no phase information is available.

Example

```
int log2N = 8;  // log2(256) = 8
int fftSamples = 256;

int16c  *fftDin;
int16c  fftDout[fftSamples];
int16c  scratch[fftSamples];
int16c  fftCoefs[fftSamples];

int16c *fftc;
fftc = &fftCoefs;

DSP_TransformFFT16_setup(fftc, log2N);  // call setup function

while (1)
{
    fftDin = &fftin_8Khz_long_window16;  // get 256 point complex data
```

```
    DSP_TransformFFT16(fftDout, fftDin, fftc, scratch, log2N);

    // do something with the output, fftDout
};
```

C

```
void DSP_TransformFFT16(int16c * dout, int16c * din, int16c * twiddles, int16c * scratch, int
log2N);
```

DSP_TransformFFT16_setup Function

Creates FFT coefficients for use in the FFT16 function.

Description

Function DSP_TransformFFT16_setup:

void DSP_TransformFFT16_setup(int16c *twiddles, int log2N);

Calculates the N twiddle factors required to operate the FFT16 function. These factors are done in serial fashion, and require considerable processing power. Ideally this function would be run only once prior to an ongoing FFT, and the results held in a buffer.

Preconditions

twiddles must be N in length N is calculated (2^log2N)

Parameters

Parameters	Description
twiddles	pointer to a complex array of factors (int16c)
log2N	binary exponent of number of data points (int)

Returns

None.

Remarks

This function is of considerable length and executed in C. It is recommended it only be called once for any given FFT length in time sensitive applications.

Example

```
see DSP_TransformFFT16 for example.
```

C

```
void DSP_TransformFFT16_setup(int16c * twiddles, int log2N);
```

DSP_TransformFFT32 Function

Creates an Fast Fourier Transform (FFT) from a time domain input.

Description

Function DSP_TransformFFT32:

void DSP_TransformFFT32(int32c *dout, int32c *din, int32c *twiddles, int32c *scratch, int log2N);

Performs an complex FFT on the input, din, and stores the complex result in dout. Performs 2^log2N point calculation, and the working buffer scratch as well as the input and output must be 2^log2N in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of DSP_TransformFFT16_setup. All values are 16 bit (Q31) fractional.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as 2^(log2N) log2N must be >= 3 FFT factors must be calculated in advance, use DSP_TransformFFT32_setup

Parameters

Parameters	Description
dout	pointer to complex output array (int32c)
din	pointer to complex input array (int32c)
twiddles	pointer to an complex array of FFT factors (int32c)
scratch	pointer to a complex scratch pad buffer (int32c)
log2N	binary exponent of number of samples (int)

Returns

None.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. Din is a complex number array, but may be loaded solely with real numbers if no phase information is available.

Example

```
int log2N = 8;  // log2(256) = 8
int fftSamples = 256;

int32c  *fftDin;
int32c  fftDout[fftSamples];
int32c  scratch[fftSamples];
int32c  fftCoefs[fftSamples];

int32c *fftc;
fftc = &fftCoefs;

DSP_TransformFFT32_setup(fftc, log2N);  // call setup function

while (1)
{
    fftDin = &fftin_5Khz_long_window32;  // get 256 point complex data

    DSP_TransformFFT32(fftDout, fftDin, fftc, scratch, log2N);

    // do something with the output, fftDout
};
```

C

```
void DSP_TransformFFT32(int32c * dout, int32c * din, int32c * twiddles, int32c * scratch, int
log2N);
```

DSP_TransformFFT32_setup Function

Creates FFT coefficients for use in the FFT32 function.

Description

Function DSP_TransformFFT32_setup:

void DSP_TransformFFT32_setup(int32c *twiddles, int log2N);

Calculates the N FFT twiddle factors required to operate the FFT32 function. These factors are done in serial fashion, and require considerable processing power. Ideally this function would be run only once prior to an ongoing FFT, and the results held in a buffer.

Preconditions

twiddles must be N in length N is calculated (2^log2N)

Parameters

Parameters	Description
twiddles	pointer to a complex array of coefficients (int32c)
log2N	binary exponent of number of data points (int)

Returns

None.

Remarks

This function is of considerable length and executed in C. It is recommended it only be called once for any given FFT length in time sensitive applications.

Example

```
see DSP_TransformFFT32 for example.
```

C

```
void DSP_TransformFFT32_setup(int32c * twiddles, int log2N);
```

DSP_TransformIFFT16 Function

Creates an Inverse Fast Fourier Transform (FFT) from a frequency domain input.

Description

Function DSP_TransformIFFT16:

void DSP_TransformIFFT16(int16c *dout, int16c *din, int16c *twiddles, int16c *scratch, int log2N);

Performs an complex Inverse FFT on the input, din, and stores the complex result in dout. Performs 2^log2N point calculation, and the working buffer scratch as well as the input and output must be 2^log2N in length. Coefficient twiddle factors come from twiddles, and may be loaded with the use of DSP_TransformFFT16_setup. All values are 16 bit (Q15) fractional.

Preconditions

din, dout, twiddles and scratch must have N elements N is calculated as 2^(log2N) log2N must be >= 3 FFT factors must be calculated in advance, use DSP_TransformFFT16_setup

Parameters

Parameters	Description
dout	pointer to complex output array (int16c)
din	pointer to complex input array (int16c)
twiddles	pointer to an complex array of factors (int16c)
scratch	pointer to a complex scratch pad buffer (int16c)
log2N	binary exponent of number of samples (int)

Returns

None.

Remarks

Scratch must be declared but need not be initialized. Din may be aided with a window function prior to calling the FFT, but is not required. A very similar function to the FFT is executed for the inverse FFT. This requires twiddle factors set in advance with the same method as used in the FFT. Complex conjugate and scaling are handled within the algorithm. The output is scaled using binary shifting based on log2N. Since the algorithm reduces the output by a scale factor of log2N, the resolution is reduced proportionally to the number of data points.

Example

```
int ilog2N = 10;  // log2(64) = 6; log2(256) = 8; log2(1024) = 10;
int ifftSamples = pow(2,ilog2N);
```

```
int16c  *ifftDin;
int16c ifftDout[ifftSamples];
int16c  iscratch[ifftSamples];
int16c  ifftCoefs[ifftSamples];
int16c  ifftTimeOut[ifftSamples];

// set up twiddle factors, these are used for both FFT and iFFT

int16c *ifftc;
ifftc = &ifftCoefs;
DSP_TransformFFT16_setup( ifftc, ilog2N);  // call to coef setup

// in this example, we take an FFT of an original time domain (sine wave)
// the output of the FFT is used as the input of the iFFT for comparison

ifftDin = &fftin_800hz_verylong16;
DSP_TransformFFT16(ifftDout, ifftDin, ifftc, iscratch, ilog2N);

// ifftDout = frequency domain output, complex number array

DSP_TransformIFFT16(ifftTimeOut, ifftDout, ifftc, iscratch, ilog2N);

// do something with the output, fftTimeOut, time domain
```

C

```
void DSP_TransformIFFT16(int16c * dout, int16c * din, int16c * twiddles, int16c * scratch, int
log2N);
```

DSP_TransformWindow_Bart16 Function

Perform a Bartlett window on a vector.

Description

Function DSP_TransformWindow_Bart16:

void DSP_TransformWindow_Bart16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Bartlett (Triangle) Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
```

```
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Bart16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x1000, 0x2000, 0x3000, 0x4000, 0x3000, 0x2000, 0x1000
```

C

```
void DSP_TransformWindow_Bart16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Bart32 Function

Perform a Bartlett window on a vector.

Description

Function DSP_TransformWindow_Bart32:

void DSP_TransformWindow_Bart32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Bartlett (Triangle) Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Bart32(OutVector32, InVector32, WindowN);
// OutWindow = 0x0, 0x10000000, 0x20000000, 0x30000000, 0x40000000,
//                       0x30000000, 0x20000000, 0x10000000
```

C

```
void DSP_TransformWindow_Bart32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWindow_Black16 Function

Perform a Blackman window on a vector.

Description

Function DSP_TransformWindow_Black16:

void DSP_TransformWindow_Black16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Blackman Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Black16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0071, 0x0665, 0x1DF1, 0x3B00, 0x3B00, 0x1DF1, 0x0665, 0x0071
```

C

```
void DSP_TransformWindow_Black16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Black32 Function

Perform a Blackman window on a vector.

Description

Function DSP_TransformWindow_Black32:

void DSP_TransformWindow_Black32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Blackman Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Black32(OutVector32, InVector32, WindowN);
// OutWindow = 0x0070B490, 0x06649680, 0x1DF13240, 0x3B003D80, 0x3B003D80,
//                       0x1DF13240, 0x06649680, 0x0070B490
```

C

```
void DSP_TransformWindow_Black32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWindow_Cosine16 Function

Perform a Cosine (Sine) window on a vector.

Description

Function DSP_TransformWindow_Cosine16:

void DSP_TransformWindow_Cosine16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Cosine Window on the first N samples of the input vector, InVector. The output is stored in OutWindow. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
```

```
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Cosine16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x1BC5, 0x320A, 0x3E65, 0x3E65, 0x320A, 0x1BC5, 0x0071
```

C

```
void DSP_TransformWindow_Cosine16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Cosine32 Function

Perform a Cosine (Sine) window on a vector.

Description

Function DSP_TransformWindow_Cosine32:

void DSP_TransformWindow_Cosine32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Cosine Window on the first N samples of the input vector, InVector. The output is stored in OutWindow. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Cosine32(OutVector32, InVector32, WindowN);
// OutWindow = 0x00000000, 0x1BC4C060, 0x32098700, 0x3E653800, 0x3E653800,
//                        0x32098700, 0x1BC4C060, 0x00000000
```

C

```
void DSP_TransformWindow_Cosine32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWindow_Hamm16 Function

Perform a Hamming window on a vector.

Description

Function DSP_TransformWindow_Hamm16:

void DSP_TransformWindow_Hamm16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Hamming Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Hamm16(OutVector16, InVector16, WindowN);
// OutWindow = 0x051F, 0x0DBE, 0x228F, 0x3761, 0x4000, 0x3761, 0x228F, 0x0DBE
```

C

```
void DSP_TransformWindow_Hamm16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Hamm32 Function

Perform a Hamming window on a vector.

Description

Function DSP_TransformWindow_Hamm32:

void DSP_TransformWindow_Hamm32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Hamming Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Hamm32(OutVector32, InVector32, WindowN);
// OutWindow = 0x051EB860, 0x0DBE26C0, 0x228F5C40, 0x37609200, 0x40000000,
//                       0x37609200, 0x228F5C40, 0x0DBE26C0
```

C

```
void DSP_TransformWindow_Hamm32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWindow_Hann16 Function

Perform a Hanning window on a vector.

Description

Function DSP_TransformWindow_Hann16:

void DSP_TransformWindow_Hann16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Hanning Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Hann16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0000, 0x095F, 0x2000, 0x36A1, 0x4000, 0x36A1, 0x2000, 0x095F
```

C

```
void DSP_TransformWindow_Hann16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Hann32 Function

Perform a Hanning window on a vector.

Description

Function DSP_TransformWindow_Hann32:

void DSP_TransformWindow_Hann32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Hanning Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Hann32(OutVector32, InVector32, WindowN);
// OutWindow = 0x00000000, 0x095F61C0, 0x20000000, 0x36A09E80, 0x40000000,
//                       0x36A09E80, 0x20000000, 0x095F61C0
```

C

```
void DSP_TransformWindow_Hann32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWindow_Kaiser16 Function

Perform a Kaiser window on a vector.

Description

Function DSP_TransformWindow_Kaiser16:

void DSP_TransformWindow_Kaiser16(int16_t *OutVector, int16_t *InVector, int N);

Compute a Kaiser Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q15 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int16_t OutVector16[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector16[i]= 0x4000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Kaiser16(OutVector16, InVector16, WindowN);
// OutWindow = 0x0031, 0x0325, 0x1375, 0x304F, 0x3FCF, 0x304F, 0x1375, 0x0325
```

C

```
void DSP_TransformWindow_Kaiser16(int16_t * OutVector, int16_t * InVector, int N);
```

DSP_TransformWindow_Kaiser32 Function

Perform a Kaiser window on a vector.

Description

Function DSP_TransformWindow_Kaiser32:

void DSP_TransformWindow_Kaiser32(int32_t *OutVector, int32_t *InVector, int N);

Compute a Kaiser Window on the first N samples of the input vector, InVector. The output is stored in OutVector. Operations are performed at higher resolution and rounded for the most accuracy possible. Input and output values are in Q31 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples The functional output computes WinVector(n) = Window(n) * InVector(n)

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library.

Example

```
int32_t OutVector32[8]={0};
int  WindowN = 8;

for (i=0;i<WindowN;i++)
{
   InVector32[i]= 0x40000000;  // constant 0.5 for functional testing
}

   DSP_TransformWindow_Kaiser32(OutVector32, InVector32, WindowN);
// OutWindow = 0x003126F6, 0x032555C8, 0x1374BCA0, 0x304F66C0, 0x3FCED900,
//                       0x304F66C0, 0x1374BCA0, 0x032555C8
```

C

```
void DSP_TransformWindow_Kaiser32(int32_t * OutVector, int32_t * InVector, int N);
```

DSP_TransformWinInit_Bart16 Function

Create a Bartlett window.

Description

Function DSP_TransformWinInit_Bart16:

void DSP_TransformWinInit_Bart16(int16_t *OutWindow, int N);

Create a N-element Bartlett (Triangle) Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of

significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Bart16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Bart32 Function

Create a Bartlett window.

Description

Function DSP_TransformWinInit_Bart32:

void DSP_TransformWinInit_Bart32(int32_t *OutWindow, int N);

Create a N-element Bartlett (Triangle) Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values Q31 fractional format. The Bartlett Window follows the equation:

Window(n) = 1 - (abs(2*n - N)/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Bart32(int32_t * OutWindow, int N);
```

DSP_TransformWinInit_Black16 Function

Create a Blackman window.

Description

Function DSP_TransformWinInit_Black16:

void DSP_TransformWinInit_Black16(int16_t *OutWindow, int N);

Create a N-element Blackman Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are Q16 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is

the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Black16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Black32 Function

Create a Blackman window.

Description

Function DSP_TransformWinInit_Black32:

void DSP_TransformWinInit_Black32(int32_t *OutWindow, int N);

Create a N-element Blackman Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are Q31 fractional format. The Blackman Window follows the equation:

Window(n) = 0.42659 - 0.49656 * COS(2*Pi*n/(N-1)) + 0.076849 * COS(4*Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Black32(int32_t * OutWindow, int N);
```

DSP_TransformWinInit_Cosine16 Function

Create a Cosine (Sine) window.

Description

Function DSP_TransformWinInit_Cosine16:

void DSP_TransformWinInit_Cosine16(int16_t *OutWindow, int N);

Create a N-element Cosine Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Cosine16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Cosine32 Function

Create a Cosine (Sine) window.

Description

Function DSP_TransformWinInit_Cosine32:

void DSP_TransformWinInit_Cosine32(int32_t *OutWindow, int N);

Create a N-element Cosine Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Cosine Window follows the equation:

Window(n) = SIN(Pi*n/(N-1)) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Cosine32(int32_t * OutWindow, int N);
```

DSP_TransformWinInit_Hamm16 Function

Create a Hamming window.

Description

Function DSP_TransformWinInit_Hamm16:

void DSP_TransformWinInit_Hamm16(int16_t *OutWindow, int N);

Create a N-element Hamming Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Hamm16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Hamm32 Function

Create a Hamming window.

Description

Function DSP_TransformWinInit_Hamm32:

void DSP_TransformWinInit_Hamm32(int32_t *OutWindow, int N);

Create a N-element Hamming Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Hamming Window follows the equation:

Window(n) = 0.54 - 0.46 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Hamm32(int32_t * OutWindow, int N);
```

DSP_TransformWinInit_Hann16 Function

Create a Hanning window.

Description

Function DSP_TransformWinInit_Hann16:

void DSP_TransformWinInit_Hann16(int16_t *OutWindow, int N);

Create a N-element Hanning Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Hann16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Hann32 Function

Create a Hanning window.

Description

Function DSP_TransformWinInit_Hann32:

void DSP_TransformWinInit_Hann32(int32_t *OutWindow, int N);

Create a N-element Hanning Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Hanning Window follows the equation:

Window(n) = 0.5 - 0.5 * COS(2*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Hann32(int32_t * OutWindow, int N);
```

DSP_TransformWinInit_Kaiser16 Function

Create a Kaiser window.

Description

Function DSP_TransformWinInit_Kaiser16:

void DSP_TransformWinInit_Kaiser16(int16_t *OutWindow, int N);

Create a N-element Kaiser Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q15 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Kaiser16(int16_t * OutWindow, int N);
```

DSP_TransformWinInit_Kaiser32 Function

Create a Kaiser window.

Description

Function DSP_TransformWinInit_Kaiser32:

void DSP_TransformWinInit_Kaiser32(int32_t *OutWindow, int N);

Create a N-element Kaiser Window, and store the output to OutWindow. Operations are performed at higher resolution floating point, and rounded for the most accuracy possible. Output values are in Q31 fractional format. The Kaiser Window follows the equation:

Window(n) = 0.402 - 0.498 * COS(2*Pi*n/N) + 0.098 * cos(4*Pi*n/N) + 0.001 * cos(6*Pi*n/N) where n is the window sample number N is the total number of samples

Preconditions

N must be a positive number. OutWindow must be declared with N elements or larger.

Parameters

Parameters	Description
OutWindow	pointer to output array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This function is performed in C. The function may be optimized for the library. It is dependent on the floating point math library. The functional window is an intermediate result that needs to be multiplied by an input vector prior to FFT processing. Because of significant processing time the window need only be computed once and the multiply of the (window * input) vector done during recurring loop processing.

Example

C

```
void DSP_TransformWinInit_Kaiser32(int32_t * OutWindow, int N);
```

DSP_VectorAbs16 Function

Calculate the absolute value of a vector.

Description

Function DSP_VectorAbs16:

void DSP_VectorAbs16(int16_t *outdata, int16_t *indata, int N);

Computes the absolute value of each element of indata and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with the absolute value of elements of indata

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100,200,127,-127,-2};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAbs16(pOutdata, inBufTest, Num);
```

```
// outVal[i] = {5,2,3,4,1,0,2,8}
```

C

```
void DSP_VectorAbs16(int16_t * outdata, int16_t * indata, int N);
```

DSP_VectorAbs32 Function

Calculate the absolute value of a vector.

Description

Function DSP_VectorAbs32:

void DSP_VectorAbs32(int32_t *outdata, int32_t *indata, int N);

Computes the absolute value of each element of indata and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of abs(indata[i])

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100,200,127,-127,-2};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAbs32(pOutdata, inBufTest, Num);

// outVal[i] = {5,2,3,4,1,0,2,8}
```

C

```
void DSP_VectorAbs32(int32_t * outdata, int32_t * indata, int N);
```

DSP_VectorAdd16 Function

Calculate the sum of two vectors.

Description

Function DSP_VectorAdd16:

void DSP_VectorAdd16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);

Computes the sum value of each element of indata1 + indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q15 fractional format. outdata[i] filled with N elements of indata1[i] + indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int16_t inBuf2[16]=     { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7,  0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAdd16(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] + inBuf2[i] = {-4,4,0,8,4,6,5,0}
```

C

```
void DSP_VectorAdd16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

DSP_VectorAdd32 Function

Calculate the sum of two vectors.

Description

Function DSP_VectorAdd32:

void DSP_VectorAdd32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);

Computes the sum value of each element of indata1 + indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata1[i] + indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
N	number of samples

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int32_t inBuf2[16]=     { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7,  0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorAdd32(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] + inBuf2[i] = {-4,4,0,8,4,6,5,0}
```

C

```
void DSP_VectorAdd32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

DSP_VectorAddc16 Function

Calculate the sum of a vector and a constant.

Description

Function DSP_VectorAddc16:

void DSP_VectorAddc16(int16_t *outdata, int16_t *indata, int16_t c, int N);

Computes the sum value of each element of (indata + c) and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata[i] + c

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
c	constant value added to all indata1 elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int16_t constValue =  3;
int Num = 8;

pOutdata = &outVal;

DSP_VectorAddc16(pOutdata, inBufTest, constValue, Num);

// outVal[i] = inBufTest[i] + constValue = {-2,5,0,7,2,3,1,-5}
```

C

```
void DSP_VectorAddc16(int16_t * outdata, int16_t * indata, int16_t c, int N);
```

DSP_VectorAddc32 Function

Calculate the sum of a vector and a constant.

Description

Function DSP_VectorAddc32:

void DSP_VectorAddc32(int32_t *outdata, int32_t *indata, int32_t c, int N);

Computes the sum value of each element of (indata + c) and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata1[i] + c

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata	pointer to input array of 16-bit elements (int32_t)
c	constant value added to all indata1 elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int32_t constValue =  3;
int Num = 8;

pOutdata = &outVal;

DSP_VectorAddc32(pOutdata, inBufTest, constValue, Num);

// outVal[i] = inBufTest[i] + constValue = {-2,5,0,7,2,3,1,-5}
```

C

```
void DSP_VectorAddc32(int32_t * outdata, int32_t * indata, int32_t c, int N);
```

DSP_VectorAutocorr16 Function

Computes the Autocorrelation of a Vector.

Description

Function DSP_VectorAutocorr16:

void DSP_VectorAutocorr16(int16_t *outCorr, int16_t *inVector, int N, int K);

Calculates the autocorrelation, with a lag of 1 to K, on the first N elements of inVector and returns the 16-bit scalar result in outCorr. The autocorrelation is calculated from other statistical equations including mean and variance. While in some cases these

equations exist inside the DSP library, the functions are executed in a serial fashion within this code to provide enhanced performance. The unbiased function has the form -

mean (M) = sum[0..N-1](x(n) / N) variance (V) = sum[0..N-1]((x(n) - M)^2) / (N-1) autocovariance (ACV)[k] = sum[0..(N-k)]((x(n) - M) * (x(n+k) - M) / (N-k)) autocorrelation (AC)[k] = CV[k] / V where N is the number of vector elements, n is the index of those elements x(n) is a single element in the input vector M is the mean of the N elements of the vector k is the lag or series index

The output of the function will return K elements, and the outCorr array should be sized to accept those 16-bit results. The outputs correspond to k=1, k=2, .., k=K delay states. The function returns a 16-bit value in rounded, saturated Q15 format.

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. outCorr must be an array with at least K values. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outCorr	pointer to output array (int16_t)
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)
K	lag value, number of output elements (int)

Returns

None.

Remarks

This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and the _LIBQ_Q16Div specifically.

Example

```
int  autocvN = 16;                    // N value, number of samples in array
int  autocvLag = 4;                   // Lag value, output shifts to observe
int16_t outAC[16]={0};
int16_t CVIn16[16] = {0x1999, 0xD99A, 0x1000, 0x6000, 0x1999, 0x1999, 0x2666,
     0x3333, 0x1000, 0x6000, 0x1999, 0x1999, 0x2666, 0x3333, 0x1999, 0x0CCC};
//  = { .2, -.3, .125, .75, .2, .2, .3, .4, .125, .75, .2, .2, .3, .4, .2, .1};

DSP_VectorAutocorr16(outAC, CVIn16, autocvN, autocvLag);

// outAC = {0xF406, D46C, 0x098F, 0x191A}
//    = -0.093567, -0.34045, 0.07468, 0.19611
```

C

```
void DSP_VectorAutocorr16(int16_t * outCorr, int16_t * inVector, int N, int K);
```

DSP_VectorBexp16 Function

Computes the maximum binary exponent of a vector.

Description

Function DSP_VectorBexp16:

int DSP_VectorBexp16(int16_t *DataIn, int N);

Calculates the maximum binary exponent on the first N elements of the input vector DataIn, and stores the integer result. The returned value represents the potential binary scaling of the vector, and may be used with other functions that auto scale their output without saturation. Inputs are given in Q15 fractional data format.

Preconditions

N must be a multiple of 2 and greater or equal to 2.

Parameters

Parameters	Description
DataIn	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

Binary exponent [log2 multiplier] (int)

Remarks

None.

Example

```
int valN = 4;
int16_t  dummy16[valN]={0x3004, 0x00CC, 0xFC04, 0xFFF0};
//                          0.375, 0.0062, -0.0311, -0.00049
int answer;

answer =   DSP_VectorBexp16(dummy16, valN);

// answer = 1, maximum binary gain is 2.
```

C

```
int DSP_VectorBexp16(int16_t * DataIn, int N);
```

DSP_VectorBexp32 Function

Computes the maximum binary exponent of a vector.

Description

Function DSP_VectorBexp32:

int DSP_VectorBexp32(int32_t *DataIn, int N);

Calculates the maximum binary exponent on the first N elements of the input vector DataIn, and stores the integer result. The returned value represents the potential binary scaling of the vector, and may be used with other functions that auto scale their output without saturation. Inputs are given in Q31 fractional data format.

Preconditions

None.

Parameters

Parameters	Description
DataIn	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

Returns

Binary exponent [log2 multiplier] (int)

Remarks

None.

Example

```
int valN=4;
int32_t  datInput32[4]={0xFF000000, 0x07000000,0x000CCCCC, 0x08000000};
//                          -0.007183, 0.054688, 0.0003906, 0.0625
```

```
int answer32;

answer32 =   DSP_VectorBexp32(datInput32, valN);

// answer = 3, maximum binary gain is 8.
```

C

```
int DSP_VectorBexp32(int32_t * DataIn, int N);
```

DSP_VectorChkEqu32 Function

Compares two input vectors, returns an integer '1' if equal, and '0' if not equal.

Description

Function DSP_VectorChkEqu32:

int DSP_VectorChkEqu32(int32_t *indata1, int32_t *indata2, int N);

Compares the first N values of indata1 to the same elements of indata2. The comparison requires that all numbers be in Q31 fractional data format. Returns the integer value '1' if all numbers are equal, and '0' if they are not equal. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata1	pointer to input array 1 of elements (int32_t)
indata2	pointer to input array 2 of elements (int32_t)
N	number of samples (int)

Returns

(int) - '1' if vectors are equal, '0' if vectors are not equal

Remarks

None.

Example

```
int     outCheck;
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6
int32_t inBufTestB[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x80000000, 0x40000000, 0x7FFFFFFF, 0x20000000};
//                  1,     -1,      0.9,    0.8,    -1,   0.5,    1,     0.25

outCheck = DSP_VectorChkEqu32(inBufTestA, inBufTestB, Num);

// outCheck = 1  // true for first 4 numbers of series
```

C

```
int DSP_VectorChkEqu32(int32_t * indata1, int32_t * indata2, int N);
```

DSP_VectorCopy Function

Copies the elements of one vector to another.

Description

Function DSP_VectorCopy:

void DSP_VectorCopy(int32_t *outdata, int32_t *indata, int N);

Fills the first N values of an input vector outdata with the elements from indata. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to destination array of values (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

None.

Example

```
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6
int32_t inBufTestB[8]={0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
                        0x40000000, 0x60000000, 0x80000000, 0x20000000};
//                  0,      1,      0.5,    0.1,    0.75,   0.5,    -1,     0.25

DSP_VectorCopy(inBufTestA, inBufTestB, Num);

// inBufTestA =  {0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
//           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values copied
```

C

```
void DSP_VectorCopy(int32_t * outdata, int32_t * indata, int N);
```

DSP_VectorCopyReverse32 Function

Reverses the order of elements in one vector and copies them into another.

Description

Function DSP_VectorCopyReverse32:

void DSP_VectorCopyReverse32(int32_t *outdata, int32_t *indata, int N);

Fills the first N values of an input vector Outdata with the reverse elements from INDATA. N must be a multiple of 4 and greater than 4 or will be truncated to the nearest multiple of 4. The vectors are both Q31 fractional format.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to destination array of values (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

None.

Example

```
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,      0.9,    0.8,    0.2,   0.5,    1,      -0.6
int32_t inBufTestB[8]={0x00000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
                        0x40000000, 0x60000000, 0x80000000, 0x20000000};
//                  0,      1,      0.5,    0.1,    0.75,   0.5,    -1,      0.25

DSP_VectorCopyReverse32(inBufTestA, inBufTestB, Num);

// inBufTestA =  {0x0CCCCCCC, 0x40000000, 0x7FFFFFFF, 0x00000000,
//           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334}
//   first 4 values copied reverse order
```

C

```
void DSP_VectorCopyReverse32(int32_t * outdata, int32_t * indata, int N);
```

DSP_VectorDivC Function

Divides the first N elements of inVector by a constant divisor, and stores the result in outVector.

Description

Function DSP_VectorDivC:

void DSP_VectorDivC(_Q16 *outVector, _Q16 *inVector, _Q16 divisor, int N);

Divides each element of the first N elements of inVector by a constant, divisor. The output is stored to outVector. Both vectors and the scalar are _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or minimum respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Div function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
divisor	scalar divisor for the input vector (_Q16)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Div function see the LibQ documentation for _LIBQ_Q16Div.

Example

```
int divNum = 4;
_Q16 divScalar = 0x00020000;  // 2.0
_Q16 inDivVec[8] = {0x08000000, 0xfffc0000,0x00024000,0x00100000,0x00038000,
       0x00400000,0xfffe0000,0x00058000};
//     2048, -4, 2.25, 16, 3.5, 64, -2, 5.5
_Q16 outDivVec[8] = {0};

DSP_VectorDivC(outDivVec, inDivVec,  divScalar, divNum);

// outDivVec =  0x04000000, 0xFFFFE0000, 0x00012000, 0x00080000, 0, 0, 0, 0
//                  1024.0,        -2.0,      1.125,       16.0, 0, 0, 0, 0
```

C

```
void DSP_VectorDivC(_Q16 * outVector, _Q16 * inVector, _Q16 divisor, int N);
```

DSP_VectorDotp16 Function

Computes the dot product of two vectors, and scales the output by a binary factor.

Description

Function DSP_VectorDotp16:

int16_t DSP_VectorDotp16(int16_t *indata1, int16_t *indata2, int N, int scale);

Calculates the dot product of two input vectors, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after accumulation in a 32 bit register. All calculations are done in Q15 fractional format. return = 1/(2^scale) * sum(indata1[i] * indata2[i])

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

Returns

int16_t - scaled output of calculation, Q15 format

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
//                      1,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6
int16_t inBufMultB[8]={0x0CCD, 0x0CCD, 0x4000, 0xC000, 0xE667, 0x4000, 0x0000, 0x0CCD};
//                      0.1,    0.1,    0.5,   -0.5,   -0.2,   0.5,    0,      0.1
int Num = 8;
int scaleVal = 2;
int16_t outScalar;
```

```
int Num = 8;

outScalar = DSP_VectorDotp16(inBufMultA, inBufMultB, Num, scaleVal);

// outScalar = 1/(2^scaleVal)*(inBufMultA[] dot inBufMultB[]) =
//   (1/4) * (0.1 + -0.1 + 0.45 + -0.4 + -0.04 + 0.25 + 0 + -0.06) = 0.25 * 0.20 = 0.05
//      = (int16_t)0x0666
```

C

```
int16_t DSP_VectorDotp16(int16_t * indata1, int16_t * indata2, int N, int scale);
```

DSP_VectorDotp32 Function

Computes the dot product of two vectors, and scales the output by a binary factor

Description

Function DSP_VectorDotp32:

int32_t DSP_VectorDotp32(int32_t *indata1, int32_t *indata2, int N, int scale);

Calculates the dot product of two input vectors, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after calculation of the result. All calculations are done in Q31 fractional format. return = 1/(2^scale) * sum(indata1[i] * indata2[i])

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

Returns

int16_t - scaled output of calculation, Q31 format

Remarks

This must be assembled with .set microMIPS.

Example

```
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,       0.9,     0.8,     0.2,    0.5,     1,      -0.6
int32_t inBufMultB[8]={0x0CCCCCCD, 0x0CCCCCCD, 0x40000000, 0xC0000000,
                        0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCCD};
//                  0.1,     0.1,     0.5,    -0.5,    -0.2,    0.5,     0,       0.1
int Num = 8;
int scaleVal = 2;
int32_t outScalar;

int Num = 8;

outScalar = DSP_VectorDotp32(inBufMultA, inBufMultB, Num, scaleVal);

// outScalar = 1/(2^scaleVal)*(inBufMultA[] dot inBufMultB[]) =
//   (1/4) * (0.1 + -0.1 + 0.45 + -0.4 + -0.04 + 0.25 + 0 + -0.06) = 0.25 * 0.20 = 0.05
//      = (int32_t)0x06666666
```

C

```
int32_t DSP_VectorDotp32(int32_t * indata1, int32_t * indata2, int N, int scale);
```

DSP_VectorExp Function

Computes the EXP (e^x) of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorExp:

void DSP_VectorExp(_Q16 *outVector, _Q16 *inVector, int N);

Computes the Exp value, e to the power of X, on the first N elements of inVector. The output is stored to outVector. Both vectors are _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Exp and Div functions from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
N	number of samples (int)

Returns

None.

Remarks

Inclusion of the LibQ header file and library is mandatory to use this function.

Example

```
int expNum = 4;
_Q16 inExpVec[8] = {0x00010000, 0xffff0000,0x00020000,0x00030000,0x00038000,
            0x00040000,0xfffe0000,0x00058000};
//          1.0,  -1.0,  2.0,  3.0,   3.5,   4.0,  -2.0,  5.5
_Q16 outExpVec[8] = {0};

DSP_VectorExp(outExpVec, inExpVec, expNum);

// outExpVec = 0x0002B7E1, 0x00005E2D, 0x00076399, 0x001415E6, 0, 0, 0, 0
//                2.71828,   0.26787,     7.3891,    20.0855, 0, 0, 0, 0
```

C

```
void DSP_VectorExp(_Q16 * outVector, _Q16 * inVector, int N);
```

DSP_VectorFill Function

Fills an input vector with scalar data.

Description

Function DSP_VectorFill:

void DSP_VectorFill(int32_t *indata, int32_t data, int N);

Fills the first N values of an input vector indata with the value data. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
data	scalar value to fill the array (int32_t)
N	number of samples (int)

Returns

None.

Remarks

None.

Example

```
int32_t fillValue = 0x3FFFFFFF;
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6

DSP_VectorFill(inBufTestA, fillValue, Num);

// inBufTestA =  {0x3FFFFFFF, 0x3FFFFFFF, 0x3FFFFFFF, 0x3FFFFFFF,
//           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values filled
```

C

```
void DSP_VectorFill(int32_t * indata, int32_t data, int N);
```

DSP_VectorLn Function

Computes the Natural Log, Ln(x), of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorLn:

void DSP_VectorLn(_Q4_11 *outVector, _Q16 *inVector, int N);

Computes the Ln(x) value, on the first N elements of inVector. The output is stored to outVector. Input vector is _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution Q4.11 format, which is a 16-bit integer format with 11 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Ln function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q4_11)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Ln function see the LibQ documentation for _LIBQ_Q4_11_ln_Q16. A negative number input will return a saturated negative value (0x8000).

Example

```
int lnNum = 4;
_Q16 inLnVal[8] = {0x40000000, 0xffff0000,0x00020000,0x00100000,0x00038000,
       0x00400000,0xfffe0000,0x00058000};
//          16384.0,  -1.0,  2.0,  16.0,   3.5,   64.0,  -2.0,  5.5
_Q4_11 outLnVal[8] = {0};

DSP_VectorLn(outLnVal, inLnVal,  lnNum);

// outLnVal =     0x4DA2,        0x8000,   0x058C,   0x162E, 0, 0, 0, 0
//                 9.704,  sat negative,   0.6934,    2.772, 0, 0, 0, 0
```

C

```
void DSP_VectorLn(_Q4_11 * outVector, _Q16 * inVector, int N);
```

DSP_VectorLog10 Function

Computes the Log10(x), of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorLog10:

void DSP_VectorLog10(_Q3_12 *outVector, _Q16 *inVector, int N);

Computes the Log10(x) value, on the first N elements of inVector. The output is stored to outVector. Input vector is _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution Q3.12 format, which is a 16-bit integer format with 12 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Log10 function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q3_12)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Log10 function see the LibQ documentation for _LIBQ_Q3_12_log10_Q16. A negative number input will return a saturated negative value (0x8000).

Example

```
int logNum = 4;
_Q16 inLogVal[8] = {0x40000000, 0xffff0000,0x00020000,0x00100000,0x00038000,
                    0x00400000,0xfffe0000,0x00058000};
//          16384.0,  -1.0,  2.0,  16.0,   3.5,   64.0,  -2.0,  5.5
_Q3_12 outLogVal[8] = {0};
```

```
DSP_VectorLog10(outLogVal, inLogVal,  logNum);

// outLogVal =    0x436E,        0x8000,    0x04D1,   0x1344, 0, 0, 0, 0
//                 4.2144,  sat negative,   0.3010,   1.2041, 0, 0, 0, 0
```

C

```
void DSP_VectorLog10(_Q3_12 * outVector, _Q16 * inVector, int N);
```

DSP_VectorLog2 Function

Computes the Log2(x) of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorLog2:

void DSP_VectorLog2(_Q5_10 *outVector, _Q16 *inVector, int N);

Computes the Log2 value, where log2(x) = ln(x) * log2(e), on the first N elements of inVector. The output is stored to outVector. Input vector is _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. The output vector is reduced resolution Q5.10 format, which is a 16-bit integer format with 10 bits representing the fractional resolution. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Log2 function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q5_10)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Log2 function see the LibQ documentation for _LIBQ_Q5_10_log2_Q16. A negative number input will return a saturated negative value (0x8000).

Example

```
int log2Num = 4;
_Q16 inLog2Val[8] = {0x40000000, 0xffff0000,0x00020000,0x00030000,0x00038000,
           0x00040000,0xfffe0000,0x00058000};
//          16384.0,  -1.0,  2.0,  3.0,   3.5,   4.0,  -2.0,  5.5
_Q5_10 outLog2Val[8] = {0};

DSP_VectorLog2(outLog2Val, inLog2Val,  log2Num);

// outLog2Val =    0x3800,        0x8000,   0x0400,   0x0657, 0, 0, 0, 0
//                  14.0,  sat negative,      1.0,    1.585, 0, 0, 0, 0
```

C

```
void DSP_VectorLog2(_Q5_10 * outVector, _Q16 * inVector, int N);
```

DSP_VectorMax32 Function

Returns the maximum value of a vector.

Description

Function DSP_VectorMax32:

int32_t DSP_VectorMax32(int32_t *indata, int N);

Returns the highest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Parameters

Parameters	Description
indata	pointer to input array of elements (int32_t)
N	number of samples (int)

Returns

(int32_t) - maximum value within the vector, Q31 format

Remarks

None.

Example

```
int32_t   outCheck;
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6

outCheck = DSP_VectorMax32(inBufTestA, Num);

// outCheck = 0x7FFFFFFF  // first 4 values
```

C

```
int32_t DSP_VectorMax32(int32_t * indata, int N);
```

DSP_VectorMaxIndex32 Function

Returns the index of the maximum value of a vector.

Description

Function DSP_VectorMaxIndex32:

int DSP_VectorMaxIndex32(int32_t *indata, int N);

Returns the index of the highest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

int - index of the position of the maximum array element

Remarks

Index values range from 0 .. (n-1).

Example

```
int     indexValue;
int     Num = 8;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  0.5,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6

indexValue = DSP_VectorMaxIndex32(inBufTestA, Num);

// returnValue =  6 (position corresponding to 0x7FFFFFFF)
```

C

```
int DSP_VectorMaxIndex32(int32_t * indata, int N);
```

DSP_VectorMean32 Function

Calculates the mean average of an input vector.

Description

Function DSP_VectorMean32:

int32_t DSP_VectorMean32(int32_t *indata, int N);

Calculates the mean average of the first N elements of the vector indata. The values of indata1 are in Q31 fractional format. The value N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

int32_t - mean average value of the vector

Remarks

None.

Example

```
int32_t   returnValue;
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6

returnValue = DSP_VectorMean32(inBufTestA, Num);

// returnValue =  0x36666666 = (1-1+0.9+0.8)/4 = 0.425
```

C

```
int32_t DSP_VectorMean32(int32_t * indata1, int N);
```

DSP_VectorMin32 Function

Returns the minimum value of a vector.

Description

Function DSP_VectorMin32:

int32_t DSP_VectorMin32(int32_t *indata, int N);

Returns the lowest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Parameters

Parameters	Description
indata	pointer to input array of elements (int32_t)
N	number of samples (int)

Returns

(int32_t) - minimum value within the vector, Q31 format

Remarks

None.

Example

```
int32_t   outCheck;
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                            0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,      0.9,    0.8,    0.2,   0.5,    1,      -0.6

outCheck = DSP_VectorMin32(inBufTestA, Num);

// outCheck = 0x80000000  // first 4 values
```

C

```
int32_t DSP_VectorMin32(int32_t * input, int N);
```

DSP_VectorMinIndex32 Function

Returns the index of the minimum value of a vector.

Description

Function DSP_VectorMinIndex32:

int DSP_VectorMinIndex32(int32_t *indata, int N);

Returns the relative position index of the lowest value of the first N elements of the vector indata. The comparison requires that all numbers be in Q31 fractional data format. N must be greater than or equal to four and a multiple of four, or it will be truncated to the nearest multiple of four.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

int32_t - mean average value of the vector

Remarks

Index values range from 0 .. (n-1).

Example

```
int     indexValue;
int     Num = 8;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  0.5,     -1,      0.9,    0.8,    0.2,   0.5,    1,     -0.6

indexValue = DSP_VectorMinIndex32(inBufTestA, Num);

// returnValue =  1 (position corresponding to 0x80000000)
```

C

```
int DSP_VectorMinIndex32(int32_t * indata, int N);
```

DSP_VectorMul16 Function

Multiplication of a series of numbers in one vector to another vector.

Description

Function DSP_VectorMul16:

void DSP_VectorMul16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);

Multiples the value of each element of indata1 * indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q15 fractional format. outdata[i] filled with N elements of indata1[i] * indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
```

```
//                          1,      -1,      0.9,    0.8,    0.2,   0.5,    1,      -0.6
int16_t inBufMultB[8]={0x0CCD, 0x0CCD, 0x4000, 0xC000, 0xE667, 0x4000, 0x0000, 0x0CCD};
//                          0.1,    0.1,    0.5,   -0.5,   -0.2,   0.5,    0,      0.1
int Num = 8;

pOutdata = &outVal;

DSP_VectorMul16(pOutdata, inBufMultA, inBufMultB, Num);

// outVal[i] = inBufTest[i] * inBuf2[i] =
//    {0x0CCD, 0xF333, 0x399A, 0xCCCD, 0xFAE2, 0x2000, 0x0000, 0xF852}
//      0.1,   -0.1,   0.45,  -0.4,   -0.04,   0.25,    0,     -0.06
```

C

```
void DSP_VectorMul16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

DSP_VectorMul32 Function

Multiplication of a series of numbers in one vector to another vector.

Description

Function DSP_VectorMul32:

void DSP_VectorMul32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);

Multiples the value of each element of indata1 * indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in the Q31 fractional format. outdata[i] filled with N elements of indata1[i] * indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                          0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,      0.9,    0.8,    0.2,   0.5,    1,      -0.6
int32_t inBufMultB[8]={0x0CCCCCCD, 0x0CCCCCCD, 0x40000000, 0xC0000000,
                          0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCCD};
//                  0.1,    0.1,    0.5,   -0.5,   -0.2,   0.5,    0,      0.1
int Num = 8;

pOutdata = &outVal;

DSP_VectorMul32(pOutdata, inBufMultA, inBufMultB, Num);

// outVal[i] = inBufTest[i] * inBuf2[i] =
```

```
//    {0x0CCCCCCD, 0xF3333333, 0x3999999A, 0xCCCCCCCD, 0xFAE147AE,
//                                         0x20000000, 0x00000000, 0xF851EB86}
//      0.1,   -0.1,   0.45,  -0.4,   -0.04,   0.25,    0,     -0.06
```

C

```
void DSP_VectorMul32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

DSP_VectorMulc16 Function

Multiplication of a series of numbers in one vector to a scalar value.

Description

Function DSP_VectorMulc16:

void DSP_VectorMulc16(int16_t *outdata, int16_t *indata, int16_t c, int N);

Multiples the value of each element of indata1 * c and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata[i] * c

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata	pointer to input array of 16-bit elements (int16_t)
c	scalar multiplicand (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x0000, 0xB334};
//                        1,     -1,      0.9,    0.8,    0.2,   0.5,    0,     -0.6
int16_t constValue = 0x4000;
int Num = 8;

pOutdata = &outVal;

DSP_VectorMulc16(pOutdata, inBufMultA, constValue, Num);

// outVal[i] = inBufTest[i] * constValue =
//    {0x4000, 0xC000, 0x399A, 0x3333, 0x1999, 0x2000, 0x0000, 0xD99A}
//      0.5,   -0.5,   0.45,    0.4,    0.1,   0.25,    0,     -0.3
```

C

```
void DSP_VectorMulc16(int16_t * outdata, int16_t * indata, int16_t c, int N);
```

DSP_VectorMulc32 Function

Multiplication of a series of numbers in one vector to a scalar value.

Description

Function DSP_VectorMulc32:

void DSP_VectorMulc32(int32_t *outdata, int32_t *indata, int32_t c, int N);

Multiples the value of each element of indata * c and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata[i] * c

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata	pointer to input array of 16-bit elements (int32_t)
c	scalar multiplicand (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x00000000, 0xB3333334};
//                    1,    -1,    0.9,    0.8,    0.2,   0.5,    1,    -0.6
int32_t constValue = 0x4000;
int Num = 8;

pOutdata = &outVal;

DSP_VectorMulc32(pOutdata, inBufMultA, constValue, Num);

// outVal[i] = inBufTest[i] * constValue =
//    {0x40000000, 0xC0000000, 0x3999999A, 0x33333333, 0x19999999,
//                                          0x20000000, 0x00000000, 0xD999999A}
//      0.5,   -0.5,   0.45,    0.4,    0.1,   0.25,    0,     -0.3
```

C

```
void DSP_VectorMulc32(int32_t * outdata, int32_t * indata, int32_t c, int N);
```

DSP_VectorNegate Function

Inverses the sign (negates) the elements of a vector.

Description

Function DSP_VectorNegate:

void DSP_VectorNegate(int32_t *outdata, int32_t *indata, int N);

Sign inversion of the first N values of an indata are assigned to outdata. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result and the scalar value to fill are both Q31 fractional format.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

None.

Example

```
int     Num = 4;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,      0.9,    0.8,    0.2,   0.5,    1,      -0.6
int32_t outBufTest[8]={0x0CCCCCCD, 0x0CCCCCCD, 0x40000000, 0xC0000000,
                        0xE6666667, 0x40000000, 0x00000000, 0x0CCCCCCD};
//                  0.1,    0.1,    0.5,   -0.5,   -0.2,   0.5,    0,      0.1;

DSP_VectorNegate(outBufTest, inBufTestA, Num);

// inBufTestA =  {0x80000000, 0x7FFFFFFF, 0x8CCCCCCD, 0x9999999A,
//           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334} // first 4 values neg
//              -1,   1,   -0.9,   -0.8,   -0.2,   0.5,   0,   0.1
```

C

```
void DSP_VectorNegate(int32_t * outdata, int32_t * indata, int N);
```

DSP_VectorRecip Function

Computes the reciprocal (1/x) of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorRecip:

void DSP_VectorRecip(_Q16 *outVector, _Q16 *inVector, int N);

Computes the reciprocal (1/x) on the first N elements of inVector. The output is stored to outVector. Both vectors are _Q16 format, which is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or minimum respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Div function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the

design in order to operate this function. For more information on the Div function see the LibQ documentation for _LIBQ_Q16Div. A value of zero in the array will not cause an error, but will return 0.

Example

```
int recNum = 4;
_Q16 inRecVec[8] = {0x08000000, 0xfffc0000,0x00020000,0x00100000,0x00038000,
                0x00400000,0xfffe0000,0x00058000};
//             2048.0, -4.0, 2.0, 16.0, 3.5, 64.0, -2.0, 5.5
_Q16 outRecVec[8] = {0};

DSP_VectorRecip(outRecVec, inRecVec,  recNum);

// outRecVec =  0x00000020, 0xFFFFC0000, 0x00008000, 0x00001000, 0, 0, 0, 0
//                 0.000488,       -0.25,        0.5,     0.0625, 0, 0, 0, 0
```

C

```
void DSP_VectorRecip(_Q16 * outVector, _Q16 * inVector, int N);
```

DSP_VectorRMS16 Function

Computes the root mean square (RMS) value of a vector.

Description

Function DSP_VectorRMS16:

int16_t DSP_VectorRMS16(int16_t *inVector, int N);

Computes the root mean square value of the first N values of inVector. Both input and output are Q15 fractional values. The function will saturate if maximum or minimum values are exceeded.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

int16_t - RMS function output, Q15 format

Remarks

This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and uses the _LIBQ_Q16Sqrt external function call.

Example

```
int16_t vecRMSIn[32]={0x1999, 0xD99A, 0x4000, 0x2666,0x1999,0x1999,0x2666, 0x3333};
// 0.2, -0.3, 0.5, 0.3, 0.2, 0.2, 0.3, 0.4
int16_t RMSOut=0;
int Nrms = 8;

RMSOut = DSP_VectorRMS16(vecRMSIn, Nrms);

// RMSOut = 0x287C (= 0.31628)
```

C

```
int16_t DSP_VectorRMS16(int16_t * inVector, int N);
```

DSP_VectorShift Function

Shifts the data index of an input data vector.

Description

Function DSP_VectorShift:

void DSP_VectorShift(int32_t *outdata, int32_t *indata, int N, int shift);

Shifts N data elements of indata to outdata, with an index change of shift. The amount of data shifted includes zero padding for the first (shift) elements if shift is positive. The vector size of indata and outdata need not be the same, however, N must not exceed either array size.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must not exceed the amount of elements in the source array. shift must not exceed the number of elements in the destination array.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (int32_t)
indata	pointer to source array of elements (int32_t)
N	number of samples (int)
shift	number of indexes to shift (int)

Returns

None.

Remarks

Destination array values shift to left (relative to the input vector) when shift is positive (back filled with zeros) and shift to the right when shift is negative. The total amount of values copied to the destination array is the length of N less the shift amount.

Example

```
int shiftValue = 3;
int     Num = 8;
int32_t inBufTestA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  1,      -1,       0.9,     0.8,     0.2,    0.5,     1,      -0.6
int32_t inBufTestB[8]={0x80000000, 0x7FFFFFFF, 0x40000000, 0x0CCCCCCC,
                        0x40000000, 0x60000000, 0x80000000, 0x20000000};
//                  -1,      1,       0.5,     0.1,     0.75,    0.5,     -1,      0.25

DSP_VectorShift(inBufTestA, inBufTestB, Num, shiftValue);

// inBufTestA =  {0x00000000, 0x00000000, 0x00000000, 0x80000000,
//           0x7FFFFFFF, 0x40000000, 0x0CCCCCCC, 0x40000000} // shifted 3 positive
```

C

```
void DSP_VectorShift(int32_t * outdata, int32_t * indata, int N, int shift);
```

DSP_VectorSqrt Function

Computes the square root of the first N elements of inVector, and stores the result in outVector.

Description

Function DSP_VectorSqrt:

void DSP_VectorSqrt(_Q16 *outVector, _Q16 *inVector, int N);

Computes the Sqrt(x) on the first N elements of inVector. The output is stored to outVector. Both vectors are _Q16 format, which

is 32-bit data with 15 bits for the integer and 16 bits for the fractional portion. If values exceed maximum or minimum they will saturate to the maximum or zero respectively.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. This function uses the Sqrt function from the LibQ library. That library must be compiled as part of the project.

Parameters

Parameters	Description
outdata	pointer to destination array of elements (_Q16)
indata	pointer to source array of elements (_Q16)
N	number of samples (int)

Returns

None.

Remarks

This function uses the Microchip PIC32MZ LibQ library to function. The user must include that library and header file into the design in order to operate this function. For more information on the Sqrt function see the LibQ documentation for _LIBQ_Q16Sqrt. A negative number input will return a saturated value (0x00FFFFxx).

Example

```
int sqrtNum = 4;
_Q16 inSqrtVec[8] = {0x40000000, 0xffff0000,0x00020000,0x00100000,0x00038000,
           0x00400000,0xfffe0000,0x00058000};
//          16384.0,  -1.0,  2.0,  16.0,   3.5,   64.0,  -2.0,  5.5
_Q16 outSqrtVec[8] = {0};

DSP_VectorSqrt(outSqrtVec, inSqrtVec,  sqrtNum);

// outSqrtVec =  0x00800000, 0x00FFFF80, 0x00016A0A, 0x00040000, 0, 0, 0, 0
//                 128.0,  sat negative,   1.41422,      4.0,    0, 0, 0, 0
```

C

```
void DSP_VectorSqrt(_Q16 * outVector, _Q16 * inVector, int N);
```

DSP_VectorStdDev16 Function

Computes the Standard Deviation of a Vector.

Description

Function DSP_VectorStdDev16:

int16_t DSP_VectorStdDev16(int16_t *inVector, int N);

Calculates the standard deviation on the first N elements of inVector and returns the 16-bit scalar result. The standard deviation is the square root of the variance, which is a measure of the delta from mean values. The mean value of the vector is computed in the process. The function has the form -

StdDev = SQRT(sum[0..N]((x(i) - M(N))^2) / (N-1)) where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four. Dependent on use of the LibQ library.

Parameters

Parameters	Description
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)

Returns

int16_t - Standard Deviation of N selected elements

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions. This function is dependent on the LibQ library, and the _LIBQ_Q16Sqrt specifically.

Example

```
int16_t vecStDevIn[32]={0x4000, 0xD99A, 0x1000, 0x6000,0x1999,0x1999,0x2666, 0x3333};
        // .2, -.3, .125, .75, .2, .2, .3, .4
int16_t StDevOut, Var16Out;
int Nstdev = 4;

StDevOut = DSP_VectorStdDev16(vecStDevIn, Nstdev);
        // StDevOut = 0x3A9E (= 0.45797)
```

C

```
int16_t DSP_VectorStdDev16(int16_t * inVector, int N);
```

DSP_VectorSub16 Function

Calculate the difference of two vectors.

Description

Function DSP_VectorSub16:

void DSP_VectorSub16(int16_t *outdata, int16_t *indata1, int16_t *indata2, int N);

Computes the difference value of each element of indata1 - indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q15 fractional format. outdata[i] filled with N elements of indata1[i] - indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int16_t)
indata1	pointer to input array of 16-bit elements (int16_t)
indata2	pointer to input array of 16-bit elements (int16_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int16_t outVal[8];
int16_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int16_t inBuf2[16]=     { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7,  0};
```

```
int Num = 8;

pOutdata = &outVal;

DSP_VectorSub16(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] - inBuf2[i] = {-6,0,-6,0,-6,-6,-9,-16}
```

C

```
void DSP_VectorSub16(int16_t * outdata, int16_t * indata1, int16_t * indata2, int N);
```

DSP_VectorSub32 Function

Calculate the difference of two vectors.

Description

Function DSP_VectorSub32:

void DSP_VectorSub32(int32_t *outdata, int32_t *indata1, int32_t *indata2, int N);

Computes the difference value of each element of indata1 - indata2 and stores it to outdata. The number of samples to process is given by the parameter N. Data is in a Q31 fractional format. outdata[i] filled with N elements of indata1[i] - indata2[i]

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
outdata	pointer to output array of 16-bit elements (int32_t)
indata1	pointer to input array of 16-bit elements (int32_t)
indata2	pointer to input array of 16-bit elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t *pOutdata;
int32_t outVal[8];
int32_t inBufTest[16] = {-5,2,-3,4,-1,0,-2,-8,-21,21,10,100, 200, 127,-127,-2};
int32_t inBuf2[16]=     { 1,2, 3,4, 5,6, 7, 8, 9, 10,-1,-100,-127,127,-7,  0};
int Num = 8;

pOutdata = &outVal;

DSP_VectorSub32(pOutdata, inBufTest, inBuf2, Num);

// outVal[i] = inBufTest[i] - inBuf2[i] = {-6,0,-6,0,-6,-6,-9,-16}
```

C

```
void DSP_VectorSub32(int32_t * outdata, int32_t * indata1, int32_t * indata2, int N);
```

DSP_VectorSumSquares16 Function

Computes the sum of squares of a vector, and scales the output by a binary factor.

Description

Function DSP_VectorSumSquares16:

int16_t DSP_VectorSumSquares16(int16_t *indata, int N, int scale);

Calculates the sum of the squares of each element of an input vector, and scales the output. Function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after accumulation in a 32 bit register. All calculations are done in Q15 fractional format. return = 1/(2^scale) * sum(indata[i]^2)

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to eight and a multiple of eight.

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int16_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

Returns

int16_t - scaled output of calculation, Q15 format

Remarks

This must be assembled with .set microMIPS.

Example

```
int16_t inBufMultA[8]={0x7FFF, 0x8000, 0x7333, 0x6666, 0x1999, 0x4000, 0x7FFF, 0xB334};
//                          1,      -1,       0.9,     0.8,     0.2,    0.5,     1,      -0.6
int Num = 8;
int scaleVal = 3;
int16_t outScalar;

outScalar = DSP_VectorSumSquares16(inBufMultA, Num, scaleVal);

// outScalar = 1/(2^scaleVal)* sum(inBufMultA[i]^2) =
//   (1/8) * (1 + 1 + 0.81 + 0.64 + 0.04 + 0.25 + 1 + 0.36) = 0.125 * 5.1 = 0.6375
//      = (int16_t)0x5199
```

C

```
int16_t DSP_VectorSumSquares16(int16_t * indata, int N, int scale);
```

DSP_VectorSumSquares32 Function

Computes the sum of squares of a vector, and scales the output by a binary factor.

Description

Function DSP_VectorSumSquares32:

int32_t DSP_VectorSumSquares32(int32_t *indata, int N, int scale);

Calculates the sum of the squares of each element of an input vector, and scales the output. The function will saturate if it exceeds maximum or minimum values. Scaling is done by binary shifting, after calculation of the results. All calculations are done in Q31 fractional format. return = 1/(2^scale) * sum(indata[i]^2)

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and multiple of four.

Parameters

Parameters	Description
indata	pointer to input array of 16-bit elements (int32_t)
scale	number of bits to shift return right (int)
N	number of samples (int)

Returns

int32_t - scaled output of calculation, Q15 format

Remarks

This must be assembled with .set microMIPS.

Example

```
int32_t inBufMultA[8]={0x7FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                        0x19999999, 0x40000000, 0x00000000, 0xB3333334};
//                    1,    -1,    0.9,    0.8,    0.2,   0.5,    1,    -0.6
int Num = 8;
int scaleVal = 3;
int32_t outScalar;

outScalar = DSP_VectorSumSquares32(inBufMultA, Num, scaleVal);

// outScalar = 1/(2^scaleVal)* sum(inBufMultA[i]^2) =
//   (1/8) * (1 + 1 + 0.81 + 0.64 + 0.04 + 0.25 + 1 + 0.36) = 0.125 * 5.1 = 0.6375
//      = (int32_t)0x51999999
```

C

```
int32_t DSP_VectorSumSquares32(int32_t * indata, int N, int scale);
```

DSP_VectorVari16 Function

Computes the variance of N elements of a Vector.

Description

Function DSP_VectorVari16:

int16_t DSP_VectorVari16(int16_t *inVector, int N);

Calculates the variance on the first N elements of inVector and returns the 16-bit scalar result. The variance is a measure of the delta from mean values, and the mean value of the vector is computed in the process. The function has the form -

var = sum[0..N]((x(i) - M(N))^2) / (N-1) where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q15 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
inVector	pointer to source array of elements (int16_t)
N	number of samples (int)

Returns

int16_t - Variance of N selected elements

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions.

Example

```
int16_t vecStDevIn[32]={0x4000, 0xD99A, 0x1000, 0x6000,0x1999,0x1999,0x2666, 0x3333};
        // .2, -.3, .125, .75, .2, .2, .3, .4
int16_t Var16Out;
int Nvar = 4;

Var16Out= DSP_VectorVari16(vecStDevIn, Nvar);  // 16-bit variance function
        // Var16Out = 0x1AD8 (= 0.20974)
```

C

```
int16_t DSP_VectorVari16(int16_t * inVector, int N);
```

DSP_VectorVariance Function

Computes the variance of N elements of inVector.

Description

Function DSP_VectorVariance:

int32_t DSP_VectorVariance(int32_t *inVector, int N);

Calculates the variance on the first N elements of inVector and returns the 32-bit scalar result. The variance is a measure of the delta from mean values, and the mean value of the vector is computed in the process. The function has the form -

var = sum[0..N]((x(i) - M(N))^2) / (N-1) where N is the number of vector elements x(i) is a single element in the vector M(N) is the mean of the N elements of the vector

Input values of the vector and output scalar value is Q31 fractional format. This format has data that ranges from -1 to 1, and has internal saturation limits of those same values. Some care has been taken to reduce the impact of saturation by adding processing steps to effectively complete the processing in blocks. However, in some extreme cases of data variance it is still possible to reach the saturation limits.

Preconditions

The pointers inVector must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
inVector	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

int32_t - Variance of N selected elements

Remarks

The input vector elements number, N, must be at least 4 and a multiple of 4. This function is optimized with microMIPS and M14KCe ASE DSP instructions.

Example

```
int varN = 8;

int32_t inVarVec[8] = {0xE6666667, 0x40000000,0x40000000,0x0CCCCCCC,
           0x00000000,0x59999999,0x20000000,0xC0000000};
//     -0.2, 0.5, 0.5, 0.1, 0, 0.7, 0.25, -0.5
int32_t outVar = 0;

outVar = DSP_VectorVariance(inVarVec, varN);
```

```
// outVar == 0x1490D2A6, = 0.1606696
```

C

```
int32_t DSP_VectorVariance(int32_t * inVector, int N);
```

DSP_VectorZeroPad Function

Fills an input vector with zeros.

Description

Function DSP_VectorZeroPad:

void DSP_VectorZeroPad(int32_t *indata, int N);

Fills the first N values of an input vector indata with the value zero. N must be a multiple of four and greater than or equal to four or it will be truncated to the nearest multiple of four. The vector result is in Q31 fractional format.

Preconditions

The pointers outdata and indata must be aligned on 4-byte boundaries. N must be greater than or equal to four and a multiple of four.

Parameters

Parameters	Description
indata	pointer to source array of elements (int32_t)
N	number of samples (int)

Returns

None.

Remarks

None.

Example

```
int     Num = 4;
int32_t inBufTestA[8]={0x3FFFFFFF, 0x80000000, 0x73333333, 0x66666666,
                           0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  0.5,     -1,      0.9,    0.8,    0.2,   0.5,    1,    -0.6

DSP_VectorZeroPad(inBufTestA, Num);

// inBufTestA =  {0x00000000, 0x00000000, 0x00000000, 0x00000000,
//                              0x19999999, 0x40000000, 0x7FFFFFFF, 0xB3333334};
//                  0,     0,      0,    0,    0.2,   0.5,    1,     -0.6
```

C

```
void DSP_VectorZeroPad(int32_t * indata, int N);
```

mul16 Function

multiply and shift integer

C

```
static inline int16_t mul16(int16_t a, int16_t b);
```

mul16r Function

multiply and shift Q15

C

```
static inline int16_t mul16r(int16_t a, int16_t b);
```

mul32 Function

multiply and shift Q31

C

```
static inline int32_t mul32(int32_t a, int32_t b);
```

SAT16 Function

saturate both positive and negative Q15

C

```
static inline int32_t SAT16(int32_t x);
```

SAT16N Function

saturate negative Q15

C

```
static inline int32_t SAT16N(int32_t x);
```

SAT16P Function

saturate positive Q15

C

```
static inline int32_t SAT16P(int32_t x);
```

LibQ

LibQ Fixed-Point Math Library

Introduction

This topic describes the LibQ Fixed-Point Math Library.

Description

The LibQ Fixed-Point Math Library is available for the PIC32MZ family of microcontrollers. This library was created from optimized assembly routines written specifically for devices with microAptiv™ core features.

The LibQ Fixed-Point Math Library simplifies writing fixed point algorithms, supporting Q15, Q31 and other 16-bit and 32-bit data formats. Using the simple, C callable functions contained in the library, fast fixed point mathematical operations can be easily executed. Fixed-point mathematical calculations may replace some functions implemented in the floating point library (math.h), depending on performance and resolution requirements.

Functions included in the LibQ library include capabilities for trigonometric, power and logarithms, and data conversion. In many cases the functions are identical other than the precision of their operands and the corresponding value that they return.

These functions are implemented in efficient assembly, and generally tuned to optimize performance over code size. In some cases the library breaks out functions that enable one to be optimized for accuracy, while another version is optimized for speed. These functions such as

_LIBQ_Q2_29_acos_Q31 and _LIBQ_Q2_29_acos_Q31_Fast are otherwise identical and can be used interchangeably. Each of these functions are typically used in computationally intensive real-time applications where execution time is a critical parameter.

Using the Library

This topic describes the basic architecture of the LibQ Fixed-Point Math Library and provides information and examples on its use.

Description

Interface Header File: `libq.h`

The interface to the LibQ Fixed-Point Math Library is defined in the `libq.h` header file. Any C language source (.c) file that uses the LibQ Fixed-Point Library should include `libq.h`.

Library File:

The LibQ Fixed-Point Math Library archive (.a) file is installed with MPLAB Harmony.

Configuring the Library using MHC

Select the Audio/Math/Math Libraries component to load the libq_c, libq/dsp libraries, as shown below

The project configuration should now contain the Math Libraries block.

The configuration window (when the block is selected) looks like the following:

Either the Speed Optimized (as shown above) or the Space Optimized versions of the library can be selected (only 1) to be generated to the project.

Library Overview

The LibQ Fixed-Point Math Library contains functions for manipulating Q15, Q31 and other intermediate integer representations of real numbers. The Library Interface section details the operation of the data formats and explains each function in detail.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the DSP Fixed-Point Math Library.

Library Interface Section	Description
Divide Functions	_Q16 fixed point divide function.
Square Root Functions	Square root of a positive _Q16 fixed point value function.
Log Functions	Log calculation functions.
Power Functions	Power calculation functions.
Exponential Functions	Exponential calculation functions.
Sine Functions	Sine calculation functions.
Cosine Functions	Cosine calculation functions.
Target Functions	Target calculation functions.
Arcsin Functions	Arcsin calculation functions.
Arccos Functions	Arccos calculation functions.
Arctan2 Functions	Arctan2 calculation functions.
Random Number Functions	_Q15 and _Q31 pseudo-random value functions.
Float Functions	Float conversion functions.
String Functions	ASCII to _Q15 conversions.

Signed fixed-point types are defined as follows: Qn_m where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied

For convenience, short names are also defined:

Exact Name	Number of Bits Required	Short Name
_Q0_15	16	_Q15
_Q15_16	16	_Q16
_Q0_31	32	_Q31

Qn_m numerical values are used by the library processing data as integers. In this format the n represents the number of integer bits, and the m represents the number of fractional bits. All values assume a sign bit in the most significant bit. The range of the numerical value therefore is:

$$-2^{(n-1)} \text{ to } [2^{(n-1)} - 2^{(-m)}]; \text{ with a resolution of } 2^{(-m)}.$$

A _Q16 format number (_Q15_16) would range from -32768.0 (0x8000 0000) to 32767.99998474 with a precision of 0.000015259 (or 2^{-16}). For example, a numerical representation of the number 3.14159 in _Q2_13 notation would be:

$$3.14159 * 2^{13} = 25735.9 \Rightarrow 0x6488$$

And converting from the _Q7_8 format with the value 0x1D89 would be:

$$0x1D89 / 2^{8} = 7561 / 256 \Rightarrow 29.5316, \text{ accurate to } 0.00391$$

Functions in the library are prefixed with the type of the return value. For example, _LIBQ_Q16Sqrt returns a _Q16 value equal to the square root of its argument. Argument types do not always match the return type. Refer to the function prototype for a specification of its arguments.

In cases where the return value is not a fixed-point type, the argument type is appended to the function name. For example,

_LIBQ_ToFloatQ31 accepts a type _Q31 argument.

In some cases, both the return type and the argument type are specified within the function name. For example:

Function Name	Return Type	Argument Type
_LIBQ_Q15_sin_Q2_13	_Q15	Q2_13
_LIBQ_Q31_sin_Q2_29	_Q31	_Q2_29

Library Interface

Functions

	Name	Description
	_LIBQ_Q15_cos_Q2_13	Calculates the value of cosine(x).
	_LIBQ_Q15_sin_Q2_13	Calculates the value of sine(x).
	_LIBQ_Q15FromFloat	Converts a float to a _Q15 value.
	_LIBQ_Q15FromString	ASCII to _Q15 conversion.
	_LIBQ_Q15Rand	Generate a _Q15 random number.
	_LIBQ_Q16_tan_Q2_29	Calculates the value of tan(x).
	_LIBQ_Q16Div	_Q16 fixed point divide.
	_LIBQ_Q16Exp	Calculates the exponential function e^x.
	_LIBQ_Q16Power	Calculates the value of x raised to the y power (x^y).
	_LIBQ_Q16Sqrt	Square root of a positive _Q16 fixed point value.
	_LIBQ_Q2_13_acos_Q15	Calculates the value of acos(x).
	_LIBQ_Q2_13_asin_Q15	Calculates the asin value of asin(x).
	_LIBQ_Q2_13_atan_Q7_8	Calculates the value of atan(x).
	_LIBQ_Q2_13_atan2_Q7_8	Calculates the value of atan2(y, x).
	_LIBQ_Q2_29_acos_Q31	Calculates the value of acos(x).
	_LIBQ_Q2_29_acos_Q31_Fast	Calculates the value of acos(x). This function executes faster than _LIBQ_Q2_29_acos_Q31 but is less precise.
	_LIBQ_Q2_29_asin_Q31	Calculates the value of asin(x).
	_LIBQ_Q2_29_asin_Q31_Fast	Calculates the value of asin(x). This function executes faster than the _LIBQ_Q2_29_asin_Q31 function, but is less precise.
	_LIBQ_Q2_29_atan_Q16	Calculates the value of atan(x).
	_LIBQ_Q2_29_atan2_Q16	Calculates the value of atan2(y, x).
	_LIBQ_Q3_12_log10_Q16	Calculates the value of Log10(x).
	_LIBQ_Q31_cos_Q2_29	Calculates the value of cosine(x).
	_LIBQ_Q31_sin_Q2_29	Calculates the value of sine(x).
	_LIBQ_Q31FromFloat	Converts a float to a _Q31 value.
	_LIBQ_Q31Rand	Generate a _Q31 random number.
	_LIBQ_Q4_11_ln_Q16	Calculates the natural logarithm ln(x).
	_LIBQ_Q5_10_log2_Q16	Calculates the value of log2(x).
	_LIBQ_Q7_8_tan_Q2_13	Calculates the value of tan(x).
	_LIBQ_ToFloatQ15	Converts a _Q15 value to a float.
	_LIBQ_ToFloatQ31	Converts a _Q31 value to a float.
	_LIBQ_ToStringQ15	_Q15 to ASCII conversion.

Description

Functions

_LIBQ_Q15_cos_Q2_13 Function

Calculates the value of cosine(x).

Description

Function _LIBQ_Q15_cos_Q2_13:

_Q15 _LIBQ_Q15_cos_Q2_13 (_Q2_13 x);

Calculates the cosine(x), where x is of type _Q2_13 radians and the resulting value is of type _Q15.

Preconditions

None

Parameters

Parameters	Description
x	The _Q2_13 input value from which to calculate cosine(x).

Returns

_LIBQ_Q15_cos_Q2_13 returns the _Q15 fixed point result from the calculation cosine(x).

Remarks

Execution Time (cycles): 102 cycles Program Memory 224 bytes

Error <= 0.00003052 (accurate to least significant _Q15 bit)

Example

```
_Q15 resultCos;

resultCos = _LIBQ_Q15_cos_Q2_13 ((_Q2_13)0x2171);  // _LIBQ_Q15_cos_Q2_13(1.045044) = 0.501862
(0x403d)
```

C

```
_Q15 _LIBQ_Q15_cos_Q2_13(_Q2_13 x);
```

_LIBQ_Q15_sin_Q2_13 Function

Calculates the value of sine(x).

Description

Function _LIBQ_Q15_sin_Q2_13:

_Q15 _LIBQ_Q15_sin_Q2_13 (_Q2_13 x);

Calculates the sine(x), where x is of type _Q2_13 radians and the resulting value is of type _Q15.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q2_13 input value from which to calculate sine(x).

Returns

_LIBQ_Q15_sin_Q2_13 returns the _Q15 fixed point result from the calculation sine(x).

Remarks

Execution Time (cycles): 100 typical (100 to 102) Program Memory 220 bytes

Error <= 0.00003052 (accurate to least significant _Q15 bit)

Example

```
_Q15 resultSin;

resultSin = _LIBQ_Q15_sin_Q2_13 ((_Q2_13)0x4093);  // _LIBQ_Q15_sin_Q2_13(2.017944) = 0.901672
(0x736a)
```

C

```
_Q15 _LIBQ_Q15_sin_Q2_13(_Q2_13 x);
```

_LIBQ_Q15FromFloat Function

Converts a float to a _Q15 value.

Description

Function _LIBQ_Q15FromFloat:

_Q15 _LIBQ_Q15FromFloat(float x);

Converts a floating point value to a _Q15 fixed point representation. The _Q15 fixed point value is returned by the function. The conversion will saturate if the value is outside the range of the _Q15 representation.

Preconditions

None.

Parameters

Parameters	Description
x	The float point value to convert to _Q15 fixed point

Returns

_LIBQ_Q15FromFloat returns the _Q15 fixed point value corresponding to the floating point (float) input value.

Remarks

The C library functions __gesf2, __lesf2, __addsf3, __mulsf3, and __fixsfsi are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 213 typical (158 to 224) Program Memory 96 bytes

Example

```
_Q15 q15;

q15 = _LIBQ_Q15FromFloat((float)0.5);  // q15 now equals (_Q15)0x4000

q15 = _LIBQ_Q15FromFloat((float)-1.0);  // q15 now equals (_Q15)0x8000

q15 = _LIBQ_Q15FromFloat((float)-0.233828);  // q15 now equals (_Q15)0xe212
```

C

```
_Q15 _LIBQ_Q15FromFloat(float x);
```

_LIBQ_Q15FromString Function

ASCII to _Q15 conversion.

Description

Function _LIBQ_Q15FromString:

_Q15 _LIBQ_Q15FromString(char *s);

Convert an ASCII string into a _Q15 fixed point value. The ASCII string must be in an -N.NNNNNN format. Leading spaces are ignored. The conversion stops at either the first non-conforming character in the string or the Null string terminator. There must be no spaces within the string value itself.

Preconditions

None.

Parameters

Parameters	Description
s	A pointer to the ASCII input string representing the _Q15 fixed point value.

Returns

_LIBQ_Q15FromString returns the _Q15 fixed point value represented by the input string.

Remarks

Execution Time (cycles): 296 typical (28 to 346) Program Memory 172 bytes

Example

```
_Q15  x;

x = _LIBQ_Q15FromString("0.125");      // x will equal 0.125 using
                              // an internal value of 0x1000

x = _LIBQ_Q15FromString("-1.0");       // x will equal -1.0 using
                              // an internal value of 0x8000

x = _LIBQ_Q15FromString("0.999969");   // x will equal 0.999969 using
                              // an internal value of 0x7FFF
```

C

```
_Q15 _LIBQ_Q15FromString(char * s);
```

_LIBQ_Q15Rand Function

Generate a _Q15 random number.

Description

Function _LIBQ_Q15Rand:

_Q15 _LIBQ_Q15Rand (int64_t *pSeed);

Generates a _Q15 pseudo-random value based on the seed supplied as a parameter. The first time this function is called, the seed value must be supplied by the user; this initial seed value can either be constant or random, depending on whether the user wants to generate a repeatable or a non-repeatable pseudo-random sequence.

The function updates the *pSeed value each time it is called. The updated *pSeed value must be passed back to the function with each subsequent call.

Warning: The pseudo-random sequence generated by this function may be insufficient for cryptographic use.

Preconditions

None.

Parameters

Parameters	Description
pSeed	A pointer to the seed value used by the function to generate a pseudo-random sequence.

Returns

_LIBQ_Q15Rand returns a random _Q15 value. _LIBQ_Q15Rand also updates the int64_t *pSeed value.

Remarks

Execution Time (cycles): 32 Program Memory 92 bytes

Example

```
// Initialize seed to a constant or random value
static int64_t  randomSeed = 0xA71078BE72D4C1F1;

_Q15           randomValue;

randomValue = _LIBQ_Q15Rand(&randomSeed);
...
randomValue = _LIBQ_Q15Rand(&randomSeed);
```

C

```
_Q15 _LIBQ_Q15Rand(int64_t * pSeed);
```

_LIBQ_Q16_tan_Q2_29 Function

Calculates the value of tan(x).

Description

Function _LIBQ_Q16_tan_Q2_29:

_Q16 _LIBQ_Q16_tan_Q2_29 (_Q2_29 x);

Calculates the tan(x), where x is of type _Q2_29 radians and the resulting value is of type _Q16.

Preconditions

None

Parameters

Parameters	Description
x	The _Q2_29 input value from which to calculate tan(x).

Returns

_LIBQ_Q16_tan_Q2_29 returns the _Q16 fixed point result from the calculation tan(x). The resulting value is saturated.

Remarks

The functions _LIBQ_Q31_sin_Q2_29, _LIBQ_Q31_cos_Q2_29, and _LIBQ_Q16Div are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 703 typical (22 to 796) Program Memory 88 bytes

Error <= 0.000015259 (accurate to least significant _Q16 bit for the input range -1.568 .. 1.568) Error rises from 0.0 to 0.065 for the input range -1.568 .. -1.570765808 and 1.568 .. 1.570765808)

Example

```
_Q16 resultTan;

resultTan = _LIBQ_Q16_tan_Q2_29 ((_Q2_29)0x16720c36);  // _LIBQ_Q16_tan_Q2_29( 0.701421838) =
0.844726562 (0x0000d840)
```

C

```
_Q16 _LIBQ_Q16_tan_Q2_29(_Q2_29 x);
```

_LIBQ_Q16Div Function

_Q16 fixed point divide.

Description

Function _LIBQ_Q16Div:

_Q16 _LIBQ_Q16Div (_Q16 dividend, _Q16 divisor);

Quotient (_Q16) = Dividend (_Q16) / Divisor (_Q16).

Preconditions

Divisor must not equal 0.

Parameters

Parameters	Description
dividend	The divide operation dividend (_Q16)
divisor	The divide operation divisor (_Q16)

Returns

_Q16 quotient of the divide operation

Remarks

The _LIBQ_Q16Div operation saturates its result.

Execution Time (cycles): 143 typical (80 to 244) Program Memory 204 bytes

Error <= 0.000015258789 (accurate to least significant _Q16 bit within the non-saturated range)

Example

```
_Q16  quotient, dividend, divisor;

dividend = (_Q16)0x00010000;  // 1
divisor  = (_Q16)0x00008000;  // 0.5

quotient = _LIBQ_Q16Div (dividend, divisor);

// quotient now equals 2;  i.e.,  (_Q16)0x00020000;
```

C

```
_Q16 _LIBQ_Q16Div(_Q16 dividend, _Q16 divisor);
```

_LIBQ_Q16Exp Function

Calculates the exponential function e^x.

Description

Function _LIBQ_Q16Exp:

_Q16 _LIBQ_Q16Exp(_Q16 x);

Calculates the exponential function e^x. The calculation will saturate if the resulting value is outside the range of the _Q16 representation. For x > 10.3972015380859375, the resulting value will be saturated to 0x7fffffff. For x < -10.3972015380859375 the resulting value will be saturated to 0.

Preconditions

None.

Parameters

Parameters	Description
x	The exponent value

Returns

_LIBQ_Q16Exp returns the _Q16 fixed point result from the calculation e^x.

Remarks

The function _LIBQ_Q16Div is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 170 typical (18 to 292) Program Memory 446 bytes

Error <= 0.000015258789 (accurate to least significant _Q16 bit within the non-saturated range)

Example

```
_Q16 expResult;

expResult = _LIBQ_Q16Exp((_Q16)0x00010000);   // _LIBQ_Q16Exp(1.000000) = 2.718277 (0x0002b7e1)
```

C

```
_Q16 _LIBQ_Q16Exp(_Q16 x);
```

_LIBQ_Q16Power Function

Calculates the value of x raised to the y power (x^y).

Description

Function _LIBQ_Q16Power:

_Q16 _LIBQ_Q16Power (_Q16 x, _Q16 y);

Calculates the x raised to the y power. Both x and y are of type _Q16. x must be positive. The calculation will saturate if the resulting value is outside the range of the _Q16 representation.

Preconditions

x must be positive.

Parameters

Parameters	Description
x	The _Q16 input value x from which to calculate x raised to the y.
y	The _Q16 input value y from which to calculate x raised to the y.

Returns

_LIBQ_Q16Power returns the _Q16 fixed point result from the calculation x raised to the y.

Remarks

Execution Time (cycles): 882 typical (586 to 1042) Program Memory 1038 bytes

Error <= 0.000015258789 (accurate to least significant _Q16 bit within the non-saturated range)

Example

```
_Q16 resultPower;

resultPower = _LIBQ_Q16Power ((_Q16)0x00020000, (_Q16)0xffff0000);  // _LIBQ_Q16Power(2.000000,
-1.000000) = 0.500000 (0x00008000)
```

C

```
_Q16 _LIBQ_Q16Power(_Q16 x, _Q16 y);
```

_LIBQ_Q16Sqrt Function

Square root of a positive _Q16 fixed point value.

Description

Function _LIBQ_Q16Sqrt:

_Q16 _LIBQ_Q16Sqrt(_Q16 x);

Calculate the square root of a positive _Q16 fixed point value, and return the _Q16 result.

Preconditions

The input value must be positive.

Parameters

Parameters	Description
x	The _Q16 fixed point value input from which to find the square root.

Returns

_LIBQ_Q16Sqrt returns the _Q16 fixed point value which is the square root of the input parameter.

Remarks

Execution Time (cycles): 240 typical (104 to 258) Program Memory 152 bytes

Error <= 0.000015258789 (accurate to least significant _Q16 bit)

Example

```
_Q16 squareRoot;

squareRoot = _LIBQ_Q16Sqrt((_Q16)0x01000000);  // The square root of 256.0 is 16.0 (0x00100000)

squareRoot = _LIBQ_Q16Sqrt((_Q16)0x00004000);  // The square root of 0.25 is 0.5 (0x00008000)

squareRoot = _LIBQ_Q16Sqrt((_Q16)0x5851f42d);  // The square root of 22609.953125 is 150.366074
(0x00965db7)
```

C

```
_Q16 _LIBQ_Q16Sqrt(_Q16 x);
```

_LIBQ_Q2_13_acos_Q15 Function

Calculates the value of acos(x).

Description

Function _LIBQ_Q2_13_acos_Q15:

_Q2_13 _LIBQ_Q2_13_acos_Q15 (_Q15 x);

Calculates the acos(x), where x is of type _Q15 and the resulting value is of type _Q2_13. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q15 input value from which to calculate acos(x).

Returns

_LIBQ_Q2_13_acos_Q15 returns the _Q2_13 fixed point result from the calculation acos(x).

Remarks

The function _LIBQ_Q2_13_asin_Q15 is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 588 typical (32 to 666) Program Memory 24 bytes

Error <= 0.00012207 (accurate to least significant _Q2_13 bit)

A higher precision function with equivalent performance exists, see _LIBQ_Q2_29_acos_Q31_Fast

Example

```
_Q2_13 resultAcos;

resultAcos = _LIBQ_Q2_13_acos_Q15((_Q15)0x2993); // _LIBQ_Q2_13_acos_Q15(0.324799) = 1.239990
(0x27ae)
```

C

```
_Q2_13 _LIBQ_Q2_13_acos_Q15(_Q15 x);
```

_LIBQ_Q2_13_asin_Q15 Function

Calculates the asin value of asin(x).

Description

Function _LIBQ_Q2_13_asin_Q15:

_Q2_13 _LIBQ_Q2_13_asin_Q15 (_Q15 x);

Calculates asin(x), where x is of type _Q15 and the resulting value is of type _Q2_13. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q15 input value from which to calculate asin(x).

Returns

_LIBQ_Q2_13_asin_Q15 returns the _Q2_13 fixed point result from the calculation asin(x).

Remarks

The functions _LIBQ_Q16Sqrt and _LIBQ_Q16Div are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 578 typical (22 to 656) Program Memory 336 bytes

Error <= 0.00012207 (accurate to least significant _Q2_13 bit)

A higher resolution version of this function exists with equivalent performance, see _LIBQ_Q2_29_asin_Q31_Fast

Example

```
_Q2_13 resultAsin;

resultAsin = _LIBQ_Q2_13_asin_Q15 ((_Q15)0x3231);  // _LIBQ_Q2_13_asin_Q15(0.392120) = 0.402954
(0x0ce5)
```

C

```
_Q2_13 _LIBQ_Q2_13_asin_Q15(_Q15 x);
```

_LIBQ_Q2_13_atan_Q7_8 Function

Calculates the value of atan(x).

Description

Function _LIBQ_Q2_13_atan_Q7_8:

_Q2_13 _LIBQ_Q2_13_atan_Q7_8 (_Q7_8 x);

Calculates the atan(x), where x is of type _Q7_8 and the resulting value is of type _Q2_13. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q7_8 input value from which to calculate atan(x).

Returns

_LIBQ_Q2_13_atan_Q7_8 returns the _Q2_13 fixed point result from the calculation atan(x).

Remarks

The function _LIBQ_Q2_13_atan2_Q7_8 is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 240 typical (202 to 256) Program Memory 16 bytes

Error <= 0.00012207 (accurate to least significant _Q2_13 bit)

Example

```
_Q2_13 resultAtan;

resultAtan = _LIBQ_Q2_13_atan_Q7_8 ((_Q7_8)0x0097);  // _LIBQ_Q2_13_atan_Q7_8(0.589844) =
0.532959 (0x110e)
```

C

```
_Q2_13 _LIBQ_Q2_13_atan_Q7_8(_Q7_8 x);
```

_LIBQ_Q2_13_atan2_Q7_8 Function

Calculates the value of atan2(y, x).

Description

Function _LIBQ_Q2_13_atan2_Q7_8:

_Q2_13 _LIBQ_Q2_13_atan2_Q7_8 (_Q7_8 y, _Q7_8 x);

Calculates the atan2(y, x), where y and x are of type _Q7_8 and the resulting value is of type _Q2_13. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
y	The _Q7_8 input value from which to calculate atan2(y, x).
x	The _Q7_8 input value from which to calculate atan2(y, x).

Returns

_LIBQ_Q2_13_atan2_Q7_8 returns the _Q2_13 fixed point result from the calculation atan2(y, x).

Remarks

The function _LIBQ_Q16Div is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 220 typical (22 to 250) Program Memory 288 bytes

Error <= 0.00012207 (accurate to least significant _Q2_13 bit)

Example

```
_Q2_13 resultAtan2;

resultAtan2 = _LIBQ_Q2_13_atan2_Q7_8 ((_Q7_8)0x589d, (_Q7_8)0xf878); //
_LIBQ_Q2_13_atan2_Q7_8(88.613281, -7.531250) = 1.655518 (0x34fa)
```

C

```
_Q2_13 _LIBQ_Q2_13_atan2_Q7_8(_Q7_8 y, _Q7_8 x);
```

_LIBQ_Q2_29_acos_Q31 Function

Calculates the value of acos(x).

Description

Function _LIBQ_Q2_29_acos_Q31:

_Q2_29 _LIBQ_Q2_29_acos_Q31 (_Q31 x);

Calculates the acos(x), where x is of type _Q31 and the resulting value is of type _Q2_29. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate acos(x).

Returns

_LIBQ_Q2_29_acos_Q31 returns the _Q2_29 fixed point result from the calculation acos(x).

Remarks

The functions _LIBQ_Q2_29_asin_Q31_Fast and _LIBQ_Q31_cos_Q2_29 are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 3370 typical (70 to 4824) Program Memory 142 bytes

Error <= 0.0000000019 (accurate to least significant _Q2_29 bit for the range -0.9993..0.9993) Error <= 0.0000000355 (accurate to 5th least significant _Q2_29 bit for the range -1.0 .. -0.9993 and 0.9993 .. 1.0)

A similar function with higher performance and reduced precision exists, see _LIBQ_Q2_29_acos_Q31_Fast

Example

```
_Q2_29 resultAcos;

resultAcos = _LIBQ_Q2_29_acos_Q31 ((_Q31)0xee63708c);  // _LIBQ_Q2_29_acos_Q31(-0.1375903431)
=  1.7088244837 (0x36aeb0af)
```

C

```
_Q2_29 _LIBQ_Q2_29_acos_Q31(_Q31 x);
```

_LIBQ_Q2_29_acos_Q31_Fast Function

Calculates the value of acos(x). This function executes faster than _LIBQ_Q2_29_acos_Q31 but is less precise.

Description

Function _LIBQ_Q2_29_acos_Q31_Fast:

_Q2_29 _LIBQ_Q2_29_acos_Q31_Fast (_Q31 x);

Calculates the acos(x), where x is of type _Q31 and the resulting value is of type _Q2_29. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate acos(x).

Returns

_LIBQ_Q2_29_acos_Q31_Fast returns the _Q2_29 fixed point result from the calculation acos(x).

Remarks

The function _LIBQ_Q2_29_asin_Q31_Fast is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 517 typical (32 to 1310) Program Memory 28 bytes

Error <= 0.000000911 (accurate to 9 least significant _Q2_29 bits)

A higher precision function with reduced performance exists, see _LIBQ_Q2_29_acos_Q31

Example

```
_Q2_29 resultAcos;

resultAcos = _LIBQ_Q2_29_acos_Q31_Fast ((_Q31)0xee63708c);  //
_LIBQ_Q2_29_acos_Q31_Fast(-0.1375903431) =  1.7088244837 (0x36aeb0af)
```

C

```
_Q2_29 _LIBQ_Q2_29_acos_Q31_Fast(_Q31 x);
```

_LIBQ_Q2_29_asin_Q31 Function

Calculates the value of asin(x).

Description

Function _LIBQ_Q2_29_asin_Q31:

_Q2_29 _LIBQ_Q2_29_asin_Q31 (_Q31 x);

Calculates the asin(x), where x is of type _Q31 and the resulting value is of type _Q2_29. The output value will be in radians the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate asin(x).

Returns

_LIBQ_Q2_29_asin_Q31 returns the _Q2_29 fixed point result from the calculation asin(x).

Remarks

The functions _LIBQ_Q2_29_asin_Q31_Fast and_LIBQ_Q31_sin_Q2_29 are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 2525 typical (286 to 4330) Program Memory 138 bytes

Error <= 0.0000000019 (accurate to least significant _Q2_29 bit for the range -0.9993..0.9993) Error <= 0.0000000346 (accurate to 5th least significant _Q2_29 bit for the range -1.0 .. -0.9993 and 0.9993 .. 1.0)

A faster version of this function exists with modestly reduced accuracy, see _LIBQ_Q2_29_asin_Q31_Fast

Example

```
_Q2_29 resultAsin;

resultAsin = _LIBQ_Q2_29_asin_Q31 ((_Q31)0x7fe50658);  // _LIBQ_Q2_29_asin_Q31( 0.9991767816)
=  1.5302172359 (0x30f78a23)
```

C

```
_Q2_29 _LIBQ_Q2_29_asin_Q31(_Q31 x);
```

_LIBQ_Q2_29_asin_Q31_Fast Function

Calculates the value of asin(x). This function executes faster than the _LIBQ_Q2_29_asin_Q31 function, but is less precise.

Description

Function _LIBQ_Q2_29_asin_Q31_Fast:

_Q2_29 _LIBQ_Q2_29_asin_Q31_Fast (_Q31 x);

Calculates the asin(x), where x is of type _Q31 and the resulting value is of type _Q2_29. The output value will be in radians the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q31 input value from which to calculate asin(x).

Returns

_LIBQ_Q2_29_asin_Q31_Fast returns the _Q2_29 fixed point result from the calculation asin(x).

Remarks

Execution Time (cycles): 507 typical (22 to 1300) Program Memory 638 bytes

Error <= 0.000000911 (accurate to 9 least significant _Q2_29 bits)

A higher resolution version of this function exists with reduced performance, see _LIBQ_Q2_29_asin_Q31

Example

```
_Q2_29 resultAsin;

resultAsin = _LIBQ_Q2_29_asin_Q31_Fast ((_Q31)0x7fe50658);  // _LIBQ_Q2_29_asin_Q31_Fast(
0.9991767816) =  1.5302172359 (0x30f78a23)
```

C

```
_Q2_29 _LIBQ_Q2_29_asin_Q31_Fast(_Q31 x);
```

_LIBQ_Q2_29_atan_Q16 Function

Calculates the value of atan(x).

Description

Function _LIBQ_Q2_29_atan_Q16:

_Q2_29 _LIBQ_Q2_29_atan_Q16 (_Q16 x);

Calculates the atan(x), where x is of type _Q16 and the resulting value is of type _Q2_29. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q16 input value from which to calculate atan(x).

Returns

_LIBQ_Q2_29_atan_Q16 returns the _Q2_29 fixed point result from the calculation atan(x).

Remarks

The function _LIBQ_Q2_29_atan2_Q16 is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 354 typical (178 to 360) Program Memory 16 bytes

Error <= 0.000000003 (accurate within one least significant _Q2_29 bit)

Example

```
_Q2_29 resultAtan;

resultAtan = _LIBQ_Q2_29_atan_Q16 ((_Q16)0x00098b31);  // _LIBQ_Q2_29_atan_Q16(9.543716) =
1.466396 (0x2eecb7ee)
```

C

```
_Q2_29 _LIBQ_Q2_29_atan_Q16(_Q16 x);
```

_LIBQ_Q2_29_atan2_Q16 Function

Calculates the value of atan2(y, x).

Description

Function _LIBQ_Q2_29_atan2_Q16:

_Q2_29 _LIBQ_Q2_29_atan2_Q16 (_Q16 y, _Q16 x);

Calculates the atan(y, x), where y and x are of type _Q16 and the resulting value is of type _Q2_29. The output value will be radians in the range pi >= result >= -pi.

Preconditions

None.

Parameters

Parameters	Description
y	The _Q16 input value from which to calculate atan2(y, x).
x	The _Q16 input value from which to calculate atan2(y, x).

Returns

_LIBQ_Q2_29_atan2_Q16 returns the _Q2_29 fixed point result from the calculation atan2(y, x).

Remarks

The C function __divdi3 is called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 348 typical (20 to 376) Program Memory 464 bytes

Error <= 0.000000003 (accurate within one least significant _Q2_29 bit)

Example

```
_Q2_29 resultAtan2;

resultAtan2 = _LIBQ_Q2_29_atan2_Q16 ((_Q16)0xf6276270, x(_Q16)0x34b4b4c0);  //
_LIBQ_Q2_29_atan2_Q16(-2520.615479, 13492.706055) = -0.184684 (0xfa1710c7)
```

C

```
_Q2_29 _LIBQ_Q2_29_atan2_Q16(_Q16 y, _Q16 x);
```

_LIBQ_Q3_12_log10_Q16 Function

Calculates the value of Log10(x).

Description

Function _LIBQ_Q3_12_log10_Q16:

_Q3_12 _LIBQ_Q3_12_log10_Q16 (_Q16 x);

Calculates the log10(x), where log10(x) = ln(x) * log10(e). x is of type _Q16 and must be positive. The resulting value is of type _Q3_12.

Preconditions

The input x must be positive.

Parameters

Parameters	Description
x	The input value from which to calculate log10(x).

Returns

_LIBQ_Q3_12_log10_Q16 returns the _Q3_12 fixed point result from the calculation log10(x).

Remarks

Execution Time (cycles): 301 typical (14 to 346) Program Memory 176 bytes

Error <= 0.000244140625 (accurate to least significant _Q3_12 bit)

Example

```
_Q3_12 resultLog10;

resultLog10 = _LIBQ_Q3_12_log10_Q16 ((_Q16)0x12ed7d91);  // _LIBQ_Q3_12_log10_Q16(4845.490494)
= 3.685303 (0x3af7)
```

C

```
_Q3_12 _LIBQ_Q3_12_log10_Q16(_Q16 x);
```

_LIBQ_Q31_cos_Q2_29 Function

Calculates the value of cosine(x).

Description

Function _LIBQ_Q31_cos_Q2_29:

_Q31 _LIBQ_Q31_cos_Q2_29 (_Q2_29 x);

Calculates the cosine(x), where x is of type _Q2_29 radians and the resulting value is of type _Q31.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q2_29 input value from which to calculate cosine(x).

Returns

_LIBQ_Q31_cos_Q2_29 returns the _Q31 fixed point result from the calculation sine(x).

Remarks

Execution Time (cycles): 265 typical (22 to 288) Program Memory 746 bytes

Error <= 0.00000000047 (accurate to least significant _Q31 bit)

Example

```
_Q31 resultCos;

resultCos = _LIBQ_Q31_cos_Q2_29 ((_Q2_29)0x07e2e1c2);  // _LIBQ_Q31_cos_Q2_29( 0.246445540) =
0.969785686 (0x7c21eff7)
```

C

```
_Q31 _LIBQ_Q31_cos_Q2_29(_Q2_29 x);
```

_LIBQ_Q31_sin_Q2_29 Function

Calculates the value of sine(x).

Description

Function _LIBQ_Q31_sin_Q2_29:

_Q31 _LIBQ_Q31_sin_Q2_29 (_Q2_29 x);

Calculates the sine(x), where x is of type _Q2_29 radians and the resulting value is of type _Q31.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q2_29 input value from which to calculate sine(x).

Returns

_LIBQ_Q31_sin_Q2_29 returns the _Q31 fixed point result from the calculation sine(x).

Remarks

Execution Time (cycles): 246 typical (244 to 266) Program Memory 598 bytes

Error <= 0.00000000047 (accurate to least significant _Q31 bit)

Example

```
_Q31 resultSin;

resultSin = _LIBQ_Q31_sin_Q2_29 ((_Q2_29)0x5a637cfe);  // _LIBQ_Q31_sin_Q2_29( 2.824644562) =
```

```
0.311668121 (0x27e4bdb1)
```

C

```
_Q31 _LIBQ_Q31_sin_Q2_29(_Q2_29 x);
```

_LIBQ_Q31FromFloat Function

Converts a float to a _Q31 value.

Description

Function _LIBQ_Q31FromFloat:

_Q31 _LIBQ_Q31FromFloat(float x);

Converts a floating point value to a _Q31 fixed point representation. The _Q31 fixed point value is returned by the function. The conversion will saturate if the value is outside the range of the _Q31 representation.

Preconditions

None.

Parameters

Parameters	Description
x	The floating point value to convert to _Q31 fixed point.

Returns

_LIBQ_Q31FromFloat returns the _Q31 fixed point value corresponding to the floating point (float) input value.

Remarks

The C library functions __gesf2, __lesf2, __addsf3, __mulsf3, and __fixsfsi are called by this routine and thus must be linked into the executable image.

Execution Time (cycles): 210 typical (158 to 214) Program Memory 100 bytes

Example

```
_Q31 q31;

q31 = _LIBQ_Q31FromFloat((float)0.000008);  // q31 now equals (_Q31)0x00004000

q31 = _LIBQ_Q31FromFloat((float)-1.0);  // q31 now equals (_Q31)0x80000000

q31 = _LIBQ_Q31FromFloat((float)0.690001);  // q31 now equals (_Q31)0x5851f400
```

C

```
_Q31 _LIBQ_Q31FromFloat(float x);
```

_LIBQ_Q31Rand Function

Generate a _Q31 random number.

Description

Function _LIBQ_Q31Rand:

_Q31 _LIBQ_Q31Rand (int64_t *pSeed);

Generates a _Q31 pseudo-random value based on the seed supplied as a parameter. The first time this function is called, the seed value must be supplied by the user; this initial seed value can either be constant or random, depending on whether the user wants to generate a repeatable or a non-repeatable pseudo-random sequence.

The function updates the *pSeed value each time it is called. The updated *pSeed value must be passed back to the function with each subsequent call.

Warning: The pseudo-random sequence generated by this function may be insufficient for cryptographic use.

Preconditions

None.

Parameters

Parameters	Description
pSeed	A pointer to the seed value used by the function to generate a pseudo-random sequence.

Returns

_LIBQ_Q31Rand returns a pseudo-random _Q31 value. _LIBQ_Q31Rand also updates the int64_t *pSeed value.

Remarks

Execution Time (cycles): 32 Program Memory 88 bytes

Example

```
// Initialize seed to a constant or random value
static int64_t  randomSeed = 0x7F18BA710E72D4C1;

_Q31  randomValue;

randomValue = _LIBQ_Q31Rand(&randomSeed);
...
randomValue = _LIBQ_Q31Rand(&randomSeed);
```

C

```
_Q31 _LIBQ_Q31Rand(int64_t * pSeed);
```

_LIBQ_Q4_11_ln_Q16 Function

Calculates the natural logarithm ln(x).

Description

Function _LIBQ_Q4_11_ln_Q16:

_Q4_11 _LIBQ_Q4_11_ln_Q16 (_Q16 x);

Calculates the natural logarithm ln(x). x is of type _Q16 and must be positive. The resulting value is of type _Q4_11.

Preconditions

The input x must be positive.

Parameters

Parameters	Description
x	The input value from which to calculate ln(x).

Returns

_LIBQ_Q4_11_ln_Q16 returns the _Q4_11 fixed point result from the calculation ln(x).

Remarks

Execution Time (cycles): 301 typical (14 to 346) Program Memory 176 bytes

Error <= 0.00048828 (accurate to least significant _Q4_11 bit)

Example

```
_Q4_11 resultLN;

resultLN = _LIBQ_Q4_11_ln_Q16 ((_Q16)0x00004000);  // _LIBQ_Q4_11_LN_Q16(0.250000) = -1.386230
(0xf4e9)
```

C

```
_Q4_11 _LIBQ_Q4_11_ln_Q16(_Q16 x);
```

_LIBQ_Q5_10_log2_Q16 Function

Calculates the value of log2(x).

Description

Function _LIBQ_Q5_10_log2_Q16:

_Q5_10 _LIBQ_Q5_10_log2_Q16 (_Q16 x);

Calculates the log2(x), where log2(x) = ln(x) * log2(e). x is of type _Q16 and must be positive. The resulting value is of type _Q5_10.

Preconditions

The input x must be positive.

Parameters

Parameters	Description
x	The input value from which to calculate log2(x).

Returns

_LIBQ_Q5_10_log2_Q16 returns the _Q5_10 fixed point result from the calculation log2(x).

Remarks

Execution Time (cycles): 227 typical (14 to 268) Program Memory 164 bytes

Error <= 0.0009765625 (accurate to least significant _Q5_10 bit)

Example

```
_Q5_10 resultLog2;

resultLog2 = _LIBQ_Q5_10_log2_Q16 ((_Q16)0x40000000);  // _LIBQ_Q5_10_log2_Q16(16384.000000) =
14.000000 (0x3800)
```

C

```
_Q5_10 _LIBQ_Q5_10_log2_Q16(_Q16 x);
```

_LIBQ_Q7_8_tan_Q2_13 Function

Calculates the value of tan(x).

Description

Function _LIBQ_Q7_8_tan_Q2_13:

_Q7_8 _LIBQ_Q7_8_tan_Q2_13 (_Q2_13 x);

Calculates the tan(x), where x is of type _Q2_13 radians and the resulting value is of type _Q7_8.

Preconditions

None

Parameters

Parameters	Description
x	The _Q2_13 input value from which to calculate tan(x).

Returns

_LIBQ_Q7_8_tan_Q2_13 returns the _Q7_8 fixed point result from the calculation tan(x).

Remarks

Execution Time (cycles): 288 typical (18 to 346) Program Memory 980 bytes

Error <= 0.00390625 (accurate to least significant _Q7_8 bit)

Example

```
_Q7_8 resultTan;

resultTan = _LIBQ_Q7_8_tan_Q2_13 ((_Q2_13)0x2e20);  // _LIBQ_Q7_8_tan_Q2_13(1.441406) =
7.683594 (0x07af)
```

C

```
_Q7_8 _LIBQ_Q7_8_tan_Q2_13(_Q2_13 x);
```

_LIBQ_ToFloatQ15 Function

Converts a _Q15 value to a float.

Description

Function _LIBQ_ToFloatQ15:

float _LIBQ_ToFloatQ15(_Q15 x);

Converts a _Q15 fixed point value to a floating point representation. The floating point value is returned by the function.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q15 fixed point value to convert to float

Returns

_LIBQ_ToFloatQ15 returns the floating point (float) value corresponding to the _Q15 input value.

Remarks

The C library functions __floatsisf and __divsf3 are called by this routine and thus must be linked in to the executable image.

Execution Time (cycles): 158 typical (54 to 176) Program Memory 28 bytes

Example

```
float f;

f = _LIBQ_ToFloatQ15((_Q15)0x4000);  // f now equals 0.5

f = _LIBQ_ToFloatQ15((_Q15)0x8000);  // f now equals -1.0

f = _LIBQ_ToFloatQ15((_Q15)0xb7ff);  // f now equals -0.562531
```

C

```
float _LIBQ_ToFloatQ15(_Q15 x);
```

_LIBQ_ToFloatQ31 Function

Converts a _Q31 value to a float.

Description

Function _LIBQ_ToFloatQ31:

float _LIBQ_ToFloatQ31(_Q31 x);

Converts a _Q31 fixed point value to a floating point representation. The floating point value is returned by the function.

Preconditions

None.

Parameters

Parameters	Description
x	The _Q31 fixed point value to convert to float

Returns

_LIBQ_ToFloatQ31 returns the floating point (float) value corresponding to the _Q31 input value.

Remarks

The C library functions __floatsisf and __divsf3 are called by this routine and thus must be linked in to the executable image.

Execution Time (cycles): 163 typical (54 to 176) Program Memory 28 bytes

Example

```
float f;

f = _LIBQ_ToFloatQ31((_Q31)0x00004000);  // f now equals 0.000008

f = _LIBQ_ToFloatQ31((_Q31)0x80000000);  // f now equals -1.0

f = _LIBQ_ToFloatQ31((_Q31)0x5851f42d);  // f now equals 0.690001
```

C

```
float _LIBQ_ToFloatQ31(_Q31 x);
```

_LIBQ_ToStringQ15 Function

_Q15 to ASCII conversion.

Description

Function _LIBQ_ToStringQ15:

void _LIBQ_ToStringQ15(_Q15 x, char *s);

Convert a _Q15 fixed point value to an ASCII string representation in a -N.NNNNNN format.

Preconditions

The character string "s" must be at least 10 characters long, including the Null string terminator.

Parameters

Parameters	Description
x	The fixed point value to be converted into an ASCII string (_Q15)
s	A pointer to the output string of at least 10 characters

Returns

An ASCII string that represents the _Q15 fixed point value in -N.NNNNNN format. The output string will be terminated by a Null (0x00) character.

Remarks

Execution Time (cycles): 118 typical (28 to 132) Program Memory 200 bytes

Example

```
char  s[10];

_LIBQ_ToStringQ15((_Q15)0x1000, s);      // s will equal "0.125000"
```

```
_LIBQ_ToStringQ15((_Q15)0x8000, s);      // s will equal "-1.000000"

_LIBQ_ToStringQ15((_Q15)0x7FFF, s);      // s will equal "0.999969"
```

C

```
void _LIBQ_ToStringQ15(_Q15 x, char * s);
```

LibQ_c

LibQ Fixed-Point “C” Math Library

Introduction

This topic describes the LibQ Fixed-Point 'C' Math Library.

Description

The LibQ Fixed-Point 'C' Math Library is written in C and is available for all processors supported by Harmony 3.

The libq_c Fixed-Point Math Library provides fixed-point math functions written in C for portability between core processors. The library uses signed fixed point types (fractional Q types specified by Qn.m) which are specifed as follows in the library names:

Qndm where:

- n is the number of data bits to the left of the radix point
- m is the number of data bits to the right of the radix point
- a signed bit is implied (unless stated otherwise)

For convenience, short names are also defined for arbitrary scaled fractional types:

- q15 is signed fractional 16 bit value
- q31 is signed fractional 32 bit value
- i16 is signed integer, i.e. Q16d0

In addition, A pseudo floating point 32 bit format (FxQFloat32) is defined that consists of 16 mantissa and a 16 bit exponent (base 2). Functions in the library are prefixed with the type of the return value and followed by argument types (in order):

libq_<result>_<name>_<args>

For example, libq_q1d15_Sin_q10d6 returns a Q1.15 value equal to the to the sine of an angle specified as a Q10.6 value (in degrees between 0 and 360).

For arbitrary scaled types (q15, q16, q31, and q32) the scaling of the result will depend on the function and the scaling of the arguments. For instance, libq_q15_Add_q15_q15(a,b) will return a scaled value type that is the two input types (which must have equivalent scaled value type).

Using the Library

Configuring the Library using MHC

Select the Audio/Math/Math Libraries component to load the libq_c, libq/dsp libraries, as shown below

The project configuration should now contain the Math Libraries block.

The configuration window (when the block is selected) looks like the following:

Remarks

The libq_c functions do not correspond to the libq fixed-point library optimized for the microaptive core processor and written in asm.

Library Overview

Table of LIBQ_C math functions:

Sine: libq_q1d15_Sin_q10d6 libq_q20d12_Sin_q20d12 Abs: libq_q15_Abs_q15 libq_q31_Abs_q31

Negate: libq_q15_Negate_q15 libq_q31_Negate_q31 Round: libq_q15_RoundL_q31

Deposit: libq_q31_DepositH_q15 libq_q31_DepositL_q15 Extract: libq_q15_ExtractH_q31 libq_q15_ExtractL_q31 Add: libq_q15_Add_q15_q15 libq_q31_Add_q31_q31 Subtract: libq_q15_Sub_q15_q15 libq_q31_Sub_q31_q31

Shift(Scale): libq_q15_ShiftLeft_q15_q15 libq_q31_ShiftLeft_q31_q15 libq_q15_ShiftRight_q15_q15 libq_q31_ShiftRight_q31_q15 libq_q15_ShiftRightRound_q15_q15 libq_q31_ShiftRightRound_q31_q15

Multiply: libq_q15_Mult_q15_q15 libq_q15_MultipyR2_q15_q15 libq_q31_Multi_q15_q31 Divide: libq_q15_DivisionWithSaturation_q15_q15

Multiply-Accumulate: libq_q31_Mac_q31_q15_q15 libq_q15_MacR_q31_q15_q15 Multiply-Subtract: libq_q31_Msu_q31_q15_q15 libq_q15_MsuR_q31_q15_q15 Exponential-Averaging: libq_q15_ExpAvg_q15_q15_q1d15

Table of LIBQ_C conversion functions: Normalize Q value: Fx16Norm Fx32Norm

Float-to-Q value: Fl2Fract16 Fl2Fract32 Fl2FxPnt16 Fl2FxPnt32 Fl2FxPnt Float-To-Integer: Fl2Int16 Fl2Int32

Library Interface

Functions

	Name	Description
	CosInDegs	Calculates fixed point Cos(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)
	Fract_10toX_Q15bfp	Calculates pow(2,x) with fixed point math.
	Fract_10toX_Q15bfpFromQ15d16	Calculates pow(2,x) with fixed point math.
	Fract_10toX_Q15d16	Calculates pow(2,x) with fixed point math.
	Fract_2toMinusX_Q15	Calculates pow(2,-x) with fixed point math.
	Fract_2toX_Q15bfp	Calculates pow(2,x) with fixed point math.

	Fract_2toX_Q15bfpFromQ15d16	Calculates pow(2,x) with fixed point math.
	Fract_2toX_Q15d16	Calculates pow(2,x) with fixed point math.
	Fract_Convert_Q15bfpToQ15d16	Convert Q15 block floating point to Q15.16 fixed point.
	Fract_Convert_Q15bfpToQ31	Convert Q15 block floating point to Q31 fixed point.
	Fract_Convert_Q15d16ToQ15bfp	Convert from Q15.16 fixed point to Q15 Block Floating Point
	Fract_Convert_Q15ToQ15bfp	Convert from Q15 fixed point to Q15 Block Floating Point
	Fract_InversePower	Computes pow(nBase,-nExponent).
	Fract_log10fix	Calculates fixed point base 10 logarithm using number of fractional bits specified by precision argument. Log base 10 is calculated using the log base 2 function Fract_log2fix.
	Fract_log2fix	Calculates fixed point base 2 logarithm using number of fractional bits specified by precision argument.
	Fract_Power	Computes pow(nBase,nExponent) using for loop.
	Fract_sqrtQ15	Fixed point Q15 squareroot.
	Fract_XminusY_Q15bfp	Subtract two Q15 block floating point numbers.
	Fract_XoverY_Q15bfp	Division x/y as Q15 block floating point.
	Fract_XoverY_Q15bfpFromQ15	Calculate ratio of X/Y as Q15's but return result as Q15 block floating point.
	Fract_XoverY_Q15d16FromQ15	Calculates ratio X/Y for two Q15 fixed points
	Fract_XplusY_Q15bfp	Add two Q15 block floating point numbers.
	Fract_XtimesY_Q15	Calculates product of X*Y for Q15 fixed point
	Fract_XtimesY_Q15bfp	Multiply X times Y, for Q15 block floating point arguments.
	Fract_XtimesY_Q15d16	Calculates product of X*Y for Q15.16 fixed point
	Fract_XtimesY_Q31	Calculates product of X*Y for Q31 fixed point
	Fx16Norm	Normalize the 16-bit fractional value.
	Fx32Norm	Normalize the 32-bit number.
	libq_q15_Abs_q15	Saturated Absolute value.
	libq_q15_Add_q15_q15	Add two 16-bit 2s-complement fractional values.
	libq_q15_DivisionWithSaturation_q15_q15	Fractional division with saturation.
	libq_q15_ExpAvg_q15_q15_q1d15	Exponential averaging
	libq_q15_ExtractH_q31	Extracts upper 16 bits of input 32-bit fractional value.
	libq_q15_ExtractL_q31	Extracts lower 16-bits of input 32-bit fractional value. Descriptionf Extracts lower 16-bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the bottom 16-bits of the 32-bit input. This function relates to the ETSI extract_l function.
	libq_q15_MacR_q31_q15_q15	Multiply accumulate with rounding.
	libq_q15_MsuR_q31_q15_q15	Multiply-Subtraction with rounding
	libq_q15_MultiplyR2_q15_q15	fractional multiplication of two 16-bit fractional values giving a 16 bit rounded result.
	libq_q15_Negate_q15	Negate 16-bit 2s-complement fractional value with saturation.
	libq_q15_RoundL_q31	Rounds the lower 16-bits of the 32-bit fractional input.
	libq_q15_ShiftLeft_q15_i16	'Arithmetic' Shift of the 16-bit input argument.
	libq_q15_ShiftRight_q15_i16	'Arithmetic' RIGHT Shift on a 16-bit value.
	libq_q15_ShiftRightR_q15_i16	Performs an 'Arithmetic' RIGHT Shift on a 16-bit input.
	libq_q15_Sub_q15_q15	Subtract two 16-bit 2s-complement fractional values
	libq_q1d15_Sin_q10d6	Approximates the sine of an angle.
	libq_q20d12_Sin_q20d12	3rd order Polynomial apprx. of a sine function
	libq_q31_Abs_q31	Saturated Absolute value.
	libq_q31_Add_q31_q31	Add two 32-bit 2s-complement fractional values.
	libq_q31_DepositH_q15	Place 16 bits in the upper half of 32 bit word.
	libq_q31_DepositL_q15	Place 16 bits in the lower half of 32 bit word.

	libq_q31_Mac_q31_q15_q15	Multiply-Accumulate function WITH saturation
	libq_q31_Msu_q31_q15_q15	L_msu(a,b,c)
	libq_q31_Mult2_q15_q15	fractional multiplication of two 16-bit fractional values.
	libq_q31_Multi_q15_q31	Implement 16 bit by 32 bit multiply.
	libq_q31_Negate_q31	Negate 32-bit 2s-complement fractional value with saturation.
	libq_q31_ShiftLeft_q31_i16	'Arithmetic' Shift of the 32-bit value.
	libq_q31_ShiftRight_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value.
	libq_q31_ShiftRightR_q31_i16	'Arithmetic' RIGHT Shift on a 32-bit value
	libq_q31_Sub_q31_q31	Subtract two 32-bit 2s-complement fractional values
	LUTwInterpolation_Q15	Calculates calculates y = f(x) based on a lookup table with linear interpolation between table entries.
	SinInDegs	Calculates fixed point Sin(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)

Description

Functions

CosInDegs Function

fract_q0d15_t CosInDegs(fract_q9d6_t theta);

Summary

Calculates fixed point Cos(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)

Description

Calculates fixed point Cos(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)

Preconditions

None

Parameters

Parameters	Description
theta	Q9.6 fixed point angle

Returns

cos(theta) in Q0.15 fixed point

Remarks

CosInDegs uses SinInDegs.

Example

```
int32_t nTests = 400;
float theta;
float delta_theta = 360.0/(nTests-1);
float theta0 = 0;

int32_t i;
int32_t x;

for ( i = 0; i < nTests; i++ )
{
    theta = theta0 + i*delta_theta;
```

```
        x = theta*(32768/512); //Q9.6
        printf("%d, %d rn",(int)x,CosInDegs(x));
    }
```

C

```
fract_q0d15_t CosInDegs(fract_q9d6_t theta);
```

Fract_10toX_Q15bfp Function

fract_q15bfp_t Fract_10toX_Q15bfp(fract_q15bfp_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, with x as Q15 block floating point, returning the result in the same format.

Preconditions

None.

Parameters

Parameters	Description
x	exponent, in Q15 block floating point format

Returns

pow(2,x), in Q15 block floating point format

Remarks

Fract_10toX(x) is basically Fract_2toX(log2(10)*x).

Example

```
fract_q15bfp_t X,Y;

X.m = 0x4000; // 1/2
X.e = 0;
Y = Fract_10toX_Q15bfp(X);
// Y.m = 0x6532 = 25906
// Y.e = 2
// Y = ( 25906 / 32768 )<<2 = 3.162353 (vs. 3.162277) , rel error = 1/41688

X.m = 0x6488; // (3.14159/4)
X.e = 2;
Y = Fract_10toX_Q15bfp(X);
// Y.m = 0x5697 = 22167 / 32768 = 0.676483
// Y.e = 0x000B = 11
// Y = (22167 / 32768)<<11 = 1385.4375 (vs. 1385.435475), rel error = 1/684,023
```

C

```
fract_q15bfp_t Fract_10toX_Q15bfp(fract_q15bfp_t x);
```

Fract_10toX_Q15bfpFromQ15d16 Function

fract_q15bfp_t Fract_10toX_Q15bfpFromQ15d16(fract_q15d16_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, with x as Q15.16, returning result as Q15 block floating point.

Preconditions

None.

Parameters

Parameters	Description
x	exponent as Q15.16

Returns

pow(2,x) as Q15 block floating point

Remarks

Fract_10toX(x) is basically Fract_2toX(log2(10)*x).

Example

```
fract_q15bfp_t Y;

x = 0x00008000; // 0.5
Y = Fract_10toX_Q15bfpFromQ15d16(x);
// Y.m = 0x6532 = 25906
// Y.e = 2
// Y = ( 25906 / 32768 )<<2 = 3.162353 (vs. 3.162277) , rel error = 1/41688

x = 0x0003243F; // 205,887 / 65536 = 3.141586 (vs. 3.14159)
Y = Fract_10toX_Q15bfpFromQ15d16(x);
// Y.m = 0x5696 = 22167 / 32768 = 0.676483
// Y.e = 0x000B = 11
// Y = (22167 / 32768)<<11 = 1385.4375 (vs. 1385.435475), rel error = 1/684,023
```

C

```
fract_q15bfp_t Fract_10toX_Q15bfpFromQ15d16(fract_q15d16_t x);
```

Fract_10toX_Q15d16 Function

fract_q15d16_t Fract_10toX_Q15d16(fract_q15d16_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, with x as Q15.16, returning result as Q15.16.

Preconditions

None.

Parameters

Parameters	Description
x	exponent as Q15.16

Returns

pow(2,x) as Q15.16

Remarks

Fract_10toX(x) is basically Fract_2toX(log2(10)*x).

Example

```
fract_q15d16_t x,y;

x = 0x00008000; // 0.5
y = Fract_10toX_Q15d16(x);
// y = 0x0003 2990 = 207,248/65536 = 3.162353 (vs. 3.162277) , rel error = 1/41688

x = 0x0003243F; // 205,887 / 65536 = 3.141586 (vs. 3.14159)
y = Fract_10toX_Q15d16(x);
// y = 0x0569 6000 = 90,791,936/65536 = 1385.375 (vs. 1385.435475) , rel error = 1/22909
```

C

```
fract_q15d16_t Fract_10toX_Q15d16(fract_q15d16_t x);
```

Fract_2toMinusX_Q15 Function

fract_q15_t Fract_2toMinusX_Q15(fract_q15_t x);

Summary

Calculates pow(2,-x) with fixed point math.

Description

Calculates pow(2,-x) with fixed point math, using three term Taylor Series and 256 entry lookup table.

Preconditions

None.

Parameters

Parameters	Description
x	Exponent in calculating 2^-x. If x < 0 then -x is used.

Returns

Value of pow(2,-x)

Remarks

If 0 <= x < 1, then pow(2,-x) is in the range 0.5 < pow(2,-x) <= 1.

Example

```
fract_q15_t x,y;

x = 3277; // 0.1
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.1) = 0.933033 = 30574 / 32768
// y = 0x776D = 30573

x = 6554; // 0.2
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.2) = 0.870551 = 28526 / 32768
// y = 0x6F6E = 28526

x = 9830; // 0.3
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.3) = 0.812252 = 26616 / 32768
// y = 0x67F9 = 26617

x = 13107; // 0.4
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.4) = 0.757858 = 24834 / 32768
// y = 0x6102 = 24834
```

```
x = 16384; // 0.5
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.5) = 0.707107 = 23170 / 32768
// y = 0x5A81 = 23169

x = 3277 + 16384; // 0.1 + 0.5 = 0.6
y = Fract_2toMinusX_Q15(x);
// pow(2,0.6) = 0.659754 = 21619 / 32768
// y = 0x5472 = 21618

x = 6554 + 16384; // 0.2 + 0.5 = 0.7
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.7) = 0.615572 = 20171 / 32768
// y = 0x4ECA = 20170

x = 9830 + 16384; // 0.3 + 0.5 = 0.8
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.8) = 0.574349 = 18820 / 32768
// y = 0x4984 = 18820

x = 13107 + 16384; // 0.4 + 0.5 = 0.9
y = Fract_2toMinusX_Q15(x);
// pow(2,-0.9) = 0.535887 = 17560 / 32768
// y = 0x4498 = 17560

x = 32767; // 0.5 + 0.5 = 1.0
y = Fract_2toMinusX_Q15(x);
// pow(2,-1.0) = 0.5 = 16384 / 32768
// y = 0x4001 = 16385
```

C

```
fract_q15_t Fract_2toMinusX_Q15(fract_q15_t x);
```

Fract_2toX_Q15bfp Function

fract_q15bfp_t Fract_2toX_Q15bfp(fract_q15bfp_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, from Q15 block floating point x, returning Q15 block floating point result.

Preconditions

none.

Parameters

Parameters	Description
x	exponent, in Q15 block floating point format

Returns

pow(2,x), in Q15 block floating point format

Remarks

None.

Example

```
fract_q15bfp_t x, y;

x.m = 16384; // 0.5
x.e = 0;
```

```
y = Fract_2toX_Q15bfp(x);
// pow(2,0.5) = sqrt(2) = 46341/32768;
// y.m = 0x5A85 = 23173
// y.e = 1
// y = (23173/32768)<<1 = 1.414368 (vs. 1.414214)

x.m = 24576; // 0.75
x.e = 0;
y = Fract_2toX_Q15bfp(x);
// pow(2,0.75) = 1.681793 = 55109/32768;
// y.m = 0x6BA4 = 27556
// y.e = 1
// y = (27556/32768)<<1 = 1.681885 (vs. 1.681793)
```

C

```
fract_q15bfp_t Fract_2toX_Q15bfp(fract_q15bfp_t x);
```

Fract_2toX_Q15bfpFromQ15d16 Function

fract_q15bfp_t Fract_2toX_Q15bfpFromQ15d16(fract_q15d16_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, based on input in Q15.16 format, returning result in Q15 block floating point format.

Preconditions

None.

Parameters

Parameters	Description
x	exponent, in Q15.16 format

Returns

pow(2,x), in Q15 block floating point format

Remarks

None.

Example

```
fract_q15d16_t x;
fract_q15bfp_t Y;

x = 0x00008000; // 0.5 in Q15.16
Y = Fract_2toX_Q15bfpFromQ15d16(x);
// pow(2,0.5) = sqrt(2) = 1.414213 = 46341 / 32768 = 0x5A82<<1
// Y.m = 0x5A85 = 23173
// Y.e = 1
// Y = ( 23173 / 32768 )<<1 = 1.414367 (vs. 1.414213)

x = 0x00014000; // 1.25 in Q15.16
Y = Fract_2toX_Q15bfpFromQ15d16(x);
// pow(2,1.25) = 2.378414 = 77936 / 32768 = 0x4C1C<<2
// Y.m = 0x4C1 = 19484
// Y.e = 2
// Y = ( 19484/32768 ) << 2 = 2.378418 (vs. 2.378414)
```

C

```
fract_q15bfp_t Fract_2toX_Q15bfpFromQ15d16(fract_q15d16_t x);
```

Fract_2toX_Q15d16 Function

fract_q15d16_t Fract_2toX_Q15d16(fract_q15d16_t x)

Summary

Calculates pow(2,x) with fixed point math.

Description

Calculates pow(2,x) using fixed point math, with Q15.16 x, returning Q15.16 result

Preconditions

None.

Parameters

Parameters	Description
x	exponent, in Q15.16 format

Returns

pow(2,x), in Q15.16 format

Remarks

None.

Example

```
fract_q15d16_t x,y;

x = 0x00008000; // 0.5 in Q15.16
y = Fract_2toX_Q15d16(x);
// pow(2,0.5) = sqrt(2) = 1.414213 = 92682/65536 = 0x0001 6A0A
// y = 0x0001 6A14 = 92692/65536 = 1.414367 (vs. 1.414213)

x = 0x00014000; // 1.25 in Q15.16
y = Fract_2toX_Q15d16(x);
// pow(2,1.25) = 2.378414 = 155,872 / 65536 = 0x0002 60E0
// y = 0x0002 60E0
```

C

```
fract_q15d16_t Fract_2toX_Q15d16(fract_q15d16_t x);
```

Fract_Convert_Q15bfpToQ15d16 Function

fract_q15d16_t Fract_Convert_Q15bfpToQ15d16(fract_q15bfp_t x)

Summary

Convert Q15 block floating point to Q15.16 fixed point.

Description

Convert Q15 block floating point to Q15.16 fixed point.

Preconditions

None.

Parameters

Parameters	Description
x	Q15 Block Floating Point

Returns

Input converted to Q15.16 fixed point.

Remarks

No effort is made to trap overflows of underflows,

Example

```
fract_q15d16_t x_Q15d16;
fract_q15bfp_t x_Q15bfp;

x_Q15bfp.m = 0x4000; // 1/2
x_Q15bfp.e  = -1;    // 1/2>>1 = 1/4

x_Q15d16 = Fract_Convert_Q15bfpToQ15d16( x_Q15bfp );
// x_Q15d16 = 0x0000 4000
```

C

```
fract_q15d16_t Fract_Convert_Q15bfpToQ15d16(fract_q15bfp_t x);
```

Fract_Convert_Q15bfpToQ31 Function

fract_q31_t Fract_Convert_Q15bfpToQ31(fract_q15bfp_t x)

Summary

Convert Q15 block floating point to Q31 fixed point.

Description

Convert Q15 block floating point to Q31 fixed point.

Preconditions

None.

Parameters

Parameters	Description
x	Q15 Block Floating Point

Returns

Input converted to Q31 fixed point.

Remarks

No effort is made to trap overflows of underflows,

Example

C

```
fract_q31_t Fract_Convert_Q15bfpToQ31(fract_q15bfp_t x);
```

Fract_Convert_Q15d16ToQ15bfp Function

fract_q15bfp_t Fract_Convert_Q15d16ToQ15bfp(fract_q15d16_t x)

Summary

Convert from Q15.16 fixed point to Q15 Block Floating Point

Description

Convert from Q15.16 fixed point to Q15 Block Floating Point

Preconditions

None.

Parameters

Parameters	Description
x	Q15.16 fixed point

Returns

Input convertd to Q15 Block Floating Point.

Remarks

Mantissa and exponent are adjusted to remove redundant leading bits.

Example

```
fract_q15d16_t x_Q15d16;
fract_q15bfp_t x_Q15bfp;

x_Q15d16 = 0xFFFFC000; //-1/4
x_Q15bfp = Fract_Convert_Q15d16ToQ15bfp(x_Q15d16);
//x_Q15bfp.m = 0xC000  // -1/2
//x_Q15bfp.e = 0xFFFF = -1; // -1/2>>1 = -1/4
```

C

```
fract_q15bfp_t Fract_Convert_Q15d16ToQ15bfp(fract_q15d16_t x);
```

Fract_Convert_Q15ToQ15bfp Function

fract_q15bfp_t Fract_Convert_Q15ToQ15bfp(fract_q15_t x)

Summary

Convert from Q15 fixed point to Q15 Block Floating Point

Description

Convert from Q15 fixed point to Q15 Block Floating Point

Preconditions

None.

Parameters

Parameters	Description
x	Q15 (aka Q1.15) fixed point

Returns

Input convertd to Q15 Block Floating Point.

Remarks

Mantissa and exponent are adjusted to remove redundant leading bits.

Example

```
fract_q15_t    x_Q15;
fract_q15bfp_t x_Q15bfp;

x_Q15    = 0xE000; //-1/4
x_Q15bfp = Fract_Convert_Q15d16ToQ15bfp(x_Q15d16);
//x_Q15bfp.m = 0xC000  // -1/2
//x_Q15bfp.e = 0xFFFF = -1; // -1/2>>1 = -1/4
```

C

```
fract_q15bfp_t Fract_Convert_Q15ToQ15bfp(fract_q15_t x);
```

Fract_InversePower Function

fract_q15bfp_t Fract_InversePower(uint16_t nBase, uint16_t nExponent)

Summary

Computes pow(nBase,-nExponent).

Description

Computes pow(nBase,-nExponent).

Preconditions

None

Returns

pow(nBase,-nExponent), as a Q15bfp block floating point number.

Remarks

pow(nBase,nExponent) must fit inside a uint32_t variable.

Example

```
uint16_t nBase, nExponent;
fract_q15bfp_t Answer;

nBase = 5;
nExponent = 3;
Answer = Fract_InversePower(nBase,nExponent);
// Answer = pow(5,-3) = 1/125
// Answer.m = 0x4189 = 16777
// Answer.e = 0xFFFA = -6
// Answer   = (16777/32768)>>6 = 1/(125.001609)
```

C

```
fract_q15bfp_t Fract_InversePower(uint16_t nBase, uint16_t nExponent);
```

Fract_log10fix Function

int32_t Fract_log10fix (int32_t x, size_t precision)

Calculates fixed point base 10 logarithm using number of fractional bits

specified by precision argument.

Description

Calculates fixed point base 10 logarithm using number of fractional bits specified by precision argument. Log base 10 is calculated using the log base 2 function Fract_log2fix.

Preconditions

None.

Parameters

Parameters	Description
x	Fixed point input, if x == 0, function returns INT32_MIN to represent negative infinity. If x is input as a negative number it will be cast to be a unsigned 32 bit integer.
precision	number of fractional bits in input and output, 1<= precision <= 31. if precision < 1 or precision > 31 function returns INT32_MAX.

Returns

Logarithm base 10 of input, in same Q format as input.

Remarks

If precision = N, then the input and output are Q(31-N).N . For example, if precision = 16 then the input and output are Q15.16. This means that there are 16 fractional bits (the LS word) and the upper 16 bits (MS word) represents an int16_t integer.

Example

See the code example under Fract_log2fix.

C

```
int32_t Fract_log10fix(int32_t x, size_t precision);
```

Fract_log2fix Function

int32_t Fract_log2fix (int32_t X32, size_t precision)

Summary

Calculates fixed point base 2 logarithm using number of fractional bits specified by precision argument.

Description

Calculates fixed point base 2 logarithm using number of fractional bits specified by precision argument. This implementation is based on Clay. S. Turner's fast binary logarithm algorithm[1].

Preconditions

None.

Parameters

Parameters	Description
X32	Fixed point input, if x == 0, function returns INT32_MIN to represent negative infinity. If x < 0 function returns INT32_MAX as error.
precision	number of fractional bits in input and output, 1<= precision <= 31. if precision < 1 or precision > 31 function returns INT32_MAX as error.

Returns

Logarithm base 2 of input, in same Q format as input. The Q format used is based on the value of precision. For example, if precision == 16 then the Q format used is Q15.16.

Remarks

If precision = N, then the input and output are Q(31-N).N . For example, if precision = 16 then the input and output are Q15.16. This means that there are 16 fractional bits (the LS word) and the upper 16 bits (MS word) represents an int16_t integer.

References: [1] C. S. Turner, "A Fast Binary Logarithm Algorithm", IEEE Signal Processing Mag., pp. 124,140, Sep. 2010.

Example

```
// Make log output Q15.16
#define PRECISION 16

#define STRING_MAX_SIZE 133
```

```
    char  ioString[STRING_MAX_SIZE];

    double Xdouble;
    double scale = 1U << PRECISION;
    float  Xfloat;

    int16_t X16;
    int32_t X32;

    while ( Xdouble != -1.0 )
    {
        sprintf(ioString,"rnX: ");
        SendDataBuffer(ioString, strlen(ioString) );
        GetDataBuffer(ioString,STRING_MAX_SIZE);
        sscanf(ioString,"%f",&Xfloat);
        Xdouble = Xfloat;

        if ( Xdouble == -1 )
        {
            sprintf(ioString,"rnWe're done!rn");
            SendDataBuffer(ioString, strlen(ioString) );
            return 0;
        }
        else if ( Xdouble < 0 )
        {
            sprintf(ioString,"%f is Negative!rn",Xdouble);
            SendDataBuffer(ioString, strlen(ioString) );
            continue;
        }
        else if ( Xdouble >= 1 << (32 - PRECISION) )
        {
            sprintf(ioString, "%f is too bigrn", Xdouble);
            SendDataBuffer(ioString, strlen(ioString) );
            continue;
        }

        sprintf(ioString,"   log10(%f) = %frn", Xdouble, (double)log10(Xdouble));
        SendDataBuffer(ioString, strlen(ioString) );

        X32 = Fract_log10fix(Xdouble * scale, PRECISION);
        sprintf(ioString,"Fract_log10fix(%f) = %f (0x%08X)rn", Xdouble, X32/scale, X32);
        SendDataBuffer(ioString, strlen(ioString) );

        X32 = Fract_log2fix(Xdouble * scale, PRECISION) ;
        sprintf(ioString," Fract_log2fix(%f) = %f (0x%08X)rn", Xdouble, X32/scale, X32 );
        SendDataBuffer(ioString, strlen(ioString) );

    }//end while ( Xdouble != -1.0 )
```

C

```
int32_t Fract_log2fix(int32_t X32, size_t precision);
```

Fract_Power Function

uint32_t Fract_Power(uint16_t nBase, uint16_t nExponent)

Summary

Computes pow(nBase,nExponent) using for loop.

Description

Computes pow(nBase,nExponent) using for loop.

Preconditions

None

Returns

pow(nBase,nExponent), this must fit into a 32 bit unsigned integer.

Remarks

none

Example

```
uint16_t nBase, nExponent;
uint32_t answer;

nBase = 5;
nExponent = 3;
answer = Fract_Power(nBase,nExponent);
// Answer = pow(5,3) = 125 = 0x007D
```

C

```
uint32_t Fract_Power(uint16_t nBase, uint16_t nExponent);
```

Fract_sqrtQ15 Function

fract_q15_t Fract_sqrtQ15(fract_q15_t X)

Summary

Fixed point Q15 squareroot.

Description

Returns square root of Q15 (Q0.15) fraction. Since -1 <= X < 1, we know that the 0<= SquareRoot(||X||) < 1.

Preconditions

None.

Parameters

Parameters	Description
X	Q15 input, should be non-negative.

Returns

Square root of abs(X).

Remarks

Given integer X, solve for Y such that

```
  Y     Y       X
---- * ----  =  ----
2^15   2^15     2^15
```

This is equivalent to

Y * Y = X * 2^15 = X<<15

So as integers:

Y = integer sqrt(X<<15)

thus

fixed_point_sqrt(X) = integer_sqrt(X<<15)

For X = 0,1,...32767, define

ErrSqrt = (sqrt(X/32768.0) - FixedPointSQRT(X))/sqrt(X/32768.0)

This error ranges from 0 to 0.015259, with an average error of 0.0054096

The fast integer square root function used is based on
http://stackoverflow.com/questions/1100090/looking-for-an-efficient-integer-square-root-algorithm-for-arm-thumb2

Example

C

```
fract_q15_t Fract_sqrtQ15(fract_q15_t X);
```

Fract_XminusY_Q15bfp Function

fract_q15bfp_t Fract_XminusY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y)

Summary

Subtract two Q15 block floating point numbers.

Description

Subtract two Q15 block floating point numbers.

Preconditions

none.

Parameters

Parameters	Description
x	Q15 block floating point term
y	Q15 block floating point term

Returns

x-y as Q15 block floating point

Remarks

None.

Example

```
fract_q15d16_t x, y;
fract_q15bfp_t X, Y, XminusY;

x = 0x00030000; // 3
y = 0x00070000; // 7
X = Fract_Convert_Q15d16ToQ15bfp(x);
Y = Fract_Convert_Q15d16ToQ15bfp(y);
XminusY = Fract_XminusY_Q15bfp( X, Y );
// XplusY.m = 0xC000 = -16384/32768 = -0.5
// XPlusY.e = 3
// XPlusY = -0.5<<3 = -4
```

C

```
fract_q15bfp_t Fract_XminusY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y);
```

Fract_XoverY_Q15bfp Function

fract_q15bfp_t Fract_XoverY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y)

Summary

Division x/y as Q15 block floating point.

Description

Division x/y as Q15 block floating point.

Preconditions

None.

Parameters

Parameters	Description
x	Numerator as Q15 block floating point
y	Denominator as Q15 block floating point

Returns

Ratio x/y as Q15 block floating point.

Remarks

None.

Example

```
fract_q15_t x,y;
fract_q15bfp_t X, Y, XoverY;

x = 0x1234; //  4660/32768 = 0.142219
y = 0x5678; // 22136/32768 = 0.675537
X = Fract_Convert_Q15d16ToQ15bfp((fract_q15d16_t)x);
Y = Fract_Convert_Q15d16ToQ15bfp((fract_q15d16_t)y);
XoverY = Fract_XoverY_Q15bfp(X,Y);
// x/y = 4660/22136 = 6898/32768 = 0.210516
// XoverY.m = 0x6BC8 = 27592
// XoverY.e = 0xFFFE = -2
// XoverY   = (27592/32768)>>2 = 0.210510
```

C

```
fract_q15bfp_t Fract_XoverY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y);
```

Fract_XoverY_Q15bfpFromQ15 Function

fract_q15bfp_t Fract_XoverY_Q15bfpFromQ15(fract_q15_t x, fract_q15_t y)

Summary

Calculate ratio of X/Y as Q15's but return result as Q15 block floating point.

Description

Calculate ratio of X/Y as Q15's but return result as Q15 block floating point.

Preconditions

None.

Parameters

Parameters	Description
x	Q15 numerator
y	Q15 denominator

Returns

Ratio x/y as Q15 block floating point

Remarks

None.

Example

```
fract_q15_t x,y;
fract_q15bfp_t XoverY;

x = 0x1234; //  4660/32768 = 0.142219
y = 0x5678; // 22136/32768 = 0.675537
XoverY = Fract_XoverY_Q15bfpFromQ15(x,y);
// x/y = 4660/22136 = 6898/32768 = 0.210516
// XoverY.m = 0x6BC8 = 27592
// XoverY.e = 0xFFFE = -2
// XoverY   = (27592/32768)>>2 = 0.210510
```

C

```
fract_q15bfp_t Fract_XoverY_Q15bfpFromQ15(fract_q15_t x, fract_q15_t y);
```

Fract_XoverY_Q15d16FromQ15 Function

fract_q15d16_t Fract_XoverY_Q15d16FromQ15(fract_q15_t x, fract_q15_t y);

Summary

Calculates ratio X/Y for two Q15 fixed points

Description

Calculates ratio X/Y for two Q15 fixed points

Preconditions

None.

Parameters

Parameters	Description
x	Q15 numerator
y	Q15 denominator

Returns

Ratio x/y as Q15.16 fixed point

Remarks

Since ratio x/y can be bigger than unity, result must be Q15.16.

Example

```
fract_q15_t    x,y;
fract_q15d16_t xovery;

x = 0xDEAD; //  -8531/32768 = -0.260345
y = 0xBEEF; // -16657/32768 = -0.508331
xovery = Fract_XoverY_Q15d16FromQ15(x,y);
// x/y   = -8531/-16657 = 33564/65536 = 0.512157
// xovery = 0x0000 831C = 33564
```

C

```
fract_q15d16_t Fract_XoverY_Q15d16FromQ15(fract_q15_t x, fract_q15_t y);
```

Fract_XplusY_Q15bfp Function

fract_q15bfp_t Fract_XplusY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y)

Summary

Add two Q15 block floating point numbers.

Description

Add two Q15 block floating point numbers.

Preconditions

none.

Parameters

Parameters	Description
x	Q15 block floating point term
y	Q15 block floating point term

Returns

x+y as Q15 block floating point

Remarks

None.

Example

```
fract_q15d16_t x, y;
fract_q15bfp_t X, Y, XplusY;

x = 0x00030000; // 3
y = 0x00070000; // 7
X = Fract_Convert_Q15d16ToQ15bfp(x);
Y = Fract_Convert_Q15d16ToQ15bfp(y);
XplusY = Fract_XplusY_Q15bfp( X, Y );
// XplusY.m = 0x5000 = 20480/32768 = 0.625
// XPlusY.e = 4
// XPlusY = 0.625<<4 = 10
```

C

```
fract_q15bfp_t Fract_XplusY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y);
```

Fract_XtimesY_Q15 Function

fract_q15_t Fract_XtimesY_Q15(fract_q15_t x, fract_q15_t y);

Summary

Calculates product of X*Y for Q15 fixed point

Description

Calculates product of X*Y for Q15 fixed point

Preconditions

None

Parameters

Parameters	Description
x	Q15 multiplicand
y	Q15 multiplicand

Returns

Q15 product x*y

Remarks

Since both x and y are Q15 fixed point numbers the product x*y is also Q15.

Example

```
fract_q15_t x, y, xtimesy;

x = 0x1234; //  4660/32768 = 0.142219
y = 0x5678; // 22136/32768 = 0.675537
xtimesy = Fract_XtimesY_Q15(x,y);
// x*y = 0.142219 * 0.675537     = 0.096069
// xtimesy = 0x0C4C = 3148/32768 = 0.096069
```

C

```
fract_q15_t Fract_XtimesY_Q15(fract_q15_t x, fract_q15_t y);
```

Fract_XtimesY_Q15bfp Function

fract_q15bfp_t Fract_XtimesY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y)

Summary

Multiply X times Y, for Q15 block floating point arguments.

Description

Multiply X times Y, for Q15 block floating point arguments.

Preconditions

None.

Parameters

Parameters	Description
x	Q15 block floating point multiplicand
y	Q15 block floating point multiplicand

Returns

Product x*y as Q15 block floating point

Remarks

None.

Example

```
fract_q15bfp_t X,Y,XtimesY;

X.m = 0x1234;
X.e = 0; //  4660/32768 = 0.142219
Y.m = 0x5678;
Y.e = 0; // 22136/32768 = 0.675537
XtimesY = Fract_XtimesY_Q15bfp(X,Y);
// XtimesY.m = 0x6260 = 25184/32768 = 0.768554
// XtimesY.e = 0xFFFD = -3
```

```
// XtimesY   = 0.768554>>3   = 0.096069
// X*Y = 0.142219 * 0.675537 = 0.096069
```

C

```
fract_q15bfp_t Fract_XtimesY_Q15bfp(fract_q15bfp_t x, fract_q15bfp_t y);
```

Fract_XtimesY_Q15d16 Function

fract_q15d16_t Fract_XtimesY_Q15d16(fract_q15d16_t x, fract_q15d16_t y);

Summary

Calculates product of X*Y for Q15.16 fixed point

Description

Calculates product of X*Y for Q15.16 fixed point

Preconditions

None

Parameters

Parameters	Description
x	Q15.16 multiplicand
y	Q15.16 multiplicand

Returns

Q15.16 product x*y

Remarks

Since both x and y are Q15.16 fixed point numbers the product x*y is also Q15.16.

C

```
fract_q15d16_t Fract_XtimesY_Q15d16(fract_q15d16_t x, fract_q15d16_t y);
```

Fract_XtimesY_Q31 Function

fract_q31_t Fract_XtimesY_Q31(fract_q31_t x, fract_q31_t y);

Summary

Calculates product of X*Y for Q31 fixed point

Description

Calculates product of X*Y for Q31 fixed point

Preconditions

None

Parameters

Parameters	Description
x	Q31 multiplicand
y	Q31 multiplicand

Returns

Q31 product x*y

Remarks

Since both x and y are Q31 fixed point numbers the product x*y is also Q31.

C

```
fract_q31_t Fract_XtimesY_Q31(fract_q31_t x, fract_q31_t y);
```

Fx16Norm Function

Normalize the 16-bit fractional value.

Description

Function Fx16Norm:

Produces then number of left shifts needed to Normalize the 16-bit fully fractional input. If the input 'a' is a positive number, it will produce the number of left shifts required to normalized it to the range of a minimum of [(MAXFRACT16+1)/2] to a maximum of [MAXFRACT16]. If the input 'a' is a negative number, it will produce the number of left shifts required to normalized it to the range of a minimum of [MINFRACT16] to a maximum of [MINFRACT16/2]. This function does not actually normalize the input, it just produces the number of left shifts required. To actually normalize the value the left shift function should be used with the value returned From this function.

the 16-bit input on range: 0 => result < 16 (i.e. NUMBITSFRACT16) If a>0: 0x4000 > Normalized Value <= 0x7fff i.e. (MAXFRACT16+1)/2 > aNorm <= MAXFRACT16 If a<0: 0x8000 >= Normalilzed Value < 0xC000 i.e. MINFRACT16>= aNorm < MINFRACT16/2

This function relates to the ETSI norm_s function.

Parameters

Parameters	Description
q15	a in Q1.15

Returns

i16 result - The number of left shifts required to normalize

C

```
int16_t Fx16Norm(q15);
```

Fx32Norm Function

Normalize the 32-bit number.

Description

Function Fx32Norm:

Produces then number of left shifts needed to Normalize the 32-bit fractional input. If the input 'a' is a positive number, it will produce the number of left shifts required to normalized it to the range of a minimum of [(MAXFRACT32+1)/2] to a maximum of [MAXFRACT32]. If the input 'a' is a negative number, it will produce the number of left shifts required to normalized it to the range of a minimum of [MINFRACT32] to a maximum of [MINFRACT32/2]. This function does not actually normalize the input, it just produces the number of left shifts required. To actually normalize the value the left-shift function should be used with the value returned From this function.

32-bit input on range: 0 => result < 32 (i.e. NUMBITSFRACT32) If a>0: 0x40000000 > Normalized Value <= 0x7fffffff i.e. (MAXFRACT32+1)/2 > aNorm <= MAXFRACT32 If a<0: 0x80000000 >= Normalized Value < 0xC0000000 i.e. MINFRACT32>= aNorm < MINFRACT32/2

This function relates to the ETSI norm_l function.

Parameters

Parameters	Description
q31 a	32-bit Q1.d31 to be normalized

Returns

int16_t result - The number of left shifts required to normalize the

C

```
int16_t Fx32Norm(q31);
```

libq_q15_Abs_q15 Function

Saturated Absolute value.

Description

Function libq_q15_Abs_q15:

Creates a saturated Absolute value. It takes the absolute value of the 16-bit 2s-complement fractional input with saturation. The saturation is for handling the case where taking the absolute value of MINFRACT16 is greater than MAXFRACT16, or the allowable range of 16-bit values. This function relates to the ETSI abs function.

Parameters

Parameters	Description
q15 a	input argument

Returns

q15 result - abs(input) <= MAXFRACT16

C

```
q15 libq_q15_Abs_q15(q15);
```

libq_q15_Add_q15_q15 Function

Add two 16-bit 2s-complement fractional values.

Description

Function libq_q15_Add_q15_q15: f

Add two 16-bit 2s-complement fractional (op1 + op2) to produce a 16-bit 2s-complement fractional result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX16 when an overflow occurs and the result is set to MIN16 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occured. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output.

Returns

q15 - a+b on Range: MINFRACT16 <= result <= MAXFRACT16

C

```
q15 libq_q15_Add_q15_q15(q15, q15);
```

libq_q15_DivisionWithSaturation_q15_q15 Function

Fractional division with saturation.

Description

Function libq_q15_DivisionWithSaturation_q15_q15():

Performs fractional division with saturation. There are three restrictions that the calling code must satisfy.

1. Both the numerator and denominator must be positive.
2. In order to obtain a non-saturated result, the numerator must be LESS than or equal to the denominator.

3. The denominator must not equal zero.

If 'num' equals 'den', then the result equals MAXINT16.

This function relates to the ETSI div_s function.

Parameters

Parameters	Description
q15 num	16-bit fractional numerator
q15 den	16-bit fractional denumerator

Returns

q15 result - ratio a/b in 16-bit fractional format

C

```
q15 libq_q15_DivisionWithSaturation_q15_q15(q15, q15);
```

libq_q15_ExpAvg_q15_q15_q1d15 Function

Exponential averaging

Description

Function libq_q15_ExpAvg_q15_q15_q1d15()

Exponential averaging implements a smoothing function based on the form: avg[i+1] = avg[i] * lamda + new * (1-lamda) In this implementation, is has been optimized as follows. avg[i+1] = (avg[i] - new) * lamda + new

The optimization precludes accurate processing of new numbers that differ from the current average by more than unity. If the difference is greater than unity or less than negative unity, the difference is saturated.

The effect is akin to a smaller lambda, e.g., the new value will have a greater weight than expected. If the smoothing is of data that is entirely positive or entirely negative, then the saturation will not be an issue.

Parameters

Parameters	Description
q15 S(k)	Previous exponential average
q15 X(k)	New value to be averaged in
q15 L	exponential averaging constant in Q1.15

Returns

q15 result - S(k+1) = S(k)*L + X(k)*(1-L)

C

```
q15 libq_q15_ExpAvg_q15_q15_q1d15(q15 prevAvgQ15, q15 newMeasQ15, q15 lamdaQ1d15);
```

libq_q15_ExtractH_q31 Function

Extracts upper 16 bits of input 32-bit fractional value.

Description

Function libq_q15_ExtractH_q31:

Extracts upper 16 bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the top 16-bits of the 32-bit input. This function relates to the ETSI extract_h function.

Returns

q15 result - Upper 16 bits of 32-bit argument a

C

```
q15 libq_q15_ExtractH_q31(q31);
```

libq_q15_ExtractL_q31 Function

Extracts lower 16-bits of input 32-bit fractional value.

Descriptionf Extracts lower 16-bits of input 32-bit fractional value and returns them as 16-bit fractional value. This is a bit-for-bit extraction of the bottom 16-bits of the 32-bit input. This function relates to the ETSI extract_l function.

Description

Function libq_q15_ExtractL_q31:

Returns

q15 - Lower 16 bits of 32-bit argument a

C

```
q15 libq_q15_ExtractL_q31(q31);
```

libq_q15_MacR_q31_q15_q15 Function

Multiply accumulate with rounding.

Description

Function libq_q15_MacR_q31_q15_q15:

This function is multiply-accumulate WITH Rounding applied to the accumulator result before it is saturated and the top 16-bits taken. This function first multiplies the two 16-bit input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fractional-type multiply. So, the shifted number now has a '0' in the Lsb. The shifted multiplier output is then added to the 32-bit fractional input 'a'. Then the 32-bits of the accumulator output are rounded by adding '2^15'. This value is then saturated to be within the q15 range. It is assumed that the binary point of the 32-bit input value a is in the same bit position as the shifted multiplier output. This function is for fractional Qtype format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI L_mac_r function.

Parameters

Parameters	Description
q31 a	32-bit accumulator operand
q15 b	16-bit multiplication operand
q15 c	16-bit multiplication operand

Returns

q15 result - a+b*c rounded

C

```
q15 libq_q15_MacR_q31_q15_q15(q31, q15, q15);
```

libq_q15_MsuR_q31_q15_q15 Function

Multiply-Subtraction with rounding

Description

Function libq_q15_MsuR_q31_q15_q15:

This function is like Multiply-Subtract but WITH Rounding applied to the subtractor result before it is saturated and the top 16-bits taken. This function first multiplies the two 16-bit input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fractional-type multiply. So, the shifted number now has a '0' in the Lsb. The shifted multiplier output is then SUBTRACTED From the 32-bit fractional input 'a'. Then the 32-bits output From this subtraction are rounded by adding '2^15'. This value is then saturated to be within the q15 range. It is assumed that the binary point of the 32-bit

input value a is in the same bit position as the shifted multiplier output.

This function is for fractional Q-type format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI msu_r function.

Parameters

Parameters	Description
q31 a	Value which is subtracted from
q15 b	multiplication operand 1
q15 c	multiplication operand 2

Returns

q15 result - a-b*c rounded to Q1.15

C

```
q15 libq_q15_MsuR_q31_q15_q15(q31, q15, q15);
```

libq_q15_MultiplyR2_q15_q15 Function

fractional multiplication of two 16-bit fractional values giving a 16 bit rounded result.

Description

Function libq_q15_MultiplyR2_q15_q15:

Performs fractional multiplication of two 16-bit fractional values and returns a ROUNDED 16-bit fractional result. The function performs a Q15xQ15->Q30 bit multiply with a left shift by '1' to give a Q31 result. This automatic shift left is done to get rid of the extra sign bit that occurs in the interpretation of the fractional multiply result. Saturation is applied to any 32-bit result that overflows. Rounding is applied to the 32-bit SHIFTED result by adding in a weight factor of 2^15, again any overflows are saturated. The TOP 16-bits are extracted and returned. This function is for fractional 'Qtype' data only and it therefore will not give the correct results for true integers (because left shift by '1'). This function assumes that the binary point in the 32-bit shifted multiplier output is between bit_16 and bit_15 when the rounding factor is added. For the special case where both inputs equal the MINFACT16, the function returns a value equal to MAXFACT16, i.e. 0x7fff = 'libq_q15_Mult2_q15_q15(0x8000,0x8000)'. This function internally calls the libq_q15_mult_q15_q31() routine to perform the actual multiplication and the rounding routine to perform the actual rounding.

This function relates to the ETSI mult_r function.

Parameters

Parameters	Description
q15 a	value in Q1.15
q15 b	value in Q1.15

Returns

q15 result - a*b rounded 16-bit signed integer (Q1.15) output value

C

```
q15 libq_q15_MultiplyR2_q15_q15(q15, q15);
```

libq_q15_Negate_q15 Function

Negate 16-bit 2s-complement fractional value with saturation.

Description

Function libq_q15_Negate_q15:

Negate 16-bit 2s-complement fractional value with saturation. The saturation is for handling the case where negating a MINFRACT16 is greater than MAXFRACT16, or the allowable range of values. This function relates to the ETSI negate function.

Returns

q15 result on range: MINFRACT16 <= result <= MAXFRACT16

C

```
q15 libq_q15_Negate_q15(q15);
```

libq_q15_RoundL_q31 Function

Rounds the lower 16-bits of the 32-bit fractional input.

Description

Function libq_q15_RoundL_q31:

Rounds the lower 16-bits of the 32-bit fractional input into a 16-bit fractional value with saturation. This converts the 32-bit fractional value to 16-bit fractional value with rounding. This function calls the 'Add' function to perform the 32-bit rounding of the input value and 'ExtractH' function to extract to top 16-bits. This has the effect of rounding positive fractional values up and more positive, and has the effect of rounding negative fractional values up and more positive. This function relates to the ETSI round function.

Returns

q15 result

C

```
q15 libq_q15_RoundL_q31(q31);
```

libq_q15_ShiftLeft_q15_i16 Function

'Arithmetic' Shift of the 16-bit input argument.

Description

Function libq_q15_ShiftLeft_q15_i16:

Performs an 'Arithmetic' Shift of the 16-bit input argument 'a' left by the input argument 'b' bit positions. If 'b' is a positive number, a 16-bit left shift is performed with 'zeros' inserted to the right of the shifted bits. If 'b' is a negative number, a right shift by abs(b) positions with 'sign extention' Saturation is applied if shifting causes an overflow or an underflow.

positive value: # of bits to left shift (zeros inserted at LSB's) {To not always saturate: if 'a=0', then max b=15, else max b=14}
negative value: # of bits to right shift (sign extend)

This function relates to the ETSI shl function.

Parameters

Parameters	Description
q15 a	16-bit signed integer value to be shifted.
i16 b	16-bit signed integer shift value

Returns

q15 result - arithmetically shifted 16-bit signed integer output

C

```
q15 libq_q15_ShiftLeft_q15_i16(q15, i16);
```

libq_q15_ShiftRight_q15_i16 Function

'Arithmetic' RIGHT Shift on a 16-bit value.

Description

Function libq_q15_ShiftRight_q15_q15:

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input by 'b' bit positions. For positive shift directions (b>0), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions (b<0), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform any 16-bit left shifts. This function does not provide any status-type information to indicate when overflows occur.

positive value: # of bits to right shift (sign extend) { To get all sign bits, b>=15 } negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI shr function.

Parameters

Parameters	Description
q15 a	16-bit signed input value to shift
i16 b	16-bit signed integer shift index

Returns

q15 result - 16-bit signed shifted output

C

```
q15 libq_q15_ShiftRight_q15_i16(q15, i16);
```

libq_q15_ShiftRightR_q15_i16 Function

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input.

Description

Function libq_q15_ShiftRightR_q15_q15:

Performs an 'Arithmetic' RIGHT Shift on a 16-bit input by 'b' bits with Rounding applied. The rounding occurs by adding a bit weight of "1/2 Lsb", where the "Lsb" is the Ending (shifted) Lsb. For example: The initial Bit#(b) is after the right shift Bit#(0), so the rounding bit weight is Bit#(b-1). Rounding does not occur on either left shifts or on no shift needed cases. For positive shift directions (b>0), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions (b<0), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform the actual 16-bit left shift. This function does not provide any status-type information to indicate when overflows occur.

positive value: # of bits to right shift (sign extend) {b > 15, results in all sign bits} negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI shr_r function.

Parameters

Parameters	Description
q15 a	16-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

Returns

q15 result - Arithmetically shifted 16-bit signed integer output

C

```
q15 libq_q15_ShiftRightR_q15_i16(q15, i16);
```

libq_q15_Sub_q15_q15 Function

Subtract two 16-bit 2s-complement fractional values

Description

Function libq_q15_Sub_q15_q15:

Subtract two 16-bit 2s-complement fractional (op1 - op2) to produce a 16-bit 2s-complement fractional difference result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX16 when an overflow occurs and the result is set to MIN16 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occured. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output. This function relates to the ETSI sub function.

Returns

q15 result a+b on range: MINFRACT16 <= result <= MAXFRACT16

C

```
q15 libq_q15_Sub_q15_q15(q15, q15);
```

libq_q1d15_Sin_q10d6 Function

Approximates the sine of an angle.

Description

Function libq_q1d15_Sin_q10d6:

This function approximates the sine of an angle using the following algorithm: sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5. The approximation is accurate for any value of x from 0 degrees to 90 degrees. Because sin(-x) = - sin(x) and sin(x) = sin(180 - x), the sine of any angle can be inferred from an angle in the first quadrant. Therefore, any angle > 90 is converted to an angle between 0 & 90. The coefficients of the algorithm have been scaled by 1/8 to fit a Q1d15 format. So the result is scaled up by 8 to obtain the proper magnitudes. The algorithm expects the angle to be in degrees and represented in Q10.6 format. The computed sine value is returned in Q1.15 format.

Preconditions

none.

Parameters

Parameters	Description
q15 angle	The angle in degrees for which the sine is computed in Q10.6

Returns

q15 sine(angle) value in Q1.15

C

```
q15 libq_q1d15_Sin_q10d6(q15 angleQ10d6);
```

libq_q20d12_Sin_q20d12 Function

3rd order Polynomial apprx. of a sine function

Description

Function libq_q20d12_Sin_q20d12:

3rd order Polynomial apprx. of a sine function

Preconditions

none.

Parameters

Parameters	Description
q31 angle	The angle in radians for which the sine is computed in Q20.12

Returns

q31 Sine(angle) value in Q20.12

C

```
q31 libq_q20d12_Sin_q20d12(q31 angQ20d12);
```

libq_q31_Abs_q31 Function

Saturated Absolute value.

Description

Function libq_q31_Abs_q31:

Creates a saturated Absolute value. It takes the absolute value of the 32-bit 2s-complement fractional input with saturation. The saturation is for handling the case where taking the absolute value of MINFRACT32 is greater than MAXFRACT32, or the allowable range of 32-bit values.

This function relates to the ETSI L_abs function.

Returns

q31 result - abs(a) <= MAXFRACT32

C

```
q31 libq_q31_Abs_q31(q31);
```

libq_q31_Add_q31_q31 Function

Add two 32-bit 2s-complement fractional values.

Description

Function libq_q31_Add_q31_q31:

Add two 32-bit 2s-complement fractional (op1 + op2) to produce a 32-bit 2s-complement fractional result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX32 when an overflow occurs and the result is set to MIN32 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occured. It is assumed that the binary point is in exactly the same bit position for both 32-bit inputs and the resulting 32-bit output. This function relates to the ETSI L_add function.

Returns

q31 result a+b on range: MINFRACT32 <= result <= MAXFRACT32

C

```
q31 libq_q31_Add_q31_q31(q31, q31);
```

libq_q31_DepositH_q15 Function

Place 16 bits in the upper half of 32 bit word.

Description

Function libq_q31_DepositH_q15:

Composes a 32-bit fractional value by placing the input 16-bit fractional value in the composite MSB's and zeros the composite 16-bit LSB's This is a bit-for-bit placement of input 16-bits into the upper part of 32-bit result.

This function relates to the ETSI L_deposit_H function.

Returns

q31 result 16-bits of a in upper MSB's and zeros in the lower LSB's

C

```
q31 libq_q31_DepositH_q15(q15);
```

libq_q31_DepositL_q15 Function

Place 16 bits in the lower half of 32 bit word.

Description

Function libq_q31_DepositL_q15:

Composes a 32-bit fractional value by placing the 16-bit Fraction input value into the lower 16-bits of the 32-bit composite value. The 16-bit MSB's of the composite output are sign extended. This is a bit-for-bit placement of input 16-bits into the bottom portion of the composite 32-bit result with sign extention. This function relates to the ETSI L_deposit_l function.

Returns

q31 result - SignExtended 16-bit MSB's and a Value in lower 16-bit LSB's

C

```
q31 libq_q31_DepositL_q15(q15);
```

libq_q31_Mac_q31_q15_q15 Function

Multiply-Accumulate function WITH saturation

Description

Function libq_q31_Mac_q31_q15_q15():

Performs a Multiply-Accumulate function WITH saturation. This routine returns the fully fractional 32-bit result From the accumulator output 'SAT(addOut_Q1d31)=outQ1d15' where 'multOut_Q1d31 + a_Q1d31 = addOut_Q1d31', and 'b_Q1d15 x c_Q1d15 = multOut_Q1d31'. The multiply is performed on the two 16-bit fully fully fractional input values 'b x c' which results in a 32-bit value. This result is left shifted by one to account for the extra sign bit inherent in the fully fully fractional-type multiply. The shifted number represents a Q1d31 number with the lsb set to '0'. This Q1d31 number is added with the 32-bit fully fully fractional input argument 'a'. Saturation is applied on the output of the accumulator to keep the value within the 32-bit fully fractional range and then this value is returned. This function is for fully fractional Q-type format data only and it therefore will not give the correct results for true integers.

This function relates to the ETSI L_mac function.

Parameters

Parameters	Description
q31 a	32-bit accumulator operand 1 in Q1d31
q15 b	16-bit multiplication operand 1 in Q1d15
q15 c	16-bit multiplication operand 2 in Q1d15

Returns

q31 result, a+b*c saturated

C

```
q31 libq_q31_Mac_q31_q15_q15(q31, q15, q15);
```

libq_q31_Msu_q31_q15_q15 Function

L_msu(a,b,c)

C

```
q31 libq_q31_Msu_q31_q15_q15(q31, q15, q15);
```

libq_q31_Mult2_q15_q15 Function

fractional multiplication of two 16-bit fractional values.

Description

Function libq_q31_Mult2_q15_q15:

Performs fractional multiplication of two 16-bit fractional values and returns the 32-bit fractional scaled result. The function performs the Q15xQ15->Q30 fractional bit multiply. It then shifts the result left by '1', to give a Q31 type result, (the lsb is zero-filled). This automatic shift left is done to get rid of the extra sign bit that occurs in the interpretation of the fractional multiply result. Saturation is applied to any results that overflow, and then the function returns the 32-bit fractional q31 result. This function is for fractional 'Q' data only and it therefore will not give correct results for true integers (because left shift by '1'). For the special case where both inputs equal the MINFRACT16, the function returns a value equal to MAXFACT32, i.e. 0x7fffffff = libq_q15_mult_q15_q31(0x8000,0x8000).

This function relates to the ETSI L_mult function.

Parameters

Parameters	Description
q15 a	multiplicand a
q15 b	multiplicand b

Returns

q31, a*b

C

```
q31 libq_q31_Mult2_q15_q15(q15, q15);
```

libq_q31_Multi_q15_q31 Function

Implement 16 bit by 32 bit multiply.

Description

Function libq_q31_Multi_q15_q31():

Implement 16 bit by 32 bit multiply as shown below The 's' and 'u' notation shows the processing of signed and unsigned numbers.

-B1- -B0- s u 2nd argument is 32 bits -A0- s 1st argument is 16 bits

----

A0B0 A0B0 s=s*u 1st 32-bit product is A0*B0 A0B1 A0B1 s=s*s 2nd 32-bit product is A0*B1

-S2- -S1- -S0- s=s+s 48-bit result is the sum of products -P1- -P0- 32-bit return is the most significant bits of sum

The algorithm is implemented entirely with the fractional arithmetic library. The unsigned by signed multiply is implemented by shifting bits 15:1 to bits 14:0 of a 16-bit positive fractional number, which throws away bit 0 of the 32-bit number. Since that affects result bits that are used for rounding, rounding processing is included. Saturation processing is handled implicitly in the fractional arithmetic library, except for the case of maximum negative numbers.

Returns

q31 result - a*b rounded

C

```
q31 libq_q31_Multi_q15_q31(q15 argAQ1d15, q31 argBQ1d31);
```

libq_q31_Negate_q31 Function

Negate 32-bit 2s-complement fractional value with saturation.

Description

Function libq_q31_Negate_q31:

Negate 32-bit 2s-complement fractional value with saturation. The saturation is for handling the case where negating a MINFRACT32 is greater than MAXFRACT32, or the allowable range of values. This function relates to the ETSI L_negate function.

Returns

q31 result on range: MINFRACT32 <= result <= MAXFRACT32

C

```
q31 libq_q31_Negate_q31(q31);
```

libq_q31_ShiftLeft_q31_i16 Function

'Arithmetic' Shift of the 32-bit value.

Description

Function libq_q31_ShiftLeft_q31_i16:

Performs an 'Arithmetic' Shift of the 32-bit input argument 'a' left by the input argument 'b' bit positions. If 'b' is a positive number, a 32-bit left shift is performed with 'zeros' inserted to the right of the shifted bits. If 'b' is a negative number, a 32-bit right shift by b bit positions with 'sign extention':

positive value: # of bits to left shift (zeros inserted at LSB's) negative value: # of bits to right shift (sign extend)

Saturation is applied if shifting causes an overflow or an underflow.

This function relates to the ETSI L_shl function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16	16-bit signed integer shift index

Returns

q31 result - arithmetically shifted 32-bit signed integer output

C

```
q31 libq_q31_ShiftLeft_q31_i16(q31, i16);
```

libq_q31_ShiftRight_q31_i16 Function

'Arithmetic' RIGHT Shift on a 32-bit value.

Description

Function libq_q31_ShiftRight_q31_q15:

Performs an 'Arithmetic' RIGHT Shift on a 32-bit input by 'b' bit positions. For positive shift directions (b>0), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions (b<0), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform any 32-bit left shifts. This function does not provide any status-type information to indicate when overflows occur.

positive value: # of bits to right shift (sign extend) negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI L_shr function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

Returns

q31 result - Arithmetically shifted 32-bit signed integer output

C

```
q31 libq_q31_ShiftRight_q31_i16(q31, i16);
```

libq_q31_ShiftRightR_q31_i16 Function

'Arithmetic' RIGHT Shift on a 32-bit value

Description

Function libq_q31_ShiftRightR_q31_q15:

Performs an 'Arithmetic' RIGHT Shift on a 32-bit input by 'b' bits with Rounding applied. The rounding occurs before any shift by adding a bit weight of "1/2 Lsb", where the "Lsb" is the Ending (shifted) Lsb. For example: The initial Bit#(i+b) is after the right shift Bit#(i), so the rounding bit weight is Bit#(i+b-1). Rounding does not occur on left shifts, when b is negative. After rounding, this function calls the right-shift function to perform the actual 32-bit right shift. For positive shift directions (b>0), 'b' Lsb-bits are shifted out to the right and 'b' sign-extended Msb-bits fill in From the left. For negative shift directions (b<0), 'b' Lsb's are shifted to the LEFT with 0's filling in the empty lsb position. The left shifting causes 'b' Msb-bits to fall off to the left, saturation is applied to any shift left value that overflows. This function calls the left-shift function to perform the actual 32-bit left shift. This function does not provide any status-type flag to indicate occurence of overflow.

positive value: # of bits to right shift (sign extend) negative value: # of bits to left shift (zeros inserted at LSB's)

This function relates to the ETSI L_shr_r function.

Parameters

Parameters	Description
q31 a	32-bit signed integer value to be shifted
i16 b	16-bit signed integer shift index

Returns

q31 result - Arithmetically shifted 32-bit signed integer output

C

```
q31 libq_q31_ShiftRightR_q31_i16(q31, i16);
```

libq_q31_Sub_q31_q31 Function

Subtract two 32-bit 2s-complement fractional values

Description

Function libq_q31_Sub_q31_q31:

Subtract two 32-bit 2s-complement fractional (op1 - op2) to produce a 16-bit 2s-complement fractional difference result with saturation. The saturation is for handling the overflow/underflow cases, where the result is set to MAX32 when an overflow occurs and the result is set to MIN32 when an underflow occurs. This function does not produce any status flag to indicate when an overflow or underflow has occured. It is assumed that the binary point is in exactly the same bit position for both 16-bit inputs and the resulting 16-bit output. This function relates to the ETSI sub function.

Returns

q31 result a-b on range: MINFRACT31 <= result <= MAXFRACT31

C

```
q31 libq_q31_Sub_q31_q31(q31, q31);
```

LUTwInterpolation_Q15 Function

fract_q15_t LUTwInterpolation_Q15(fract_q15_t x,
const fract_q15_t * pX_LUT_Vector,
const fract_q15_t * pY_LUT_Vector,
const uint16_t nLUTSize);

Summary

Calculates calculates y = f(x) based on a lookup table with linear interpolation between table entries.

Description

Calculates calculates y = f(x) based on a lookup table with linear interpolation between table entries.

Preconditions

None.

Parameters

Parameters	Description
x	Q15 value, independent variable
pX_LUT_Vector	X column of LUT in Q15
pY_LUT_Vector	Y column of LUT in Q15
nLutSize	number of rows (entries) in the LUT

Returns

y = f(x) based on the LUT using linear interpolation between table entries.

Remarks

It is assumed that X LUT vector is in strictly monotonically increasing order. That is: pX_LUT_Vector[i] < pX_LUT_Vector[i+1] for i = 0,1,...nLUTSize-2

Internal math is done in signed 32 integers and truncated to Q15.

If x <= pX_LUT_Vector[0] then pY_LUT_Vector[0] is returned.

if pX_LUT_Vector[nLUTSize-1] <= x then pY_LUT_Vector[nLUTSize-1] is returned.

WARNING: It is strongly recommended that you test out the LUT and compare actual vs. desired values in Matlab/Octave or Excel! You may need to add additional rows where the interpolation does not produce the needed accuracy.

C

```
fract_q15_t LUTwInterpolation_Q15(fract_q15_t x, const fract_q15_t * pX_LUT_Vector, const
fract_q15_t * pY_LUT_Vector, const uint16_t nLUTSize);
```

SinInDegs Function

fract_q0d15_t SinInDegs(fract_q9d6_t theta);

Summary

Calculates fixed point Sin(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)

Description

Calculates fixed point Sin(theta) (Q0.15 fixed point), where theta is in degrees (Q8.6 fixed point)

Preconditions

None

Parameters

Parameters	Description
theta	Q9.6 fixed point angle

Returns

sin(theta) in Q0.15 fixed point

Remarks

None.

Example

```
int32_t nTests = 400;
float theta;
float delta_theta = 360.0/(nTests-1);
float theta0 = 0;

int32_t i;
int32_t x;

for ( i = 0; i < nTests; i++ )
{
    theta = theta0 + i*delta_theta;
    x = theta*(32768/512); //Q9.6
    printf("%d, %d rn",(int)x,SinInDegs(x)));
}
```

C

```
fract_q0d15_t SinInDegs(fract_q9d6_t theta);
```

Index

_

_LIBQ_Q15_cos_Q2_13 function 460
_LIBQ_Q15_sin_Q2_13 function 460
_LIBQ_Q15FromFloat function 461
_LIBQ_Q15FromString function 462
_LIBQ_Q15Rand function 462
_LIBQ_Q16_tan_Q2_29 function 463
_LIBQ_Q16Div function 464
_LIBQ_Q16Exp function 464
_LIBQ_Q16Power function 465
_LIBQ_Q16Sqrt function 466
_LIBQ_Q2_13_acos_Q15 function 466
_LIBQ_Q2_13_asin_Q15 function 467
_LIBQ_Q2_13_atan_Q7_8 function 468
_LIBQ_Q2_13_atan2_Q7_8 function 468
_LIBQ_Q2_29_acos_Q31 function 469
_LIBQ_Q2_29_acos_Q31_Fast function 470
_LIBQ_Q2_29_asin_Q31 function 470
_LIBQ_Q2_29_asin_Q31_Fast function 471
_LIBQ_Q2_29_atan_Q16 function 472
_LIBQ_Q2_29_atan2_Q16 function 472
_LIBQ_Q3_12_log10_Q16 function 473
_LIBQ_Q31_cos_Q2_29 function 473
_LIBQ_Q31_sin_Q2_29 function 474
_LIBQ_Q31FromFloat function 475
_LIBQ_Q31Rand function 475
_LIBQ_Q4_11_ln_Q16 function 476
_LIBQ_Q5_10_log2_Q16 function 477
_LIBQ_Q7_8_tan_Q2_13 function 477
_LIBQ_ToFloatQ15 function 478
_LIBQ_ToFloatQ31 function 478
_LIBQ_ToStringQ15 function 479

A

Abstraction Model 149, 230, 269, 304
Abstration Model 187
ADPCM Decoder Library 350
ADPCM Ecncoder Library 362
ADPCM_Decoder function 351
ADPCM_Decoder_ConfigByteOrder function 351
adpcm_enc.h 364
adpcm_encode_frame function 363
adpcm_encoder_config_byteorder function 363
adpcm_encoder_free function 363
adpcm_encoder_init function 363
ADPCM_GetChannels function 351
ADPCM_HdrGetBitsPerSample function 352
ADPCM_HdrGetBlockAlign function 352
ADPCM_HdrGetBytesPerSec function 352
ADPCM_HdrGetDataLen function 352
ADPCM_HdrGetFileSize function 352
ADPCM_HdrGetFormat function 352
ADPCM_HdrGetNumOfChan function 353
ADPCM_HdrGetSamplesPerSec function 353
ADPCM_HEADER_SIZE macro 353
ADPCM_Initialize function 351
ADPCMHEADER type 353
AK4953 CODEC Driver Library Help 149
AK4954 CODEC Driver Library Help 186
Audio Demonstrations 8
Audio Libraries Help 350
Audio Overview 2
audio_enc 8
audio_player_basic 15
audio_signal_generator 25
audio_tone 32
audio_tone_linkeddma 54

B

Building the Application 13, 22, 28, 47, 75, 90, 102, 116, 133, 145
Building the Library 194, 237, 276, 311
    I2S Driver Library 311
Bulding the Application 59
Bulding the Library 154, 354

C

c) Data Types and Constants 340
Client Access 151, 188, 232, 270, 306
Client Operations 152, 189, 233, 271
Client Operations - Buffered 306
CMSIS-DSP 367
Codec Libraries Help 149
Configuring MHC 153, 193, 275, 310
Configuring the Hardware 13, 23, 29, 49, 60, 76, 91, 102, 117, 133, 145
Configuring the Library 152, 190, 233, 272, 309, 333, 337, 341, 345
Configuring the Library using MHC 368, 370, 457, 480
Confuguring MHC 236
CosInDegs function 483

D

Data Types and Constants 364, 366
DATA_LENGTH type 298
Decoder Library Help 350
Demonstrations 8
    Audio Demonstrations (audio_microphone_loopback) 61
Digital Audio Basics 2
Digital Audio in Harmony 5
Driver Libraries Help 149
DRV_AK4953_AUDIO_DATA_FORMAT type 182
DRV_AK4953_BUFFER_EVENT type 182
DRV_AK4953_BUFFER_EVENT_HANDLER type 182
DRV_AK4953_BUFFER_HANDLE type 183
DRV_AK4953_BufferAddRead function 165
DRV_AK4953_BufferAddWrite function 166
DRV_AK4953_BufferAddWriteRead function 168
DRV_AK4953_BufferEventHandlerSet function 164
DRV_AK4953_CHANNEL type 184
DRV_AK4953_Close function 161
DRV_AK4953_COMMAND_EVENT_HANDLER type 184
DRV_AK4953_CommandEventHandlerSet function 161
DRV_AK4953_Deinitialize function 156
DRV_AK4953_DIGITAL_BLOCK_CONTROL type 185
DRV_AK4953_EnableInitialization function 157
DRV_AK4953_GetI2SDriver function 179
DRV_AK4953_INIT type 185

DRV_AK4953_Initialize function 157
DRV_AK4953_INT_EXT_MIC type 185
DRV_AK4953_IntExtMicSet function 171
DRV_AK4953_IsInitializationDelayed function 159
DRV_AK4953_LRCLK_Sync function 180
DRV_AK4953_MIC type 186
DRV_AK4953_MicGainGet function 172
DRV_AK4953_MicGainSet function 172
DRV_AK4953_MicMuteOff function 173
DRV_AK4953_MicMuteOn function 174
DRV_AK4953_MicSet function 174
DRV_AK4953_MONO_STEREO_MIC type 186
DRV_AK4953_MonoStereoMicSet function 175
DRV_AK4953_MuteOff function 175
DRV_AK4953_MuteOn function 176
DRV_AK4953_Open function 162
DRV_AK4953_ReadQueuePurge function 169
DRV_AK4953_SamplingRateGet function 177
DRV_AK4953_SamplingRateSet function 177
DRV_AK4953_Status function 159
DRV_AK4953_Tasks function 160
DRV_AK4953_VersionGet function 181
DRV_AK4953_VersionStrGet function 181
DRV_AK4953_VolumeGet function 178
DRV_AK4953_VolumeSet function 179
DRV_AK4953_WriteQueuePurge function 170
drv_ak4954.h 229
DRV_AK4954_AUDIO_DATA_FORMAT type 222
DRV_AK4954_AUDIO_DATA_FORMAT_I2S macro 227
DRV_AK4954_AUDIO_DATA_FORMAT_MACRO macro 190
DRV_AK4954_AUDIO_SAMPLING_RATE macro 191
DRV_AK4954_BUFFER_EVENT type 223
DRV_AK4954_BUFFER_EVENT_HANDLER type 223
DRV_AK4954_BUFFER_HANDLE type 224
DRV_AK4954_BUFFER_HANDLE_INVALID macro 227
DRV_AK4954_BufferAddRead function 206
DRV_AK4954_BufferAddWrite function 207
DRV_AK4954_BufferAddWriteRead function 208
DRV_AK4954_BufferEventHandlerSet function 203
DRV_AK4954_CHANNEL type 224
DRV_AK4954_CLIENTS_NUMBER macro 191
DRV_AK4954_Close function 202
DRV_AK4954_COMMAND_EVENT_HANDLER type 225
DRV_AK4954_CommandEventHandlerSet function 204
drv_ak4954_config_template.h 229
DRV_AK4954_COUNT macro 227
DRV_AK4954_Deinitialize function 199
DRV_AK4954_DIGITAL_BLOCK_CONTROL type 225
DRV_AK4954_ENABLE_MIC_BIAS macro 191
DRV_AK4954_EnableInitialization function 198
DRV_AK4954_GetI2SDriver function 221
DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDXx macro 191
DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDXx macro 191
DRV_AK4954_INDEX_0 macro 227
DRV_AK4954_INDEX_1 macro 228
DRV_AK4954_INDEX_2 macro 228
DRV_AK4954_INDEX_3 macro 228
DRV_AK4954_INDEX_4 macro 228
DRV_AK4954_INDEX_5 macro 228
DRV_AK4954_INIT type 226
DRV_AK4954_Initialize function 197
DRV_AK4954_INSTANCES_NUMBER macro 192
DRV_AK4954_INT_EXT_MIC type 226
DRV_AK4954_IntExtMicSet function 217
DRV_AK4954_IsInitializationDelayed function 198
DRV_AK4954_LRCLK_Sync function 220
DRV_AK4954_MASTER_MODE macro 192
DRV_AK4954_MIC type 226
DRV_AK4954_MIC_GAIN macro 192
DRV_AK4954_MicGainGet function 212
DRV_AK4954_MicGainSet function 212
DRV_AK4954_MicMuteOff function 213
DRV_AK4954_MicMuteOn function 214
DRV_AK4954_MicSet function 214
DRV_AK4954_MONO_STEREO_MIC type 226
DRV_AK4954_MonoStereoMicSet function 215
DRV_AK4954_MuteOff function 215
DRV_AK4954_MuteOn function 216
DRV_AK4954_Open function 201
DRV_AK4954_ReadQueuePurge function 210
DRV_AK4954_SamplingRateGet function 217
DRV_AK4954_SamplingRateSet function 218
DRV_AK4954_Status function 200
DRV_AK4954_Tasks function 200
DRV_AK4954_VersionGet function 222
DRV_AK4954_VersionStrGet function 221
DRV_AK4954_VOLUME macro 192
DRV_AK4954_VolumeGet function 218
DRV_AK4954_VolumeSet function 219
DRV_AK4954_WHICH_MIC_INPUT macro 193
DRV_AK4954_WriteQueuePurge function 211
drv_genericcodec.h 267
DRV_GENERICCODEC_AUDIO_DATA_FORMAT type 267
DRV_GENERICCODEC_AUDIO_DATA_FORMAT_MACRO macro 234
DRV_GENERICCODEC_AUDIO_SAMPLING_RATE macro 234
DRV_GENERICCODEC_BUFFER_EVENT type 262
DRV_GENERICCODEC_BUFFER_EVENT_HANDLER type 262
DRV_GENERICCODEC_BUFFER_HANDLE type 263
DRV_GENERICCODEC_BUFFER_HANDLE_INVALID macro 265
DRV_GENERICCODEC_BufferAddRead function 247
DRV_GENERICCODEC_BufferAddWrite function 248
DRV_GENERICCODEC_BufferAddWriteRead function 250
DRV_GENERICCODEC_BufferEventHandlerSet function 244
DRV_GENERICCODEC_CHANNEL type 264
DRV_GENERICCODEC_CLIENTS_NUMBER macro 234
DRV_GENERICCODEC_Close function 244
DRV_GENERICCODEC_COMMAND_EVENT_HANDLER type 264
DRV_GENERICCODEC_CommandEventHandlerSet function 246
drv_genericcodec_config_template.h 267
DRV_GENERICCODEC_COUNT macro 265
DRV_GENERICCODEC_Deinitialize function 240
DRV_GENERICCODEC_GetI2SDriver function 261
DRV_GENERICCODEC_I2C_DRIVER_MODULE_INDEX_IDXx macro 235
DRV_GENERICCODEC_I2S_DRIVER_MODULE_INDEX_IDXx macro 235

DRV_GENERICCODEC_INDEX_0 macro 266
DRV_GENERICCODEC_INDEX_1 macro 266
DRV_GENERICCODEC_INDEX_2 macro 266
DRV_GENERICCODEC_INDEX_3 macro 266
DRV_GENERICCODEC_INDEX_4 macro 266
DRV_GENERICCODEC_INDEX_5 macro 266
DRV_GENERICCODEC_INIT type 265
DRV_GENERICCODEC_Initialize function 239
DRV_GENERICCODEC_INSTANCES_NUMBER macro 235
DRV_GENERICCODEC_LRCLK_Sync function 261
DRV_GENERICCODEC_MicGainGet function 253
DRV_GENERICCODEC_MicGainSet function 254
DRV_GENERICCODEC_MicMuteOff function 254
DRV_GENERICCODEC_MicMuteOn function 255
DRV_GENERICCODEC_MuteOff function 256
DRV_GENERICCODEC_MuteOn function 256
DRV_GENERICCODEC_Open function 242
DRV_GENERICCODEC_ReadQueuePurge function 251
DRV_GENERICCODEC_SamplingRateGet function 257
DRV_GENERICCODEC_SamplingRateSet function 257
DRV_GENERICCODEC_Status function 241
DRV_GENERICCODEC_Tasks function 242
DRV_GENERICCODEC_VersionGet function 258
DRV_GENERICCODEC_VersionStrGet function 259
DRV_GENERICCODEC_VOLUME macro 235
DRV_GENERICCODEC_VolumeGet function 259
DRV_GENERICCODEC_VolumeSet function 260
DRV_GENERICCODEC_WriteQueuePurge function 252
DRV_I2C_INDEX macro 303
drv_i2s.h 331
DRV_I2S_BUFFER_EVENT enumeration 328
DRV_I2S_BUFFER_EVENT_HANDLER type 329
DRV_I2S_BUFFER_HANDLE type 330
DRV_I2S_BUFFER_HANDLE_INVALID macro 330
DRV_I2S_BufferCompletedBytesGet function 324
DRV_I2S_BufferEventHandlerSet function 320
DRV_I2S_BufferStatusGet function 323
DRV_I2S_ClockGenerationSet function 327
DRV_I2S_Close function 316
drv_i2s_config_template.h 331
DRV_I2S_ErrorGet function 325
DRV_I2S_Initialize function 313
DRV_I2S_LRCLK_Sync function 326
DRV_I2S_Open function 315
DRV_I2S_ProgrammableClockSet function 328
DRV_I2S_ReadBufferAdd function 316
DRV_I2S_ReadQueuePurge function 321
DRV_I2S_SERIAL_SETUP type 330
DRV_I2S_SerialSetup function 326
DRV_I2S_Status function 314
DRV_I2S_WriteBufferAdd function 317
DRV_I2S_WriteQueuePurge function 322
DRV_I2S_WriteReadBufferAdd function 319
drv_wm8904.h 303
DRV_WM8904_AUDIO_DATA_FORMAT enumeration 298
DRV_WM8904_AUDIO_DATA_FORMAT_MACRO macro 272
DRV_WM8904_AUDIO_SAMPLING_RATE macro 272
DRV_WM8904_BUFFER_EVENT enumeration 298
DRV_WM8904_BUFFER_EVENT_HANDLER type 299
DRV_WM8904_BUFFER_HANDLE type 300
DRV_WM8904_BUFFER_HANDLE_INVALID macro 302
DRV_WM8904_BufferAddRead function 286
DRV_WM8904_BufferAddWrite function 287
DRV_WM8904_BufferAddWriteRead function 288
DRV_WM8904_BufferEventHandlerSet function 283
DRV_WM8904_CHANNEL enumeration 300
DRV_WM8904_CLIENTS_NUMBER macro 273
DRV_WM8904_Close function 282
DRV_WM8904_COMMAND_EVENT_HANDLER type 301
DRV_WM8904_CommandEventHandlerSet function 284
drv_wm8904_config_template.h 303
DRV_WM8904_COUNT macro 302
DRV_WM8904_Deinitialize function 279
DRV_WM8904_ENABLE_MIC_BIAS macro 273
DRV_WM8904_ENABLE_MIC_INPUT macro 273
DRV_WM8904_GetI2SDriver function 296
DRV_WM8904_I2C_DRIVER_MODULE_INDEX_IDXx macro 273
DRV_WM8904_I2S_DRIVER_MODULE_INDEX_IDXx macro 274
DRV_WM8904_INDEX_0 macro 302
DRV_WM8904_INIT structure 301
DRV_WM8904_Initialize function 278
DRV_WM8904_INSTANCES_NUMBER macro 274
DRV_WM8904_LRCLK_Sync function 297
DRV_WM8904_MASTER_MODE macro 274
DRV_WM8904_MuteOff function 292
DRV_WM8904_MuteOn function 292
DRV_WM8904_Open function 281
DRV_WM8904_ReadQueuePurge function 290
DRV_WM8904_SamplingRateGet function 293
DRV_WM8904_SamplingRateSet function 293
DRV_WM8904_Status function 280
DRV_WM8904_Tasks function 280
DRV_WM8904_VersionGet function 296
DRV_WM8904_VersionStrGet function 297
DRV_WM8904_VOLUME macro 274
DRV_WM8904_VolumeGet function 294
DRV_WM8904_VolumeSet function 295
DRV_WM8904_WriteQueuePurge function 291
DSP 369
DSP_ComplexAdd32 function 374
DSP_ComplexConj16 function 375
DSP_ComplexConj32 function 376
DSP_ComplexDotProd32 function 376
DSP_ComplexMult32 function 377
DSP_ComplexScalarMult32 function 378
DSP_ComplexSub32 function 379
DSP_FilterFIR32 function 379
DSP_FilterFIRDecim32 function 381
DSP_FilterFIRInterp32 function 382
DSP_FilterIIR16 function 383
DSP_FilterIIRBQ16 function 384
DSP_FilterIIRBQ16_cascade8 function 385
DSP_FilterIIRBQ16_cascade8_fast function 386
DSP_FilterIIRBQ16_fast function 388
DSP_FilterIIRBQ16_parallel8 function 389
DSP_FilterIIRBQ16_parallel8_fast function 391

DSP_FilterIIRBQ32 function 392
DSP_FilterIIRSetup16 function 393
DSP_FilterLMS16 function 394
DSP_MatrixAdd32 function 395
DSP_MatrixEqual32 function 396
DSP_MatrixInit32 function 397
DSP_MatrixMul32 function 398
DSP_MatrixScale32 function 399
DSP_MatrixSub32 function 400
DSP_MatrixTranspose32 function 401
DSP_TransformFFT16 function 402
DSP_TransformFFT16_setup function 403
DSP_TransformFFT32 function 403
DSP_TransformFFT32_setup function 404
DSP_TransformIFFT16 function 405
DSP_TransformWindow_Bart16 function 406
DSP_TransformWindow_Bart32 function 407
DSP_TransformWindow_Black16 function 407
DSP_TransformWindow_Black32 function 408
DSP_TransformWindow_Cosine16 function 409
DSP_TransformWindow_Cosine32 function 410
DSP_TransformWindow_Hamm16 function 411
DSP_TransformWindow_Hamm32 function 411
DSP_TransformWindow_Hann16 function 412
DSP_TransformWindow_Hann32 function 413
DSP_TransformWindow_Kaiser16 function 414
DSP_TransformWindow_Kaiser32 function 414
DSP_TransformWinInit_Bart16 function 415
DSP_TransformWinInit_Bart32 function 416
DSP_TransformWinInit_Black16 function 416
DSP_TransformWinInit_Black32 function 417
DSP_TransformWinInit_Cosine16 function 418
DSP_TransformWinInit_Cosine32 function 418
DSP_TransformWinInit_Hamm16 function 419
DSP_TransformWinInit_Hamm32 function 420
DSP_TransformWinInit_Hann16 function 420
DSP_TransformWinInit_Hann32 function 421
DSP_TransformWinInit_Kaiser16 function 422
DSP_TransformWinInit_Kaiser32 function 422
DSP_VectorAbs16 function 423
DSP_VectorAbs32 function 424
DSP_VectorAdd16 function 424
DSP_VectorAdd32 function 425
DSP_VectorAddc16 function 426
DSP_VectorAddc32 function 427
DSP_VectorAutocorr16 function 427
DSP_VectorBexp16 function 428
DSP_VectorBexp32 function 429
DSP_VectorChkEqu32 function 430
DSP_VectorCopy function 430
DSP_VectorCopyReverse32 function 431
DSP_VectorDivC function 432
DSP_VectorDotp16 function 433
DSP_VectorDotp32 function 434
DSP_VectorExp function 435
DSP_VectorFill function 435
DSP_VectorLn function 436
DSP_VectorLog10 function 437
DSP_VectorLog2 function 438
DSP_VectorMax32 function 438
DSP_VectorMaxIndex32 function 439
DSP_VectorMean32 function 440
DSP_VectorMin32 function 441
DSP_VectorMinIndex32 function 441
DSP_VectorMul16 function 442
DSP_VectorMul32 function 443
DSP_VectorMulc16 function 444
DSP_VectorMulc32 function 444
DSP_VectorNegate function 445
DSP_VectorRecip function 446
DSP_VectorRMS16 function 447
DSP_VectorShift function 448
DSP_VectorSqrt function 448
DSP_VectorStdDev16 function 449
DSP_VectorSub16 function 450
DSP_VectorSub32 function 451
DSP_VectorSumSquares16 function 451
DSP_VectorSumSquares32 function 452
DSP_VectorVari16 function 453
DSP_VectorVariance function 454
DSP_VectorZeroPad function 455
dWAVHEADER type 362

E

Encoder Library Help 362
Example Audio Projects 6

F

Files 186, 228, 267, 303, 331, 335, 340, 344, 349, 364, 366
Fract_10toX_Q15bfp function 484
Fract_10toX_Q15bfpFromQ15d16 function 484
Fract_10toX_Q15d16 function 485
Fract_2toMinusX_Q15 function 486
Fract_2toX_Q15bfp function 487
Fract_2toX_Q15bfpFromQ15d16 function 488
Fract_2toX_Q15d16 function 489
Fract_Convert_Q15bfpToQ15d16 function 489
Fract_Convert_Q15bfpToQ31 function 490
Fract_Convert_Q15d16ToQ15bfp function 490
Fract_Convert_Q15ToQ15bfp function 491
Fract_InversePower function 492
Fract_log10fix function 492
Fract_log2fix function 493
Fract_Power function 494
Fract_sqrtQ15 function 495
Fract_XminusY_Q15bfp function 496
Fract_XoverY_Q15bfp function 496
Fract_XoverY_Q15bfpFromQ15 function 497
Fract_XoverY_Q15d16FromQ15 function 498
Fract_XplusY_Q15bfp function 499
Fract_XtimesY_Q15 function 499
Fract_XtimesY_Q15bfp function 500
Fract_XtimesY_Q15d16 function 501
Fract_XtimesY_Q31 function 501
Fx16Norm function 502
Fx32Norm function 502

G

GENERIC AUDIO DRIVER CODEC Library 229

H

How the Library Works 151, 188, 232, 270, 305

I

I2S Driver Library Help 303
I2S Peripheral Library Help 332
I2SC Peripheral Library Help 335
I2SCx_Initialize function 339
I2SCx_LRCLK_Get function 339
I2Sx_Initialize function 343
I2Sx_LRCLK_Get function 343
Introduction 8, 149, 186, 229, 268, 303, 332, 336, 341, 344, 350, 353, 358, 362, 364, 367, 370, 456, 480
isMP3decoder_enabled function 355
isWAVdecoder_enabled function 361

L

LibQ 456
LibQ_c 480
libq_q15_Abs_q15 function 503
libq_q15_Add_q15_q15 function 503
libq_q15_DivisionWithSaturation_q15_q15 function 503
libq_q15_ExpAvg_q15_q15_q1d15 function 504
libq_q15_ExtractH_q31 function 504
libq_q15_ExtractL_q31 function 505
libq_q15_MacR_q31_q15_q15 function 505
libq_q15_MsuR_q31_q15_q15 function 505
libq_q15_MultiplyR2_q15_q15 function 506
libq_q15_Negate_q15 function 506
libq_q15_RoundL_q31 function 507
libq_q15_ShiftLeft_q15_i16 function 507
libq_q15_ShiftRight_q15_i16 function 507
libq_q15_ShiftRightR_q15_i16 function 508
libq_q15_Sub_q15_q15 function 508
libq_q1d15_Sin_q10d6 function 509
libq_q20d12_Sin_q20d12 function 509
libq_q31_Abs_q31 function 510
libq_q31_Add_q31_q31 function 510
libq_q31_DepositH_q15 function 510
libq_q31_DepositL_q15 function 511
libq_q31_Mac_q31_q15_q15 function 511
libq_q31_Msu_q31_q15_q15 function 511
libq_q31_Mult2_q15_q15 function 512
libq_q31_Multi_q15_q31 function 512
libq_q31_Negate_q31 function 513
libq_q31_ShiftLeft_q31_i16 function 513
libq_q31_ShiftRight_q31_i16 function 513
libq_q31_ShiftRightR_q31_i16 function 514
libq_q31_Sub_q31_q31 function 514
Library Interface 155, 195, 238, 276, 312, 335, 339, 343, 347, 350, 355, 358, 363, 364, 369, 372, 459, 481
Library Overview 150, 187, 231, 269, 305, 369, 371, 458, 481
LUTwInterpolation_Q15 function 515

M

Math Libraries Help 367
microphone_loopback 61
MP3 Decoder Library 353
MP3_Decoder function 355
MP3_FillReadBuffer function 356
MP3_FindSyncWord function 356
MP3_GetAudioSize function 356
MP3_GetBitRate function 356
MP3_GetBitsPerSample function 356
MP3_GetChannels function 356
MP3_GetDuration function 356
MP3_GetSampleRate function 357
MP3_HdrGetBytesPerSec function 357
MP3_HdrGetDataLen function 357
MP3_HdrGetFileSize function 357
MP3_HdrGetSamplesPerSec function 357
MP3_Initialize function 357
MP3_Initialize_N function 357
MP3_UpdatePlaytime function 358
mul16 function 455
mul16r function 455
mul32 function 456

P

PCM Encoder Library 364
pcm_enc.h 366
pcm_encode_frame function 365
pcm_encoder_free function 365
pcm_encoder_init function 365
Peripheral Libraries Help 332
plib_i2s_u2224.h 335
plib_i2sc_11241.h 340
plib_spi_01329.h 344
plib_ssc_6078.h 349

R

read_ID3Tags function 358
Running the Application 145
Running the Demonstration 14, 23, 29, 51, 60, 78, 92, 102, 119, 135
    Audio Demonstrations (audio_microphone_loopback) 78

S

SAMPLE_LENGTH type 226
SAT16 function 456
SAT16N function 456
SAT16P function 456
Setup (Initialization) 151, 188, 232, 270
SinInDegs function 515
SPI-I2S Peripheral Library Help 340
SSC Peripheral Library Help 344
SSC_BaudSet function 348
SSC_Initialize function 347
SSC_LRCLK_Get function 348
System Access 305
System Configuration 153, 190, 234, 272, 310

U

universal_audio_decoders 81
usb_microphone 94
usb_speaker 106

usb_speaker_bass_boost 121
usb_speaker_hi_res 137
Using the Library 149, 186, 230, 268, 304, 334, 338, 342, 346, 363, 364, 368, 370, 457, 480

W

WAV Decoder Library 358
WAV_DEC type 362
WAV_Decoder function 359
WAV_GetAudioSize function 360
WAV_GetBitRate function 360
WAV_GetChannels function 359
WAV_GetDuration function 359
WAV_GetSampleRate function 360
WAV_HdrGetBytesPerSec function 360
WAV_HdrGetDataLen function 360
WAV_HdrGetFileSize function 360
WAV_HdrGetSamplesPerSec function 360
WAV_HEADER_SIZE macro 361
WAV_HEAP_SIZE macro 361
WAV_Initialize function 361
WAV_Initialize_N function 359
WAV_INPUT_BUFFER_SIZE macro 362
WAV_OUTPUT_BUFFER_SIZE macro 362
WAV_UpdatePlaytime function 359
WAVE_FORMAT_ALAW macro 361
WAVE_FORMAT_MULAW macro 361
WAVE_FORMAT_PCM macro 361
WM8904 CODEC Driver Library Help 267