Application Report
SPMA042B–June 2012–Revised June 2013

Dual-SPI Emulating I²S on Tiva™ C Series TM4C123x MCUs

Michael Risley
Steve Sager

Stellaris® Microcontrollers
Digital Field Applications

ABSTRACT

This application report presents an example using two serial peripheral interfaces (SPIs) to emulate an integrated interchip sound (I²S) peripheral on the Tiva™ C Series TM4C123GH6PGE microcontroller. Utilizing the sound API, .wav audio files stored in the on-board SD card are played via pseudo-I²S to an external Texas Instruments' TLV320AIC3107 codec. This generic software platform can be customized to add audio functionality to embedded systems.

NOTE: This document applies to both the Tiva C Series and the Stellaris® Cortex-M4 MCUs.

Contents

List of Figures

1 Introduction

Integrated Interchip Sound (I^2S) is a widely used electrical serial bus interface standard for digital audio transmission. Many embedded applications use I^2S for audio playback, and most modern microcontrollers generally provide I^2 support. If this peripheral is unavailable on the microcontroller, or there is a need for an additional I^2S, the peripheral can be imitated using existing resources. This application report examines an implementation of a pseudo-I^2S interface using two serial peripheral interfaces (SPIs) that allow the user to overcome the hardware limitations or lack of dedicated peripherals. The basis of this application report is a demo using a Tiva C Series TM4C123GH6PGE microcontroller playing audio with this pseudo-I^2S interface to a TI TLV320AIC3107EVM-K. The demo is open source and can be downloaded from the link found in Section 12, *References*. This document describes the demo's specific setup, how to create a pseudo-I^2S on a generic Tiva C Series microcontroller, and the configuration of important characteristics of the sound driver.

2 General Overview, Dual-SPI to I^2S

The I^2S interface typically consists of a bus with at least three signals: Bit Clock (BCLK), Data Input (DIN), and Word Clock (WCLK). These three signals are used in the dual-SPI implementation as shown in the connection diagram in Figure 1. With an external I^2S device configured to be the master, the SPI module is controlled by the bit and word clocks of the I^2S interface. By using the word clock signal to enable or disable the slave SPI ports on the microcontroller, a pseudo-I^2S interface is created.

NOTE: The SPI frame select lines are inverted, causing the data transmission to toggle between the two SPI ports.

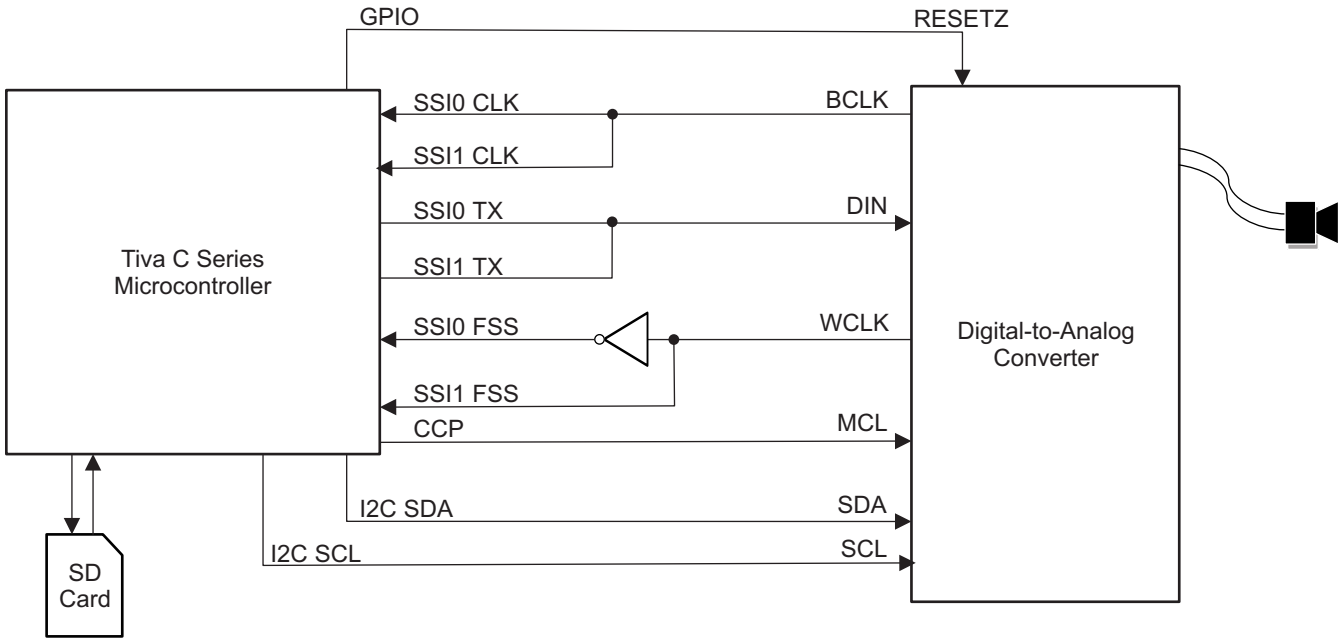

Figure 1. General System Connect

A capture and compare output (CCP) of the Tiva C Series microcontroller is used to provide the codec master clock (MCLK). In this implementation, the codec is provided with a 25-MHz MCLK. The codec internal PLLs then generate the desired BCLK and WCLK signals based on the required audio sampling frequency.

The GPIO output at the top of Figure 1 is used to pull the codec RESETZ line low to reset the codec. This GPIO is toggled before any data is sent to the codec, to trigger a reset, ensuring proper functionality.

The Inter-Integrated Circuit (I^2C) signals, SDA and SCL, are used to write to the internal codec registers in order to configure the data path, clocking, line outputs, and additional provisions.

The SD card shown in Figure 1 is used to store the pulse-code modulation (PCM) .wav audio data. The .wav files could be stored in alternate locations; however, this example reads the data from an SD card to demonstrate a more comprehensive and larger storage medium.

3 Hardware Implementation, TM4C123G to TLV320AIC3107

The dual-SPI audio example was built and verified using a Tiva C Series TM4C123GH6PGE microcontroller connected to an external TLV320AIC3107EVM-K. Figure 2 shows the required external wiring of these boards. This section describes the hardware connections necessary to build this configuration.

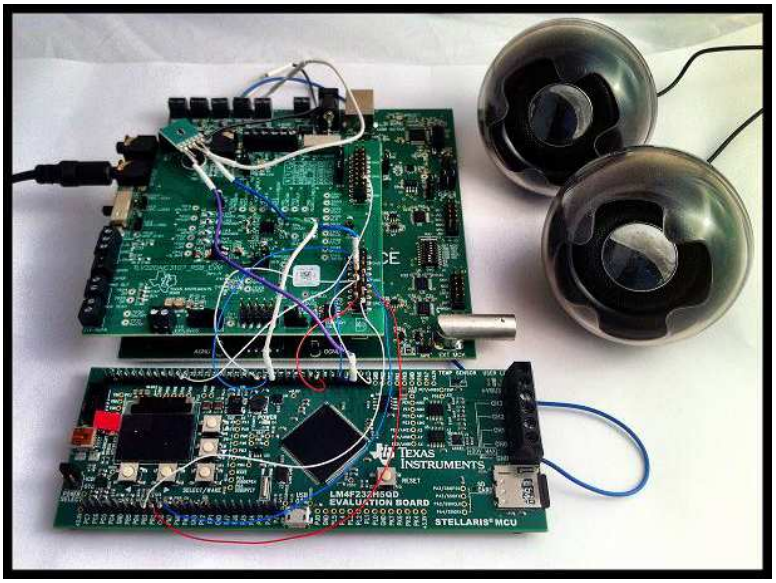

Figure 2. Complete System Setup

Table 1 shows the pin-to-pin wiring required for this configuration. For example, the Tiva TM4C123x transmission line of the third SPI peripheral (SSI3Tx) must be routed to the I^2S data input (DIN) of the codec EVM. The corresponding pins to be connected are PH3 to P11 (of the J5 header).

Table 1. LM4F to TLV320AIC3107 EVM Connections

Tiva C Series TM4C123GH6PGE Pin (Peripheral)	TI TLV320AIC2107 Codec EVM Pin (Peripheral)
PF2 (SSI1Clk)	J5/P3 (BCLK)
PF3 (SSI1Fss)	J5/P7 (WCLK)
PF1 (SSI1Tx)	J5/P11 (DIN)
PH0 (SSI3Clk)	J5/P3 (BCLK)
PH1 (SSI3Fss)	J5/P7 (WCLK)
PH3 (SSI3Tx)	J5/P11 (DIN)
PB2 (I2C0SCL)	J5/P16 (SCL)
PB3 (I2C0SDA)	J5/P20 (SDA)
PD0 (GPIO)	J4/P14 (RESETZ)
PG3 (T5CCP1)	J5/P17 (MCLK)

The codec and USB baseboard can be configured for multiple operations. To ensure proper functionality, the switches and jumpers must be properly set. Table 2 and Table 3 show the required configurations for both the codec EVM and USB baseboard. For switches (SWn), **1** indicates the *on* state, and **0** indicates *off*. For jumpers (Jn), an **N** indicates *not populated* and **P** indicates *populated*.

Table 2. Codec USB-MODEVM Hardware Configuration

Description	Switch or Jumper Name	State
Regulator Enable	SW1 (2 switches)	3.3VD EN and 1.8VD EN
USB Config	SW2 (8 switches)	EXT MCK, USB RST, USB SPI, USB MCK, USB I2S, OFF, OFF, OFF
IOVDD Select	SW3 (8 switches)	3.3V, OFF, OFF, OFF, OFF, OFF, OFF, OFF
External I2C	J6 (2 jumpers)	N, N
External Audio Data	J14 (6 jumpers)	N, N, N, N, N, N
External SPI	J15 (6 jumpers)	N, N, N, N, N, N
SDA Pull-up	JMP3	P
SCL Pull-up	JMP4	P
CNTL or FSX Select	JMP5	FSX
MCLKI	JMP7	2–3
External MCLK	JMP8	N

Table 3. TLV320AIC3107 EVM Hardware Configuration

Description	Switch or Jumper Name	State
Single-Ended or Differential Inputs	SW1	SE
Cap or Cap-Less	SW2	CAP
VDD Select	SW3	+5VA
Mic Bias Select	W9	1–2
On-Board Microphone	W10	P
On-Board Microphone	W11	P
Headset	W12	N
Headset	W13	N
IOVDD Select	W14	1–2
GPIO1 Input	W15	P
RESET Input	W16	P
EEPROM Select	W17	N
SPOM Select	W18	N
SPOP Select	W19	N

NOTE: For proper operation of the I^2C, populate JMP3 and JMP4 of the USB baseboard to provide a 2.7-kΩ pull-up resistance. Route 1-kΩ resistors in parallel from R1 and R2 vias to +V. This configuration ensures a weak pull-up resistance of approximately 720 Ω. Certain slaves might pose enough load that silent or noise levels make it difficult for pull-ups to raise the SCL signal.

4 Software Codec Setup

In this implementation, the Texas Instruments TLV320AIC3107EVM-K is used for the external codec. This evaluation module is a complete stereo audio codec with several inputs and outputs, extensive audio routing, mixing, and effects capabilities. As shown in the connection diagram in Figure 1, there are seven separate lines communicating with the codec: RESETZ, BCLK, WCLK, MCLK, DIN, SDA, and SCL. In addition, fully-differential stereo speakers should be connected to HP_OUTPUT at J10. The codec functional block diagram with registers can be found in the TLV320AIC3107 data sheet.

The software-configured data path is highlighted in red on the codec functional block diagram, Figure 3. The seven lines communicating with the microcontroller can be found on the top and bottom of this diagram. The two outputs to the right, HPLOUT and HPROUT, go to the speakers.

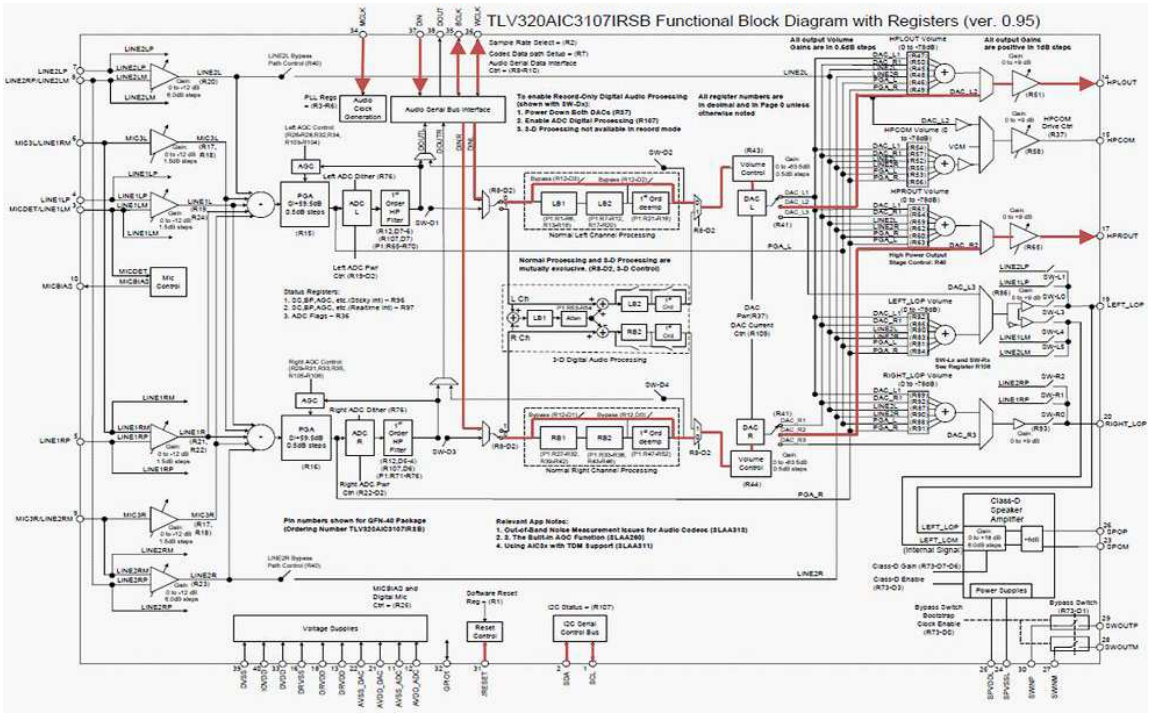

Figure 3. TLV320AIC3107 Data Path

During initialization of the codec, the data path in Figure 3 is configured by writing to various codec control registers. Set up the codec in a specific sequence for best results. The Tiva microcontroller performs these register writes using I^2C.

The DACInit( ) function in dac.c is called during the initialization of the sound driver. This function writes to a series of registers in the codec by calling the DACWriteRegister( ) function each time. In addition to setting up the MUXes for the control path in Figure 3, the following are configured:

- PLL divisors, to define the codec sample rate (8 kHz, 11.025 kHz, 16 kHz …)
- Fsref (44.1 kHz, 48 kHz)
- Headset detection
- Left and right output volume
- Pop reduction
- Volume

If using a codec other than the TLV320AIC3107, the register write calls in the DACInit( ) function must be modified accordingly. The DACInit( ) function is thoroughly commented with the name of the registers, register number, effect of write, and the write value. The following three code samples show these writes:

```
//
// DAC Output Switching Control Register
// ----------------------
// Left DAC output selects DAC_L2 path to left high power output drivers.
// Right DAC output selects DAC_R2 path to right high power output
// drivers.
// ----------------------
// D[7:0] = 1010 0000
//
DACWriteRegister(41, 0xA0);
```

Multiple output paths exist from the left and right codec. Using L2 and R2 bypasses the analog volume controls and mixing networks. This output provides the highest quality codec playback performance with reduced power consumption, but can only be used if the codec output is not routed to multiple output drivers simultaneously, and if mixing of the codec output with other analog signals is not required. If mixing is required by the application, L1 and R1 should be used. However, for differential speakers, the implementation in this application report is sufficient. Select DAC_L2 and DAC_R2 to choose the path shown in red on Figure 3. The sample code above is performing an I^2C transmission to write the value 0xA0 to the 41st register, the *Codec Output Switching Control Register*. As seen from the control registers section of the TLV320AIC3107 data sheet, this write selects the left codec to route to DAC_L2 and the right codec to route to DAC_R2. This configuration also allows sets each channel to have an independent volume control.

The Tiva C Series microcontroller drives the MCLK at 25 MHz. The data converters are based on the concept of an Fsref rate that is used internally to the part, and is related to the actual sampling rates of the conversions through a series of ratios. For typical sampling rates, Fsref is either 44.1 kHz or 48 kHz. To acquire the required audio sampling frequencies, MCLK must be divided to allow for an Fsref of 44.1 kHz or 48 kHz.

```
//
// PLL Programming Register A
// ----------------------
// PLL Disabled (Enable Later)
// Q = 2
// P = 2
// ----------------------
// D[7:0] = 0001 0010
//
DACWriteRegister(3, 0x12);
```

The above code is performing an I^2C transmission to write the value 0x12 to the third register, the PLL Programming Register A. With the PLL enabled, Fsref = (PLLCLK_IN x K x R) / (2048 x P). Additional details on this clock generation can be found in the Audio Clock Generation section of the TLV320AIC3107 data sheet. To acquire the desired Fsref of 44.1 kHz: Q = 2, P = 2, J = 7, and K = 2253. These values are all set in PLL Programming Registers A-D. The code above sets Q=2 and P=2. The PLL is left disabled until all of the PLL programming registers are properly configured.

```
//
// Codec Sample Rate Select Register
// ----------------------
// DAC Fs = Fsref/4.
// ----------------------
// D[7:0] = 0000 0110
//
DACWriteRegister(2, 0x06);
```

The sampling rate of the codec can be set to Fsref/NDAC or 2xFsref/NDAC, with NDAC being 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, or 6. For example, with Fsref = 44.1 kHz, the codec sampling rate can be set to 44.1 kHz by using NDAC=1. In the code above, the codec sampling frequency is being set to 11.025 kHz (44.1 / 4), by writing the value 0x06 to the second register, the *Codec Sample Rate Select Register*.

Because the sampling frequency of the audio to be played is unknown when the codec is initialized, the SoundSetFormat( ) function in the sound API handles the support for multiple sampling frequencies by hard coding the Fsref, K, J, and D values required for a particular sampling frequency. To add support for a sampling frequency, a case must be added to check the value against the ulSampleRate variable. Set the Fsref, along with the PLL K, J, and D values. Enable the codec dual-rate mode (if necessary) and set the NDAC value. The TLV320AIC3107 data sheet can be useful in determining the appropriate values; however, the TLV320AIC3107EVM-K Graphical User Interface Software might be the easiest method. Using the Clocks tab, enter the settings and click, **Search for Ideal PLL Settings**. Table 4 lists the supported sampling frequencies pre-coded into this software example.

Table 4. Supported Audio Formats

Supported Bits per Sample	Supported Channels	Supported Sampling Frequencies (kHz)
16	Mono (1)	8.000
	Stereo (2)	11.025
		16.000
		32.000
		48.000
		64.000

5 Software Model Overview

The dual-SPI software example makes use of multiple drivers. A simplified software stack for the dual_spi.c application is shown in Figure 4.

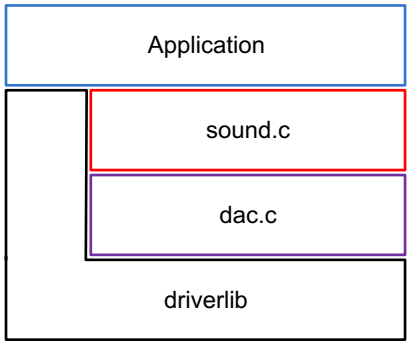

Figure 4. Dual-SPI Application Software Stack

At the user level is the dual_spi.c main application that initializes the sound driver, graphics, SD card I/O, and user button polling. Dual_spi.c depends on many TivaWare™ resources including driverlib, utils, grlib, and third-party fat fs. This application interacts to audio only through sound.c. When the application starts, a list is populated with .wav files found on the SD card. The display then shows the name of the first audio clip. Refer to Figure 5 for an example of this GUI, eight seconds into the 12-second audio file named DEMO16'1.WAV at 50% volume.

Use the GUI to control audio clip play as follows:

- Press the **LEFT** or **RIGHT** push buttons to cycle through the available audio clips. The name of each clip selected displays.
- Select a sound clip by pressing the **SELECT/WAKE** push button.
- During playback, pause or unpause the clip by pressing the **SELECT/WAKE** push button.
- To stop the clip and return to the main menu, press the **LEFT** push button.
- The **UP** and **DOWN** push buttons correspond to increasing and decreasing the audio volume by 10% increments. This volume setting is reflected by a vertical volume bar on the right side of the screen.
- During playback, check both the time stamp and horizontal time bar on the bottom of the screen to see the audio file progression.

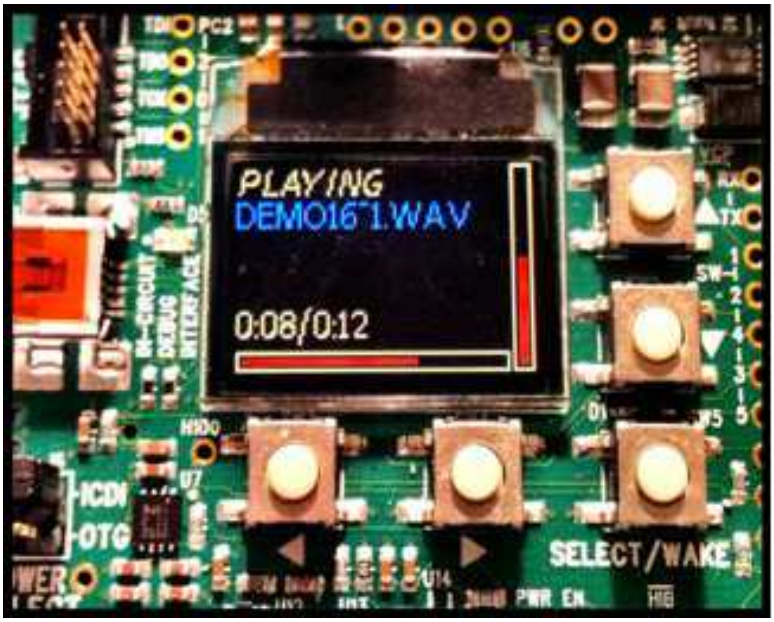

Figure 5. Dual-SPI Software Example GUI

The primary application configures the necessary audio infrastructure and begins playback through the sound API located in sound.c. The SoundInit( ) function must be called before any other API calls. After the sound driver is initialized, use the SoundOpen( ) function to open a .wav file. Finally, to begin playback of the opened audio file, call the SoundPlay( ) function. Use the following additional functions to modify or receive data feedback, which includes:

- Set and get volume
- Get a time string
- Get song length
- Get playback status
- Get sample rate
- Turn volume up or down

The sound API initializes and configures the TLV320AIC3107 through the codec API located in dac.c. The SoundInit( ) function initializes the codec by calling the DACInit( ) function. This must be called before any writes are made to the codec registers.

6 Sound APIs

The API required to initialize and configure the sound driver includes the following three functions.

NOTE: These APIs must be called in the order shown.

```
void SoundInit(void);
tBoolean SoundOpen(const char *pcFileName,
                   tWaveHeader *pWaveHeader);
void SoundPlay(void);
```

The SoundInit( ) function initializes the peripherals of the Tiva microcontroller that are necessary for audio playback, including two SPI modules, a periodic timer interrupt, µDMA, and all codec initializations.

The SoundOpen( ) function opens a .wav file and parses its header information. This function populates a caller-supplied header structure with the file audio characteristics (Format, Number of Channels, Sample Rate, Average Byte Rate, Bits per Sample). Validation of the audio file is confirmed before proceeding.

The SoundPlay( ) function begins playback of the audio file opened by the SoundOpen( )function. The SoundPlay( ) function sets the necessary flags, enables a periodic interrupt (based on sampling frequency), and enables the next uDMA transfer for audio data buffering. The SoundPlay( ) function continues playing the audio file opened by the SoundOpen( ) function.

The remaining functions constitute the optional API in the sound driver:

```
void SoundStop(void);
void SoundPause(void);
void SoundVolumeSet(unsigned long ulPercent);
void SoundVolumeDown(unsigned long ulPercent);
void SoundVolumeUp(unsigned long ulPercent);
unsigned char SoundVolumeGet(void);
unsigned long SoundGetTime(tWaveHeader *pWaveHeader, char *pcTime,
                           unsigned long ulSize);
unsigned long SoundSampleRateGet(void);
unsigned long SoundGetLength(void);
tBoolean SoundPlaybackStatus(void);
```

The SoundStop( ) function changes the state of playback to *Stopped* so that the audio clip stops playing. This function returns the playback position to zero and disables all relevant interrupts.

The SoundPause( ) function changes the state of playback to *Paused* and halts the audio clip until the SoundPlay( ) function is called again, at which point playback is resumed from the paused state.

The SoundVolumeSet( ) function sets the audio volume to a user-specified percentage. The supplied level is expressed as a percentage between 0% (mute) and 100% (full volume), in increments of 10%.

The SoundVolumeDown( ) function decreases the volume by a user-specified percentage. The adjusted volume cannot go below 0%.

The SoundVolumeUp( ) function increases the volume by a user-specified percentage. The adjusted volume will not go above 100%.

The SoundVolumeGet( ) function returns the current volume setting specified as a percentage between 0 and 100%, inclusive.

The SoundGetTime( ) function formats a text string to represent the elapsed and total playing time of the opened audio file.

The SoundSampleRateGet( ) function returns the sampling rate of the opened audio file.

The SoundGetLength( ) function returns the total length of the opened audio file in seconds.

The SoundPlaybackStatus( ) function returns the current playback status of the opened audio file: playing or not playing.

7 Codec APIs

The following five functions comprise the codec initialization and configuration API.

NOTE: The DACInit( ) function must be called first.

```
tBoolean DACInit(void);
void DACVolumeSet(unsigned long ulVolume);
void  DACClassDEn(void);
void  DACClassDDis(void);
tBoolean DACWriteRegister(unsigned char ucRegister,
                          unsigned long ulData);
```

The DACInit( ) function initializes the I^2C interface and writes to the appropriate codec registers to set up the path shown in Figure 3. This function must be called prior to any other API in the codec module. Configured specifically for the TLV320AIC3107, the sequence of register writes can be altered to successfully interface with a different codec.

The DACVolumeSet( ) function sets the volume of the codec based on the given percentage. This function uses a lookup table to translate a 0-100% scale to MUTE-0dB.

The DACClassDEn( ) function enables the class D amplifier in the codec.

NOTE: This example does not utilize the Class D amplifier, which is bypassed as shown in Figure 3.

The DACClassDDis( ) function disables the class D amplifier in the codec.

NOTE: This example does not utilize the Class D amplifier, which is bypassed as shown in Figure 3.

The DACWriteRegister( ) function writes given data to a given codec register .

8 Getting Started With the Software Example

Follow these steps to open the dual-SPI audio demo in CCS and load the example to a Tiva C Series TM4C123GH6PGE microcontroller:

1. Move the compressed dual_spi_audio.zip folder to the LM4F board examples folder in *../TivaWare/boards/ek-lm4f232/*.
2. Right-click the dual_spi_audio.zip folder and select **Extract All**.
3. Click Next > Next > Finish to extract the CCS project containing the C source files reviewed in Section 5, *Software Model Overview*.
4. Open Code Composer Studio™ and select your preferred workspace.
5. Import the dual_spi_audio CCS project.
6. Select Project > Import Existing CCS/CCE Eclipse Project.
7. Click **Browse**, and select the extracted ***dual_spi_audio*** folder in *C:/t/TivaWare_C_Series_1.0/boards/ek-lm4f232/*.
8. Click **Finish**.
9. Build the project.
10. Select Project > Build Active Project. With the exception of one warning, the project builds successfully.

    Note: Both driverlib/ccs-cm4f and grlib/ccs-cm4f must be built prior to building this project.
11. To load the software to the target, make sure that the EVM is connected, with drivers installed. Select Target > Launch TI Debugger.

9 CPU Usage

This section describes the procedure and results of the CPU usage calculations. Results are shown for each sampling frequency and the corresponding CPU utilization percentage.

NOTE: The timer interrupt handler responsible for updating the graphics during playback was omitted during calculations to remove unnecessary overhead.

To calculate the CPU usage, the application is put to sleep repeatedly when idle in the main loop. With all peripherals sleep-enabled, an interrupt causes the microcontroller to wake from sleep for the duration of the interrupt handler and then return to sleep mode. The ratio of time per period (T_p) and summation of all interrupt durations over this period (T_i), yields the percentage of the CPU bandwidth being used, where *N* is the average number of interrupts that take T_i per period as some periodic interrupts might wake to find nothing needs to be done.

$[(N \times T_i) / T_p] \times 100$

To capture the duration of the sleep and interrupt times, GPIOs are toggled and viewed with an oscilloscope. A GPIO dedicated to viewing the sleep times and is toggled high prior to calling the sleep function. The same GPIO is then toggled low at the start of any interrupt handler. A separate GPIO is used for each interrupt and is toggled at entry and exit of the interrupt handler. Below is a simple pseudo-code example of this operation:

```
INTERRUPTHANDLER()
{
     // WRITE GPIO A LOW.
     // WRITE GPIO B HIGH.
     ...
     // WRITE GPIO B LOW.
}

MAIN()
{
     SOUNDINIT();
     SOUNDOPEN(...);
     SOUNDPLAY();
     WHILE(1)
     {
          // WRITE GPIO A HIGH.
          SYSCTLSLEEP();
     }
}
     }
```

The interrupt sequence for a 16-bit, 8-kHz stereo audio sample is characterized by the waveforms shown in Figure 6. Waveform 1 (yellow) displays the frequency and duration of the microcontroller entering sleep mode. When high, the microcontroller is in sleep mode. The low pulses demonstrate the microcontroller waking for the duration of an interrupt routine. Waveform 2 (blue) displays the frequency and duration of the Timer2AIntHandler( ) function used in sound.c to periodically call the SoundPlayContinue( )function. When pulsed high, the interrupt is active. This interrupt is set up to refill the audio buffers if necessary and perform any necessary housekeeping to keep the codec supplied with audio data. This interrupt occurs every 55 mS to accommodate the 8-kHz sampling frequency. As shown in Figure 6, this interrupt occurred eight times within the two vertical cursors (–126 ms to 318 ms, a 444-ms window). When the first interrupt occurred (too small to be viewed here), the buffers did not require servicing, meaning the interrupt required no action and immediately returned to sleep. Waveform 3 (pink) displays the frequency and duration at which the SoundIntHandler( ) function is called by the processor due to an interrupt from the SPI peripheral (too small to be viewed here). Due to the miniscule duration of these interrupts, this time was excluded from calculations.

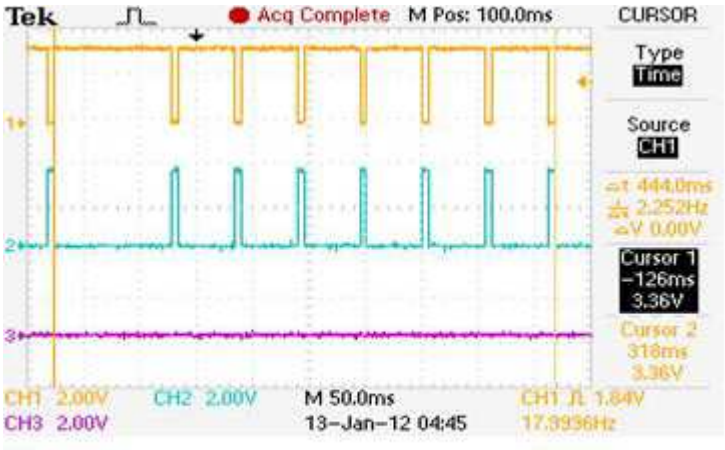

Figure 6. 8-kHz Interrupt Waveforms

The duration of the period is 444 ms as measured by the vertical cursors. Each timer Interrupt was found to have a duration of 5.2 ms. Eight of these interrupts occur per period, with one requiring minuscule time. Thus, the CPU utilization is calculated as follows:

[(7 × 5.2) / 444] × 100 = 8.2%

Using this method for all sampling frequencies, the corresponding CPU utilization was determined. Note that these points were taken while running the microcontroller at 50 MHz, during 16-bit stereo audio playback. The CPU utilization stacks up linearly with the sampling frequency, just as the data retrieval from the SD card increases. A majority of the CPU utilization is reading from the SD card.

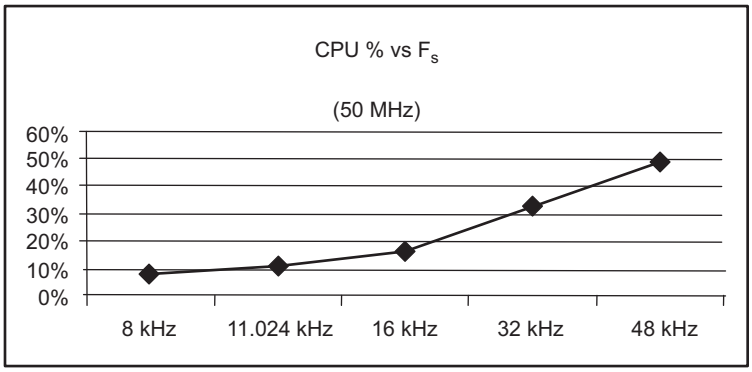

Figure 7. CPU Utilization Graph Based on Audio Sampling Frequency

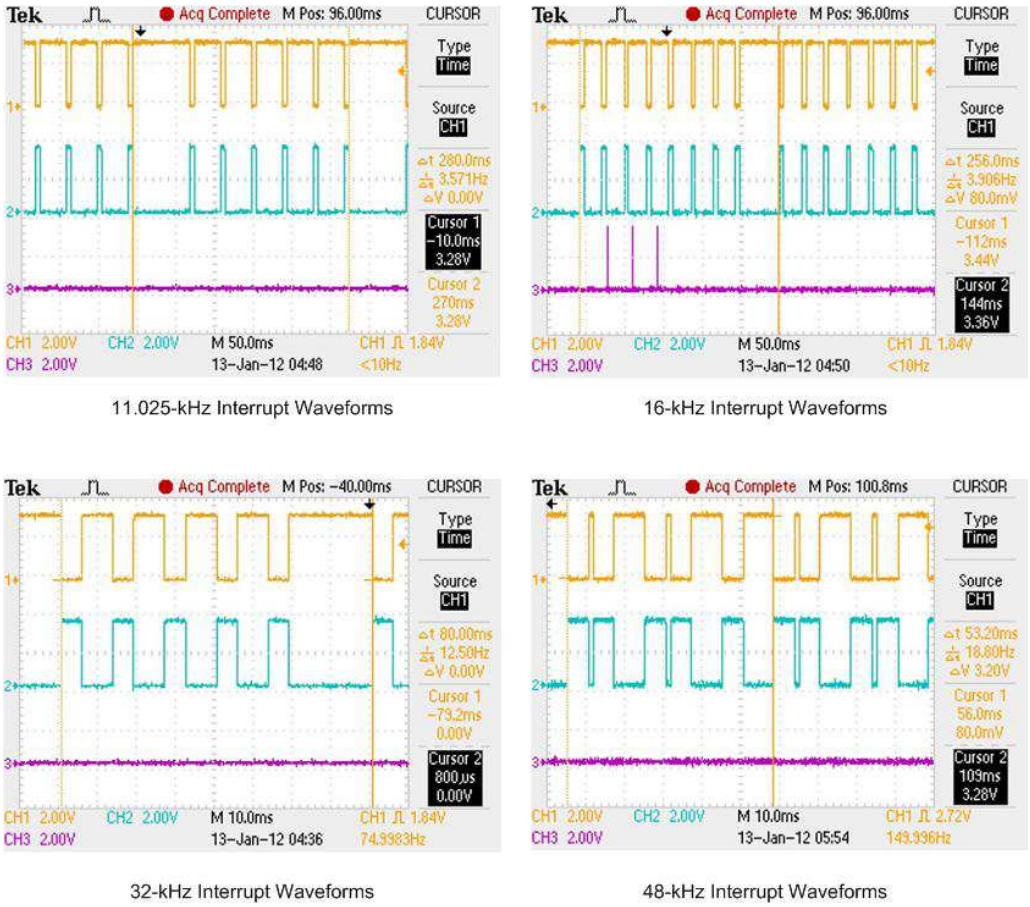

Figure 8. Interrupt Waveforms

10 Modifications

The purpose of the dual-SPI software example is to introduce the usage and capability of the sound API. The dual_spi.c driver represents just one simple use of the sound API, and is intended as a reference. Modification to the sound API is expected and required to tailor to the end-user application, whether a Tiva C Series TM4C123GH6PGE (or Stellaris LM4F232) I^2S substitute or additional I^2S peripheral to a different Tiva C Series microcontroller. This section discusses several characteristics that can be added or modified by the user, and the necessary steps to make these changes.

10.1 Adding Support for Additional Sampling Frequencies

The sound API supports six different sampling frequencies by default: 8, 11.025, 16, 32, 48, and 64 kHz. If an additional sampling frequency is desired, edits to the SoundSetFormat( ) function in sound.c are necessary. By default, the sampling frequency of an audio file is read and stored into the pWaveHeader as ulSampleRate. This value is then passed into the SoundSetFormat( ) function. An if statement is used to select the current sampling rate, and write the required data to the codec registers. More detail on writing to these registers can be found in Section 4, *Software codec Setup*.

For example, if a sampling frequency support of 22.05 kHz is needed, add an additional else-if block for this case. To support 22.05 kHz, set the codec Fsref to 44.1 kHz and scale down by 2. Using the TLV320AIC3107EVM Software Evaluation Tool, shown in Figure 9, the desired Fsref and PLLCLK_IN can be configured to calculate the optimal K, J, and D values. Using these values together with an NDAC of 2, the codec sample rate reaches 22050.

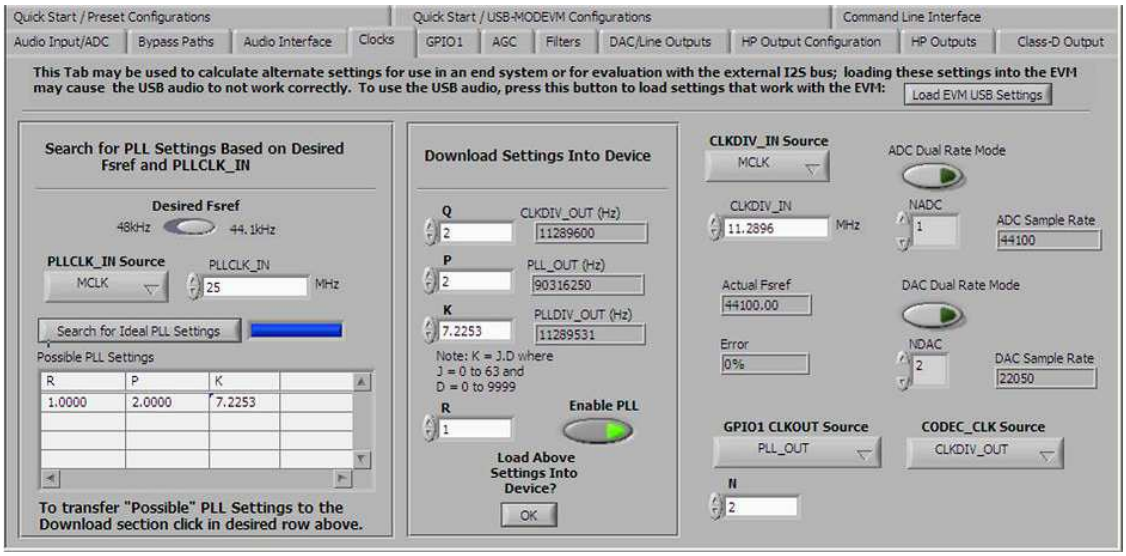

Figure 9. TLV320AIC3107EVM-K GUI Software

10.2 Adding 8- or 32-Bit Audio Support

The sound API by default supports 16-bit audio which is the most intuitive as the SPI transmit FIFO is 16 bits wide. For 8-bit or 32-bit audio support, multiple modifications to the sound API are necessary.

In the beginning of the SoundSetFormat( ) function in sound.c, the SPI protocol, mode of operation, bit rate, and data width are configured. Change the ulDataWidth parameter of the SSIConfigSetExpClk( ) function from 16 to 8 to support 8-bit operation. This parameter defines the width of the data transfers and can be a value between 4 and 16, inclusive. For a 32-bit mode, 16-bit widths are sufficient. For stereo 32-bit, the left and right channels can be divided into the even and odd data blocks. For mono 32-bit, each data frame is divided in two.

After the SPI configurations are made for both mono and stereo formats at the top of the SoundSetFormat( ) function, another modification is required near the bottom of this function, where the DMA settings are configured for the channels used. The ulDMASetting is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. To include 8-bit functionality for mono, set as follows:

```
//
// The transfer size is 8 bits from the TX buffer to the TX FIFO.
//
ulDMASetting = UDMA_SIZE_8 | UDMA_SRC_INC_8 |
               UDMA_DST_INC_NONE | UDMA_ARB_2;
```

For 8-bit stereo, the DMA can be configured to hold the left and right data in chunks, resulting in 16-bit sizes:

```
//
// The transfer size is 16 bits(stereo 8 bits) from the TX buffer
// to the TX FIFO.
//
ulDMASetting = UDMA_SIZE_16 | UDMA_SRC_INC_16 |
               UDMA_DST_INC_NONE | UDMA_ARB_2;
```

The final modifications are required where the transfer parameters for the µDMA channel control structure are set. These modifications are made using the uDMAChannelTransferSet( ) function in the SoundBufferPlay( ) function of sound.c, prior to enabling the µDMA channels. The current implementation configures the µDMA channel for 16-bit data sizes with a source address increment of 32, allowing for splitting the left and right data upon calling the uDMAChannelTransferSet( ) function. The first 16 bits (left channel) are transferred to the first SPI channel:

```
ROM_uDMAChannelTransferSet(ulChannel,
                UDMA_MODE_PINGPONG,
                (unsigned short *)g_sOutBuffers[g_ulPlaying].pulData,
                (void *)(I2S_CHAN0_BASE | SSI_O_DR),
                g_sOutBuffers[g_ulPlaying].ulSize);
```

The next 16 bits (right channel) are transferred to the second SPI channel:

```
ROM_uDMAChannelTransferSet(ulChannel2,
                UDMA_MODE_PINGPONG,
                (unsigned short *)g_sOutBuffers[g_ulPlaying].pulData+1,
                (void *)(I2S_CHAN1_BASE | SSI_O_DR),
                g_sOutBuffers[g_ulPlaying].ulSize );
```

Note the difference in the third and fourth arguments. The 32-bit data chunks, set by the prior source increment size, are split into blocks of 16 and sent out the respective SPI bases.

10.3 Using the Codec On-Board Class D Amplifier

The TLV320AIC3107 is equipped with an on-board class-D speaker amplifier that is not used by this sound API. If a higher gain is needed, reconfigure the codec to include this amplifier in the data path, yielding a gain of up to +18 dB.

This example uses the data path shown in the TLV320AIC3107 functional block diagram of Figure 3. Register 41, controlling the DAC L and DAC R, must be changed to select DAC_L3 and DAC_R3. The Class-D Amp must then be configured and enabled through register 73. See the TLV320AIC3107 data sheet for specifics on these registers. Make these modifications to the driver, dac.c, in the DACInit( ) function. Additional information on function writes to the codec registers is explained in Section 4, *Software codec Setup*.

10.4 Using a Codec Other Than the TLV320AIC3107

The sound API used in this dual-SPI implementation is tailored specifically for the TLV320AIC3107. However, following both the hardware and software configurations shown in this document, these APIs can translate to work successfully on a different codec.

See Figure 1 for the necessary hardware interface for the codec and the required support for External MCLK, External Reset, I^2C, and I^2S. The hardware interface is straightforward and might consist of only minor changes. The majority of the modifications are in the dac.c driver. This driver sets up the necessary infrastructure to read and write values to the codec registers. The DACInit( ) function configures these peripherals and then makes multiple DACWriteRegister( ) calls, specific to the TLV320AIC3107. Each write is thoroughly commented with the description, functionality, and binary data being sent to the codec. The corresponding functionalities must be re-implemented if migrating to a different codec.

Apart from the DACInit( ) function, a few register writes exist in the sound API (for example, when setting the sampling frequency based on the audio file). Because the MCLK of the codec has a frequency of 25 MHz, the SoundSetFormat( ) function hard codes the necessary PLL, Fsref, and NDAC values needed to change the codec sampling frequency so that the audio is sampled from the Tiva SPI FIFO at the required rate. When not using the TLV320AIC3107, every DACWriteRegister( ) call must be tailored to the equivalent register or data to achieve the same functionality. See the substituted codec data sheet and the TLV320AIC3107 data sheet for specific register information.

10.5 Changing the CPU Frequency or Codec Master Clock

The Tiva C Series microcontroller generates the master clock (MCLK) for the TLV320AIC3107 codec, which is a 25-MHz clock signal driven by a 16-bit timer. This timer is simply configured with a unity load value, meaning the timer divides the system clock in half. Because the codec clocking (PLLs, Fsref, NDAC, and so on) is configured based on this predefined MCLK of 25 MHz, an adjustment of the main system clock skews the MCLK and thus the codec sampling rate. For example, if the Tiva C Series system clock is changed from 50 MHz to 40 MHz, the new MCLK would be 40/2= 20 MHz. Because this frequency is slower, the codec samples data from the Tiva SPI FIFOs at a lower rate resulting in sluggish audio output.

The modifications necessary to restore playback rate are in the SoundSetFormat( ) function. Each codec register write in the if statement for each sampling frequency must be re-evaluated for the new MCLK. Using the TLV320AIC3107 EVM Software Evaluation Tool, the PLLCLK_IN field can be changed to the new MCLK value. The newly-calculated PLL settings replace the values written in the SoundSetFormat( ) function. When changing the CPU frequency or codec system clock, this method can be used to accommodate the altered MCLK.

10.6 Using a Microcontroller Other Than LM4F232

Although the primary purpose of this example is to bring audio functionality to the Tiva C Series of microcontrollers, the example could also be used to introduce additional I^2S peripherals on different platforms.

In this case, definitions of the hardware mappings in dac.c and sound.c must be modified for the alternate microcontroller. The driver dac.c contains *#defines* for the I^2C and reset pins used to communicate with the external codec. The driver sound.c contains *#defines* for all pseudo-I^2S pins used in the sound driver. These definitions must be changed, aligning the alternate microcontroller available pins and peripherals to the existing. Port and pin definitions for Tiva C Series microcontrollers can be found in pin_map.h of the TivaWare for C Series DriverLib.

10.7 Using Flash to Store Audio Instead of the SD Card

This dual-SPI sound API reads audio files from an on-board SD card. These read times constitute a majority of CPU usage and can be alleviated by storing the audio in Flash memory. However, this methodology consumes large amounts of memory and is not recommended for audio clips of more than a couple seconds in length.

C arrays containing 8-bit sound clips in .wav format can be created using the makefsutility in TivaWare. This array can be stored in a header file and referenced by the SoundOpen( ) function of the sound API instead of a file object. See the Stellaris EVALBOT demo for an example. The source code can be found in StellarisWare/boards/ek-evalbot/sound_demo/sound_demo.c to understand the correlation.

10.8 Playing Mono versus Stereo Audio

The dual-SPI audio example supports both Mono and Stereo formats, and it is important to understand how the sound driver distinguishes between them.

The sound.c sound driver supports multiple 16-bit sampling frequencies in both mono and stereo formats. These attributes are parsed from the .wav file and populated in the SoundOpen( ) function. After this information is retrieved, configurations are automatically used for the particular format. From the user's perspective, opening a 16-bit, mono 8-kHz .wav file from the SD card is no different than opening a 16-bit stereo 64-kHz .wav file. Multiple formatted audio files can be played back-to-back. The GoldWave freeware software can be useful for exporting and configuring PCM, N-bit, N-kHz audio files.

11 Conclusion

The dual-SPI audio example provides a basis for developing applications incorporating audio on Tiva C Series microcontrollers that do not provide an I^2S module. The provided sound API allows for a straightforward implementation for interfacing a Tiva C Series TM4C123x microcontroller to an external codec over I^2S. The sound API provides a robust, easily manipulated development platform to fit the end-user's LM4F audio application.

12 References

The following related documents and software are available on the Tiva C Series web site at www.ti.com/tiva-c:

- Tiva TM4C123GH6PGE data sheet, literature number SPMS375
- Associated files for this application report, including the source code for the sound API and the dual_spi example: Related files
- Tiva TM4C123GH6PGE microcontroller product folder: TM4C123GH6PGE

The following resources contain additional information about the external codec used in this application report:

- TLV320AIC3107 data sheet, literature number SLAU261
- TLV320AIC3107EVM-K GUI software: SLAC250
- TLV320AIC3107 product folder: TLV320AIC3107

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265