AIC27 Example for the TMS320C5510 Prototype Board

Arthur McClelland *Digital Signal Processing Solutions*

ABSTRACT

This application report is intended to help demystify the AIC27 example program that comes with the TMS320C5510™ prototype board. The example program is intended to display some of the audio capabilities of the C5510™ prototype board. Because it performs all these actions through header files and macros, however, it may seem unclear as to what is actually going on. This document clarifies what is happening, and enables the user to modify the concepts put forth here to their own applications. All of the functions within main() are callable C routines. The syntax for calling, and the location of source code, are given in the C5510 prototype board help file.†

Contents

List of Tables

† The help file for the C5510 prototype board currently is not integrated into the Code Composer Studio™ help file. It is located under the Start Menu –> Programs –> Texas Instruments –> Code Composer Studio 2 –> Documentation –> C5510 prototype board help.

1 Introduction

The complete Audio Codec ’97 Component Specification v2.2 can be downloaded from the Intel Corporation® web site. The Audio Codec ’97 Specification v2.2 contains a complete table of all the codec control registers. Excerpts of the table of control registers are included where appropriate.

The AIC27 program basically performs three actions. First, it generates a tone and plays it to the left channel, then to the right channel, and finally to both channels simultaneously. Next, it takes the line in audio jack of the codec and loops it to the line out audio jack of the codec. (There is a compile option for the digital loop back, to go from the microphone to the line out.) Finally, it sets the three user-configurable LEDs to blink until the board is reset. This spplication report steps through the program line by line, giving descriptions and explanations of the functions and actions of each line.

The lines of code that are against the left margin are lines of code in the *main ()* function. The functions are listed exactly as they appear in the main () function. Functions that are called multiple times in the main () function are listed multiple times. Functions that are called by functions in main () are shown in boldfaced font. All the source code is included in Appendix A for reference. The functions are hyperlinked to the appropriate section of Appendix A for ease of reference. The file name for the source code appears next to each line in braces { }.

The source files are found in one of two folders, C:\ti\c5500\evm\target\lib\sources\evm5510\c or C:\ti\c5500\evm\target\lib\sources\drv55x\c.

2 Functions

brd_init(160) {board.c}

This line is a macro that initializes the C5510 prototype board for use. It takes the parameter of a 16-bit, unsigned integer for the desired board operating frequency. In aic27, the board is initialized at 160 MHz. (Please note board frequencies must be a multiple of 10, with the lowest being 20 MHz and the highest being 160 MHz.) The brd_init function is called from the board.h header file. The source code can be found in board.c and is given in Appendix A. The file board.c can be found in the directory …c5500\evm\target\lib\sources\evm5510\c. For users wishing to create their own programs, understanding exactly how the board is initialized may be important, so each line of this macro will be discussed.

brd_set_cpu_freq(freq); {board.c}

This function is found in board.c. (See Appendix A for source code for this function.) It first ensures that the freq value passed to it is valid. Once the value of freq is deemed valid, or changed to a valid value, the function assigns the value of freq to the global variable cpu_freq. This value is then passed to the function clock_init(EVM_OSC, cpuFreq, PLL_DIV_2);.

clock_init(EVM_OSC, cpuFreq, PLL_DIV_2); {clock.c}

The clock_init() function can be found in clock.c in the directory …c5500\evm\target\lib\ sources\drv55x\c. (See Appendix A for source code for this function.) The purpose of this function is to decide which revision of silicon is being used, and to call either APLL_init(inclk, outclk, plldiv) or DPLL_init(inclk, outclk, plldiv) appropriately. It is assumed that the reader has revision 1.1 or higher, so DPLL_init(inclk, outclk, plldiv) will be not be discussed here.

APLL_init(inclk, outclk, plldiv); {clock.c}

This function sets the clock frequency. The source code is found in clock .c and is included in Appendix A.

brd_emif_init(); {board.c}

The source code for this function is found in board.c, and is also included in Appendix A. It initializes the external memory interface (EMIF).

3 External Memory Interface

The EMIF provides a glueless interface to three types of memory devices:

- Asynchronous devices, including ROM, flash memory, and asynchronous SRAM. See *Using Asynchronous Memory* in the Code Composer Studio help file.
- Synchronous burst SRAM (SBSRAM) running at 1/2 or 1 times the central processing unit (CPU) clock rate. See *Using SBSRAM* in the Code Composer Studio help file.
- Synchronous DRAM (SDRAM) running at either 1/2 or 1 times the CPU clock rate.See *Using SDRAM* in the Code Composer Studio help file.

The EMIF supports the following types of accesses:

- Program accesses
- 32-bit data accesses
- 16-bit data accesses
- 8-bit data accesses

To see what parts of the digital signal processor (DSP) can send external-memory requests to the EMIF, and the order in which the EMIF services simultaneous requests, see *EMIF Request Priorities* in the Code Composer Studio help file.

If you want the DSP to share memory chips with an external device, see *HOLD Requests* in the Code Composer Studio help file.

If you would like to buffer CPU write operations to reduce delays, see *Write Posting* in the Code Composer Studio help file.

TIMER_HALT(TIMER_PORT0); {timer.h}

This line halts the timer on DMA port 0. Timer_Halt is a macro that is called from timer.h.

4 DMA Controller

The direct memory access (DMA) controller allows movement of data to and from internal memory, external memory, and peripherals. This occurs without intervention from the CPU, and takes place in the background of CPU operation. The DMA controller has a dedicated idle domain, and you can put the DMA controller into a low-power state by turning off this domain. Each multichannel buffered serial port (McBSP) on the TMS320C55x™ DSP has the ability to temporarily take the DMA domain out of this idle state when the McBSP needs the DMA controller (see *Reducing Power Consumed by a McBSP* in the Code Composer Studio help file).

The DMA controller has the following important features:

- Operation that is independent of the CPU
- Four standard ports, one for each data resource. They are internal dual-access RAM (DARAM), internal single-access RAM (SARAM), external memory, and peripherals
- An auxiliary port, to enable certain transfers between the enhanced host port interface (EHPI) and memory
- Six channels that allow the DMA controller to keep track of the context of six independent block transfers among the standard ports
- Bits for assigning each channel a low priority or high priority. See *Service Chains* in the Code Composer Studio help file.
- Event synchronization. DMA transfers in each channel can be made dependent on the occurrence of selected events. *See Synchronizing Channel Activity* in the Code Composer Studio help file.
- An interrupt for each channel. Each channel can send an interrupt to the CPU upon completion of certain operational events. See *Channel Interrupt* in the Code Composer Studio help file.
- Software-selectable options for updating addresses for the sources and destinations of data transfers.

For information about the registers used to program the DMA controller, *see Summary of DMA Controller Registers* in the Code Composer Studio help file.

The control registers for the DMA and the McBSP are mapped in input/output (I/O) space, and require the C keyword *ioport* to write to them directly. For example:

#define DMCSSAU_ADDR(port) (DMCSDP_ADDR(port)+5)

#define DMCSSAU(port) *(ioport volatile unsigned int *)DMCSSAU_ADDR(port)

TIMER_HALT(TIMER_PORT1); {timer.h}

This line halts the timer on DMA port 1.

I**NTR_GLOBAL_DISABLE;**

This line disables the global interrupts.

TMS320C55x is a trademark of Texas Instruments.

IER0 = 0;

This line clears the interrupt enable register 0. By setting all the IER bits to 0, all the maskable interrupts are disabled.

To enable a maskable interrupt, set its corresponding enable bit to 1. To disable a maskable interrupt, clear its corresponding enable bit to 0. At reset, all the IER bits are cleared to 0, disabling all the maskable interrupts.

NOTE: IER1 and IER0 are not affected by a software-reset instruction or by a DSP hardware reset. The user should initialize these registers before globally enabling (INTM = 0) the maskable interrupts.

IFR0 = 0;

This line clears the interrupt flag register 0. The 16-bit interrupt flag registers, IFR1 and IFR0, contain flag bits for all the maskable interrupts. When a maskable interrupt request reaches the CPU, the corresponding flag is set to 1 in one of the IFRs. This indicates that the interrupt is pending, or waiting for acknowledgement from the CPU.

The IFRs can be read to identify pending interrupts, and can be written to, to clear pending interrupts. To clear an interrupt request (and clear its IFR flag to 0), write a 1 to the corresponding IFR bit. For example:

MOV #0100000000000100b, mmap(@IFR0); Clear flags IF14 and IF2.

All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR. Acknowledgement of a hardware interrupt request also clears the corresponding IFR bit. A device reset clears all IFR bits.

IER1 = 0;

This line clears the Interrupt Enable Register 1. By setting all the IER bits to 0, all the maskable interrupts are disabled.

IFR1 = 0;

This line clears the Interrupt Flag Register 1.

CPLD_CTRL1_REG = CPLD_CTRL1_DEFAULT;

This line initializes the Complex Programmable Logic Device Control 1 register to a predefined default value. The CPLD_CTRL1_DEFAULT is defined in board.h.

#define CPLD_CTRL1_DEFAULT 0x10

This default enables the NMI (non-maskable interrupt) from the host to be routed to the DSP's NMI pin.

CPLD_CTRL2_REG = CPLD_CTRL2_DEFAULT;

This line initializes the Complex Programmable Logic Device Control 2 register to a predefined default value. The CPLD_CTRL2_DEFAULT is defined in board.h.

#define CPLD_CTRL2_DEFAULT 0x00

CPLD_DBIO_REG = CPLD_DBIO_DEFAULT;

The DBIO register provides four general-purpose bits of I/O to a daughterboard. The four signals can be defined as inputs or outputs under software control. By default, two signals default to inputs, and two signals default to outputs. This ensures backward compatibility with the existing daughterboards' two STAT inputs and two CNTL output signals. The DB_IODIR bits select whether the signals are inputs or outputs. The DB_IO bits select the output values ,or provide the current signal values.

This line initializes the complex programmable logic device daughterboard input/output register to a predefined default value. The CPLD_DBIO_DEFAULT is defined in board.h.

#define CPLD_DBIO_DEFAULT 0x30

The default initializes DBIODIR3 and DBIODIR2 to be input bits, and DBIODIR1 and DBIODIR0 to be output bits.

CPLD_SEM0_REG = CPLD_SEM0_DEFAULT;

This line initializes the complex programmable logic device semaphore 0 register to a predefined default value. The CPLD_SEM0_DEFAULT is defined in board.h.

#define CPLD_SEM0_DEFAULT 0x00

This corresponds to a not-owned state.

CPLD_SEM1_REG = CPLD_SEM1_DEFAULT;

This line initializes the complex programmable logic device semaphore 0 register to a predefined default value. The CPLD_SEM1_DEFAULT is defined in board.h.

#define CPLD_SEM1_DEFAULT 0x00

This corresponds to a not-owned state.

//CPLD_SLIC_REG = CPLD_SLIC_DEFAULT;

The Subscriber Line Interface Circuit (SLIC) register was in the early prototypes, but never made it into the production board. Therefore, it is commented out and can be safely deleted, if so desired. This register does not appear in Table 1 of the CPLD registers.

Table 1. DSP CPLD Register Definitions for the TMS320C5510 EVM Board

Offset + 0xC00008	Name	Description	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	CNTL1	Controls 1	–	DB_RST R/W 0 (No Reset)	DB_INT R/W 0 (No INT)	NMI/EN RW 0 (No INT)	–	USR LED2 R/W 0 (Off)	USR LED1 R/W 0 (Off)	USR LED0 R/W 0 (Off)
1	STAT	Status	DSP_EHP IENA SW R	USR SW1 R	USR SW0 R	NMI R	DB_INT3 R	HST_INT3 R	DB_DET R	DAA RING R
2	–	–	–	–	–	–	–	–	–	–
3	DBIO	Daughter-board GP I/O	IODIR3 R/W 0 (Input)	IODIR2 R/W 0 (Input)	IODIR1 R/W 1 (Output)	IODIR0 R/W 1 (Output)	DBIO3 R/W 0 (Inactive)	DBIO2 R/W 0 (Inactive)	DBIO1 R/W 0 (Inactive)	DBIO0 R/W 0 (Inactive)

Table 1. DSP CPLD Register Definitions for the TMS320C5510 EVM Board (Continued)

Offset + 0xC00008	Name	Description	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
4	CNTL2	Controls 2	–	–	–	–	–	–	BSP1 SEL R/W 0 (MIC)	AU/DIO OUT SEL RW 0 (Stereo)
5	SEM0	Semaphore 0	–	–	–	–	–	–	–	SEM0 R/W 0 (Not Owned)
6	SEM1	Semaphore 1	–	–	–	–	–	–	–	SE/M1 RW 0 (Not owned)
7	–	–	–	–	–	–	–	–	–	–

TIMER_RESET(TIMER_PORT0);

TIMER_RESET(TIMER_PORT1);

These two lines reset the timers for DMA port 0 and port 1.

REG_WRITE(IVPH_ADDR,(((u32)&_vectors & 0xffff00) >> 8));

REG_WRITE(IVPD_ADDR,(((u32)&_vectors & 0xffff00) >> 8));

These two lines initialize the interrupt vector pointers.

boardInitFlag = True;

This line sets the boardInitFlag to “true”, to let the program know that the board has been initialized.

I**NTR_GLOBAL_ENABLE**;

This line enables the global interrupts.

INTR_GLOBAL_DISABLE;

This line disables the global interrupts.

CPLD_CTRL1_REG |= 0x1;

This line turns on the first user LED. It does this by ORing the CTRL1 register with the value of 1, which makes bit 0 of the CTRL1 register the value of 1, while leaving the rest of the register alone.

- CPLD_CTRL1_REG |= 0x2; would turn on the second user LED.
- CPLD_CTRL1_REG |= 0x4; would turn on the third user LED.

Table 1 gives the descriptions of the CPLD registers for the C5510 prototype board.

Aic27ExContTonePb(LEFT, NUM_BUFFS_TO_PLAY); {aic27ex.c}

This line is a function that is given in aic27ex.c. It generates a tone, and plays it to the channel passed to it for the length of time that it takes to play the number of buffers passed to it. The source code for this function is given in Appendix A.

sig_init(TONE_FREQ, 8000); {sig.c}

This function checks to see that the frequency requested is a valid one. It then sets the frequency and the sample rate. Finally, it calls sig_reset(). The source code can be found in sig.c and Appendix A.

sig_reset() {sig.c}

This function calculates the points at which the sinusoidal is to be sampled. It resets the signal generator to a zero-phase signal.

sig(toneBuf[0], BUF_SIZE, N_CH, (SigCh)ch)

sig(toneBuf[1], BUF_SIZE, N_CH, (SigCh)ch)

These two lines actually generate the tone. They use a sine look-up table, and write the values from the sine table to the appropriate buffer, based on the channel selections for output. Buf [0] and Buf [1] are ping-pong buffers.

CPLD_CTRL2_REG &= ~1;

This line ANDs the Complex Programable Logic Device Control Register 2 with the complement of 1. This selects stereo output.

aic27_open(); {aic27.c}

This function initializes and sets up the registers for the codec.

aic27_write_reg(MASTER_VOL, 0x0000, 0xFFFF); {aic27.c}

As shown in Table 2, this line sets the master volume for both the left and right channels to level 0, which is the loudest.

Table 2. Master Volume Register

Master Volume

15	14	13	12	11	10	9	8
Mute	X	L5	L4	L3	L2	L1	L0
7	6	5	4	3	2	1	0
X	X	R5	R4	R3	R2	R1	R0

Default 8000h

aic27_sample_rate(AUDIO_ADC_RRATE, AIC27_8000HZ); {aic27.c}

The address of the Audio Analog-to-Digital Converter Receive Rate register is AUDIO_ADC_RRATE = 0x32. The definitions for the register addresses can be found in aic27.h.

aic27_sample_rate(FRONT_DAC_SRATE, AIC27_8000HZ); {aic27.c}

The address of the Front Digital-to-Analog Converter Send Rate is FRONT_DAC_SRATE = 0x2c. The definitions for the register addresses can be found in aic27.h.

These lines set the sample rate for the FRONT_DAC_SRATE and the AUDIO_ADC_RRATE. The function aic27_sample_rate () checks to see if the register being requested is a legitimate register for changing the sample rate. If the register being passed is deemed legitimate, the function aic27_write_reg() is called to write the appropriate value to the register.

The legitimate registers are FRONT_DAC_SRATE, REAR_DAC_SRATE, LFE_DAC_SRATE, AUDIO_ADC_RRATE, LINE1_SRATE, and LINE2_SRATE.

bufsPlayed = 0;

This line clears the buffer counter.

aic27_cont_play(toneBuf[0], toneBuf[1], BUF_SIZE, 0, &status); {aic27.c}

This function checks to see if the codec is open. If the codec is open, this function then disables the DMA interrupts, sets up the transmit packets, and then re-enables the DMA interrupts.

while (*status);

This line waits until first buffer is completed.

```
while (bufsPlayed++ < duration)
{
        bufNum ^= 1;
        *status = 1;
        //wait for buffer to be empty
        while (*status);
}
```

This while loop plays through all the buffers.

aic27_stop_play(); {aic27.c}

This function resets the codec transfer packets to 0, or NULL, appropriately.

aic27_close(); {aic27.c}

This function calls disable_codec_comm().

disable_codec_comm() {aic27.c}

This function sets the receive stop flag and the transfer stop flag to true, resets the DMA receive and transfer functions, disables the pending interrupts, resets the packets, and resets the McBSP.

dma_reset(AIC27TXCH); //reset Tx DMA ch {dma55x.c}

dma_reset(AIC27RXCH); //reset Rx DMA ch {dma55x.c}

These two lines reset the direct memory access (DMA) control registers for the transmit and receive channels to zero. The source code is found in DMA55x.c.

_disable_aic27_interrupt(); {aic27.c}

This function disables the pending interrupts, and unhooks the interrupt handler.

_reset_aic27_pkts(); {aic27.c}

This function resets the transmit and the receive packets.

_serial_reset(); {aic27.c}

This function resets the multichannel buffered serial port (McBSP), disables the interrupts, and clears any pending interrupt flags.

busy_delay(50); {aic27ex.c}

This is just a delay function. The source code is located at the bottom of aic27ex.c and is also included in Appendix A. The function decrements from 0xFFFF to 0x0000 the number of times passed to it (in this case, 50 times).

CPLD_CTRL1_REG |= 0x2;

This line turns on the second user LED by ORing the CPLD Control 1 register with the value of 2h. See Table 1 for the CPLD registers.

Aic27ExContTonePb(RIGHT, NUM_BUFFS_TO_PLAY); {aic27ex.c}

This line generates a tone again, and plays it to the right channel.

busy_delay(50); {aic27ex.c}

This is just a delay function. The source code is located at the bottom of aic27ex.c, and also in Appendix A. The function decrements from 0xFFFF to 0x0000 the number of times passed to it (in this case, 50 times).

Aic27ExContTonePb(BOTH, NUM_BUFFS_TO_PLAY); {aic27ex.c}

This line generates a tone and plays it through both channels.

busy_delay(50); {aic27ex.c}

This is just a delay function. The source code is located at the bottom of aic27.c, and also in Appendix A. The function decrements from 0xFFFF to 0x0000 the number of times passed to it (in this case, 50 times).

brd_led_toggle(BRD_LED0); {led.c}

This line turns off the first LED.

brd_led_toggle(BRD_LED1); {led.c}

This line turns off the second LED.

Aic27ExLb(LINE_INPUT); {aic27ex.c}

//Aic27ExLb(MIC_INPUT); {aic27ex.c}

One of these lines should be commented out. These lines determine which input selection is being passed to the function Aic27ExLib.

CPLD_CTRL2_REG &= ~1;

This line ANDs CPLD Control Register 2 with the complement of 1. It selects stereo output.

brd_led_disable(BRD_LED0); {led.c}

brd_led_disable(BRD_LED1);

brd_led_disable(BRD_LED2); {led.c}

These three lines turn off the user LEDs. The source code is located in LED.c and in Appendix A.

aic27_open(); {aic27.c}

This line is a macro that is called from aic27.h. The source code is in aic27.c, and is also included in Appendix A. This macro checks to see if the codec is already open. If the codec is not already open, then aic27_open () calls aic_init() to initialize the codec.

aic_init(); {aic27.c}

This function calls two more functions, enable_codec_comm() and _setup_aic27_registers(), both of which are located in aic27.c.

enable_codec_comm(); {aic27.c}

This calls a series of reset functions to get the codec initialized to receive data. This series of functions consists of _reset_aic27_pkts(), _serial_reset(), _setup_aic27_dma(), _enable_aic27_interrupt(), and _setup_aic27_serial_port().

_reset_aic27_pkts(); {aic27.c}

This function resets the transfer and receive packets.

_serial_reset(); {aic27.c}

This function resets the McBSP, disables the interrupts, and clears any pending interrupt flags.

_setup_aic27_dma(); {aic27.c}

This function sets up the DMA registers to deal with the codec.

_enable_aic27_interrupt(); {aic27.c}

This function clears any pending interrupts, hooks and enables the McBSP receive interrupt, and enables the global interrupts.

_setup_aic27_serial_port(); {aic27.c}

This function sets up the McBSP registers to capture the codec input, and loops it to the codec output. Appendix B contains the details of the McBSP setup.

aic27_write_reg(MASTER_VOL, 0x1010, 0xFFFF); {aic27.c}

As shown in Table 2, this line sets the master volume for both the left and right channels to level 1, which is the next-to-loudest level. The address of the master volume register is MASTER_VOL = 0x02. The definitions for the register addresses can be found in aic27.h.

aic27_write_reg(FRONT_MIXER_VOL, 0x1010, 0xFFFF); {aic27.c}

This line sets the volume of the front mixer. The address of the Front Mixer register is FRONT_MIXER_VOL= 0x72. The definitions for the register addresses can be found in aic27.h.

aic27_sample_rate(AUDIO_ADC_RRATE, LOOP_SAMPLE_RATE); {aic27.c}

The address of the Audio Analog-to-Digital Converter Receive Rate register is AUDIO_ADC_RRATE = 0x32. The definitions for the register addresses can be found in aic27.h.

aic27_sample_rate(FRONT_DAC_SRATE, LOOP_SAMPLE_RATE); {aic27.c}

The address of the Front Digital-to-Analog Converter Send Rate is FRONT_DAC_SRATE = 0x2c. The definitions for the register addresses can be found in aic27.h.

These lines set the sample rate for the FRONT_DAC_SRATE and the AUDIO_ADC_RRATE. The function aic27_sample_rate () checks to see if the register being requested is a legitimate register for changing the sample rate. If the register being passed is found to be a legitimate one, the function aic27_write_reg() is called to write the appropriate value to the register.

The legitimate registers are FRONT_DAC_SRATE, REAR_DAC_SRATE, LFE_DAC_SRATE, AUDIO_ADC_RRATE, LINE1_SRATE, and LINE2_SRATE.

```
if (micSel)
{
aic27_write_reg(REC_SEL, MIC_IN, 0xFFFF);//mic_in  {aic27.c}
}
else
{
aic27_write_reg(REC_SEL, LINE_IN, 0xFFFF); //line_in  {aic27.c}
}
```

This IF-ELSE statement decides which input to look for, and then sets the REC_SEL register to the appropriate value. The Record Select register selects which input to record from. The following typedef from aic27.h gives the values for passing to the aic_write_reg ().

This IF–ELSE statement decides which input to look for, and then sets the REC_SEL register to the appropriate value. The Record Select register selects which input to record from. The following typedef from aic27.h gives the values for passing to the aic_write_reg (). Table 3 shows the record select register, and Table 4 gives the valid values for the register. Both tables are excerpted from the *Audio Codec ’97 Component Specification v2.2*.

```
/*****************************************************************************/
/* REC_SEL register settings                                                 */
/*                                                                           */
/* Note that only the first 3 bits of the enumeration are written to the    */
/* register bits based on the mapping below:                                 */
/*                                                                           */
/* b0–b2:      Right Ch                                                      */
/* b8–b10:     Left  Ch                                                      */
/*****************************************************************************/
typedef enum
{
  MIC_IN                    = 0x0000,
   CD_IN                    = 0x0101,
   VIDEO_IN                 = 0x0202,
   AUX_IN                   = 0x0303,
   LINE_IN                  = 0x0404,
   STEREO_MIX               = 0x0505,
   MONO_MIX                 = 0x0606,
   PHONE_MIX                = 0x0707
} AIC27RecordSel, *PAIC27RecordSel;
```

Table 3. Record Select Register

Record Select

15	14	13	12	11	10	9	8
X	X	X	X	X	L2	L1	L0
7	6	5	4	3	2	1	0
X	X	X	X	X	R2	R1	R0

Default 0000h

Table 4. Values for the Record Select Register

Value	Description
0	Mic in
1	CD in
2	Video in
3	Aux in
4	Line in
5	Stereo mix
6	Mono mix
7	Phone

```
for (cnt=0; cnt<BUF_SIZE; cnt++)
{
toneBuf[0][cnt] = 0;
toneBuf[1][cnt] = 0;
}
```

This FOR loop zeros the ping-pong buffers.

aic27_cont_capture(toneBuf[0], toneBuf[1], BUF_SIZE, 0, &bufStatus1); {aic27.c}

This function checks to see if the codec is open. If the codec is open, this function disables the DMA interrupts, sets up the receive packets, and then re-enables the DMA interrupts.

aic27_cont_play (toneBuf[1], toneBuf[0], BUF_SIZE, 0, &bufStatus2); {aic27.c}

This function first checks to see if the codec is open. If the codec is open, this function then disables the DMA interrupts, sets up the transmit packets, and then re-enables the DMA interrupts.

aic27_close(); {aic27.c}

This function calls disable_codec_comm().

disable_codec_comm() {aic27.c}

This function sets the receive stop flag and the transfer stop flag to “true”, resets the DMA receive and transfer functions, disables the pending interrupts, resets the packets, and resets the McBSP.

dma_reset(AIC27TXCH); //reset Tx DMA ch {dma55x.c}

dma_reset(AIC27RXCH); //reset Rx DMA ch {dma55x.c}

These two lines reset the DMA control registers for the transmit and receive channels to zero. The source code is found in DMA55x.c and Appendix A.

_disable_aic27_interrupt(); {aic27.c}

This function disables the pending interrupts and unhooks the interrupt handler.

_reset_aic27_pkts(); {aic27.c}

This function resets the transmit and receive packets.

_serial_reset(); {aic27.c}

This function resets the McBSP, disables the interrupts, and clears any pending interrupt flags.

blink_leds(FOREVER); {aic27ex.c}

This function calls a while loop that loops indefinitely, toggling each user led on and off in turn, after a certain delay time.

brd_led_toggle(BRD_LED0); {led.c}

The switch-case statement in this function decides which user-defined led to toggle. The LED_MASK is declared in board.h. The led is then toggled with the following line:

(CPLD_CTRL1_REG & ledMask) ? brd_led_disable(led) : brd_led_enable(led);

busy_delay(DELAY_TIME); {aic27ex.c}

This is only a time delay function. The source code is located at the bottom of aic27ex.c, and is also shown in *Appendix A*. The function decrements from 0xFFFF to 0x0000 the number of times passed to it (in this case, 100 times, and Delay_Time = 100).

5 References

1. Intel Corporation, Audio Codec ’97 Component Specification v2.2.

Appendix A Code Sample

```
/************************************************************************/
/*  aic27ex.c – Codec sample application for TMS320VC5510 EVM           */
/*    Plays continuous tone to left, right and then both                */
/*    speakers. First LED is on during play to left speaker,            */
/*    second LED is on during play to right speaker, and both are       */
/*    on during play to both speakers. All LEDs flash on success.       */
/*                                                                      */
/************************************************************************/
#include <board.h>
#include <sig.h>
#include <aic27.h>
#include <dma55x.h>
#include <intr.h>

//**********************************************************************/
//* defines
//**********************************************************************/
#define TONE_PB_SAMPLE_RATE         8000
#define LOOP_SAMPLE_RATE            AIC27_48000HZ
#define BUF_SIZE                    1024
#define NUM_BUFFS_TO_PLAY           25
#define TONE_FREQ                   500
#define N_CH                        2
#define DELAY_TIME                  100

#define MIC_INPUT                   1
#define LINE_INPUT                  0

#define FOREVER                     0xFFFFFFFFUL

//**********************************************************************/
//* globals data
//**********************************************************************/
s16      toneBuf[2][BUF_SIZE];
s16      *pBuf[2];
volatile u16 bufsPlayed;

//**********************************************************************/
//* functions
//**********************************************************************/
static s16 Aic27ExContTonePb( s16 ch, s16 duration );
static s16 Aic27ExLb(s16 micSel);
void blink_leds(u32 cnt);
void busy_delay(u16 cnt);
```

```
//*********************************************************************/
//* MAIN
//*********************************************************************/
void main()
{
    // Initialize board
    brd_init(160);

    INTR_GLOBAL_DISABLE;

    // play continuous tone to left, right and then both speakers
    CPLD_CTRL1_REG |= 0x1;
    Aic27ExContTonePb(LEFT, NUM_BUFFS_TO_PLAY );

    busy_delay(50);

    CPLD_CTRL1_REG |= 0x2;
    Aic27ExContTonePb(RIGHT, NUM_BUFFS_TO_PLAY );

    busy_delay(50);

    Aic27ExContTonePb(BOTH, NUM_BUFFS_TO_PLAY );

    busy_delay(50);
    brd_led_toggle(BRD_LED0);
    brd_led_toggle(BRD_LED1);

    // loop line or mic input to line output
    Aic27ExLb(LINE_INPUT);
      //Aic27ExLb(MIC_INPUT);

    // Do nothing else during loopback
    blink_leds(FOREVER);
}
```

```
/*************************************************************************/
/* s16 brd_init(u16 freq) – Initialize EVM board for use                 */
/*                                                                       */
/*  Parameters:                                                          */
/*    – freq: Desired board operating frequency.                         */
/*                                                                       */
/*  Return:                                                              */
/*  – BRD_OK on success                                                  */
/*  – BRD_ERROR on failure                                               */
/*                                                                       */
/*  Notes:                                                               */
/*                                                                       */
/* Board frequencies must be a multiple of 10 with the lowest being 20 MHz */
/* and the highest being 160 MHz.                                        */
/*                                                                       */
/*************************************************************************/
s16 brd_init(u16 freq)
{
   brd_set_cpu_freq(freq);

   brd_emif_init();

   TIMER_HALT(TIMER_PORT0);
   TIMER_HALT(TIMER_PORT1);

   // disable interrupts
   INTR_GLOBAL_DISABLE;

   // clear IFR/IER registers
   IER0 = 0;
   IFR0 = 0;
   IER1 = 0;
   IFR1 = 0;

   CPLD_CTRL1_REG  = CPLD_CTRL1_DEFAULT;
   CPLD_CTRL2_REG  = CPLD_CTRL2_DEFAULT;
   CPLD_DBIO_REG   = CPLD_DBIO_DEFAULT;
   CPLD_SEM0_REG   = CPLD_SEM0_DEFAULT;
   CPLD_SEM1_REG   = CPLD_SEM1_DEFAULT;
   //CPLD_SLIC_REG   = CPLD_SLIC_DEFAULT;

   // reset DSP timer
   TIMER_RESET(TIMER_PORT0);
   TIMER_RESET(TIMER_PORT1);

   //Initialize interrupt vector pointers
   REG_WRITE(IVPH_ADDR,(((u32)&_vectors & 0xffff00) >> 8));
   REG_WRITE(IVPD_ADDR,(((u32)&_vectors & 0xffff00) >> 8));

   boardInitFlag = True;

   INTR_GLOBAL_ENABLE; // Enable Interrupts

   return(BRD_OK);
}
```

```
/*************************************************************************/
/* void brd_set_cpu_freq()                                               */
/*                                                                       */
/* This routine sets the CPU frequency                                   */
/*                                                                       */
/* Parameters:                                                           */
/* – freq frequency in MHz to set cpu clock                              */
/*                                                                       */
/* Return:                                                               */
/* – void                                                                */
/*                                                                       */
/* Notes:                                                                */
/*                                                                       */
/* Board frequencies must be a multiple of 10 with the lowest being 20 MHz  */
/* and the highest being 160 MHz.                                        */
/*************************************************************************/
void brd_set_cpu_freq(u16 freq)
{
  // limit frequency to max
  if (freq > MAX_CPU_FREQ)
      freq = MAX_CPU_FREQ;

  // limit frequency to min
  if (freq < MIN_CPU_FREQ)
      freq = MIN_CPU_FREQ;

  // if selected freq is current freq, do nothing
  if (freq == cpuFreq)
      return;

  // calculate nearest freq without going over
  if (freq % 10)
      freq = ((freq/10) * 10);

  // set global variable
  cpuFreq = freq;

  // initialize clock generator
  clock_init(EVM_OSC, cpuFreq, PLL_DIV_2);
}
```

```
/*************************************************************************/
/* clock_init() – Initialize clock frequency to specified value          */
/*                                                                       */
/* if pllmult > 1                                                        */
/*                 outclk = (pllmult / (plldiv + 1)) * inclk             */
/*                                                                       */
/* if pllmult < 1                                                        */
/*                 outclk = (1 / (plldiv + 1)) * inclk                   */
/*                                                                       */
/*************************************************************************/
void clock_init(unsigned int inclk, unsigned int outclk, unsigned int plldiv)
{
        unsigned int dieid3, dieid2;

        dieid3 = DIE_ID3;
        dieid2 = DIE_ID2;

        if ((dieid3 && 0x0001) & (dieid2 && 0x4000))
        {
                /* rev 1.1/1.1a/1.2 Si */
             APLL_init(inclk, outclk, plldiv); }
        else
        {
                /* rev 1.0/1.0a Si */
        DPLL_init(inclk, outclk, plldiv);
        }


}*************************************************************************/
/* APLL_init() – Initialize clock frequency to specified value           */
/*                                                                       */
/* if pllmult > 1                                                        */
/*                 outclk = (pllmult / (plldiv + 1)) * inclk             */
/*                                                                       */
/* if pllmult < 1                                                        */
/*                 outclk = (1 / (plldiv + 1)) * inclk                   */
/*                                                                       */
/*************************************************************************/
void APLL_init(unsigned int inclk, unsigned int outclk, unsigned int plldiv)
{
  unsigned int pllmult = (outclk * (plldiv+1)) / inclk;
  unsigned int aplldiv, apllmult = 0;

  // ensure plldiv is only 2 bits
  plldiv &= 0x3u;

        if (pllmult % 2)
        {
                  /* odd */
                  aplldiv = 0x0003;
                  apllmult = pllmult – 1;
        }
        else
```

```
        {
                /* even */
                aplldiv = 0x0002;
                apllmult = (outclk == inclk) ? 0x000F : ((pllmult/2) – 1);
        }

        //force into BYPASS mode (b4=0)
        CLKMD = 0;

        //wait for BYPASS mode to be active
        while (CLKMD & (1<<PLLENABLE));

        CLKMD =
        ((1<<VCO_ONOFF)|(apllmult<<PLLMULT)|(aplldiv<<PLLDIV)|(1<<PLLENABLE));

        //wait for PLL mode to be active if pllmult > 1
        if (pllmult > 1)
        {
                while (!(CLKMD & (1<<PLLENABLE)));
        }
}


/*************************************************************************/
/* void brd_emif_init()                                                  */
/*                                                                       */
/* This routine initializes the EMIF registers                           */
/*                                                                       */
/* Parameters:                                                           */
/* – None                                                                */
/*                                                                       */
/* Return:                                                               */
/* – None                                                                */
/*                                                                       */
/* Notes:  CE2 space, where the daughter board resides, is not configured   */
/*      so users will need to setup this space when a daughter board is     */
/*      present.                                                         */
/*************************************************************************/
void brd_emif_init()
{
        EmifSpaceCtrl ce0, ce1, ce3;
        u16 writeSetup, writeStrobe, writeHold;
        u16 readSetup, readStrobe, readHold, freq;
        u16 gctrl;

        if (cpuFreq > 120)
                gctrl = (1 << MEMCEN) | (1 << MEMFREQ);
        else
                gctrl = (1 << MEMCEN);

        //1M SBSRAM. if MTYPE selects SBSRAM, remaining
        //fields (ctrl1, ctrl2, ctrl3) do not apply
        ce0.ctrl1 = (MTYPE_32SBSRAM << MTYPE);
```

```
ce0.ctrl2 = 0;
ce0.ctrl3 = 0;

freq = cpuFreq / 10;

readSetup = READ_SETUP_NS;
readStrobe = (u16)((READ_STROBE_NS * freq) / 100) + 1;
readHold = (u16)((READ_HOLD_NS * freq) / 100) + 1;

writeSetup = WRITE_SETUP_NS;
writeStrobe = (u16)((WRITE_STROBE_NS * freq) / 100) + 1;
writeHold = (u16)((WRITE_HOLD_NS * freq) / 100) + 1;

//1M Flash

#if 0
ce1.ctrl1 = (MTYPE_32ASYNC << MTYPE) |
                 (0xf << READ_SETUP) |
                 (0x3f << READ_STROBE) |
                 (0x3 << READ_HOLD);

  ce1.ctrl2 = (0xf << WRITE_SETUP) |
                  (0x3f << WRITE_STROBE) |
                  (0x3 << WRITE_HOLD) |
                  (0x3 << EXT_HOLD_WRITE) |
                  (0x3 << EXT_HOLD_READ);

ce1.ctrl3 = 0;

#else
ce1.ctrl1 = (MTYPE_32ASYNC << MTYPE) |
                (readSetup << READ_SETUP) |
                (readStrobe << READ_STROBE) |
                (readHold << READ_HOLD);

ce1.ctrl2 = (writeSetup << WRITE_SETUP)  |
                (writeStrobe << WRITE_STROBE) |
                (writeHold << WRITE_HOLD) |
                (0x3 << EXT_HOLD_WRITE) |
                (u16)(0x3 << EXT_HOLD_READ);

ce1.ctrl3 = 0;

#endif

//CPLD
ce3.ctrl1 = (MTYPE_16ASYNC << MTYPE) |
                (0xf << READ_SETUP) |
                (0x3f << READ_STROBE) |
                (0x3 << READ_HOLD);
```

```
        ce3.ctrl2 = (0xf << WRITE_SETUP) |
                    (0x3f << WRITE_STROBE) |
                    (0x3 << WRITE_HOLD) |
                    (0x3 << EXT_HOLD_WRITE) |
                    (0x3 << EXT_HOLD_READ);

        ce3.ctrl3 = 0;

        //4M Daughter board in ce2 space defaulting for now
        //CPLD registers in ce3 space defaulting for now

        emif_init(gctrl, &ce0, &ce1, (EmifSpaceCtrl *)0, &ce3, (EmifSdramCtrl *)0);
}


/*************************************************************************/
/* Aic27ExContTonePb() – Play continuous tone                            */
/*   This function provides an example of playing back a buffer of data in  */
/*   continuous (dma autoinitialization) mode.                           */
/*                                                                       */
/* Note: The volume of the output from the front_mixer of the AIC27 codec   */
/*       should be set to mute, otherwise the audio will be heard.       */
/*                                                                       */
/*************************************************************************/
static s16 Aic27ExContTonePb( s16 ch, s16 duration )
{
  s16 *status;
  volatile u16 bufNum = 0;

  //initialize tone generator
  sig_init(TONE_FREQ, 8000);

  //synthesize tone
  sig(toneBuf[0], BUF_SIZE, N_CH, (SigCh)ch);
  sig(toneBuf[1], BUF_SIZE, N_CH, (SigCh)ch);

  //select stereo output
  CPLD_CTRL2_REG &= ~1;

  //open codec
  aic27_open();

  //set volume
  aic27_write_reg(MASTER_VOL, 0x0000, 0xFFFF);
  aic27_write_reg(REC_SEL, 0x0404, 0xFFFF);

  //set sample rate
  aic27_sample_rate(AUDIO_ADC_RRATE, AIC27_8000HZ);
  aic27_sample_rate(FRONT_DAC_SRATE, AIC27_8000HZ);

  //clear buffer counter
  bufsPlayed  = 0;

  //perform continuous play
  aic27_cont_play(toneBuf[0], toneBuf[1], BUF_SIZE, 0, &status);

  //wait until first buffer is completed
  while (*status);
```

```
  //loop through all buffers
  while (bufsPlayed++ < duration)
 {
        bufNum ^= 1;
        *status = 1;

        //wait for buffer to be empty
        while (*status);
 }

  // stop codec
  aic27_stop_play();

  //close codec
  aic27_close();

  return(BRD_OK);
}


/************************************************************************/
/* int sig_init(unsigned int freq_, unsigned int srate_)                */
/*                                                                      */
/* This functions sets the frquency and the sample rate.                */
/*                                                                      */
/* Parameters:                                                          */
/*                unsigned int freq_                                    */
/*                unsigned int srate_                                   */
/*                                                                      */
/* Return:                                                              */
/*                – BRD_OK on success                                   */
/*                – BRD_ERROR on failure                                */
/*                                                                      */
/************************************************************************/
int sig_init(unsigned int freq_, unsigned int srate_)
{
           if (freq_ > SIG_MAX_FREQ)
                     return –1;

           freq = freq_;
           srate = srate_;

           sig_reset();

           return 0;
}
```

```
/*************************************************************************/
/* void sig_reset()                                                      */
/*                                                                       */
/* This function calculates the points at which the sinusoidal           */
/* is to be sampled. It resets the signal generator to a zero phase      */
/* signal.                                                               */
/*                                                                       */
/* Parameters:    none                                                   */
/*                                                                       */
/* Return:        none                                                   */
/*                                                                       */
/*************************************************************************/
void sig_reset()
{
  u32 delta;

  delta = ((u32)freq * 256) / (u32)srate;

  st.delta = (u16)delta;
  st.offset = 0;
}


/*************************************************************************/
/* void sig(int *pBuf, unsigned int size, unsigned int num_ch, SigCh ch)  */
/*                                                                       */
/* This function generates a buffer full of tone data. It approximates   */
/* by using the look-up table.                                           */
/*                                                                       */
/* Parameters:                                                           */
/*    pBuf – Pointer to buffer to store tone data                        */
/*    num_ch – Specifies either a single channel or 2 channels. For a    */
/*    single channel consecutive samples in the buffer differ where as for */
/*    the 2 channel case identical LEFT and right tone samples are       */
/*    placed in the buffer.                                              */
/*    ch – Specifies the channel to store the data: LEFT, RIGHT, or BOTH */
/*                                                                       */
/* Return:    none                                                       */
/*                                                                       */
/*************************************************************************/
void sig(int *pBuf, unsigned int size, unsigned int num_ch, SigCh ch)
{
          unsigned int i = 0;
          int temp;

          for (i=0; i<size; i++)
          {
               pBuf[i] = sine_table[st.offset];

               if (num_ch != 1)
               {
                       temp = pBuf[i];
```

```
                        if ( (ch != LEFT) && (ch != BOTH) )
                                pBuf[i] =  0;

                        ++i;

                        pBuf[i] = (((ch == RIGHT) || (ch == BOTH))) ? temp : 0;
                }

                //calculate next offset
                st.offset += st.delta;

                //wrap offset
                st.offset &= 0xFF;
        }
}


/*************************************************************************/
/*    aic27_write_reg() – Write an AIC27 control register                */
/*                                                                       */
/*  Parameters –                                                         */
/*  reg – an element of an enumeration indicating the register to write. */
/*  value – a number containing the bits to be written, already in their */
/*          correct field locations.                                     */
/*  mask – a number indicating the position of the bits in ”value” to be */
/*         written.                                                      */
/*                                                                       */
/* Returns – BRD_OK on success, BRD_ERROR on failure.                    */
/*                                                                       */
/*************************************************************************/
u16 aic27_write_reg(AIC27Reg reg, u16 value, u16 mask)
{
  // If codec closed, just return
  if (cdx.state == AIC27_CODEC_CLOSED)
         return (u16)BRD_ERROR;

// Mask register and get write value
   if (mask != 0xFFFF)
   {
     value |= (aic27_read_reg(reg) & ~mask);
   }

  //disable receive DMA interrupt
  IER0 &= (u16)(~(1<<AIC27_RXDMA_FLAG));

  //assemble cmd addr
  //b19 in composite 20-bit word is 0
  //aic27Cmd.addrL = ((reg & 0xe) << 12);
  //aic27Cmd.addrH = ((reg & 0x70) >> 4);
    aic27Cmd.addr = ((u32)((u32)reg & 0x7e) << 12);
```

```
  //assemble cmd data
    aic27Cmd.data = value;
    aic27Cmd.data <<= 4;

   //set tag to indicate valid addr/data on next frame
     aic27Cmd.tag |= (AIC27_VALID_REGADDR_CH | AIC27_VALID_REGDATA_CH);

   //clear write flag
     writeDone = False;

   //enable receive DMA interrupt
    IER0 |= (u16)(1<<AIC27_RXDMA_FLAG);

  // Wait for write to complete
     while (writeDone == False);
     return (u16)BRD_OK;
 }


/*************************************************************************/
/* s16 brd_led_disable(BrdLed led)                                       */
/*                                                                       */
/* This routine disables the specified user-defined LED.                 */
/*                                                                       */
/* Parameters:                                                           */
/*    – led – indicates which led to disable                             */
/*                                                                       */
/* Return:                                                               */
/*                 BRD_OK on success                                     */
/*                 BRD_ERROR on failure                                  */
/*                                                                       */
/* Notes:                                                                */
/*                                                                       */
/*************************************************************************/
s16 brd_led_disable(BrdLed led)
{
   switch (led)
   {
   case BRD_LED0:
     CPLD_CTRL1_REG &= ~LED0_MASK;
     break;

   case BRD_LED1:
     CPLD_CTRL1_REG &= ~LED1_MASK;
     break;

   case BRD_LED2:
     CPLD_CTRL1_REG &= ~LED2_MASK;
     break;
```

```
   default:
     return BRD_ERROR;
   }

   return (BRD_OK);
}


/************************************************************************/
/* aic27_open() – Open AIC27 codec and connect codec to McBSP.          */
/*                                                                      */
/*                                                                      */
/*    Parameters – None                                                 */
/*                                                                      */
/*    Returns – BRD_OK on success, BRD_ERROR on failure.                */
/*                                                                      */
/************************************************************************/
int aic27_open()
{
    // If codec is already open return BRD_OK
    if (cdx.state == AIC27_CODEC_OPENED)
           return BRD_OK;

    // Set flag to opened
    cdx.state = AIC27_CODEC_OPENED;

    // Initialize AIC27
    aic27_init();

    // Return true
    return BRD_OK;
}


/************************************************************************/
/* aic27_init() – Initialize AIC27 codec by setting up the registers    */
/*                                                                      */
/*    Parameters – None                                                 */
/*                                                                      */
/*    Returns – None                                                    */
/*                                                                      */
/************************************************************************/
void aic27_init()
{
          rx_stop = tx_stop = False;

          //Enable codec communications
            enable_codec_comm();

          //Setup AIC27 codec registers
          _setup_aic27_registers();
}
```

```
static void enable_codec_comm()
{
          _reset_aic27_pkts();

          _serial_reset();

          _setup_aic27_dma();

          _enable_aic27_interrupt();

          //Setup McBSP
          _setup_aic27_serial_port();
}

static void _reset_aic27_pkts()
{
          //reset AIC27 input and output frames
          aic27Status.tag = 0;
          aic27Cmd.tag = 0;
          aic27Status.addr = aic27Cmd.addr = 0;
          aic27Status.data = aic27Cmd.data = 0;

          // Initialize buffer pointers and callback function
          cdx.rxPkt.size = 0;
          cdx.rxPkt.indx = 0;
          cdx.rxPkt.bufi = (bool)0;
          cdx.rxPkt.pBuf[0]  = NULL;
          cdx.rxPkt.pBuf[1]  = NULL;
          cdx.rxPkt.callback = NULL;

          cdx.txPkt.size = 0;
          cdx.txPkt.indx = 0;
          cdx.txPkt.bufi = (bool)0;
          cdx.txPkt.pBuf[0]  = NULL;
          cdx.txPkt.pBuf[1]  = NULL;
           cdx.txPkt.callback = NULL;

           dmaTxBuf.tag = dmaRxBuf.tag = 0;
           dmaTxBuf.data = dmaRxBuf.data = 0;
           dmaTxBuf.addr = dmaRxBuf.addr = 0;
           dmaTxBuf.leftCh = dmaRxBuf.leftCh = 0;
           dmaTxBuf.rightCh = dmaRxBuf.rightCh = 0;
}

/*************************************************************************/
/* _serial_reset() – Resets the device's serial port                     */
/*************************************************************************/
static void _serial_reset()
{
   // Reset McBSP
   MCBSP_TX_RESET(AIC27_PORT);
   MCBSP_RX_RESET(AIC27_PORT);
```

```
   // Disable interrupts
   IER0 &= (u16)(~(1<<AIC27_RXDMA_FLAG));

   // Clear pending interrupt flags
   IFR0 |= (u16)(1<<AIC27_RXDMA_FLAG);
}

/*****************************************************************************/
/* _setup_aic27_dma() – setup DMA ch 4 & 5 for AIC27 codec                   */
/* TX – Ch 4, RX – Ch 5                                                      */
/*****************************************************************************/
static void _setup_aic27_dma()
{
   u32 addr = 0;
   u16 mSpace = 0;

   DMGCR = 0x0000; //initialize gcr

   // clear all DMA registers
   reset_dma();

   /*************************************************************************/
   /* Transmit DMA setup                                                    */
   /*************************************************************************/
   DMCCR(AIC27TXCH)  = (DM_NOMOD     << DM_DST_AMODE)  |
            (DM_POST_INCR << DM_SRC_AMODE)  |
            (1 << DM_AUTOINIT)  |  // Requires ENDPROG
            (1 << DM_PRIO) |
            (DMSYNC_XEVT2 << DM_SYNC);

   // Determine memory type of source data
   mSpace = setMemSpace((u32)&dmaTxBuf);

   DMCSDP(AIC27TXCH)  = (DM_RHEA << DM_DST) |
            (mSpace << DM_SRC)  |
            (DM_DTYPE_32 << DM_DATA_TYPE);

    // Transmit interrupt is not used (only receive)

    DMCICR(AIC27TXCH)  = 0;
    // set src address (byte address)
    addr = ((u32)&dmaTxBuf) << 1;
    DMCSSAL(AIC27TXCH) = (u16)addr;
    DMCSSAU(AIC27TXCH) = (u16)((addr & 0xff0000)>>16);

    // set dst address (byte address)
    addr = ((u32)DXR2_ADDR(AIC27_PORT)) << 1;
    DMCDSAL(AIC27TXCH) = (u16)addr;
    DMCDSAU(AIC27TXCH) = (u16)((addr & 0xff0000)>>16);

    DMCEN(AIC27TXCH)   = DMA_BUFSIZE;
    DMCFN(AIC27TXCH)   = 1;
    DMCEI(AIC27TXCH)   = 0;
    DMCFI(AIC27TXCH)   = 0;
```

```
/*************************************************************************/
/* //Receive DMA setup                                                   */
/*************************************************************************/
    DMCCR(AIC27RXCH)  = (DM_POST_INCR << DM_DST_AMODE)  |
            (DM_NOMOD << DM_SRC_AMODE) |
            (1 << DM_REPEAT) |
            (1 << DM_AUTOINIT) |
            (1 << DM_PRIO)  |
            (DMSYNC_REVT2 << DM_SYNC);

    // Determine memory type of destination
    mSpace = setMemSpace((u32)&dmaRxBuf);

    DMCSDP(AIC27RXCH)  = (mSpace << DM_DST)  |
            (DM_RHEA << DM_SRC) |
            (DM_DTYPE_32 << DM_DATA_TYPE);

    DMCICR(AIC27RXCH)  = (1 << DM_FRAME);

    // set src address
    addr = ((u32)DRR2_ADDR(AIC27_PORT)) << 1;
    DMCSSAL(AIC27RXCH) = (u16)addr;
    DMCSSAU(AIC27RXCH) = (u16)((addr & 0xff0000)>>16);

    // set dst address
    addr = ((u32)&dmaRxBuf) << 1;
    DMCDSAL(AIC27RXCH) = (u16)addr;
    DMCDSAU(AIC27RXCH) = (u16)((addr & 0xff0000)>>16);

    DMCEN(AIC27RXCH)   = DMA_BUFSIZE;
    DMCFN(AIC27RXCH)   = 1;
    DMCEI(AIC27RXCH)   = 0;
    DMCFI(AIC27RXCH)   = 0;
}
static void _enable_aic27_interrupt()
{
    // Clear any pending interrupt
    IFR0 |= (u16)(1<<AIC27_RXDMA_FLAG);

    // Hook and enable McBSP receive interrupt
    hook_interrupt(AIC27_RXDMA_TRAP, _aic27_rxdma_isr);
    IER0 |= (u16)(1<<AIC27_RXDMA_FLAG);

    // Make sure global interrupts are enabled
    INTR_GLOBAL_ENABLE;
}
```

```
/*************************************************************************/
/* _setup_aic27_serial_port() – setup serial port for AIC27 codec        */
/*************************************************************************/
static void _setup_aic27_serial_port()
{
   SPCR1(AIC27_PORT) = SPCR1_VAL;
   SPCR2(AIC27_PORT) = SPCR2_VAL;

   RCR1(AIC27_PORT)  = RCR1_VAL;
   RCR2(AIC27_PORT)  = RCR2_VAL;

   XCR1(AIC27_PORT)  = XCR1_VAL;
   XCR2(AIC27_PORT)  = XCR2_VAL;

   SRGR1(AIC27_PORT) = 0;
   SRGR2(AIC27_PORT) = 0;

   MCR1(AIC27_PORT)  = 0;
   MCR2(AIC27_PORT)  = 0;

   PCR(AIC27_PORT)   = PCR_VAL;

   RCERA(AIC27_PORT) = 0;
   RCERB(AIC27_PORT) = 0;
   RCERC(AIC27_PORT) = 0;
   RCERD(AIC27_PORT) = 0;
   RCERE(AIC27_PORT) = 0;
   RCERF(AIC27_PORT) = 0;
   RCERG(AIC27_PORT) = 0;
   RCERH(AIC27_PORT) = 0;

   XCERA(AIC27_PORT) = 0;
   XCERB(AIC27_PORT) = 0;
   XCERC(AIC27_PORT) = 0;
   XCERD(AIC27_PORT) = 0;
   XCERE(AIC27_PORT) = 0;
   XCERF(AIC27_PORT) = 0;
   XCERG(AIC27_PORT) = 0;
   XCERH(AIC27_PORT) = 0;

   readDone = writeDone = True;

   //now start McBsp and DMA transfers
    DMA_ENABLE(AIC27RXCH);
    DMA_ENABLE(AIC27TXCH);

    // Enable serial port (both TX/RX)
    MCBSP_ENABLE(AIC27_PORT, MCBSP_BOTH);

    //wait until codec is ready (b15 of slot 1 must be '1')
    //while (!(aic27Status.tag & AIC27_CODEC_READY));
}
```

```
/*****************************************************************************/
/*    aic27_sample_rate() – Set the specified channel sample rate.           */
/*                                                                           */
/*    Parameters –                                                           */
/*    reg – an element of an enumeration indicating the pertinent register. */
/*    freq – an element of an enumeration indicating the frequency.          */
/*                                                                           */
/*    Returns – BRD_OK on success, BRD_ERROR on failure.                     */
/*                                                                           */
/*****************************************************************************/
int aic27_sample_rate(AIC27Reg reg, u16 freq)
{
          if ((reg == FRONT_DAC_SRATE) ||
                    (reg == REAR_DAC_SRATE)  ||
                    (reg == LFE_DAC_SRATE)   ||
                    (reg == AUDIO_ADC_RRATE) ||
                    (reg == LINE1_SRATE)     ||
                    (reg == LINE2_SRATE))
          {
                    return aic27_write_reg(reg, freq, 0xFFFF);
          }

          return BRD_ERROR;
}

 void busy_delay(u16 cnt)
{
         volatile u16 temp;

         while (cnt--)
         {
                    temp = 0xffff;
                    while(temp--);
         }
}
```

```
/*************************************************************************/
/* aic27_cont_capture()                                                  */
/*                                                                       */
/*    This function is used to capture data from the codec               */
/*                                                                       */
/*    Parameters –                                                       */
/*       pingBuf – a buffer to contain data read                         */
/*       pongBuf – a second buffer to contain data read                  */
/*       size – the number of samples in each buffer                     */
/*       pfunc – Pointer to function to run at end of buffer             */
/*       bufStatus – pointer to transfer status.                         */
/*               1 = transfer in progress, 0 = transfer complete         */
/*                                                                       */
/*    Returns – BRD_OK on success, BRD_ERROR on failure.                 */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
int aic27_cont_capture(int *pingBuf,
                int *pongBuf,
                unsigned int size,
                Fp pfunc,
                int **bufStatus)
{
    // Codec must be opened first
    if (cdx.state == AIC27_CODEC_CLOSED)
        return BRD_ERROR;

    //disable receive DMA interrupt
    IER0 &= (u16)(~(1<<AIC27_RXDMA_FLAG));

    /* Setup receive packet */
    cdx.rxPkt.size = size;
    cdx.rxPkt.indx = 0;
    cdx.rxPkt.bufi = (bool)0;
    *bufStatus = (int *)&cdx.rxPkt.status;
    cdx.rxPkt.status = 1;
    cdx.rxPkt.callback = pfunc;
    cdx.rxPkt.pBuf[0] = pingBuf;
    cdx.rxPkt.pBuf[1] = pongBuf;

    //enable receive DMA interrupt
    IER0 |= (u16)(1<<AIC27_RXDMA_FLAG);

    return BRD_OK;
}
```

```
/*************************************************************************/
/* aic27_cont_play()                                                     */
/*                                                                       */
/*    This function is used to continuously send data to the codec.      */
/*                                                                       */
/*    Parameters –                                                       */
/*       pingBuf – a buffer containing data to play                      */
/*       pongBuf – a second buffer containing data to play               */
/*       size – the number of samples in each buffer                     */
/*       pfunc – Pointer to function to run at end of buffer             */
/*       bufStatus – pointer to transfer status.                         */
/*          1 = transfer in progress, 0 = transfer complete              */
/*                                                                       */
/*    Returns – BRD_OK on success, BRD_ERROR on failure.                 */
/*                                                                       */
/*************************************************************************/
int aic27_cont_play(int *pingBuf,
                int *pongBuf,
                unsigned int size,
                Fp pfunc,
                int **bufStatus)
{
    // Codec must be opened first
    if (cdx.state == AIC27_CODEC_CLOSED)
        return BRD_ERROR;

    //disable receive DMA interrupt
    IER0 &= (u16)(~(1<<AIC27_RXDMA_FLAG));

    // Setup transmit pack
    cdx.txPkt.size = size;
    cdx.txPkt.indx = 0;
    cdx.txPkt.bufi = (bool)0;
    *bufStatus = (int *)&cdx.txPkt.status;
    cdx.txPkt.status = 1;
    cdx.txPkt.callback = pfunc;
    cdx.txPkt.pBuf[0] = pingBuf;
    cdx.txPkt.pBuf[1] = pongBuf;

    //enable receive DMA interrupt
    IER0 |= (u16)(1<<AIC27_RXDMA_FLAG);

    return BRD_OK;
}
```

```
/*************************************************************************/
/* aic27_stop_play() – Stops sending data to AIC27                       */
/*                                                                       */
/* Parameters –  None                                                    */
/*                                                                       */
/* Returns – None                                                        */
/*                                                                       */
/*************************************************************************/
void aic27_stop_play()
{
   cdx.txPkt.callback = 0;
   cdx.txPkt.pBuf[0] = NULL;
   cdx.txPkt.pBuf[1] = NULL;
}

/*************************************************************************/
/* aic27_close() – Closes codec connection                               */
/*                                                                       */
/*   Parameters – None                                                   */
/*                                                                       */
/*   Returns – BRD_OK on success, BRD_ERROR on failure.                  */
/*                                                                       */
/*************************************************************************/
int aic27_close()
{
    // If codec not opened, then just return
    if (cdx.state == AIC27_CODEC_CLOSED)
           return BRD_OK;

    // Disable codec interrupt
       disable_codec_comm();

    cdx.state = AIC27_CODEC_CLOSED;

    return BRD_OK;
}

static void disable_codec_comm()
{
          rx_stop = tx_stop = True;

          dma_reset(AIC27TXCH); //reset Tx DMA ch
          dma_reset(AIC27RXCH); //reset Rx DMA ch

          _disable_aic27_interrupt();
          _reset_aic27_pkts();
          _serial_reset();

          cdx.state = AIC27_CODEC_OPENED;
}
```

```
/*************************************************************************/
/* dma_reset –  Reset DMA ch.                                            */
/*                                                                       */
/*  This function resets the specified DMA ch by initializing            */
/*  ch control registers to their default values                         */
/*                                                                       */
/*************************************************************************/
void dma_reset(int ch)
{
        DMCCR(ch)   = 0;

        DMCSDP(ch)  = 0;
        DMCICR(ch)  = 0;

        DMCSSAL(ch) = 0;
        DMCSSAU(ch) = 0;
        DMCDSAL(ch) = 0;
        DMCDSAU(ch) = 0;

        DMCEN(ch)   = 0;
        DMCFN(ch)   = 0;
        DMCEI(ch)   = 0;
        DMCFI(ch)   = 0;
}

 static void _disable_aic27_interrupt()
{
   // Disable interrupt
   IER0 &= (u16)(~(1<<AIC27_RXDMA_FLAG));

   // Unhook interrupt handler
   unhook_interrupt(AIC27_RXDMA_TRAP);
}

/*************************************************************************/
/* Aic27ExLb –  CODEC Example: Loopback mode                             */
/*                                                                       */
/*   This function provides a loopback feature.                          */
/*                                                                       */
/*************************************************************************/
static s16 Aic27ExLb(s16 micSel)
{
    int *bufStatus1;
    int *bufStatus2;
    u16 cnt;

      CPLD_CTRL2_REG &= ~1;
```

```
        brd_led_disable(BRD_LED0);
        brd_led_disable(BRD_LED1);
        brd_led_disable(BRD_LED2);

    aic27_open();

    //select stereo output
    CPLD_CTRL2_REG &= ~1;

    // set Front_mixer and Master volume
    aic27_write_reg(MASTER_VOL, 0x0202, 0xFFFF);
    aic27_write_reg(FRONT_MIXER_VOL, 0x0000, 0xFFFF);

    //set sample rate
    aic27_sample_rate(AUDIO_ADC_RRATE, LOOP_SAMPLE_RATE);
    aic27_sample_rate(FRONT_DAC_SRATE, LOOP_SAMPLE_RATE);

      if (micSel)
      {
            aic27_write_reg(REC_SEL, MIC_IN, 0xFFFF); //mic_in
      }
      else
      {
            aic27_write_reg(REC_SEL, LINE_IN, 0xFFFF); //line_in
      }

    // clear buffers
    for (cnt=0; cnt<BUF_SIZE; cnt++)
    {
        toneBuf[0][cnt] = 0;
        toneBuf[1][cnt] = 0;
    }

    // enable simultaneous capture and playback
    aic27_cont_capture(toneBuf[0],toneBuf[1],BUF_SIZE,0,&bufStatus1);
    aic27_cont_play   (toneBuf[1],toneBuf[0],BUF_SIZE,0,&bufStatus2);

      // close codec
      aic27_close();

      return (BRD_OK);
}
```

```
/*************************************************************************/
/* blink_leds(u32 cnt) : Toggles all the user leds for a duration of cnt.   */
/*                                                                         */
/*************************************************************************/
void blink_leds(u32 cnt)
{
   while (cnt)
          {
                   brd_led_toggle(BRD_LED0);
                   busy_delay(DELAY_TIME);
                   brd_led_toggle(BRD_LED1);
                   busy_delay(DELAY_TIME);
                   brd_led_toggle(BRD_LED2);
                   busy_delay(DELAY_TIME);

                   if (cnt != FOREVER) cnt--;
          }
}


/*************************************************************************/
/* s16 brd_toggle_led(BrdLed led)                                        */
/*                                                                       */
/* This routine toggles the specified user-defined LED.                  */
/*                                                                       */
/* Parameters:                                                           */
/*    – led – indicates which led to toggle                              */
/*                                                                       */
/* Return:                                                               */
/*                 BRD_OK on success                                     */
/*                 BRD_ERROR on failure                                  */
/*                                                                       */
/* Notes:                                                                */
/*                                                                       */
/*************************************************************************/
s16 brd_led_toggle(BrdLed led)
{
  int ledMask;

  switch (led)
  {
  case BRD_LED0:
    ledMask = LED0_MASK;
    break;

  case BRD_LED1:
    ledMask = LED1_MASK;
    break;
```

```
    case BRD_LED2:
      ledMask = LED2_MASK;
      break;

    default:
      return BRD_ERROR;
    }

    ( CPLD_CTRL1_REG & ledMask ) ? brd_led_disable(led) : brd_led_enable(led);

    return (BRD_OK);
  }
```

Appendix B Implementing the AC97 Standard With Dual-Phase Frame

- (R/X)PHASE = 1: Dual-phase frame
- (R/X)FRLEN1 = 0000000b: 1 word in phase 1
- (R/X)WDLEN1 = 010b: 16 bits per word in phase 1
- (R/X)FRLEN2 = 0001011b: 12 words in phase 2
- (R/X)WDLEN2 = 011b: 20 bits per word in phase 2

CLKRP/CLKXP= 0: Receive data sampled on falling edge of internal CLKR / transmit data clocked on rising edge of internal CLKX

- FSRP/FSXP = 0: Active-high frame-sync signal
- (R/X)DATDLY = 01b: Data delay of 1 clock cycle (1-bit data delay)

```
//McBsp setting for AC’97 compliant codec in dual mode
#define WD_PER_FRAME_1   0x0
#define WD_PER_FRAME_4   0x3

#define RCR1_VAL   ((WD_PER_FRAME_1 << 8) | (WORD_LENGTH_16 << 5))

#define RCR2_VAL   ((DUAL_PHASE        << 15) | (WD_PER_FRAME_4     << 8)  |\
                    (WORD_LENGTH_20    << 5)  | (NO_COMPAND_MSB_1ST << 3)  |\
                    (NO_FRAME_IGNORE   << 2)  | (DATA_DELAY1        << 0))

#define XCR1_VAL   ((WD_PER_FRAME_1 << 8) | (WORD_LENGTH_16 << 5))

#define XCR2_VAL   ((DUAL_PHASE        << 15) | (WD_PER_FRAME_4     << 8)  |\
                    (WORD_LENGTH_20    << 5)  | (NO_COMPAND_MSB_1ST << 3)  |\
                    (NO_FRAME_IGNORE   << 2)  | (DATA_DELAY1        << 0))

#define PCR_VAL    ((FSYNC_POL_HIGH<<3)|(FSYNC_POL_HIGH<<2)|
                    (CLKX_POL_RISING<<1)| (CLKR_POL_FALLING))

#define SPCR1_VAL          0
#define SPCR2_VAL          0
#define SRGR1_VAL          0
#define SRGR2_VAL          0

#define MCBSP_RX  1
#define MCBSP_TX  2
#define MCBSP_BOTH 3

//Serial Port Control Register SPCR1
#define DLB_ENABLE         0x01   //Enable Digital Loop back Mode
#define DLB_DISABLE        0x00   //Disable Digital Loop back Mode

#define RXJUST_RJZF        0x00   //Receive Right Justify Zero Fill
#define RXJUST_RJSE        0x01   //Receive Right Justify Sign Extend
#define RXJUST_LJZF        0x02   //Receive Left Justify Zero Fill
```

```
#define CLK_STOP_DISABLED  0x00   //Normal clocking for non-SPI mode
#define CLK_START_WO_DELAY 0x02   //Clock starts without delay
#define CLK_START_W_DELAY  0x03   //Clock starts with delay

#define DX_ENABLE_OFF      0x00   //no extra delay for turn-on time
#define DX_ENABLE_ON       0x01   //enable extra delay for turn-on time

#define ABIS_DISABLE       0x00   //A-bis mode is disabled
#define ABIS_ENABLE        0x01   //A-bis mode is enabled

//Serial Port Control Registers SPCR1 and SPCR2
#define INTM_RDY           0x00   //R/X INT driven by R/X RDY
#define INTM_BLOCK         0x01   //R/X INT driven by new multi–channel block
#define INTM_FRAME         0x02   //R/X INT driven by new frame sync
#define INTM_SYNCERR       0x03   //R/X INT generated by R/X SYNCERR

#define RX_RESET           0x00   //R or X in reset
#define RX_ENABLE          0x01   //R or X enabled

//Serial Port Control Register SPCR2
#define SP_FREE_OFF        0x00   //Free running mode is disabled
#define SP_FREE_ON         0x01   //Free running mode is enabled

#define SOFT_DISABLE       0x00   //SOFT mode is disabled
#define SOFT_ENABLE        0x01   //SOFT mode is enabled

#define FRAME_GEN_RESET    0x00   //Frame Synchronization logic is reset
#define FRAME_GEN_ENABLE   0x01   //Frame sync signal FSG is generated

#define SRG_RESET          0x00   //Sample Rate Generator is reset
#define SRG_ENABLE         0x01   //Sample Rate Generator is enabled

//Pin Control Register PCR
#define IO_DISABLE         0x00   //No General Purpose I/O Mode
#define IO_ENABLE          0x01   //General Purpose I/0 Mode enabled

#define CLKR_POL_RISING    0x01   //R Data Sampled on Rising Edge of CLKR
#define CLKR_POL_FALLING   0x00   //R Data Sampled on Falling Edge of CLKR
#define CLKX_POL_RISING    0x00   //X Data Sent on Rising Edge of CLKX
#define CLKX_POL_FALLING   0x01   //X Data Sent on Falling Edge of CLKX
#define FSYNC_POL_HIGH     0x00   //Frame Sync Pulse Active High
#define FSYNC_POL_LOW      0x01   //Frame Sync Pulse Active Low

#define CLK_MODE_EXT       0x00   //Clock derived from external source
#define CLK_MODE_INT       0x01   //Clock derived from internal source

#define FSYNC_MODE_EXT     0x00   //Frame Sync derived from external source
#define FSYNC_MODE_INT     0x01   //Frame Sync derived from internal source

#define IDLE_ENABLE        0x01   //Stop all McBsp clocks when in IDLE mode
#define IDLE_DISABLE       0x00   //Do not stop all McBsp clocks when in IDLE mode
```

```
//Transmit Receive Control Register XCR/RCR
#define SINGLE_PHASE          0x00   //Selects single phase frames
#define DUAL_PHASE            0x01   //Selects dual phase frames
#define MAX_FRAME_LENGTH      0x7f   //maximum number of words per frame

#define WORD_LENGTH_8         0x00   //8 bit word length (requires filling)
#define WORD_LENGTH_12        0x01   //12 bit word length       ””
#define WORD_LENGTH_16        0x02   //16 bit word length       ””
#define WORD_LENGTH_20        0x03   //20 bit word length       ””
#define WORD_LENGTH_24        0x04   //24 bit word length       ””
#define WORD_LENGTH_32        0x05   //32 bit word length (matches DRR DXR sz

#define MAX_WORD_LENGTH       0x20   //maximum number of bits per word

#define NO_COMPAND_MSB_1ST    0x00   //No Companding, Data XFER starts w/MSb
#define NO_COMPAND_LSB_1ST    0x01   //No Companding, Data XFER starts w/LSb
#define COMPAND_ULAW          0x02   //Compand ULAW, 8 bit word length only
#define COMPAND_ALAW          0x03   //Compand ALAW, 8 bit word length only

#define FRAME_IGNORE          0x01   //Ignore frame sync pulses after 1st
#define NO_FRAME_IGNORE       0x00   //Utilize frame sync pulses

#define DATA_DELAY0           0x00   //1st bit in same clk period as fsync
#define DATA_DELAY1           0x01   //1st bit 1 clk period after fsync
#define DATA_DELAY2           0x02   //1st bit 2 clk periods after fsync

//Sample Rate Generator Register SRGR

//Clock mode (ext. / int.) see PCR
#define MAX_SRG_CLK_DIV        0xff   //max value to divide Sample Rate Gen Cl
#define MAX_FRAME_WIDTH        0xff   //maximum FSG width in CLKG periods
#define MAX_FRAME_PERIOD       0x0fff //FSG period in CLKG periods

#define FSX_DXR_TO_XSR         0x00   //Transmit FSX due to DXR to XSR copy
#define FSX_FSG                0x01   //Transmit FSX due to FSG

#define CLKS_POL_FALLING       0x00   //falling edge generates CLKG and FSG
#define CLKS_POL_RISING        0x01   //rising edge generates CLKG and FSG

#define GSYNC_OFF              0x00   //CLKG always running
#define GSYNC_ON               0x01   //CLKG and FSG synch’ed to FSR

//Multi-channel Control Register 1 and 2 MCR1/2
#define RMCM_CHANNEL_ENABLE    0x00   //all 128 channels enabled
#define RMCM_CHANNEL_DISABLE   0x01   //all channels disabled, selected by
                                      //enabling RP(A–H)BLK, RCER(A–H)
```

```
#define XMCM_CHANNEL_DX_DRIVEN 0x00   //transmit data over DX pin for as many
                                      //number of words as required
#define XMCM_XCER_CHAN_TO_DXR  0x01   //selected channels written to DXR
#define XMCM_ALL_WORDS_TO_DXR  0x02   //all words copied to DXR(1/2),
                                      //DX only driven for selected words
#define XMCM_CHANNEL_SYM_R/X   0x03   //symmetric transmit and receive
                                      //operation

#define MCM_128CHANNEL_DISABLE 0x01   //Normal mode 32 channels enabled
#define MCM_128CHANNEL_ENABLE  0x01   //Enable all 128 channels
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI 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 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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265