TEXAS INSTRUMENTS

Application Report
SPRA785 - November 2001

Programming the TMS320VC5509 I2C Peripheral

Rishi Bhatacharya *C5000 DSP Software Applications*

ABSTRACT

This application note demonstrates the procedure used to program the TMS320VC5509 I2C peripheral module. Basic operations of the I2C, including reading and writing, and the initialization of the I2C bus are covered. These operations are illustrated using the I2C routines provided in the chip support library (CSL) (See the *TMS320C55x Chip Support Library API Reference Guide* (SPRU433A)). Two examples are provided to further demonstrate the operation of the I2C peripheral: writing and reading from an I2C EEPROM and controlling an I2C codec.

Contents

1 Introduction **2**
1.1 TMS320VC5509 I2C Features 3
2 Read and Write Operations **3**
3 EEPROM Read/Write Example **8**
3.1 Code Description 10
4 AIC23 Codec Volume Control Example **11**
4.1 Example Description 14
Appendix A CAT24WC256 Serial EEPROM **18**
A.1 Device Addressing 18
A.2 Acknowledge 18
A.3 Writing to the EEPROM (page write) 19
A.4 Reading From the EEPROM (sequential read) 19
Appendix B TLV320AIC23 Stereo Codec **21**
Appendix C Sine Wave Value Calculator **23**
C.1 Sample Value Origination Description 23

List of Figures

Figure 1. Multiple C55x I2C Connections 2
Figure 2. I2C_Init 4
Figure 3. I2C_write 5
Figure 4. I2C_read 7
Figure 5. EEPROM Read/Write Example Code 9
Figure 6. AIC23 Codec Control Example Code 12
Figure 7. AIC23 Codec Control Example Code (Continued) 13
Figure A–1. Acknowledge Timing 18

Trademarks are the property of their respective owners.

Figure A–2. Slave Address Bits 18
Figure A–3. Page Write Timing 19
Figure A–4. Selective Read Timing 20
Figure B–1. 2-Wire I2C Compatible Timing 22

List of Tables

Table B–1. Slave Address Selection 21
Table B–2. AIC23 Control Register Addresses 22

1 Introduction

In modern electronic systems there are peripheral I2Cs that must communicate with both each other and the outside world. To maximize hardware efficiency and simplify circuit design, Philips developed a simple bi-directional 2-wire, serial data (SDA) and serial clock (SCL) bus for inter-I2C control. This I2C-bus supports any I2C fabrication process and, with the extremely broad range of I2C-compatible chips from Philips and other suppliers, it is the worldwide industry standard proprietary control bus.

Each device is recognized by a unique address and can operate as either a receiver-only device (e.g. an LCD driver), or a transmitter with the capability to both receive and send information (such as memory). Transmitters and/or receivers can operate in either master or slave mode, depending on whether the chip must initiate a data transfer, or is only addressed. I2C is a multi-master bus, i.e. it can be controlled by more than one I2C connected to it.

The basic I2C-bus, with a data transfer rate up to 100 kbits/s and 7-bit addressing, was introduced nearly 20 years ago. But, as data transfer rates and application functionality rapidly increased, the I2C-bus specification was enhanced to include fast-mode and 10-bit addressing, thus meeting the demand for higher speeds and more address space.

The catalog DSP multi-master I2C peripheral provides an interface between a TI DSP device and I2C-bus compatible devices that connect via the I2C serial bus. External components attached to the I2C bus can serially transmit/receive up to 8-bit data to/from the TI DSP device through the two-wire I2C interface.

The catalog DSP I2C peripheral supports any slave or master I2C compatible device. Figure 1 shows the example of these pins for multiple C55x™ I2C serial ports connected for a two-way transfer from one device to other devices.

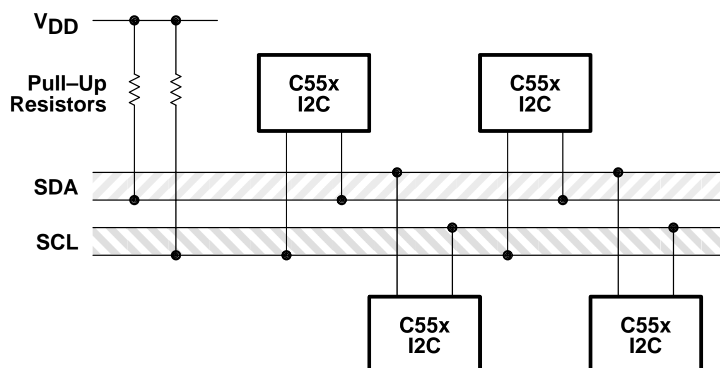


Figure 1. Multiple C55x I2C Connections

C55x is a trademark of Texas Instruments.

1.1 TMS320VC5509 I2C Features

The catalog DSP I2C contains the following features:

- Compliance with Philips' I2C-bus specification v2.1
- Bit/Byte format transfer
- 7-bit and 10-bit device addressing modes
- General call
- Start byte
- Free data format
- Multi-master transmitter/slave receiver mode
- Multi-master receiver/slave transmitter mode
- Combined master transmit/receive and receive/transmit mode
- I2C data transfer rate of from 10kbps up to 400kbps (Philips I2C rate)
- Has one read and one write DMA event that can be used by the DMA
- Has one read/write and one 'illegal operation' interrupt that can be used by the CPU
- Module enable/disable capability

Please refer to *TMS320C55x DSP Peripherals Reference Guide* (SPRU317B) for more information.

2 Read and Write Operations

The basic operations of the I2C peripheral are the same as any serial interface: reading and writing. The I2C bus initialization, the read, and write operations are illustrated by an explanation of the implementation of the corresponding CSL function I2C_init(), I2C_write(), and I2C_read(). Refer to *TMS320C55x Chip Support Library API Reference Guide* (SPRU433A) for a complete description of these functions. Source code is provided as part of CCS.

Prior to performing reading or writing operations, it is necessary to call the I2C_init() function to configure the I2C bus appropriately. After the I2C bus has been configured, the I2C_write and I2C_read CSL functions can be used to transfer data.

- **I2C_init():** Figure 2 shows a block diagram of this function. This function takes a structure that contains various parameters as input and configures the I2C. These parameters include the address mode, the own address, system clock, desired transfer rate, bits per byte to be received or transmitted, data loopback mode, and the free mode.

Calling Convention: I2C_Init

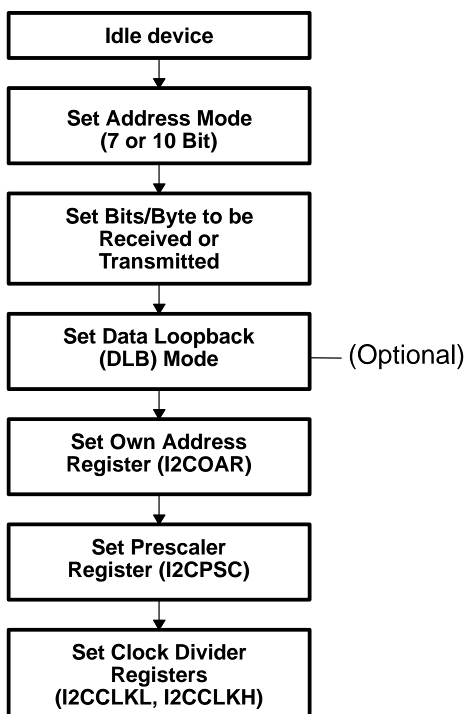


Figure 2. I2C_Init

NOTE: The Set Data Loopback (DLB) mode is optional.

Flow Graph Description:

Initially, the function idles the device and configures all of the parameters in the I2CMDR register with the values passed in the input structure. These include the address mode (XA), the bits/byte (BC), data loopback mode (DLB), and the free mode (FREE).

Next, based on the transfer rate and system clock parameters, the function sets its own address register (I2COAR) with the value given and configures the prescaler register (I2CPSC) and the clock divider registers (I2CCLKL, I2CCLKH) .

- **I2C_write():** Figure 3 illustrates a block diagram of I2C_write. This function can be used to perform the master transmitter and slave transmitter I2C modes. It takes, as input, a pointer to the data which will be sent, the number of bytes to be transmitted, the master mode (on/off), the slave address, the transfer mode, and the timeout for various errors.

Calling Convention: int I2C_write (Uint16 *data, int length, int master, Uint16 slaveaddress, int transfermode, int timeout);

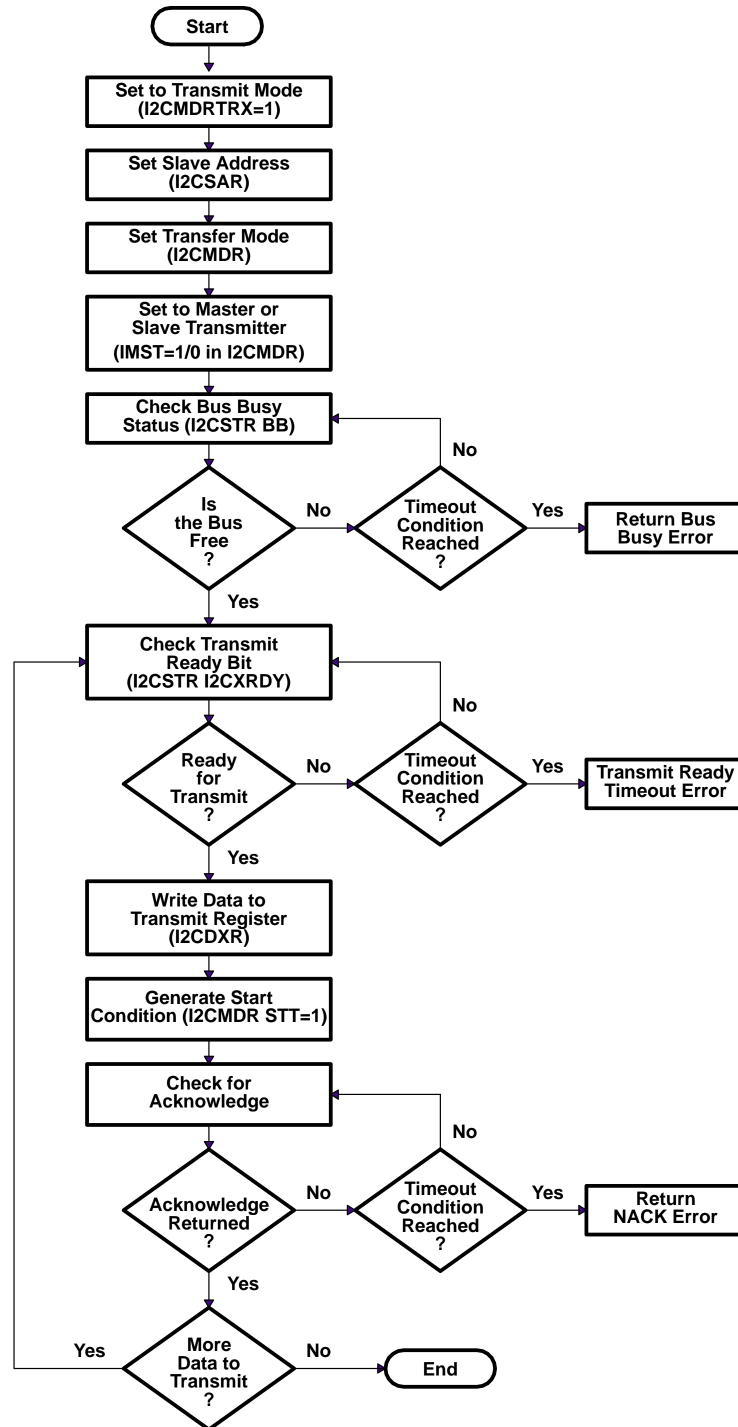


Figure 3. I2C_write

Flow Graph Description:

This process begins by disabling all interrupts (safety mechanisms in a multitasking environment) and proceeds to configure the I2C by setting the transmit mode (TRX = 1), transfer mode (S-A-D-P*, S-A-D-D*, etc.), and the master mode (MST) in the I2CMDR register.

After setting the slave address in the I2CSAR register, the function polls the bus busy (BB) bit in the I2CSTR register. If the bus is busy, the function continues to poll until either the bus becomes free, or the timeout value is reached. If the timeout value is reached, an error flag is returned.

If the bus is free, it polls the transmit ready (I2CXRDY) bit in the I2CSTR register to check for a transmit ready condition. As in the case of the bus busy polling, the bus will continue to poll until either it becomes ready or it times out, in which case it will return a different error flag. If the transmit ready condition is set, the function writes the data byte to be transmitted into the I2CDXR register and generates a start condition (sets STT to 1).

I2C_write() checks for an acknowledge by polling the NACK bit in the I2CTR register. If no-acknowledge is returned (NACK =1), the function returns an error flag. If an acknowledge is received and there is more data to transmit, the function loops back to the check for transmit ready section and continues to do this until all data has been transmitted. If there is no more data to transmit, the function restores all interrupts and ends.

NOTE: In the example provided, an inefficient method of polling the I2CXRDY bit is used. Transmission can also be implemented via DMA or interrupt methods.

- **I2C_read():** Figure 4 illustrates a block diagram of this function. This function is used to perform the master receiver and slave receiver I2C modes. It takes, as input, a pointer to where the received data will be stored, the number of bytes to be received, the master mode (on/off), the slave address, the transfer mode, the timeout for various errors, and a flag to tell the function to check for bus busy (more details about this will be found later).

* S = START; A = Address; D = Data; P = STOP

Calling Convention: int I2C_read (Uint16 *data, int length, int master, Uint16 slaveaddress, int transfermode, int timeout, int checkbus);

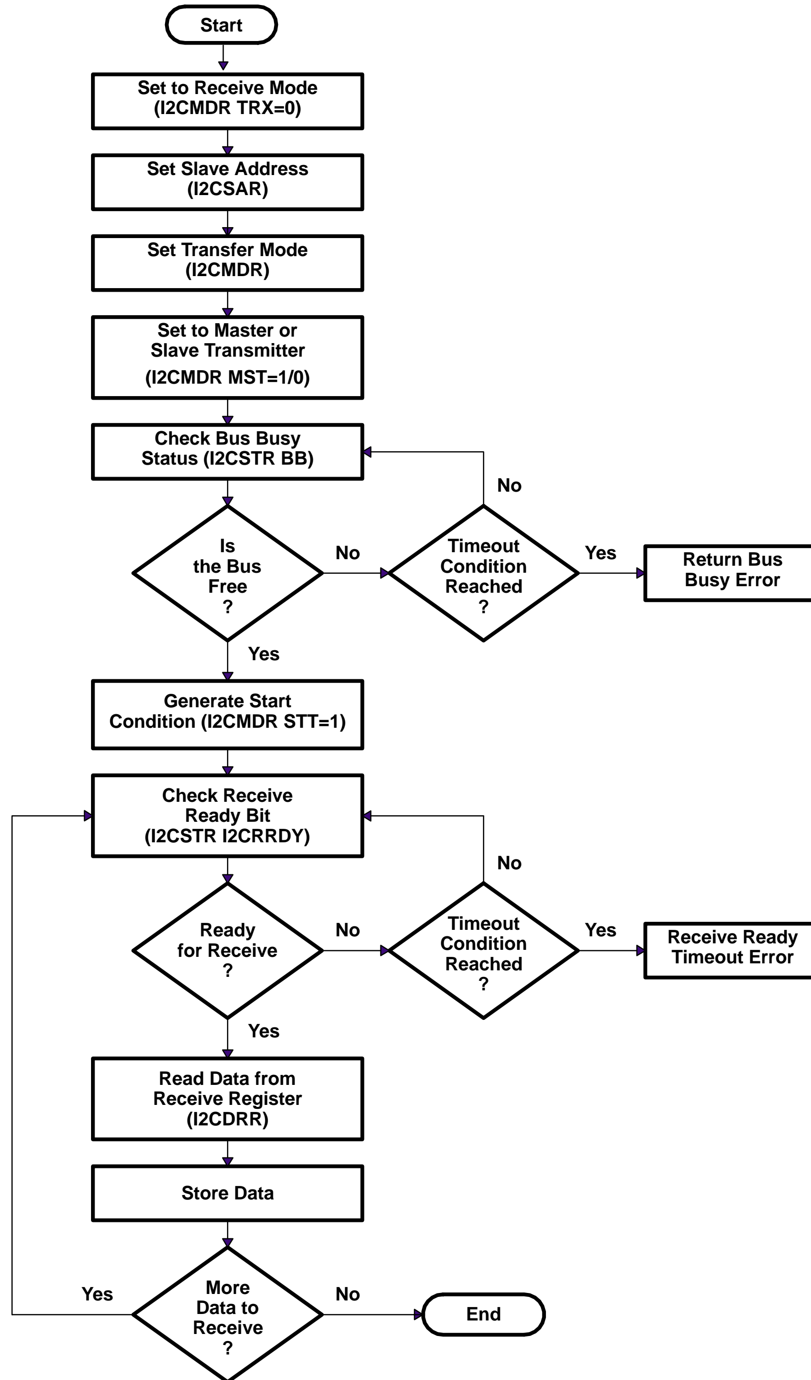


Figure 4. I2C_read

Flow Graph Description:

I2C_read begins this process first by disabling all interrupts (safety mechanism in a multitasking environment), then configuring the I2C by setting the receive mode (TRX = 0), transfer mode (S-A-D-P, S-A-D-D, etc.), and the master mode (MST) in the I2CMDR register. After setting the slave address in the I2CSAR register, the function polls the bus busy (BB) bit in the I2CSTR register. If the bus is busy, the function continues to poll until either the bus becomes free, or the timeout value is reached. If the timeout value is reached, an error flag is returned.

If the bus is free, it generates a start condition (sets STT = 1) and begins polling the receive ready (I2CRRDY) bit in the I2CSTR register to check for a receive ready condition. As in the case of the bus busy polling, it continues to poll until either it becomes ready or it times out, in which case it will return a different error flag. If the receive ready condition is set, the function reads the data byte from the I2CDRR register, stores it into the specified memory location, and increments the memory pointer by one.

If there is more data to receive, the function loops back to the check for receive ready section and continues to do this until all data has been stored. If there is no more data to receive, the function restores all interrupts and ends.

3 EEPROM Read/Write Example

In the example detailed in Figure 5, you will write 5 bytes of data (0xA, 0xB, 0xC, 0xD, 0xE) to the EEPROM and read them back. This example can be run in the Spectrum Digital 5509 EVM.

Appendix A provides a technical overview of the EEPROM.

```
#include <csl.h>
#include <csl_I2C.h>
#include <stdio.h>

int x=1,y=1,z=1;
Uint16 slaveaddressreceive[2]={0,0};
Uint16 databyte[7]={0,0,10,11,12,13,14};
Uint16 datareceive[6]={0,0,0,0,0,0};

I2C_Init Init = {
      0,          /* 7 or 10 bit address mode                          */
      0x0000,     /* own address - don't care if master                */
      144,        /* clkout value (Mhz)                                */
      400,        /* a number between 10 and 400kbs                    */
      0,          /* number of bits/byte to be received or transmitted */
      0,          /* DLB mode                                          */
      1           /* FREE mode of operation                            */
};


main()
{

CSL_init();
I2C_init(&Init);  /* initialize the I2C using the Init structure */

/* ---------- WRITE ---------- */

x=I2C_write(databyte,7,1,0x50,1,30000);   /* write five bytes of data in */
                                          /* master transmitter mode to  */
                                          /* the slave address 0x50      */

/* ---------- READ ---------- */

y=I2C_write(slaveaddressreceive,2,1,0x50,2,30000);

z=I2C_read(datareceive,5,1,0x50,3,30000,0);

I2C_stop(); /* generate stop condition */
```


Figure 5. EEPROM Read/Write Example Code

3.1 Code Description

Figure 5 illustrates an example where 5 bytes of data (0xA, 0xB, 0xC, 0xD, 0xE) is written to the EEPROM and then read back, to verify that the correct data has been written.

- **Step 0:** Data Initialization

It is necessary to set up three arrays that will be used during the send and receive modes from the EEPROM.

The first array, databyte, contains both the address (on the EEPROM) to which the data will be written, and the actual data. This array is used only in the writing phase of the example. The first two elements {0, 0} are the two address bytes shown as A15-A0 in Figure A–3 (byte address). During this process, memory will be written to location 0. The next five bytes are data that is written to the EEPROM (0xA, 0xB, 0xC, 0xD, 0xE).

```
Uint16 databyte[7]={0,0,10,11,12,13,14};
```

The next two arrays are used during the receiving phase of the example. The slaveaddressreceive array contains two elements which indicate the memory location on the EEPROM from which the data is read. Since memory location 0 was specified during the written phase, the same address is specified, allowing the written data to be read.

```
Uint16 slaveaddressreceive[2]={0,0};
```

Next, initialize array datareceive to store the data received, while reading the data from EEPROM.

```
Uint16 datareceive[6]={0,0,0,0,0,0};
```

- **Step 1:** I2C Bus Initialization

Call the initialization function with the Init structure as an argument to initialize the I2C device.

```
I2C_init(&Init);   /* configure the I2C bus*/
```

Begin by initializing various parameters that will be used during the transfer. The initialization structure that is passed to the I2C_init function to initialize the device has the following values:

```
I2C_Init Init = {
      0,          /* 7  bit address mode                               */
      0x0000,     /* own address                                       */
      144,        /* system clock value (Mhz)                          */
      400,        /* rate – a number between 10 and 400kbs             */
      0,          /* number of bits/byte to be received or transmitted (8) */
      0,          /* DLB mode off                                      */
      1           /* FREE mode on                                      */
};
```

- **Step 2:** Write Operation

To write the 5 bytes of data to the EEPROM, call the I2C_send function with the databyte array, length = 7 (2 address bytes plus 5 bytes of data), master mode on, slave address 0x50, transfer mode = S-A-D-P (see I2CMDR RM field description), and timeout = 30000. Obtain the slave address of 0x50 by viewing Figure A–2 – on your particular board. In the board used for this report, both EEPROM address pins are left unconnected, meaning that the A0 and A1 bits should be 0. Ignoring the last R/W bit (which is taken care of by the TRX field in the I2CMDR register, you have 1010000, or 0x50. S-A-D-P mode is used to conform to the requirement of the stop bit (P) generation shown in Figure A–3, Appendix A.

```
x=I2C_write(databyte,7,1,0x50,1,30000);
```

- **Step 3:** Read Operation

To read the 5 data bytes back, first perform a 'dummy' write operation (as shown in Figure A–4) by sending the two address bytes (line address in memory to be read) to the same slave address in S-A-D..(n)..D mode (see I2CMDR RM field description). notice that this mode is selected because no stop bit (P) is required.

```
y=I2C_write(slaveaddressreceive,2,1,0x50,2,30000);
```

Next, read the 5 bytes of data that is written by using the I2C_receive function, which is called with the datareceive array, length = 5, master mode on, slave address 0x50, transfer mode = S-A-D-D-D…. (repeat mode on), timeout = 30000, and check for bus busy flag off. The check for bus busy flag must be off, due to the the master mode having control of the bus after sending the two address bytes, and the function would timeout if it were to check for bus status. This function begins by generating the second start condition shown in Figure A–4 after which the EEPROM sends back a byte of data and continues to do so as long as the master responds with an acknowledge (the NOACK in figure Figure A–4 should be replaced with ACK if more data is desired).

```
z=I2C_read(datareceive,5,1,0x50,3,30000,0);
```

After the 5 bytes of data have been received, manually generate a stop condition by calling I2C_sendStop() CSL function. This ends the S-A-D-D-D mode transmission.

4 AIC23 Codec Volume Control Example

Figure 6 is an example of the AIC23 codec being set up using the I2C interface to configure the AIC23 control registers. A tone is sounded and the volume is varied to test the proper operation of the control interface. This example can be run in the Spectrum Digital 5509 EVM.

Appendix B provides a technical overview of the AIC23.

Appendix C illustrates how the tone (sine wave values) were calculated.

```
#include "csl.h"        <-- Include CSL header file.
#include <csl_I2C.h>
#include <stdio.h>
#include <csl_mbsp.h>

statI2C int SineSamples[]={0x00000, 0x018f8, 0x030fb, 0x0471c, 0x05a82, 0x06a6d,
                         0x07641, 0x07d8a, 0x07fff, 0x07d8a, 0x07641, 0x06a6d,
                         0x05a82, 0x0471c, 0x030fb, 0x018f8, 0x00000, 0x0e709,
                         0x0cf06, 0x0b8e5, 0x0a57f, 0x09594, 0x089c0, 0x08277,
                         0x08002, 0x08277, 0x089c0, 0x09594, 0x0a57f, 0x0b8e5,
                         0x0cf06, 0xe0709/*, 0x00000*/};
#define SINEBUFFSIZE 32
#define LCD_PORT_NUMBER 0
int j=5,i=1;
Uint32 xmt;
Uint16 temp=1;
Uint16 reset[2] = {0x1E,0x00};
Uint16 power_down_control[2] = {0x0C,0x07};
Uint16 analog_audio_path_control[2] = {0x08,0x10};
Uint16 digital_audio_path_control[2] = {0x0A,0x01};
Uint16 digital_audio_interface_format[2] = {0x0E,0x43};
Uint16 sample_rate_control[2] = {0x10,0x22};
Uint16 digital_interface_activation[2] = {0x12,0x01};
Uint16 left_line_input_volume_control[2] = {0x01,0x17};
Uint16 right_line_input_volume_control[2] = {0x03,0x17};
Uint16 left_headphone_volume_control[2] = {0x05,0xFF};
Uint16 right_headphone_volume_control[2] = {0x07,0xFF};
I2C_Init Init = {
      0,          /* 7 bit address mode                                     */
      0x0000,     /* own address                                            */
      144,        /* clkout value (Mhz)                                     */
      400,        /* rate - a number between 10 and 400kbs                  */
      0,          /* number of bits/byte to be received or transmitted (8) */
      0,          /* DLB mode off                                           */
      1           /* FREE mode on                                           */
};

MCBSP_Config configAIC = {
            ...
      };

main()
{
```

Callouts: "Include CSL header file." — "Initialize array for each of the 11 control registers with register address concatenated with register value." — "Define initialization structure."

Figure 6. AIC23 Codec Control Example Code

```
int ii, jj;

/* initialize the I2C using the Init structure */
    I2C_init(&Init);
/* configure reset register */
       j=I2C_send(reset,2,1,0x1A,1,30000);
/* configure power down control register */
       j=I2C_send(power_down_control,2,1,0x1A,1,30000);
/* configure analog audio path register */
       j=I2C_send(analog_audio_path_control,2,1,0x1A,1,30000);
/* configure digital audio path register */
       j=I2C_send(digital_audio_path_control,2,1,0x1A,1,30000);
/* configure digital audio interface register */
       j=I2C_send(digital_audio_interface_format,2,1,0x1A,1,30000);
/* configure sample rate control register */
       j=I2C_send(sample_rate_control,2,1,0x1A,1,30000);
/* configure digital interface activation register */
       j=I2C_send(digital_interface_activation,2,1,0x1A,1,30000);
/* configure left line input register */
       j=I2C_send(left_line_input_volume_control,2,1,0x1A,1,30000);
/* configure right line input register */
       j=I2C_send(right_line_input_volume_control,2,1,0x1A,1,30000);
/* configure left headphone register */
       j=I2C_send(left_headphone_volume_control,2,1,0x1A,1,30000);
/* configure right headphone register */
       j=I2C_send(right_headphone_volume_control,2,1,0x1A,1,30000);

/* BEGIN CODE TO TRANSFER DATA USING MCBSP */

       mhMcbsp = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
       MCBSP_config(mhMcbsp,&ConfigAI2C);  /* configure MCBSP */
       MCBSP_start (mhMcbsp, MCBSP_RCV_START | MCBSP_SRGR_START |
                             MCBSP_SRGR_FRAMESYNC, 0x3000);

do {
   for(ii=0; ii<18; ii++)  /* 18*32 samples per period = 576 samples */
   {
      for ( jj=0; jj<SINEBUFFSIZE; jj++)  /* SINEBUFFSIZE = 8 */
      {
          while (!MCBSP_xrdy(mhMcbsp));
          xmt = (unsigned int) SineSamples[jj];
           /* Now write the data to the transmit data register */
          MCBSP_write32(mhMcbsp,xmt);

       }
   }
   } while (1)

}
```


Figure 7. AIC23 Codec Control Example Code (Continued)

4.1 Example Description

In this example the AIC23 codec is set up using the I2C interface to configure the control registers. You will then play a tone and vary the volume to test the proper operation of the control interface.

Step 1: AIC23 Control register value definition

It is necessary to set up the eleven arrays that are used to configure the eleven registers found on the AIC23 codec.

In the first array, configure the reset register to take the device out of reset. Referring to figure Figure B–1 notice that the address of the register and the values of the fields it contains must be concatenated together into two bytes of data. Next, the register address and field values are sent in S-A-D-P mode (requires a stop bit) to the codec. If the register address is 0001111, then the first seven bits of the first byte sent out must be 0001111.

The registers on the codec have nine fields; therefore, the LSB of the first byte sent counts as the value for field number nine, and the second byte counts as the value for fields 1–8. Taking notice of the reset register description, writing all 1's to the register will place it in reset mode.

Therefore, in order to take the device out of reset mode, you must write zero values to the device's nine fields. For simplicity, write all 0's to the register. With the address being the first seven bits, you have: 0001111000000000, or 0x1E and 0x00.

```
Uint16 reset[2] = {0x1E,0x00};
```

The second array is set up in a similar manner. The power down control register has an address of 0000110. Field nine in the power down control register is reserved; therefore, place a 0 in that field and turn the power on (field 8 = 0), clock on, oscillator on, outputs on, DAC on, ADC off, microphone input off, and line input off. Again, with the address being the first seven bits of the two bytes to be transferred, you have: 0000110000000111, or 0x0C and 0x07.

```
Uint16 power_down_control[2] = {0x0C,0x07};
```

Here are the other nine register field values:

Analog Audio Path Control:

Sidetone enable off, sidetone attenuation 6db, dac select on, bypass off, input select adc line, microphone mute off, microphone boost 0db.

```
Uint16 analog_audio_path_control[2] = {0x08,0x10};
```

Digital Audio Path Control:

DAC soft mute off, deemphasis disabled, adc high pass filter on.

```
Uint16 digital_audio_path_control[2] = {0x0A,0x01};
```

Digital Audio Interface Format:

Master mode, input length 16 bits.

```
Uint16 digital_audio_interface_format[2] = {0x0E,0x43};
```

Sample Rate Control:

Sample rate 44.1Khz, clock mode master.

```
Uint16 sample_rate_control[2] = {0x10,0x22};
```

Digital Interface Activation:

Active Interface.

```
Uint16 digital_interface_activation[2] = {0x12,0x01};
```

Left Line Input Volume Control:

Simultaneous volume mute on, left line input mute off, left line input volume.

```
Uint16 left_line_input_volume_control[2] = {0x01,0x17};
```

Right Line Input Volume Control:

Simultaneous volume mute on, right line input mute off, right line input volume.

```
Uint16 right_line_input_volume_control[2] = {0x03,0x17};
```

Left Headphone Volume Control:

Left-right headphone simultaneous volume mute on, left zero crossing detect on, left headphone volume.

```
Uint16 left_headphone_volume_control[2] = {0x05,0xFF};
```

Right Headphone Volume Control:

Right-left headphone simultaneous volume mute on, right zero crossing detect on, right headphone volume.

```
Uint16 right_headphone_volume_control[2] = {0x07,0xFF};
```

- **Step 2:** I2C Initialization

Begin the next step by initializing various parameters that will be used during the transfer. The initialization structure that is passed to the I2C_init function to initialize the device has the following values:

```
I2C_Init Init = {
      0,          /* 7  bit address mode */
      0x0000,     /* own address */
      144,        /* system clock value (Mhz)  */
      400,        /* rate - a number between 10 and 400*/
      0,          /* number of bits/byte to be received or transmitted (8) */
      0,          /* DLB mode off */
      1           /* FREE mode on */
};
```

Call the initialization function with the Init structure as an argument to initialize the I2C device.

```
I2C_init(&Init);   /* configure the I2C */
```

- **Step 3:** Write Operation

Referring again to Figure B–1 it is necessary to use S-A-D-P mode to control the registers. Call the I2C_send function with the reset array, the length (which will always be fixed at two), master mode on, slave address 0x1A (as determine above), S-A-D-P mode, and a timeout of 30000 to configure the reset register.

```
j=I2C_write(reset,2,1,0x1A,1,30000);  /* configure reset register */
```

Configure the other registers using the same format:

```
/* configure power down control register */
j=I2C_write(power_down_control,2,1,0x1A,1,30000);

/* configure analog audio path register */
j=I2C_write(analog_audio_path_control,2,1,0x1A,1,30000);

/* configure digital audio path register */
j=I2C_write(digital_audio_path_control,2,1,0x1A,1,30000);

/*configure digital audio interface register */
j=I2C_write(digital_audio_interface_format,2,1,0x1A,1,30000);

/* configure sample rate control register */
j=I2C_write(sample_rate_control,2,1,0x1A,1,30000);

/*configure digital interface activation register*/
j=I2C_write(digital_interface_activation,2,1,0x1A,1,30000);

/* configure left line input register */
j=I2C_write(left_line_input_volume_control,2,1,0x1A,1,30000);

/* configure right line input register */
j=I2C_write(right_line_input_volume_control,2,1,0x1A,1,30000);

/* configure left headphone register */
j=I2C_write(left_headphone_volume_control,2,1,0x1A,1,30000);

/* configure right headphone register */
j=I2C_write(right_headphone_volume_control,2,1,0x1A,1,30000);
```

- **Step 4:** MCBSP configuration and testing

In order to test the control functions of the codec, you must connect a set of headphones/speakers to the headphone ports on the board.

After configuring the registers, use the MCBSP serial interface to send data to the codec's DAC in order to play a tone. This tone plays indefinitely until the user manually halts the device.

For instructions on calculating sinewave tone values, please refer to Appendix C.

Please note that the headphone volume control register is configured to output maximum volume; therefore, after running the program you should be able to hear sounds. If a tone is heard, mute the device by setting the left and right headphone volume to the lowest possible value (–34.5 db).

```
Uint16 left_headphone_volume_control[2] = {0x05,0x00};
Uint16 right_headphone_volume_control[2] = {0x07,0x00};
```

After re-compiling and running the code, no sound should be heard. At this point you have verified the I2C control interface operation.

Appendix A CAT24WC256 Serial EEPROM

The CAT24WC256 is a 256K-bit Serial CMOS EEPROM on the Skywalker board and is internally organized as 32,768 words of 8 bits each. The CAT24WC256 features a 64-byte page write buffer and operates via the I2C bus serial interface. The EEPROM operates strictly as a slave device. The transfer is controlled by the Master device which generates the serial clock and all start and stop conditions for bus access.

For more information about the CAT24WC256 Serial EEPROM, please visit the website for Cadence Design Systems, Inc.

A.1 Device Addressing

The bus master begins a transmission by sending a start condition. The master sends the address of the particular slave device it is requesting. The five most significant bits of the 8-bit slave address are fixed as 10100 (see Figure A–2). The EEPROM uses the next two bits as address bits.

The address bits A1 and A0 allow up to four devices on the same bus. These bits must compare to their hardwired input pins. The last bit of the slave address specifies whether a read or write operation is to be performed. When this bit is set to 1, a read operation is selected, and when set to 0, a write operation is selected (please remember that this bit is automatically determined when you set the TRX bit of the ICMDR register – if TRX = 1, the write function will be selected, and if TRX = 0, the read function will be enabled).

After the master sends a start condition and the slave address byte, the EEPROM monitors the bus and responds with an acknowledge (on the SDA line) when its address matches the transmitted slave address. The EEPROM then performs a read or write operation depending on the state of the R/W bit.

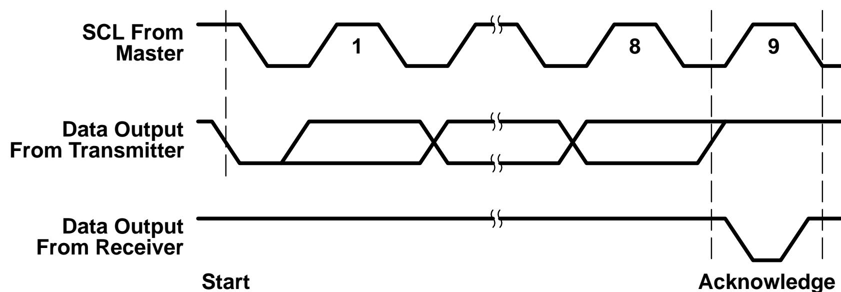


Figure A–1. Acknowledge Timing

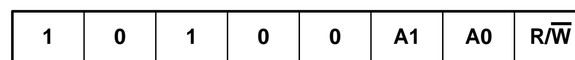


Figure A–2. Slave Address Bits

A.2 Acknowledge

After a successful data transfer, each receiving device is required to generate an acknowledge. The acknowledging device pulls down the SDA line during the ninth clock cycle, signaling that it received the 8 bits of data.

The EEPROM responds with an acknowledge after receiving a start condition and its slave address. If the device has been selected along with a write operation, it responds with an acknowledge after receiving each 8-bit byte.

When the EEPROM begins a read mode it transmits 8 bits of data, releases the SDA line, and monitors the line for an acknowledge. Once it receives this acknowledge, the EEPROM will continue to transmit data. If no acknowledge is sent by the master, the device terminates data transmission and waits for a stop condition.

A.3 Writing to the EEPROM (page write)

Using the page write operation, the EEPROM writes up to 64 bytes of data in a single write cycle. After each byte has been transmitted, the EEPROM responds with an acknowledge, and internally increments the six low order address bits by one. The high order bits remain unchanged.

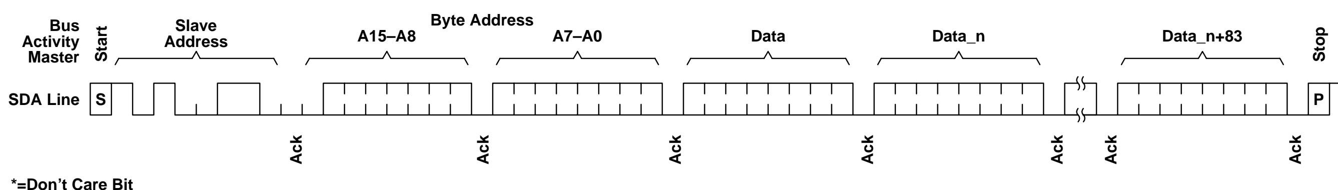


Figure A–3. Page Write Timing

Using the page write operation, the EEPROM writes up to 64 bytes of data in a single write cycle. The page write operation is initiated in the same manner as the byte write operation; however, instead of terminating after the initial byte is transmitted, the master is allowed to send up to 63 additional bytes. After each byte has been transmitted, the EEPROM will respond with an acknowledge, and internally increment the six low order address bits by one. The high order bits remain unchanged.

A.4 Reading From the EEPROM (sequential read)

Sequential read operations allow the master device to select at random any memory location for a read operation. The master device first performs a 'dummy' write operation (see Figure 6) by sending the start condition, slave address and byte addresses of the location it wishes to read.

After EEPROM acknowledges, the master device sends the start condition and the slave address again, this time with the R/W bit set to one (that is, setting the TRX bit in ICMDR register to receive).

The EEPROM then responds with its acknowledge and sends the 8-bit byte requested. After the EEPROM sends the initial 8-bit byte requested, the master will respond with an acknowledge which tells the device it requires more data.

The EEPROM will continue to output an 8-bit byte for each acknoawledge sent by the master. The operation will terminate when the master fails to respond with an acknowledge, thus sending the STOP condition. The data being transmitted from EEPROM is out-putted sequentially with data from address N followed by data from address N+1.

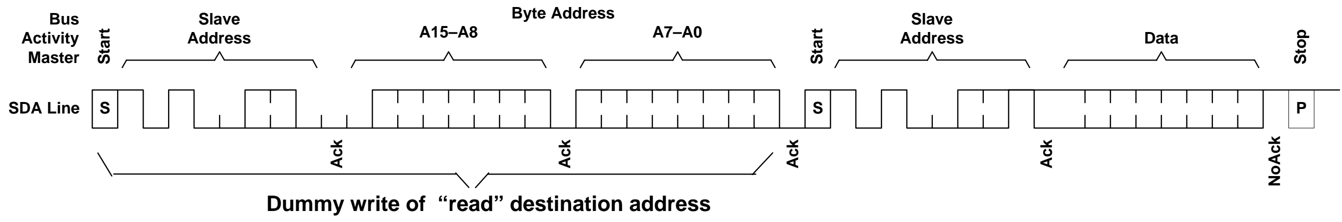


Figure A–4. Selective Read Timing

Appendix B TLV320AIC23 Stereo Codec

The TLV320AIC23 is a high-performance stereo audio codec that is found on the Skywalker board. For more details, please refer to the *TLV320AIC23 Stereo Audio D/A Converter, 8–to 96–kHz, With Integrated Headphone Amplifier* (SLWS106) data manual.

The analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) within the TLV320AIC23 use multibit sigma-delta technology with integrated oversampling digital interpolation filters. Data-transfer word lengths of 16, 20, 24, and 32 bits, with sample rates from 8 kHz to 96 kHz, are supported.

The ADC sigma-delta modulator features third-order multibit architecture with up to 90-dBA signal-to-noise ratio (SNR) at audio sampling rates up to 96 kHz, enabling high-fidelity audio recording in a compact, power-saving design.

The DAC sigma-delta modulator features a second-order multibit architecture with up to 100-dBA SNR at audio sampling rates up to 96 kHz, enabling high-quality digital audio-playback capability, while consuming less than 23 mW during playback only.

The TLV320AIC23 is the ideal analog input/output (I/O) choice for portable digital audio-player and recorder applications, such as MP3 digital audio players.

The TLV320AIC23 has many programmable features.

The control interface is used to program the registers of the device. The control interface complies with I2C specifications.

In I2C mode, the data transfer uses SDIN for the serial data and SCLK for the serial clock. The start condition is a falling edge on SDIN while SCLK is high. The seven bits following the start condition determine which device on the I2C bus receives the data. R/W determines the direction of the data transfer.

The TLV320AIC23 is a write only device and responds only if R/$\overline{W}$ is 0. The device operates only as a slave device whose address is selected by setting the state of the $\overline{CS}$ pin as follows.

Table B–1. Slave Address Selection

$\overline{CS}$ State (Default = 0)	Address
0	0011010
1	0011011

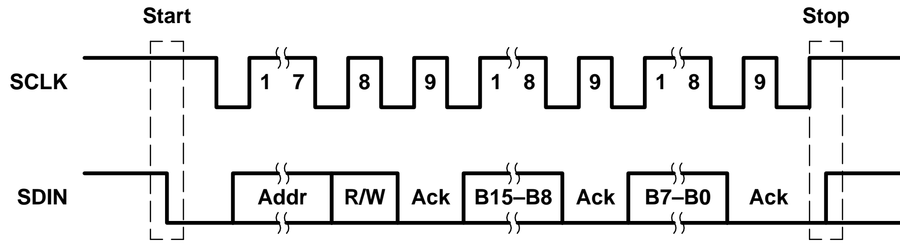
Based on the default setting of the $\overline{CS}$ pin (used in the TMS320VC5509 EVM), the slave address for the AIC23 should be 0011010, or 0x1A. The device that recognizes the address responds by pulling SDIN low during the ninth clock cycle, acknowledging the data transfer.

The control follows in the next two eight-bit blocks. The stop condition after the data transfer is a rising edge on SDIN when SCLK is high (see Table B–1).

The 16-bit control word is divided into two parts. The first part is the address block, the second part is the data block:

B[15:9] Control Address Bits

B[8:0] Control Data Bits



B[15:8] Control Address Bits
B[7–0] Control Data Bits

Figure B–1. 2-Wire I2C Compatible Timing

The AIC23 has eleven registers that are used to control operation. The addresses for the eleven registers are shown in Table B–2.

Table B–2. AIC23 Control Register Addresses

Address	Register
0000000	Left line input channel volume control
0000001	Right line input channel volume control
0000010	Left channel headphone volume control
0000011	Right channel headphone volume control
0000100	Analog audio path control
0000101	Digital audio path control
0000110	Power down control
0000111	Digital audio interface format
0001000	Sample rate control
0001001	Digital Interface activation
0001111	Reset Register

Appendix C Sine Wave Value Calculator

C.1 Sample Value Origination Description

The data sent to the codec are sampled values of a sine wave (tone) in Q15 format.

These values are determined by using the following procedure:

- **Step 1**

  Pick an arbitrary frequency, from 300 Hz to 3 kHz (this allows a tone to be heard without distortion).

- **Step 2**

  For the purposes of this example, select the frequency 1.378 kHz. Next, use the following formula to generate samples in decimal format:

  $$m[n] = Sin\left(\frac{2\pi f_{in}}{f_s}\right)$$

  where m[n] is the sampled value, fin is the selected frequency (1.378 kHz in this case), and fs is the sampling frequency of the codec (the codec was configured for a sampling frequency of 44.1 kHz).

- **Step 3**

  After obtaining the sample value, convert to the Q15 format by multiplying by 32767 and convert the resulting value into a hexadecimal number.

  For example:

n = 0:

m[n] = 0.0
m[n] * 32767 = 0.0
0.0 = 0x00000.

n = 1:

m[n] = .19507.
m[n] * 32767 = 6391.95223
6391.95223 = 0x018F8.

n = 2:

m[n] = .38265.
m[n] * 32767 = 12538.30973
12538.30973 = 0x30FB.

Using this procedure, the following values are sent to the codec:

static int SineSamples[]={0x00000, 0x018f8, 0x030fb, 0x0471c, 0x05a82, 0x06a6d, 0x07641, 0x07d8a, 0x07fff, 0x07d8a, 0x07641, 0x06a6d, 0x05a82, 0x0471c, 0x030fb, 0x018f8, 0x00000, 0x0e709, 0x0cf06, 0x0b8e5, 0x0a57f, 0x09594, 0x089c0, 0x08277, 0x08002, 0x08277, 0x089c0, 0x09594, 0x0a57f, 0x0b8e5, 0x0cf06, 0xe0709};

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

Copyright © 2001, Texas Instruments Incorporated