Implementing the DSP Paradigm on the C5510 DSK

Started: 20 July 2003
Revised: 29 January 2004
Updated: 14 September 2004

How does one

- fetch a sample from an A/D,
- process it,
- send the processed result to a D/A?

1 Introduction

This note delves into what is involved in using the TI C5510DSK to digitize an analog waveform, process the samples in some fashion, and convert the result back into the analog world. We start simple and build in stages. As noted in the opening lecture this semester this represents the basic digital signal paradigm.

TI supplies a rich set of tools as part of their Code Composer Studio system for use by experienced system implementers. These implementers are typically at the running stage of life whereas we have not yet walked or even crawled.

We will start at a very basic level and do everything ourselves. Once we understand what is being done and why we will move using the TI tools. We will, at least initially, do our programming using C.

A good way to start our investigation is to describe the steps of acquiring samples, processing them and sending them to an output at a reasonably abstract form. The C code shown in Figure 1 "implements" the basic steps. The details have been at a lower level.

The filling in of the details is the purpose of this note.

2 Analog input and output on the C5510 DSK

Consulting the TMS320VC5510DSK Technical Reference manual we learn that the DSK uses a TI TLV320AIC23 stereo audio codec chip to supply two channels of A/D conversion and two channels of D/A conversion.

The C5510 DSK manual shows in its Figure 2-1 (reproduced here as Figure 2) that the AIC23 is interfaced to the DSK via two ports of the Buffered multichannel Serial Port (McBSP). McBSP port 1 is used for setting AIC23 control words and McBSP port 2 is used to move stereo sample values values (two 16-bit values) between the C5510 and the AIC23.

In order to make use of the AIC23 on the C5510 DSK we need to understand how both the AIC23 and the C5510 McBSP are connected and and how they work.

```
void main(void)
{

    int  sample, psample;
    void setup_codec();
    int  input_sample(void);
    int  process_sample(int);
    void output_value;

    setup_codec();

    while (1) {
        sample = input_sample();
        psample = process_sample(sample);
        output_value(psample);
    }
}
```

Figure 1: The DSP paradigm, in abstract form, implmented using C.

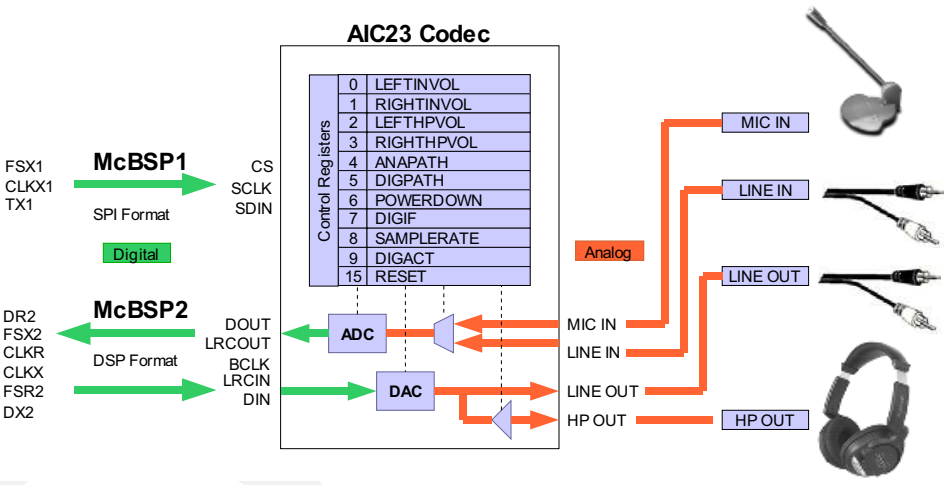


Figure 2: The AIC23 and C5510 interface. From the TMS320VC5510 DSK Technical Reference.

3 The C5510 McBSP — AIC23 connections

Figure 2 provides a high level picture of how the C5510 McBSP ports 1 and 2 connect to the AIC23. More detail can be determined studying the DSK schematics contained in the C5510 Technical Reference document.

A major function of TI's DSKs is to provide a reference design that can be studied by customers who are designing systems using TI's DSP devices.

The tracing of signals would have been make easier if signals coming from or going off-sheet had relevant sheet numbers appended.

3.1 The AIC23 control connection

The section of the AIC23 associated with programming the control registers is connected to the C5510 via McBSP port 1.

The section of the AIC23 associated with control word transfers is connected to the C5510 via McBSP port 1. Starting with schematic sheet 15 we can somewhat arduously trace the data path back to the C5510 shown on schematic sheet 2.

Sheet 13 shows the USB interface and the AIC23 as subsystems. Sheet 15 expands on the AIC23 subsystem. Sheet 8 shows the multiplexing and buffering of the signals. Sheet 2 shows which pins on the C5510 that are associated with the signals from sheet 8.

The sheet numbers and signals that propagate between sheets is shown in Figure 3.

C5510 pin	sheet 2	sheet 8	sheet 13	sheet 15
DX1	DSP_BDX1	DSP_BDX1	CTL_DX1	CTL_DATA
CLKX1	DSP_BCLKX1	DSP_BCLKX1	CTL_CLKX1	CTL_CLK
FSX1	DSP_BFSX1	DSP_BFSX1	CTL_FSX1	CTL_CS

Figure 3: Tracing the waveforms between the AIC23 data section and the C5510 chip.

3.2 The AIC23 data connections

The section of the AIC23 associated with data transfers is connected to the C5510 via McBSP port 2. Starting with schematic sheet 15 we can (still arduously) trace the data path back to the C5510 shown on schematic sheet 2.

Sheet 13 shows the USB interface and the AIC23 as subsystems. Sheet 15 expands on the AIC23 subsystem. Sheet 8 shows the buffering of the signals. Sheet 2 shows which pins on the C5510 that are associated with the signals from sheet 8.

The sheet numbers and signals that propagate between sheets is shown in Figure 4.

C5510 pin	sheet 2	sheet 8	sheet 13	sheet 15
DX2	DSP_BDX2	AIC23SDATAIN	DATA_DIN	SDIN
FSX2	DSP_BFSX2	LRCIN	DATA_SYNCIN	LRCIN
FSR2	DSP_BRSR2	LRCOUT	DATA_SYNCOUT	LRCOUT
CLKX2	DSP_BCLKX2	BCLK	DATA_BCLK	BCLK
CLKR2	DSP_BCLKR2	BCLK		
DR2	DSP_BDR2	AIC23SDATAOUT	DATA_DOUT	DOUT

Figure 4: Tracing the waveforms between the AIC23 data section and the C5510 chip.

4 The AIC23

The AIC23 codec is described in *TLV320AIC23 Stereo Audio CODEC, 8- to 96-kHz, With Integrated Headphone Amplifier Data Manual*, TI's document number SLWS106C, July 2001. CCS also contains descriptive information in particular a discussion of an API for the AIC23. Of particular interest to us is the `tone.c` program used as the tone example.

Figure 5 reproduces the block diagram of the AIC23 from the *TLV320AIC23 Data Manual*, (SLWS106C). Not shown in Figure 5 are the anti-alias filters at the A/D inputs and the anti-image filters at the D/A outputs.

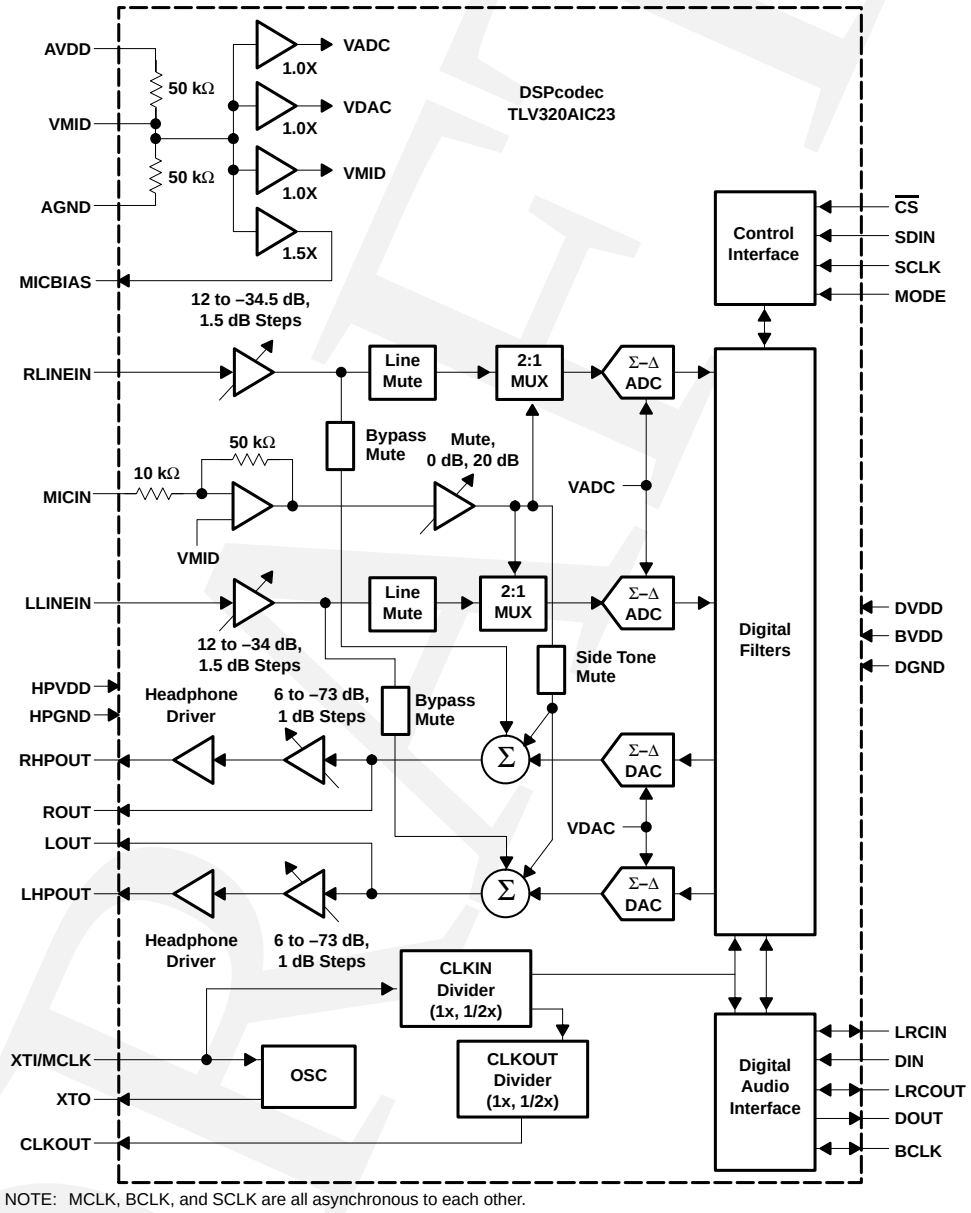
Checking the DSK schematics it determined that a 12 MHz clock is supplied to the AIC23. The AIC23 uses this clock to set its internal timings. The sample rates available for the A/D and the D/A converters are related but not necessarily the same. The anti-alias/image filter cutoff frequencies vary with the sample rates. The filter transfer function characteristics can vary depending on the sample rate. A total of 11 A/D and D/A sample rate combinations are available.

The AIC contains 10 registers that determine how the device operates and one register whose loading is used to cause a reset. The registers are listed in Figure 6.

The AIC23 control registers are 9 bits in length. Values are sent to the AIC23 as 16-bit units. The 7 most significant bits contain the address of the register to be loaded and the 9 least significant bits contain the value to be loaded. It is not possible to read the control registers back to the host processor.

5 The McBSP system

The two main sources for information about the McBSP system are:



NOTE: MCLK, BCLK, and SCLK are all asynchronous to each other.

Figure 5: Block diagram of the TI TLV320AIC23 codec. From the AIC23 data manual.

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

Figure 6: The AIC23 register set.

- *Introduction to McBSP*, TI's document number SPRU592A (This is a 275 page document indicating that some effort is going needed in order to understand its operation).
- *TMS320VC5510 Fixed-Point Digital Signal Processor Data Manual*, TI's document number SPRS607E. This documents the addresses of the McBSP registers in I/O space.

There are numerous sections of the CCS help system that deal with both the AIC23 interface and the McBSP. There are also code examples also included with CCS. The associated header (.h) files are a valuable source of information.

There are three McBSP ports on the C5510. Each port implements a full duplex bit-serial interface. A McBSP port is used to transmit and/or receive fixed length (such as 16 or 32 bit) values using a bit serial link. These ports minimize the number of wires needed to connect peripheral devices such as A/D and D/A converters to the C5510. This greatly simplifies the interconnection design on a printed circuit board. There is a corresponding cost increase for the supporting logic within the C5510 and the peripheral device but with today's technology this extra cost is very minimal.

The input and output channels associated with a McBSP port can be operated somewhat independently. For a port input or output channel

- one of the devices at the end of the channel serves as either a *master* or a

slave,
- there is a line for transmitting data values in bit-serial form,
- the master generates a timing waveform that is to be used by the slave to generate the bit-serial data,
- the master generates a *frame-sync* waveform that identifies a *frame* of several n-bit values.

The McBSP system is extremely flexible and is intended for use in a large and diverse number of applications. Because of this, a large number of decisions need to be made regarding determining the control values for any particular application. Once properly configured the McBSP is very easy to use.

The key registers associated with a McBSP port are shown in Figure 7. Each

even addr	odd addr	
DRR2	DRR1	Data Receive Registers
DXR2	DXR1	Data Transmitter Registers
SPCR2	SPCR1	Serial Port Control Registers
RCR2	RCR1	Receive Control Registers
XCR2	XCR1	Transmit Control Registers
SRGR2	SRGR1	Sample Rage Generator Registers
MCR2	MCR1	Multichannel Registers
PCR		Pin Control Register

Figure 7: Basic McBSP port control registers. The receive/transmit multichannel enable registers are not shown.

register contains 16 bits. Registers are located in I/O address space in such a way so that they can be read and written as 32-bit units. The McBSP port 0 register set starts at address 0x2800. The port 1 register set starts at 0x2C00 and the port 2 register set starts at 0x3000.

For a more complete description of the McBSP see the TI McBSP manual, TI document SPRU592A. This note assumes that the reader has at least scanned this document.

Because we are not going to be involved in making multichannel transfers there are fifteen registers that we will be dealing with. There four registers used for data movement. These are the DRR1, DRR2, DXR1 and DXR2 registers. There are four control register used to determine how the channel operates. These are RCR1, RCR2, XCR1 and XCR2. Two registers are used in setting transfer rates. Two multi-channel control registers are only of interest because we will have to

make sure that this feature is not active. The pin control register (PCR) controls the functions associated with several of the channels external pins.

5.1 Configuring the McBSP port 1

The McBSP registers lie in I/O address space. Chapter 5 of *TMS320C55x Optimizing C/C++ Compiler User's Guide* (SPRU281E) describes how to access values in I/O space.

Striking out on our own for now we need to be able to read and write the registers associated with McBSP ports 1 and 2. It would be nice if we could refer to the McBSP registers by name say as,

```
McBSP_reg(port, register_name)
```

where `port` could take on the values 0, 1, and 2 and `register_name` would be as used in the McBSP documentation. Say using the names used in the documentation:

```
DRR1   RCERA  XCERA
DRR2   RCERB  XCERB
DXR1   RCERC  XCERC
DXR2   RCERD  XCERD
SPCR1  RCERE  XCERE
SPCR2  RCERF  XCERF
RCR1   RCERG  XCERG
RCR2   RCERH  XCERH
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2   PCR
```

It turns out that this is easily accomplished using a header file. We will create a file `McBSP_452.h` for this purpose. Define statements can be used to define the desired register names. For example:

```
#define McBSP_DRR1  0x1
#define McBSP_DRR2  0x0
```

The string `McBSP_` has been prepended in order make the register names being defined very specific to the McBSP. This is considered a good practice. Unfortunately, it also becomes quite the nuisance if the names are typed frequently.

The piece d'resitance is

```
#define McBSP_reg(port,register) \
 (*((ioport unsigned *)((port*0x0400u)+0x2800u+register)))
```

which is the macro definition making it all work.

Using this macro we can write C statements of the form

```
McBSP_reg(1, McBSP_XCR1) = 0x4322;
uTemp = McBSP_reg(1, McBSP_DRR1);
```

making both life easier and our code more readable. We could use a variable rather than a constant to select the port making the code somewhat more general.

There were other choices that could have been chosen. One alternative to the above approach would be to define C macros for each I/O memory address used the the various McBSP ports. A naming convention making use of the register names found in the C5510 data manual would be quite reasonable. For example:

```
#define DRR1_0  (*((ioport unsigned *)(0x2801u)))
#define DRR2_0  (*((ioport unsigned *)(0x2800u)))
    .
    .
#define DRR1_1  (*((ioport unsigned *)(0x2C01u)))
#define DRR2_1  (*((ioport unsigned *)(0x2C00u)))
    .
    .
```

Many variations are possible. One must choose a convention and go with it or make use of what others have done prior. Often it is worthwhile to experiment a bit and see what works well and what doesn't. Two significant goals of whatever convention is chosen are to minimize the opportunities for making programming mistakes and to make the programming task easier.

Appendix A contains a listing of `McBSP_452.h`.

Next we need to figure out what values are needed in the registers for port 1. Checking the AIC23 data sheet we see that AIC23 control register interface can be configured to use either the I^2C or the SPI control protocol to read and/or write values. Which convention is used is determined by the level on the MODE pin. A quick check of the DSK schematic determines that the hardware is designed to use the SPI protocol.

At this point we don't know much if anything about either the I^2C or SPI protocols. The former has no relevance to the problem at hand while the latter has become of great interest.

Rummaging through the McBSP manual (SPRU592A) it is discovered that (not surprisingly) the McBSP supports the SPI protocol. Chapter 6 is dedicated to the SPI mode.

At this point one should sit down and read both documents focusing on the SPI.

Probably the first thing to be determined is which of the C5510 and the AIC23 is the master and which is the slave. The answer may differ depending on the McBSP port.

Because the control interface needs to be used to program the AIC23 operation we start there. The AIC23 functional diagram shows the clock going into the control interface. This indicates that for the control interface the chip is a slave.

The AIC23 control interface uses three lines.

- One line is for the control data being sent to the AIC. This is in 16-bit units. The most significant 7 bits selects a register and the least significant 9 bits are the value to be written into that register.
- One line is used for the clock that strobes values from the data line into the AIC.
- The remaining line is latch control line. It is used to latch the deserialized 9-bit data value into the appropriate register.

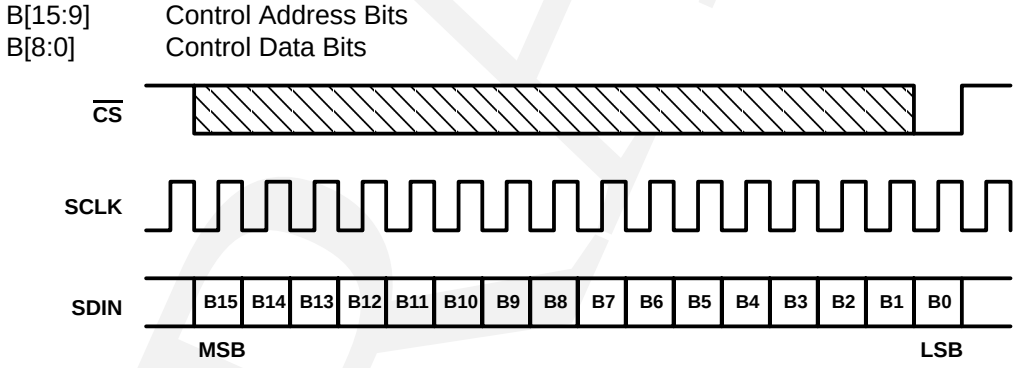


Figure 8: The AIC23 control waveform timing. From the AIC23 data manual.

The relationship between these three waveforms is shown in Figure 8. The AIC23 samples the serial data on the rising edges of the clock.

We still don't know much about SPI but the first topic we encounter in the McBSP SPI chapter is the clock stop mode. With Figure 8 in hand we should be able determine the settings needed for the McBSP SPI clock stop mode.

The McBSP gives us four timing relationship choices for the clock stop mode. All we have to do is compare the given timing diagrams with the one in the AIC23 sheet and we are home free, at least on this part.

- CLKSTP = 10b, CLKXP = 0, CLKRP = 0

  The clock transitions high to low during the middle of the bit period. Can't use.

- CLKSTP = 11b, CLKXP = 0, CLKRP = 1

  Looks good to me.

- CLKSTP = 10b, CLKXP =1, CLKRP = 0

  Probably will work. Not comfortable with the start up clock timing but should be ok.

- CLKSTP = 11b, CLKXP =1, CLKRP =1

  Negative transitions a mid bit interval. Can't use.

We will go with the second choice. We will use a 16-bit frame size.

Section 6.5 of the SPI manual is titled *Procedure for Configuring a McBSP for SPI Operation*. This is followed by section 6.6 *McBSP as the SPI Master*. There is useful information present here but not a step by step procedure that we can follow. Let's fast forward to section 8.2 and the immediately following sections.

Next we work through the outlined steps making programming notes as we go. We are not using the receiver and so will not make any changes there.

In order to guarantee a known starting point we will load the McBSP registers with the default values set into them after a hardware reset. These values are documented in the McBSP manual.

The McBSP prepend is dropped for the moment to keep line lengths (and the amount of typing) manageable.

```
McBSP_reg(1, SPCR1) = 0x0000;
McBSP_reg(1, SPCR2) = 0x0000;
McBSP_reg(1, RCR1)  = 0x0000;
McBSP_reg(1, RCR2)  = 0x0000;
McBSP_reg(1, XCR1)  = 0x0000;
McBSP_reg(1, XCR2)  = 0x0000;
McBSP_reg(1, SRGR1) = 0x00FF;
McBSP_reg(1, SRGR2) = 0x2000;
McBSP_reg(1, MCR1)  = 0;
McBSP_reg(1, MCR2)  = 0;
McBSP_reg(1, PCR)   = 0;
```

Next we will follow the steps outlined in Chapter 8 of the McBSP manual for configuring the port 1 transmitter. We will assume that the register contents are unknown and only the particular bits involved with a setting are to be altered. Such might be the case if we had previously programmed the receiver.

1. To reset the transmitter, sample generator and frame-sync logic

   ```
   McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x00C1);
   ```

2. Setting the transmitter pins to operate as McBSP pins:

   This is done by setting PCR bit 13 to a zero.

   ```
   McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x2000);
   ```

3. Disable digital loop back. This can be done by clearing bit 15 of SPCR1.

   ```
   McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x8000);
   ```

   We might should define a symbol and avoid hard wiring constants into the code.

4. Enable the clock stop mode bits in SPCR1.

   Above we decided to use CLDSTP=11.

   ```
   McBSP_reg(1, SPCR1) = (McBSP_reg(1, SPCR1)&(~0x1800))|0x1800;
   ```

   We are a bit more general than needed here.

5. Enable/disable the transmit multichannel selection. Choosing disable.

   ```
   McBSP_reg(1, MCR2) = McBSP_reg(1, MCR2)&(~0x0001);
   ```

   Again we are more general than perhaps expected.

6. Choosing 1 or 2 phases for transmit. Choosing one phase.

   ```
   McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x8000);
   ```

7. Setting the transmit word length. Setting 16 bits.

   ```
   McBSP_reg(1, XCR1) = (McBSP_reg(1, XCR1)&(~0x00E0))|0x0040;
   ```

8. Setting the transmit frame word length.

   One value per frame is needed. Since we are only using one phase we only need to set a value in XCR1.

   ```
   McBSP_reg(1, XCR1) = McBSP_reg(1, XCR1)&(~0x7F00);
   ```

9. Enable/disable transmit frame sync ignore.

   I think that this is a don't care for this application.

10. Setting the transmit companding mode. Not for the control data values.

    ```
    McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x0014);
    ```

11. Setting the transmit data delay. Setting to 0 seems reasonable.

    ```
    McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x0003);
    ```

12. Setting the transmit DXENA mode. I don't know. setting to 0 seems reasonable.

```
McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x0080);
```

13. Setting the transmit interrupt mode. Interrupts are later in the semester.

```
McBSP_reg(1, SPCR2) = McBSP_reg(1, SPCR2)&(~0x0030);
```

14. Setting the transmit frame-sync mode. The discussion on this item indicates doing the following operations for SPI use.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0800);
McBSP_reg(1, SRGR2) = McBSP_reg(1, SRGR2)&(~0x1000);
```

15. Setting the transmit frame-sync polarity. The AIC23 needs sync low.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0008);
```

16. Setting the SRG frame-sync period and pulse width. The frame sync period needs to be at least as the frame size. A value of 18 should work.

```
McBSP_reg(1, SRGR2) = (McBSP_reg(1, SRGR2)&(~0x00ff))|0x0013;
McBSP_reg(1, SRGR1) = McBSP_reg(1, SRGR1)&(~0xff00);
```

17. Setting the transmit clock mode. The McBSP needs to generate.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0200);
```

18. Setting the transmit clock polarity. This is the CLKXP signal which we decided way above needs to be 0.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0002);
```

19. Setting the SRG clock divide down value. This is determines the clock rate for the control data. The value divided down is C5510 clock (I think). Assuming a max clock of 200 MHz if we divide by 200 we get a 1 MHz control clock to the AIC23. Seems reasonable. A value of 128 gives a reasonable clock rate and is easy to convert to binary.

```
McBSP_reg(1, SRGR1) = (McBSP_reg(1, SRGR1)&(~0x00ff))|0x0080;
```

20. Choosing an input clock. Want to use the CPU clock.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0080);
McBSP_reg(1, SRGR2) = McBSP_reg(1, SRGR2)|0x2000;
```

21. Setting the input clock polarity. This involves setting the CLKSP, CLKXP and CLKRP bits. The CLKSP bit is not used for this application.

```
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0003);
```

22. Enable the transmitter.

```
McBSP_reg(1, SPCR2) = McBSP_reg(1, SPCR2)|0x00C1;
```

Away we go!

Wow! We made it through the Chapter 8 transmitter check list. TI even supplies forms to be filled out while going through this. They can be found in the McBSP manual and are also available in the CCS help system. We will have to go through this again for McBSP port 2, both for transmit and for receive.

Summarizing:

- SPCR1

```
McBSP_reg(1, SPCR1) = 0x0000;
McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x8000);
McBSP_reg(1, SPCR1) = (McBSP_reg(1, SPCR1)&(~0x1800))|0x1800;
McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x0080);
```

- SPCR2

```
McBSP_reg(1, SPCR2) = 0x0000;
McBSP_reg(1, SPCR1) = McBSP_reg(1, SPCR1)&(~0x00C1)
McBSP_reg(1, SPCR2) = McBSP_reg(1, SPCR2)&(~0x0030);
```

- RCR1

```
McBSP_reg(1, RCR1) = 0x0000;
```

- RCR2

```
McBSP_reg(1, RCR2) = 0x0000;
```

- XCR1

```
McBSP_reg(1, XCR1) = 0x0000;
McBSP_reg(1, XCR1) = (McBSP_reg(1, XCR1)&(~0x00E0))|0x0040;
McBSP_reg(1, XCR1) = (McBSP_reg(1, XCR1)&(~0x7F00));
```

- XCR2

```
McBSP_reg(1, XCR2) = 0x0000;
McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x8000);
McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x0014);
McBSP_reg(1, XCR2) = McBSP_reg(1, XCR2)&(~0x0003);
```

- SRGR1

```
McBSP_reg(1, SRGR1) = 0x00FF;
McBSP_reg(1, SRGR1) = McBSP_reg(1, SRGR1)&(~0xff00);
McBSP_reg(1, SRGR1) = (McBSP_reg(1, SRGR1)&(~0x00ff))|0x0080;
```

- SRGR2

```
McBSP_reg(1, SRGR2) = 0x2000;
McBSP_reg(1, SRGR2) = McBSP_reg(1, SRGR2)&(~0x1000);
McBSP_reg(1, SRGR2) = (McBSP_reg(1, SRGR2)&(~0x00ff)|0x0013;
McBSP_reg(1, SRGR2) = McBSP_reg(1, SRGR2)|0x2000;
```

- MCR1

```
McBSP_reg(1, MCR1) = 0x0000;
```

- MCR2

```
McBSP_reg(1, MCR2) = 0x0000;
McBSP_reg(1, MCR2) = McBSP_reg(1, MCR2)&(~0x0001);
```

- PCR

```
McBSP_reg(1, PCR) = 0x0000;
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x2000);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0800);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0008);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)|(0x0200);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0002);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0080);
McBSP_reg(1, PCR) = McBSP_reg(1, PCR)&(~0x0003);
```

This distills down to a more manageable:

```
McBSP_reg(1, McBSP_SPCR2) = 0x0000;  // stop xmtr
McBSP_reg(1, McBSP_SPCR1) = 0x1800;
McBSP_reg(1, McBSP_RCR1)  = 0x0000;
McBSP_reg(1, McBSP_RCR2)  = 0x0000;
McBSP_reg(1, McBSP_XCR1)  = 0x0040;
McBSP_reg(1, McBSP_XCR2)  = 0x0000;
McBSP_reg(1, McBSP_SRGR1) = 0x0080;
McBSP_reg(1, McBSP_SRGR2) = 0x2011;
McBSP_reg(1, McBSP_MCR1)  = 0x0000;
McBSP_reg(1, McBSP_MCR2)  = 0x0000;
McBSP_reg(1, McBSP_PCR)   = 0x0A08;
McBSP_reg(1, McBSP_SPCR2) = 0x00C1; // start xmtr
```

5.2 Using McBSP port 1 to initialize the AIC23

At this point we have the capability to send control values to the AIC23 using McBSP serial port 1. There are 10 8-bit registers in the AIC23 that need to be programmed.

Studying the DSK schematic and the AIC23 data manual we see that the AIC23 is supplied with a 12 MHz clock. The AIC23 can use this to set the sample rate. The AIC23 audio data channel also has a DSP mode designed to connect directly to TI DSP devices. Because the AIC23 is controlling the sample timing it makes sense that it be the master for audio data transfers on McBSP port 2.

It makes sense to do as was done in Spectrum Digital's example, `tone.c`, and simply place the required 10 AIC23 setup values in an array and use a function to send the array contents to the AIC23.

How does one send a 16-bit value to the AIC23 over McBSP port 1? Back to the McBSP documentation we go.

Basically the McBSP transmitter has a register that we place values into when it is empty. We check to see this register is empty and if so place a value into it. If not, we wait until it is empty. There is a bit in the SPCR2 register that we can check to see if transmitter data register is empty.

```
while((McBSP_reg(1, McBSP_SPCR2)&0x0002) == 0);
McBSP_reg(1, McBSP_DXR1) = value;
```

This can be made into a function or macro. Going the function route:

```
void McBSP_send(unsigned port, unsigned value)
{
    while((McBSP_reg(port, McBSP_SPCR2)&0x0002) == 0); // wait
    McBSP_reg(port, McBSP_DXR1) = value;
}
```

We can get some efficiency by letting the compiler expand this in line. However the savings will not be much compared to the typical wait time for the transfers.

Our AIC23 initialization code can be written in the form

```
McBSP_send(2, 0) = 0x0F00; // reset the AIC23
for (ctr = 0; ctr < N_presets; ctr++) {
    McBSP_send(2, *presets++)
}
```

The assumption here being that the preset values include the associated register address in the top 7 bits of a 16 bit value.

With this code we can get the AIC23 up and running (hopefully). Once running, the AIC takes whatever is on the McBSP port 2 output and sends it to the

D/As and sends out A/D values to the McBSP port 2 input. The AIC23, being the master, generates all the control and timing signals. The AIC23 runs autonomously *assuming* the device attached to it is accepting the A/D values being sent and sends values to be for the D/A.

However, at this point in our setup, McBSP port 2 is neither listening nor talking.

5.3 Configuring McBSP port 2

Next we setup the transfer of audio data as sets of two 16-bit values, left and right channel samples. McBSP port 2 needs to be configured to be compatible with the waveforms generated by the AIC23.

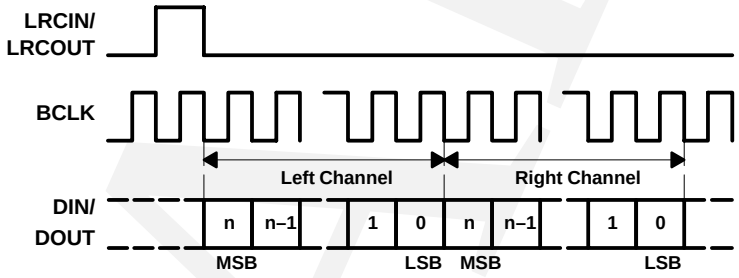


Figure 9: The AIC23 audio data digital waveform timing. From the AIC23 data manual.

We start by stopping the McBSP port 2 receiver and transmitter.

```
McBSP_reg(2, McBSP_SPCR1) = 0x0000;  // stop rcvr
McBSP_reg(2, McBSP_SPCR2) = 0x0000;  // stop xmtr
```

For the McBSP port 2 set-up, the use of the TI work sheets looks like the way to go. We are now a bit higher on the learning curve and hopefully have a better understanding of the McBSP than when we started.

This time we work through the registers on a register by register basis referring to McBSP manual chapters 7 and 8 as needed.

The AIC23 sends 2 16-bit words per frame, left channel followed by the right. This can be read by the McBSP as a 32-bit value with the most significant part, the left 16-bit value, placed into `DRR2` and the least significant part, the right 16-bit value, placed into `DRR1`.

Conversely the AIC23 expects the C5510 to send two 16-bit values per frame, left followed by right. A 32-bit frame can be used with the `DXR2` being used for the left part and `DXR1` for the right part.

The data sent by the AIC23 should be sampled on the rising edge of the receive clock. The data sent by the McBSP is expected to change on the falling edge if the transmit clock.

Working through the register list in Chapter 12 of the McBSP manual we get:

```
McBSP_reg(2, McBSP_SPCR1) = 0x0000;  // stop rcvr
McBSP_reg(2, McBSP_SPCR2) = 0x0000;  // stop xmtr
McBSP_reg(2, McBSP_RCR1)  = 0x00A0;  // 2 16-bit words/frame as 32 bits
McBSP_reg(2, McBSP_RCR2)  = 0x0000;
McBSP_reg(2, McBSP_XCR1)  = 0x00A0;  // 2 16-bit words/frame as 32 bits
McBSP_reg(2, McBSP_XCR2)  = 0x0000;
McBSP_reg(2, McBSP_SRGR1) = 0x0000;
McBSP_reg(2, McBSP_SRGR2) = 0x0000;
McBSP_reg(2, McBSP_MCR1)  = 0x0000;
McBSP_reg(2, McBSP_MCR2)  = 0x0000;
McBSP_reg(2, McBSP_PCR)   = 0x0003; // use of clock edges

McBSP_reg(2, McBSP_SPCR1) = 0x0001; // start rcvr
McBSP_reg(2, McBSP_SPCR2) = 0x00C1; // start xmtr
```

Looking at the non-zero bits that resulted it appears that life is much simpler when the peripheral acts as the master. The above set of configuration values was a lot easier figure out than the ones for the control channel!

6 Testing what we have done so far

The above code was collected together into a file named `setup_code.c`. The short program (based on Spectrum Digital's `tone.c`) shown in Figure 10 was written for use in testing.

The CCS project was built from scratch. The `rts55.lib` was used to set up the C environment. The sample linker command file, `lnk.cmd` was used for linking. Both were found in the directory `C:\ti\c5500\cgtools\lib`. A warning caused by the command file was simply ignored. With not much todo the program was found to work as expected. At least, it *appears* to work as expected.

The test program simply sends data values to both the left and right channels. The right channel (or what is nominally the right channel) is scaled by a factor of 2 to better identify it.

7 Adding sample input support

The AIC23 data manual does not appear to specify the clock rate used to shift data bits on the audio data line nor the timing relationship between the `LCRIN`

```
/* Test program for EECS 452 C5510 AIC23 standalone support
 *
 * 12Jul03 .. initial version .. KM
 *
 */

#include "McBSP_452.h"

/* 64 value sine table */

int SineTable[] = {
    0x0000, 0x0C8C, 0x18F9, 0x2528, 0x30FB, 0x3C56, 0x471C, 0x5133,
    0x5A82, 0x62F1, 0x6A6D, 0x70E2, 0x7641, 0x7A7C, 0x7D89, 0x7F61,
    0x7FFF, 0x7F61, 0x7D89, 0x7A7C, 0x7641, 0x70E2, 0x6A6D, 0x62F1,
    0x5A82, 0x5133, 0x471C, 0x3C56, 0x30FB, 0x2528, 0x18F9, 0x0C8C,
    0x0000, 0xF374, 0xE707, 0xDAD8, 0xCF05, 0xC3AA, 0xB8E4, 0xAECD,
    0xA57E, 0x9D0F, 0x9593, 0x8F1E, 0x89BF, 0x8584, 0x8277, 0x809F,
    0x8001, 0x809F, 0x8277, 0x8584, 0x89BF, 0x8F1E, 0x9593, 0x9D0F,
    0xA57E, 0xAECD, 0xB8E4, 0xC3AA, 0xCF05, 0xDAD8, 0xE707, 0xF374
};

void main()
{
    int  sample, N_samples;

    N_samples = sizeof(SineTable)/sizeof(int);

    setup_codec();

    // McBSP_send(1, 8*0x0200+0x000C); // does the sample rate change?

    /* output a sinewave on both channels */

    while (1) {
        for (sample = 0; sample < N_samples; sample++) {
            McBSP_send(2, SineTable[sample]);
            McBSP_send(2, SineTable[sample]/2);
        }
    }
}
```

Figure 10: Initial McBSP/AIC23 test program.

and the LRCOUT waveforms used for frame-sync when in master mode.

The AIC23 data manual Figure 3.8 (reproduced here as Figure 9) shows that the input and the output data streams have the same relations between their frame-sync signals (LRCIN and LRCOUT) and between the data bit timing and the clock. This figure sort of implies that LRCIN and LRCOUT are coincident (at least when the input and output data rates are the same). Is this a valid impression?

If the two frame-sync waveforms are coincident then we might use either the arrival of a sample to determine when to load the transmitter buffer with the next value for the D/A. Or, the emptying of the transmitter buffer to indicate that a value had arrived from the A/D.

Any time offset between the frame-sync waveforms reduces the amount of time the C5510 has available between having received a sample and when it has to place a processed result into the transmitter buffer to be sent to the AIC23.

A couple of test programs were written for checking the timing as seen by the McBSP. Using an oscilloscope it was observed that

- The McBSP port 2 XRDY bit becomes set about 160 ns prior to the RRDY bit becomes set.
- The clock used to shift the serial data is probably 12 MHz.

The interval between sample values for a 48 kHz sample rate is 20.8 μs.

No measurements were made for the case where the A/D and the D/A clocks differed.

If what was observed represents fact then there is approximately 20.6 μs available in which to process a sample and place the result into the McBSP transmitter buffer.

What this comes down to is that the setting of the RRDY flag not only indicates that a sample value has been received but it can also be used to flag when the McBSP buffer is empty.

The program shown in Figure 10 was modified to support receiving samples from the AIC23. The result is shown in Figure 11

8 Time delay through the system

Many control applications are sensitive to the delay through their subsystems. The amount of delay associated with the elements inside of a feedback loop has a direct effect on the loop. Less delay is good, more delay is bad. Learning how to plan and make delay measurements is a very educational exercise and one that we will doing later in the semester. When making such measurements it is useful to have some idea of what the result is going to be.

Sources of delay include:

```
/* Test program for EECS 452 C5510 AIC23 standalone support
 *
 * 12Jul03 .. initial version .. KM
 * 14Jul03 .. added AIC23 sample input .. KM
 *
 */

#include "McBSP_452.h"

/* 64 value sine table */

int SineTable[] = {
    0x0000, 0x0C8C, 0x18F9, 0x2528, 0x30FB, 0x3C56, 0x471C, 0x5133,
    0x5A82, 0x62F1, 0x6A6D, 0x70E2, 0x7641, 0x7A7C, 0x7D89, 0x7F61,
    0x7FFF, 0x7F61, 0x7D89, 0x7A7C, 0x7641, 0x70E2, 0x6A6D, 0x62F1,
    0x5A82, 0x5133, 0x471C, 0x3C56, 0x30FB, 0x2528, 0x18F9, 0x0C8C,
    0x0000, 0xF374, 0xE707, 0xDAD8, 0xCF05, 0xC3AA, 0xB8E4, 0xAECD,
    0xA57E, 0x9D0F, 0x9593, 0x8F1E, 0x89BF, 0x8584, 0x8277, 0x809F,
    0x8001, 0x809F, 0x8277, 0x8584, 0x89BF, 0x8F1E, 0x9593, 0x9D0F,
    0xA57E, 0xAECD, 0xB8E4, 0xC3AA, 0xCF05, 0xDAD8, 0xE707, 0xF374
};

int LeftInput, RightInput; // sample values go into these

/*
 *  places the two calling values into the McBSP transmitter buffer
 *  waits for a new sample to arrive
 *  places the sample value into LeftInput and RightInput
 *  then returns
 */

void AIC23_IO(unsigned port, int LeftValue, int RightValue)
{
    McBSP_reg(port, McBSP_DXR2) = LeftValue;
    McBSP_reg(port, McBSP_DXR1) = RightValue;

    while((McBSP_reg(port, McBSP_SPCR1)&0x0002) == 0); //  wait for sample

    LeftInput  = McBSP_reg(2, McBSP_DRR2);
    RightInput = McBSP_reg(2, McBSP_DRR1);
}

void main()
{
    int  sample, N_samples;

    N_samples = sizeof(SineTable)/sizeof(int);

    setup_codec();

    /* output values on both channels */

    while (1) {
        for (sample = 0; sample < N_samples; sample++) {
            AIC23_IO(2, SineTable[sample], RightInput);
        }
    }
}
```

Figure 11: Rudimentary program for receiving samples from the AIC23 A/D and sending processed results back to the AIC23 D/A

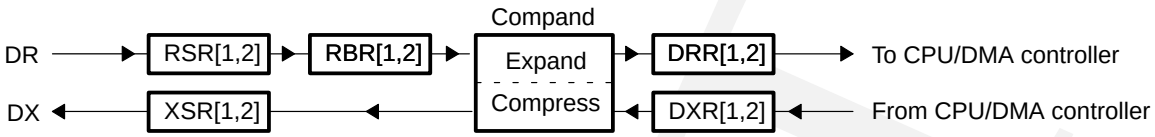


Figure 12: Registers used by the McBSP for moving data values. From the TI McBSP manual, SPRU592A.

- the A/D converter. The AIC23 is a so-called Sigma-Delta converter. These do not capture values at any given instant but rather use a signal processing technique to estimate sample values. There is generally a significant (relative the the sample rate) amount of delay with such converters.
- the time needed to move values from the codec to the DSP. This can have two major components, the time required by the bit serial data transfers, and the buffering of values within the codec and/or the DSP.
- the time required by the buffering (if any) within the DSP.
- the computation time required by whatever algorithms are being implemented.
- the time needed to values from the C5510 to the codec. Again this can have two major components.
- the D/A converter. The AIC uses a Sigma-Delta D/A converter. Such is a signal processing device in its own right. There can be a significant amount of delay (relative to the reconstruction clock) associated with this device.

Of concern in this note is the time delay associated with the initial buffering and serialization and deserialization.

The McBSP can be used to serialize and/or deserialized a number of word sizes ranging from 8-bits to 32 bits. For word sizes 16 bits and less values use 16-bit registers. For word sizes greater than 16 bits register pairs are used.

The transmitter and receiver are *double buffered*. A register (pair) is used to (de)serialize the data. While this is going on it is not possible to load a new value without corrupting the value currently being transmitted. This timing bottle neck is eliminated by providing a second buffer register (pair). Figure 12 shows the basic register of a McBSP port.

Consider the transmitter. The buffer register precedes the register used for serialization. Values to be transmitted are loaded into the buffer. When it becomes time (determined by the frame-sync waveform) to start transmitting the next value the buffer contents are transferred into the serialization. The buffer is now considered empty and a new value can be loaded. This can be safely be done at any time prior to the next frame-sync pulse.

Figure 13 illustrates the timing involved with data transfers between the AIC23 and the C5510.

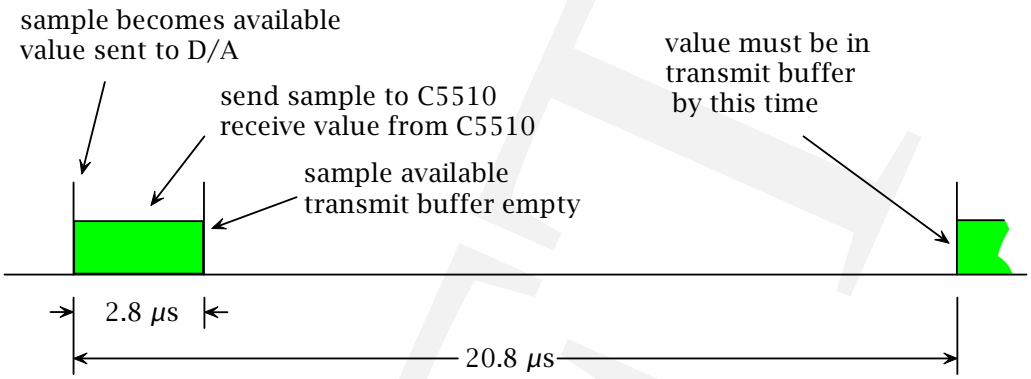


Figure 13: Timing for data transfers between the AIC23 and the McBSP. Sample rate of 48 kHz.

Starting a the time at which a sample value becomes available within the AIC23 the delays associated with our above code include:

- serial transfer
- processing
- serial transfer

A careful study of the timing in Figure 13 indicates that there should be a total of two sample interval durations between the time the codec has a sample value available and when the codec loads the processed result into its D/A processing.

9 Recapitulation

Figure 2 is a very useful and very well thought out representation the relationship between the C5510, the AIC23, and the outside world. The focus being on the AIC23. Our goal in this note was to bring the AIC32 to life. To put it to work digitizing analog waveforms, sending sample values to the C5510 for processing, receiving processed digital values and converting them into analog form.

The primary information resources that we used were the

- `tone.c` example code supplied by Spectrum Digital,
- data manual for the TLV320AIC23 codec.
- the data manual for the TMS320VC5510 DSP device,
- the manual describing the McBSP peripheral,
- the schematics for the C5510DSK.

The basic problem was divided into a series of questions:

- How to send information to the AIC23?

- How to configure the McBSP for SPI operation?
- What values to send to configure the AIC?
- Were we successful setting up the AIC23?
- How to do both output and output to the AIC?
- How to insure values are properly associated with the left and right channels?

We pretty much took these on one-by-one. We were somewhat fortunate in that what we did worked right off. Not necessarily the normal situation. The McBSP is a very flexible peripheral and there are many choices that have to be made. The existence of a 175 page manual for the McBSP is daunting in its own right.

Our focus in this note is learning how to do rather than on only how to use. TI supplies a significant amount of support for the "how to use" aspect. In EECS 452 we will focus on "to do" for most of the semester. The reasons for this include

- someone needs to know how to do,
- adding peripherals outside of those supported by TI will need this knowledge,
- reduces the dependence on others,
- the knowledge transfers to DSP devices supplied by other manufacturers (there are indeed some),
- it will lead to be better understanding and appreciation of what one is doing when using the TI tools.

TI has formed what it terms a *DSPThird Party Network* of over 600 independent companies to create and sell hardware and software to support TI's DSP devices. A lot of people are earning a living providing TI customers with support. If you want to participate you really need to know what is going on, at all levels.

A few closing comments about our code.

This is first cut, learning code. It could be used as a basis for a more mature set of support routines. Typically one would have an *open* routine, a normal use set of routines, and a *close* routine.

The AIC23 configuration was hardwired into the `setup_code` function. If we desire to use this code for a number of applications (such as a series of lab exercises) using different configurations we need to keep changing the code. If this is what is done, it is too all easy to use the wrong version on occasion. Rather than constantly modifying the code what might we do differently?

One possible modification to `setup_codec` would be to pass the AIC23 setup list as a parameter to the codec setup function. This would allow different applications configure the codec as needed without having to keep modifying the setup code itself. The size of the list could be hardwired to be 10 items, make a call parameter, or built into the list somehow. One way would to use a value that would not appear as a list terminator. One value could be used as a list terminator is the reset value, `0x1E00`.

Another alternative to the hardwired setup problem is to always set up the AIC23 using a documented default configuration and, if necessary, modify it on the fly once the AIC23 is running. For example, adding the line of code

```
McBSP_send(1, 8*0x0200+0x000C);
```

following the call to `setup_code` should result in setting the sample rate to 8 kHz. To set up use of the 96 kHz sample rate the required instruction would be

```
McBSP_send(1, 8*0x0200+0x001C);
```

Don't forget that the characteristics of the anti-alias and the anti-image filters change with the sample rate.

In order to more better isolate the application programmer from the implementation details a small function could be written to take a list of change values and send them to the AIC23. Though it isn't clear whether this isolation all that helpful.

How to best (if there is such a thing) configure the software for ease of use and reliability is the software designer's or engineer's task.

A The McBSP symbol definition header file

```
#ifndef _McBSP_452_H_

#define _McBSP_452_H_

#define McBSP_DRR1  0x1
#define McBSP_DRR2  0x0
#define McBSP_DXR1  0x3
#define McBSP_DXR2  0x2
#define McBSP_SPCR1 0x5
#define McBSP_SPCR2 0x4
#define McBSP_RCR1  0x7
#define McBSP_RCR2  0x6
#define McBSP_XCR1  0x9
#define McBSP_XCR2  0x8
#define McBSP_SRGR1 0xB
#define McBSP_SRGR2 0xA
#define McBSP_MCR1  0xD
#define McBSP_MCR2  0xC
#define McBSP_RCERA 0xE
#define McBSP_RCERB 0xF
#define McBSP_RCERC 0x13
#define McBSP_RCERD 0x14
#define McBSP_RCERE 0x17
#define McBSP_RCERF 0x18
#define McBSP_RCERG 0x1B
#define McBSP_RCERH 0x1C
#define McBSP_XCERA 0x10
#define McBSP_XCERB 0x11
#define McBSP_XCERC 0x15
#define McBSP_XCERD 0x16
#define McBSP_XCERE 0x19
#define McBSP_XCERF 0x1A
#define McBSP_XCERG 0x1D
#define McBSP_XCERH 0x1E
#define McBSP_PCR   0x12

#define McBSP_reg(port,register) \
(*((ioport unsigned *)((port*0x0400u)+0x2800u+register)))

#endif
```

B setup_codec.c

```
/* EECS 452 McBSP/AIC23 basic paradigm example support
 *
 * 12July03 .. initial version .. KM
 *
 */

#include "McBSP_452.h"

/* AIC23 setup values from tone.c modified for our use. */

int AIC23_params[] = {
    0*0x0200+0x0017, // left  input volume
    1*0x0200+0x0017, // right input volume
    2*0x0200+0x01F9, // left  headphone volume
    3*0x0200+0x01F9, // right headphone volume
    4*0x0200+0x0011, // analog audio path control
    5*0x0200+0x0000, // digital audio path control
    6*0x0200+0x0000, // power down control
    7*0x0200+0x0043, // digital audio interface format
    8*0x0200+0x0001, // sample rate control
    9*0x0200+0x0001, // digital interface activation
};

/* Function to send values to McBSP transmitter */

void McBSP_send(unsigned port, unsigned value)
{
    while((McBSP_reg(port, McBSP_SPCR2)&0x0002) == 0); // xmtr wait
    McBSP_reg(port, McBSP_DXR1) = value;
}

/* Function to set up both the McBSP ports and the AIC23.
 *
 * Returns with the data flowing between the C5510 and the AIC23.
 *
 */

void setup_codec()
{
    int N_presets, *presets, ctr;

    /* first set up McBSP port 1 */
```

```
    McBSP_reg(1, McBSP_SPCR2) = 0x0000;  // stop xmtr
    McBSP_reg(1, McBSP_SPCR1) = 0x1800;
    McBSP_reg(1, McBSP_RCR1)  = 0x0000;
    McBSP_reg(1, McBSP_RCR2)  = 0x0000;
    McBSP_reg(1, McBSP_XCR1)  = 0x0040;
    McBSP_reg(1, McBSP_XCR2)  = 0x0000;
    McBSP_reg(1, McBSP_SRGR1) = 0x0080;
    McBSP_reg(1, McBSP_SRGR2) = 0x2011;
    McBSP_reg(1, McBSP_MCR1)  = 0x0000;
    McBSP_reg(1, McBSP_MCR2)  = 0x0000;
    McBSP_reg(1, McBSP_PCR)   = 0x0A08;
    McBSP_reg(1, McBSP_SPCR2) = 0x00C1; // start xmtr

    /* reset the AIC */

    McBSP_send(1, 0x1E00);

    /* next set up the AIC23 registers */

    N_presets = sizeof(AIC23_params)/sizeof(int);
    presets = &AIC23_params[0];

    for (ctr = 0; ctr < N_presets; ctr++) {
        McBSP_send(1, *presets++);
    }

    /* finally setup McBSP port 2 */

    McBSP_reg(2, McBSP_SPCR1) = 0x0000;  // stop rcvr
    McBSP_reg(2, McBSP_SPCR2) = 0x0000;  // stop xmtr
    McBSP_reg(2, McBSP_RCR1)  = 0x00A0;  // 2 16-bit words read as 32 bits
    McBSP_reg(2, McBSP_RCR2)  = 0x0000;
    McBSP_reg(2, McBSP_XCR1)  = 0x00A0;  // 2 16-bit words read as 32 bits
    McBSP_reg(2, McBSP_XCR2)  = 0x0000;
    McBSP_reg(2, McBSP_SRGR1) = 0x0000;
    McBSP_reg(2, McBSP_SRGR2) = 0x0000;
    McBSP_reg(2, McBSP_MCR1)  = 0x0000;
    McBSP_reg(2, McBSP_MCR2)  = 0x0000;
    McBSP_reg(2, McBSP_PCR)   = 0x0003; // use of clock edges

    McBSP_reg(2, McBSP_SPCR1) = 0x0001; // start rcvr
    McBSP_reg(2, McBSP_SPCR2) = 0x00C1; // start xmtr
}
```