INEL 5326

Communication System Design Signal Processing Experiments

Shawn Hunt

January 2002

Index

Introduction

Our experience has been that students are interested in implementing stuff in hardware. In the academic setting, this means learning how to apply knowledge gained in previous classes where the emphasis has been on theory. In this class, the emphasis is placed on *doing*. Going beyond simply playing with hardware, the purpose of the class is to provide the student with a major design experience in the area of communication and signal processing. This major design experience means going through all phases of a design, from initial problem statement to implementation.

A major design experience has many phases or steps. Of all the things the student must do in the design experience, the only thing the students have not seen before will probably be the DSP hardware. Preparing for the implementation part of the design will mean learning about the hardware its software tools. In order to facilitate this, the class is broken into two parts, the first consisting of eight experiments, the second consisting of the project.

The first two experiments introduce the hardware and software tools. They first learn the assembler and debugger, and how to run programs on the DSK. They next learn how to select a desired sampling rate, and how to compile and run programs in C.
There are six experiments where the student implements signal processing algorithms, including FIR and IIR filters, multirate systems, FFTs, and adaptive filters. The seventh experiment has to do with precision and quantization, and the last experiment has to do with integrating C and Assembly programs.

There were three important considerations when designing the experiments. First, have the students learn by doing. This was done by having the students learn most of the skills needed for the project with hands-on experiments. Second, the emphasis in this class is on signal processing, not programming. If the student is interested, there are other classes in the department that deal with hardware and assembly programming. On the other hand, it is good for the students to have exposure to signal processing algorithms written in assembly. To this end, in many of the experiments the students implement two equivalent algorithms, one written in Assembly, the other in C. The assembly programs come included with the class book, and the programs in C are written by the student. The third consideration is that the class is only one semester long. Combining this with the fact that the emphasis is not on programming, many usefull subroutines are provided to the students instead of having them write code. The subroutines needed for accessing the A/D and D/A converter, and also an FFT coded in assembly are provided. Since all experiments involve the dsp board, it is inevitable that the student will learn about it and its assembly language. What we are trying to achieve is a balance between hardware/assembly and signal processing.

The experiment descriptions are purposely written in a terse manner. Since they are to be done in preparation for a capstone design project, a cookbook approach is *not* desired. Instead, each experiment is discussed in class before given to the students. The students should have a clear idea of *what* is to be done in the experiment, but they must think about *how* to do it. This is built up gradually, starting of with very simple experiments, leading to more involved ones. The first two experiment descriptions are very specific, giving the commands needed to generate the desired output. It is also desirable for the student to learn to use reference materials. This means reading about the theory in from other sources, and also reading the manuals included with the DSK and its assembler.

The equipment available in the lab is:

- PCs
- C31 DSKs
- Oscilloscopes
- Function generators
- Multimeters
- Spectrum analyzers

Software available is:

- Programs from the class book, "Digital Signal Processing:Laboratory Experiments using C and the TMS320C31 DSK," 'by Rulph Chassaing.
- Matlab
- C compiler for the PC
- C compiler for the DSK
- Assembler for the DSK

1. Using the DSK Tools and Generating a Sine Wave

Purpose

The purpose of this experiment is to learn how to use the DSK software tools, and to actually run a program on the DSK that produces an audible output.

Background

Assembler and Linker

A program written in Assembly language passes through two steps in producing an executable file.

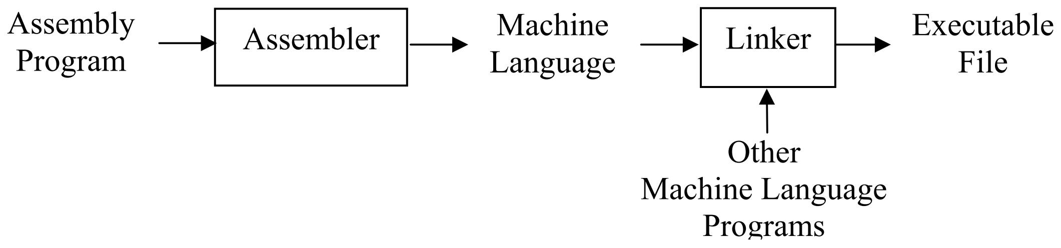


Figure 1.1. Steps from assembly program to executable file.

The assembler takes the assembly program and translates it into machine language that the processor understands. Although the resulting program is in machine language, it cannot be run without specifying external references and the memory to be used. This is the linkers job. The linker, as the name implies, takes the machine language program and links it with other machine language programs where the subroutines and/or data are defined. The linker also specifies the interface between the program and the hardware. In other words, it specifies where in memory the program and the data will be placed.

The program **dsk3a** does both the assembling and linking. The command

```
>>dsk3a matrix.asm
```

will assemble and link the program *matrix.asm* and produce an executable output called *matrix.dsk*. A more complete description of the Assembler is found in Chapter 5 of the *TMS320C3x DSP Starter Kit User's Guide*.

Boot Loader

There are two ways of loading executable files into the DSK, the boot loader, and the debugger. The boot loader is a program that downloads an executable file from the PC into the DSK and runs it. This can be done with the command

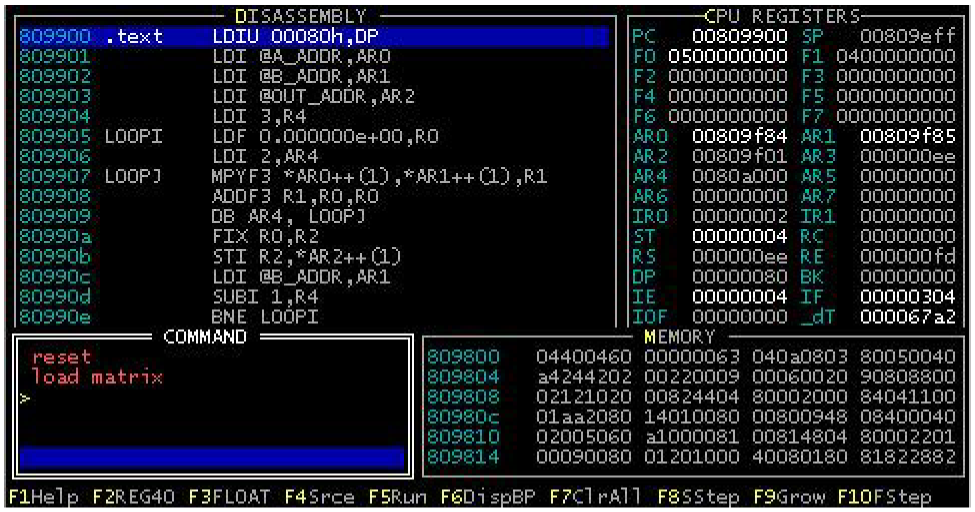
```
>>dsk3load sine4p.dsk
```

Debugger

The debugger is a program that allows not only programs to be loaded and run on the DSK, but also allows the user to view the DSK registers and memory and the program being executed. A more complete description of the Debugger is found in chapter 7 of the *TMS320C3x DSP Starter Kit User's Guide*. The debugger is run using the command

```
>>dsk3d reset
```

The debugger has four windows, the COMMAND window, the DISASSEMBLY window, the CPU REGISTERS window and the MEMORY window. See Figure 1.2 below.



```
                     DISASSEMBLY                             CPU REGISTERS
809900 .text     LDIU 00080h,DP                      PC     00809900 SP     00809eff
809901           LDI @A_ADDR,AR0                     F0   0500000000 F1   0400000000
809902           LDI @B_ADDR,AR1                     F2   0000000000 F3   0000000000
809903           LDI @OUT_ADDR,AR2                   F4   0000000000 F5   0000000000
809904           LDI 3,R4                            F6   0000000000 F7   0000000000
809905 LOOPI     LDF 0.000000e+00,R0                 AR0    00809f84 AR1    00809f85
809906           LDI 2,AR4                           AR2    00809f01 AR3    000000ee
809907 LOOPJ     MPYF3 *AR0++(1),*AR1++(1),R1        AR4    0080a000 AR5    00000000
809908           ADDF3 R1,R0,R0                      AR6    00000000 AR7    00000000
809909           DB AR4, LOOPJ                       IR0    00000002 IR1    00000000
80990a           FIX R0,R2                           ST     00000004 RC     00000000
80990b           STI R2,*AR2++(1)                    RS     000000ee RE     000000fd
80990c           LDI @B_ADDR,AR1                     DP     00000080 BK     00000000
80990d           SUBI 1,R4                           IE     00000004 IF     00000304
80990e           BNE LOOPI                           IOF    00000000 _dT    000067a2
              COMMAND                                 MEMORY
 reset                            809800    04400460 00000063 040a0803 80050040
 load matrix                      809804    a4244202 00220009 00060020 90808800
>                                 809808    02121020 00824404 80002000 84041100
                                  80980c    01aa2080 14010080 00800948 08400040
                                  809810    02005060 a1000081 00814804 80002201
                                  809814    00090080 01201000 40080180 81822882
F1Help F2REG40 F3FLOAT F4Srce F5Run F6DispBP F7ClrAll F8SStep F9Grow F10FStep
```

Figure 1.2. The Debugger program interface

The program starts in the COMMAND window. A summary of commands can be seen by typing *help* at the prompt. A few of these commands are shown below.

load test.dsk	loads the program *test.dsk*
F5	run the program
F8	single step through the program
ALT-D	go to the DISASSEMBLY window
ALT-C	go to the CPU REGISTERS window
ALT-M	go to the MEMORY window
Esc	return to the COMMAND window
reset	Reset the DSK board
quit or **exit**	Exit program
help	Summary of commands

The registers R0-R7 (shown as F0-F7 in the debugger) are 40 bits long, and are used for floating point numbers. From the Command window press

F3	to see registers R0-R7 in float
F2	to see registers R0-R7 in hexadecimal

The memory used in the C31 is 32 bits long. The values can be displayed in hexadecimal, float or integer format. The format for saving float and integer numbers is different, so the values will be different if displayed in integer or float format. For example, a float value of 2 will be shown as +2.0000000000e+00 in float and 16777216 in integer. An integer value of 2 will be shown as 2 in integer, and as +1.0000002384e+00 in float. From the Command window type

memf	to display the memory values in float
memi	to display the memory values in integer
memx	to display the memory values in hexadecimal

Experiments:

Experiment 1.1
Assemble and run *sine4p.asm* using the boot loader and the debugger. Connect the AIC output to the oscilloscope and speaker to see and hear it.

Experiment 1.2
Do example 1.1 in the book.

Skills: Upon completing this experiment you should:

- Know how to use the assembler/linker **dsk3a**
- Know how to use the boot loader, **dsk3boot**, to run a program.
- Know how to use the debugger, **dsk3d**, to load and run a program and to see memory and register values
- Know how to connect the DSK using the parallel port and the AIC

2. Initializing the AIC, and Analog Input and Output

Purpose

The purpose of this experiment is to play with and learn about the AIC on the DSK. This includes learning how to initialize the AIC for the desired sampling rate and filter bandwidths. Also, the C compiler, assembler and linker will be used.

Background

Initializing the AIC and Selecting Sampling Rates

Analog input and output is done by the AIC (Analog Interface Circuit).

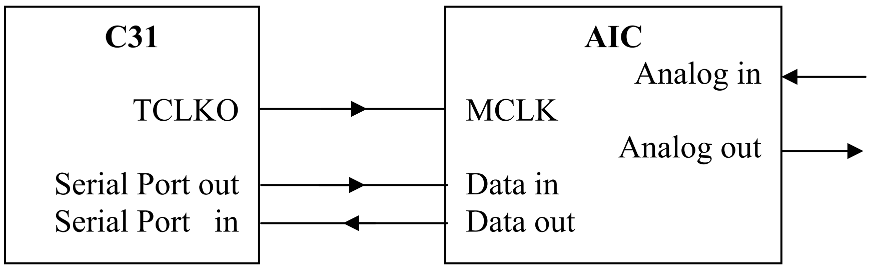


Figure 2.1. Connections between the C31 and the AIC

There are three main connections between the C31 and the AIC, a Clock and a Serial Port in and out. A timer on the C31 generates a signal (TCLKO) used as the Master Clock (MCLK) of the AIC. This timer, the C31 serial port, and the AIC must be initialized before the AIC can be used. The initialization sequence is:

Initialize the C31 timer
Initialize the C31 serial ports
Initialize the AIC

The C31 TCLKO output can have a maximum frequency of 12.5 MHz (50MHz/4), but the MCLK input on the AIC can have a maximum frequency of 10 MHz. In the supplied programs, the C31 timer is initialized for a frequency of 6.25 MHz (50MHz/8).

Initializing the AIC includes setting four registers that determine the input and output sampling rates, the input and output filter bandwidths, and control data. These are shown in figure 2.2.

bit															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	TA					X	X	RA					0	0
X	TA'						X	RA'						0	1
X	TB						X	RB						1	0
X	X	X	X	X	X	X	X	Control						1	1

Figure 2.2. AIC Registers (X indicate don't cares)

See figure 4-8 of the *TMS320C3x DSP Starter Kit User's Guide*. RA and RB determine the input sampling frequency and the cutoff frequency of the anti-aliasing filter. TA and TB determine the output sampling frequency and the cutoff frequency of the reconstruction filter. The equations are similar for both, so they will be shown only for the transmit case.
The equations for determining the filter bandwidth and the sampling frequency are:

$$BW = \frac{3600Hz}{288kHz}\left(\frac{MCLK}{2 \cdot TA}\right)$$

$$F_s = \frac{MCLK}{2 \cdot TA \cdot TB}.$$

Example 2.1

Suppose we want an input and output sampling rate of 8kHz, and input and output filters with a bandwidth of 3,600Hz. Let the master clock frequency be 6.25MHz.

From the first equation above, we have,

$$3600Hz = \frac{3600Hz}{288kHz}\left(\frac{6.25MHz}{2 \cdot TA}\right).$$

Solving for TA,

$$TA = \frac{3600Hz}{288kHz}\left(\frac{6.25MHz}{2 \cdot 3600}\right) = 10.85.$$

Represent this in binary using the closest integer

$$11_{10} = 01011_2.$$

The actual bandwidth of the filter is then,

$$BW = \frac{3600Hz}{288kHz}\left(\frac{6.25MHz}{2 \cdot 11}\right) \approx 3{,}551Hz.$$

From the second equation we have

$$8000 = \frac{MCLK}{2 \cdot TA \cdot TB},$$

or solving for TB

$$TB = \frac{MCLK}{2 \cdot TA \cdot F_s} = \frac{6.25MHz}{2 \cdot 11 \cdot 8kHz} = 35.51.$$

Again, rounding to the nearest integer and expressing this in binary is

$$36_{10} = 100100_2.$$

The actual sampling frequency is

$$F_s = \frac{6.25MHz}{2 \cdot 11 \cdot 36} \approx 7{,}891Hz.$$

Thus, TA=01011_2, and TB=100100_2. These numbers must be put into the format shown in figure 4-8 of the *TMS320C3x DSP Starter Kit User's Guide* (also shown in figure 2.2). Note that the number sent to each register in the AIC is two bytes (16 bits) long:

Using zeros for the don't cares (X's),
Register 1

$$0\ 0\ TA\ 0\ 0\ RA\ 0\ 0_2 = \underbrace{0\ 0\ 0\ 1}_{1}\underbrace{0\ 1\ 1\ 0}_{6}\underbrace{0\ 0\ 1\ 0}_{2}\underbrace{1\ 1\ 0\ 0_2}_{C} = 162C_h$$

Register 3

$$0\ TB\ 0\ RB\ 1\ 0_2 = 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0_2 = 4892_h$$

The programs supplied with the class book initialize the AIC requiring only the numbers for the four registers. An example of these numbers for C and Assembly programs is shown in example 2.2. Note that hexadecimal numbers can be identified using *0x* or *h*.

Example 2.2

C program:

```
int AICSEC[4] = {0x162C,0x1,0x4892,0x67};   /*Config data for AIC*/
```

Assembly program:

```
AICSEC  .word    162Ch,1h,4892h,67h              ; Fs = 8 kHz
```

Notice that these are the first and third numbers determined in example 2.1. This means that the sampling rate in this example is 7,891*Hz*, and that the anti-aliasing and reconstruction filters have bandwidths of 3,551*Hz*. The fourth number configures the control register of the AIC. The control register is used to set many parameters, including the anti-aliasing filter. Use figures 4-8 and 4-9 of the *TMS320C3x DSP Starter Kit User's Guide* to determine how to include or remove the anti-aliasing filter.

Using the C compiler

In this experiment you will use the DSK floating point tools to compile, assemble and link a program written in C. These tools include three programs, **cl30**, **asm30** and **lnk30**. To convert a program written in C into an executable file that can be run on the DSK, it needs to go through three steps shown in figure 2.2.

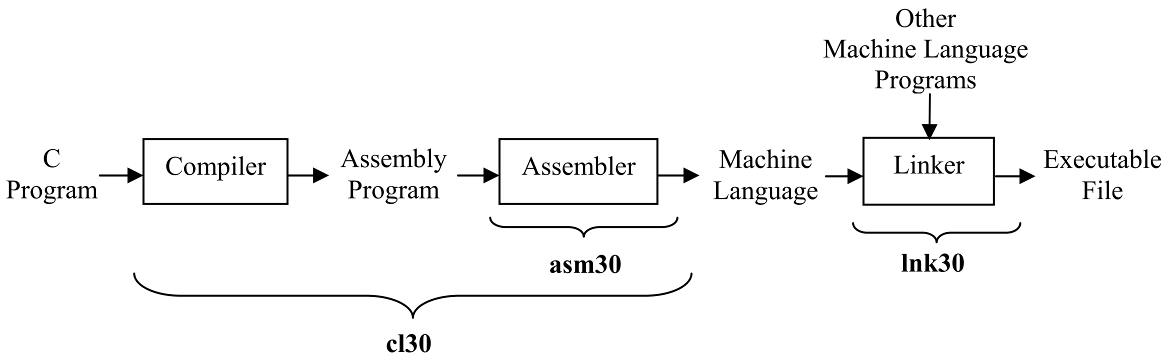


Figure 2.2. Steps from C program to executable file

The Program is first compiled and assembled. Both steps can be done with the command **cl30**. For example, to compile and assemble the program *loopc.c*, use

```
>>cl30 loopc.c
```

This program will give *loopc.obj* as output. To compile and assemble separately, use the commands

```
>>cl30 loopc.c -n
>>asm30 loopc.asm
```

The -n option for **cl30** makes it only compile, giving *loopc.asm* as output. The second program, **asm30**, will give *loopc.obj* as output.
Finally, the program is linked to produce an executable file.

```
>>lnk30 loopc.cmd
```

The linking resolves external function calls and determines how the system memory will be used. The linking program needs to know which programs to link together, what memory is available, and where to put each part of the program. This can be done from the command line, or can be written in a command file. The file *loopc.cmd* has this information for this example and is shown below.

```
****************************************************
    /*LOOPC.CMD - COMMAND FILE FOR LINKING     */
    -c                 /* USING C CONVENTION       */
    -stack 0x100       /* 256 WORDS STACK          */
    -heap  0x100       /* 256 WORDS HEAP           */
    vecs_dsk.obj       /* INSTALL INTERRUPT        */
    loopc.obj          /* MAIN PROGRAM             */
    -O loopc.out       /* LINKED COFF OUTPUT FILE */
    -l rts30.lib       /* RUN-TIME LIBRARY SUPPORT*/
    MEMORY
    {
     RAMS: org = 0x0809800, len = 0x2        /*BOOT STACK          */
     RAM0: org = 0x0809802, len = 0x03FE     /*INTERNAL BLOCK 0  */
     RAM1: org = 0x0809C00, len = 0x03C0     /*INTERNAL BLOCK 1  */
     VECS: org = 0x0809FC5, len = 0x0040     /*VECTORS              */
    }
    SECTIONS
    {
     .text:   {} > RAM0          /* CODE                     */
     .cinit:  {} > RAM0          /* INITIALIZATION TABLES    */
     .stack:  {} > RAM1          /* SYSTEM STACK             */
     .bss:    {} > RAM0          /* BSS SECTION              */
     vecs:    {} > VECS          /* VECTOR SECTION           */
    }
****************************************************
```

Figure 2.3. Command file used to link loopc

The first part of the program specifies that the programs to be linked are *loopc.obj* and *vecs_dsk.obj*. The output is written to *looopc.out*, and the library is *rts.lib*. The second part of the program called MEMORY specifies the starting address and the length of each available memory segment and defines a name for it. For example, there is a memory segment starting at the address 809C00h with length 3C0h and is given the name RAM1. The final part is called SECTIONS and defines where in memory each part of the program and data will be placed. The text is where the program is located, the stack is where the local variables are stored, and bss is where the global and static variables are stored.

Experiments

Experiment 2.1

a. Assemble and run *loopi.asm*. Connect the function generator as input, and see and hear the output. When you have verified that the hardware is working, make shure the sampling rate is 8kHz.

b. Remove the anti-aliasing filter from the input. Connect the oscilloscope so you can see both the input and output. Increase the frequency of the input to above 4kHz. Verify that the Nyquist frequency is 4kHz, and that there is aliasing in the system. For example, with the frequency of the signal generator set to 6kHz, what is the frequency of the output? Why?

c. Change the sampling rate to 10kHz. What is the Nyquist frequency? With the signal generator set to 6kHz what is the output frequency? What should the output frequency be for an input of 7kHz? Why?

Experiment 2.2

a. Compile, assemble, link and run *loopc.c*. Make sure the sampling rate is 8kHz.
b. Connect the oscilloscope so you can see both the input and output. Remove the anti-aliasing filter, input a sine of 300Hz and determine the delay from input to output. Increase the frequency and verify that the system has linear phase.
c. Modify *loopc.c* so that every other sample is multiplied by -1. With the frequency generator connected to the input, vary the input from 400 to 4kHz. Observe the output. Explain why this happens. Connect music or speech to the input and listen to the output.
d. Modify *loopc.c* so that every other sample is set to zero. Connect the function generator and input a sine wave from 400 to 4kHz. What is the frequency response of the system?
(c and d from Sorensen and Chen, 1997)

Skills: Upon completing this experiment you should:

- Know how to initialize the codec for any desired sampling rate
- Know how to change the cutoff frequency of the input and output filters
- Know how to remove the anti-aliasing filter from the input of the AIC
- Know how to compile, assemble and link a program written in C
- Know how to write a short experiment report

3. FIR filter implementations

Purpose

The purpose of this experiment is to implement FIR filters in real time.

Background

In this experiment, an FIR filter will be implemented in real time. There are various things that make this experiment challenging. First, the program for filtering must be implemented in C. The C code for implementing an FIR filter is simple, but may cause problems because this program is meant to work in real time. In previous classes, convolutions have been done by hand or in MATLAB. This is the first time a convolution will be done in real time.

In the real time filter, the convolution is implemented by using the difference equation directly. Using $x[n]$ as the input and $y[n]$ as the output, the difference equation for an FIR filter is,

$$y[n] = h_0 x[n] + h_1 x[n-1] + ... + h_{N-1} x[n-N+1].$$

This experiment is also challenging because this may be the first design implemented in hardware. The design may meet the specifications in MATLAB but fail when implemented on the DSK. The AIC used for input and output is not perfect, and this must be taken into account when designing the filter. For example, in one of the designs the output must be between 0 and -1 dB from 1000 to 3000Hz. If the AIC amplifies the signal, then it must be attenuated by the digital filter in order to stay below 0 dB.

Having to *show* that the design meets the specifications is the final challenge. This means using the function generator along with the oscilloscope and multimeter to show that the filter is linear phase and that it meets the magnitude specifications.

Design and implementation are generaly done in steps. If the problem is very simple it may be possible to design and implement all at once, but this is generally not the case. In most cases, the design and implementation are broken into parts, and each part is verified to work before moving on to the next part. In this experiment, the problem is not difficult, but it is difficult to verify what is going on in the DSK if the output is not what you expect. I have seen the same thing happen many times: a student writes the complete code for the DSK in C and runs it. It does not work, and the student has no idea why. Instead of implementing the solution all at once on the DSK, it is better to implement a very simple filter on the PC. This way every step of the problem will be easily veryfiable, and easy to debug. A program written for this experiment will also help with other experiments and probably with your final project.

I would recommend the following procedure. Start with a program on the PC that can read and write numbers from a simple ASCII file. This would basically be the loopc.c program from the DSK, but for the PC. Once this program is running, use it to implement a very simple, say third order, FIR filter. Make shure that the numbers are read and written from the files simulating what is done on the DSK. In other words, instead of reading all the values at the beginning, read one value from the file, compute the output, write it to another file, read the next value, etc. This C program on the PC is very simple to debug, because you can write the values at each step to the screen. Select the filter coefficients and the input so that the output is easy to calculate by hand and see if the program is doing what you want. Once this program is running, then go ahead and implement the filter on the DSK.

Experiments

Experiment 3.1
Design a FIR linear phase filter in Matlab for the magnitude specifications given in Figure 3.1. Implement the filter using *firnc.asm*. The filter must meet the specifications in MATLAB, but does not need to meet the specifications when implemented on the DSK.
a. Set the sampling frequency to 8kHz. What is the cutoff (-3dB) frequency in Hz?
b. Change the sampling frequency to 10kHz. What is the new cutoff frequency? What should it be theoretically?
c. Reset the sampling frequency to 8kHz and remove the anti-aliasing filter. Connect the oscilloscope so you can see both the input and output. Input a frequency of 300Hz and determine the delay from input to output. Next, verify that the filter is linear phase.
d. With a sampling frequency of 8kHz and no anti-aliasing filter, increase the input frequency from 500Hz to 7.5kHz. What happens to the signal? Explain why this happens. Is this really a lowpass filter?

Experiment 3.2
Design a minimum order FIR linear phase filter for the magnitude specifications given in Figure 3.2 in Matlab. Modify *loopc.c* to implement the filter. ***Show*** that the filter you implement on the DSK meets the specifications.

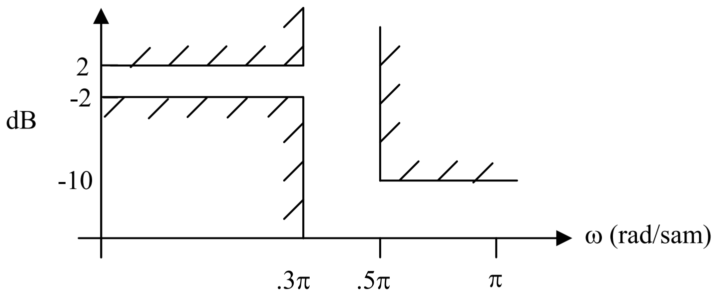


Figure 3.1. Specifications for the Lowpass filter of experiment 3.1.

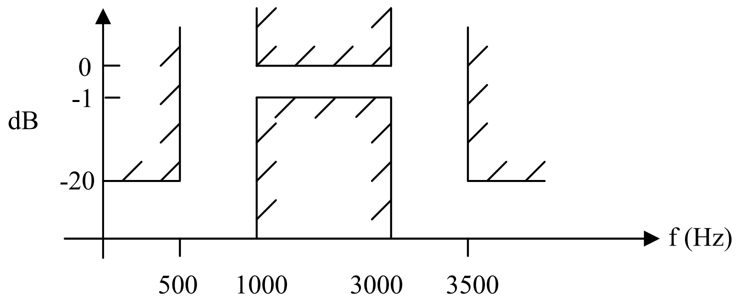


Figure 3.2. Specifications for the Bandpass filter of experiment 3.2.

Skills: Upon completing this experiment you should:

- Have reviewed how to design FIR filters using MATLAB
- Know how to implement an FIR filter on the DSK
- Know how to verify the specifications of the implemented filter using a function generator, an oscilloscope and a multimeter

4. IIR filter implementations

Purpose

The purpose of this experiment is to implement an IIR filter in real time and to work with an open ended problem.

Background

The first part of this experiment is similar to experiment 3, except that the filter is IIR instead of FIR. The difference equation for an IIR filter is shown below,

$$y[n] = b_0 x[n] + b_1 x[n-1] + \ldots + b_{P-1} x[n-P+1] - a_1 y[n-1] - \ldots - a_{M-1} y[n-M+1].$$

It is similar to the difference equation for an FIR filter, so the C program written in experiment 3 can be used for IIR filters with minor modifications.

The second part of the experiment is an open ended problem. Open ended problems are problems with no set solution. Use the supplied wave file. It is a recording of people speaking with background noise. You will design a filter (it may be FIR or IIR) so that it sounds better. This is real design! In 5309 you calculated coefficients for given specifications, but here you must select the specifications yourself. As mentioned above, there is no one 'best' solution, and you may have to go through many iterations to find one that works well. As is generally the case, you will want to do much or most of the design using MATLAB, then implement it on the DSK.

Experiments

Experiment 4.1

Design a minimum order IIR filter for the magnitude specifications given in Figure 4.1 using Matlab. Use *iir6bp.asm* to implement this filter. Verify that the system is not linear phase. Show that the filter you implement on the DSK meets the specifications. How does the implementation on the DSK compare with the magnitude and phase predicted by MATLAB?

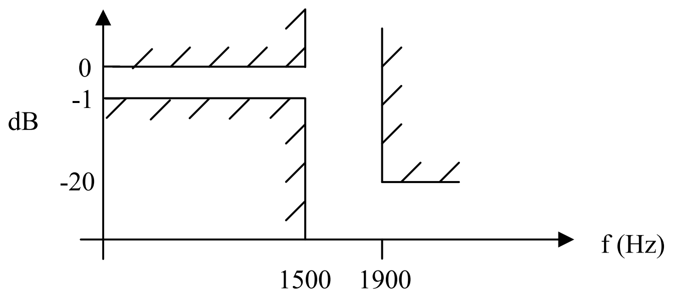


Figure 4.1. Magnitude specifications for lowpass filter.

Experiment 4.2
Modify *loopc.c* to implement an IIR filter that meets the same specifications as in experiment 4.1. Show that the filter you implement on the DSK meets the specifications.

Experiment 4.3
Design a filter (FIR or IIR) to make the given wave file sound better.

Skills: Upon completing this experiment you should:

- Have reviewed how to design IIR filters using MATLAB
- Know how to implement an IIR filter on the DSK
- Have practiced designing filters for open ended problems
- Practiced writing short experiment reports

5. Multirate System implementation

Purpose

The purpose of this experiment is to use FIR filters in a modern signal processing implementation.

Background

This process of changing the sampling rate of a signal in the discrete time domain is done by what are called multirate systems. These multirate systems can be used to increase or decrease the sampling rate. Many digital systems use multirate techniques. One case is where we need to link two digital communication systems that have different sampling rates. Other cases have to do with hardware implementation. Most of the A/D converters on the market are sigma-delta converters, which sample at a very high sampling rate, then convert the digital data into a lower sampling rate. In most CD players, there is a multirate system that does 'oversampling'. This technique of oversampling means increasing the sampling rate of the data. Both the sigma-delta A/D converters and oversampling CD players use multirate systems because of hardware considerations. In other words, it is easier to get a certain level of performance with multirate than without it. Lets look at one example of changing the sampling rate in more detail.

The first CD players that came on the market did not use oversampling. The data was sent to the D/A converter at the CD sampling rate of 44.1kHz, and converted to analog. The digital to analog converter includes a reconstruction filter to eliminate the high frequency repetitions of the signal. If the original signal had frequency components to 20kHz, then there will be repetitions starting at 24.1kHz (44.1-20). See figure 5.1.

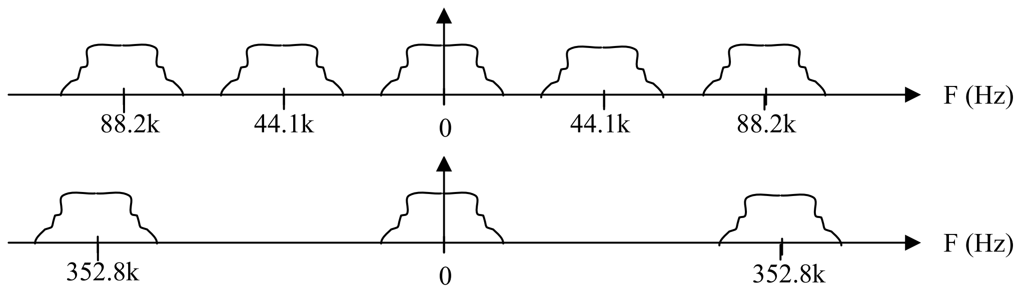


Figure 5.1. Signals before reconstruction filter with sampling rates of 44.1 and 352.8kHz

The CD system has 16 bits, and a SNR of about 98dB. Thus, we would like a analog lowpass filter with a passband to 20kHz, and a stopband at 24kHz with an attenuation of at least 98dB. Remember that there is a 90^0 phase shift for each pole in the system. Thus, this filter will have a large phase shift, especially near the cutoff frequencies. Some people theorized that this phase shift had a negative effect on the sound of these CD players. Oversampling can alleviate the situation. With 8 times oversampling, common today, the signal is converted from a sampling rate of 44.1kHz, to a sampling rate of 8*44.1kHz=352.8kHz. The process of increasing the sampling rate cannot increase the information in the signal, so it still only has content to 20kHz. The advantge is that since the sampling rate is 352.8kHz, the high frequency repetitions will start at 332.8kHz instead of 24.1kHz. The analog reconstruction filter still needs to pass frequencies to 20kHz, but the stopband is now at 352kHz. Since the cutoff is so high, the phase shift will also be at much higher, inaudible, frequencies.

So multirate techniques are usefull, but how is it done? In this experiment, we will only consider increasing or decreasing the sampling rate by an integer amount.

Suppose we have an analog signal, x(t), that is band limited to 4kHz, with the magnitude frequency response shown in figure 5.2. Let x(t) be sampled at two different sampling frequencies. Let $x_1[n]$ be the discrete signal we get sampling at 8kHz, and $x_2[n]$ be the signal we get sampling at 16kHz. The resulting magnitude frequency responces are shown in figure 5.3. As long as there was no aliasing in the sampling, we can process $x_1[n]$ to get a signal as if it had been sampled at 16kHz.

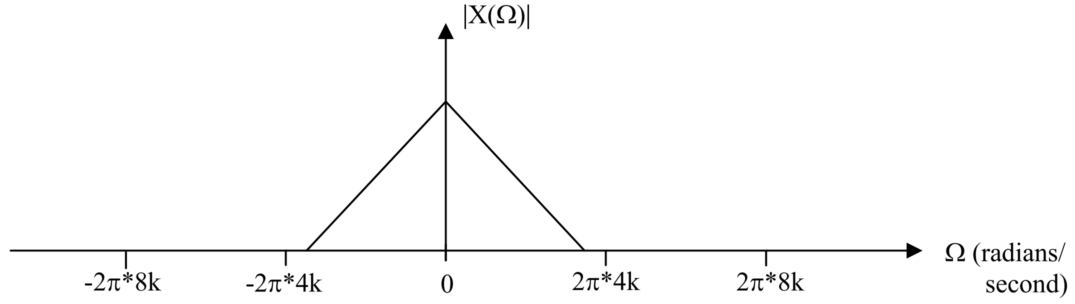


Figure 5.2. Magnitude frequency response of x(t).

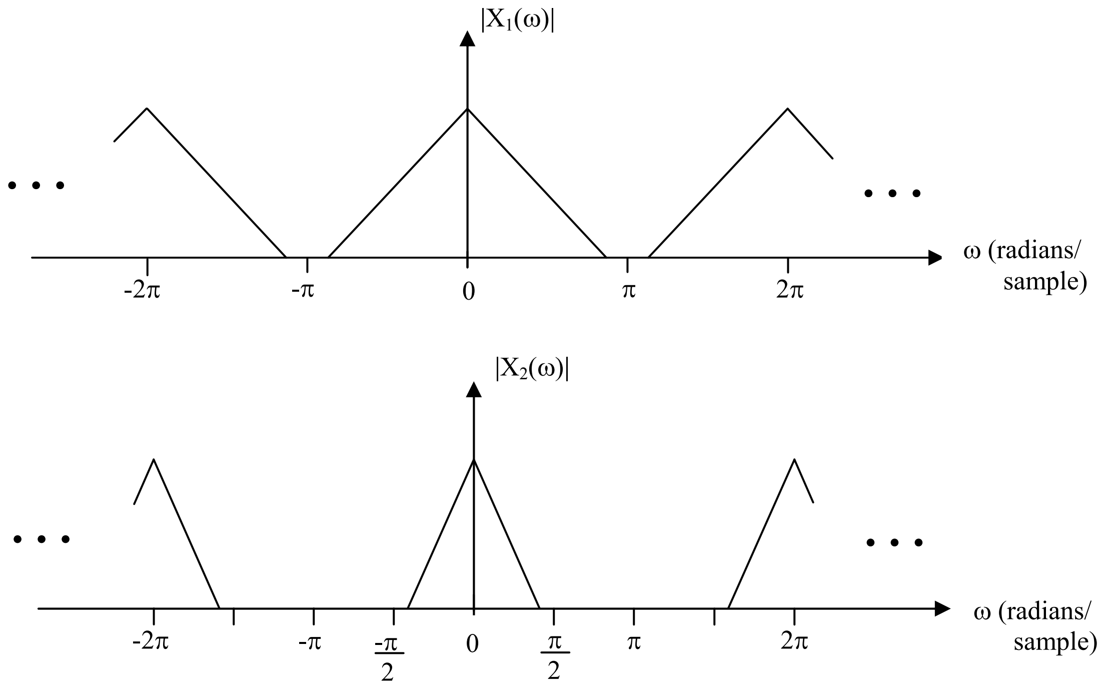
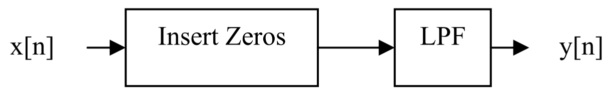
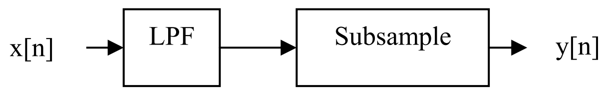


Figure 5.3. Magnitude frequency responses

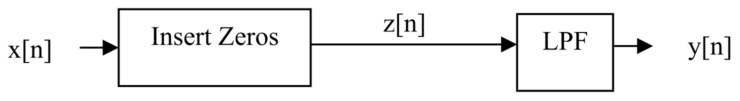
Increase
A multirate system that increases the sampling rate is shown below. The sampling rate change is done in two steps. First, zeros are added between existing samples, and second, the resulting signal is low pass filtered.



Decrease
Decreasing the sampling rate is also done in two steps. First, the signal is low pass filtered, then the signal is subsampled. (Subsampling means throwing samples away. For example, subsampling by two means discarding every other sample.)



Assume we want to increase the sampling rate by two. This means inserting one zero between each sample of x[n], then lowpass filter the resulting signal. Let the signal with the inserted zeros be z[n], as shown below.



The following figures illustrate the procedure.

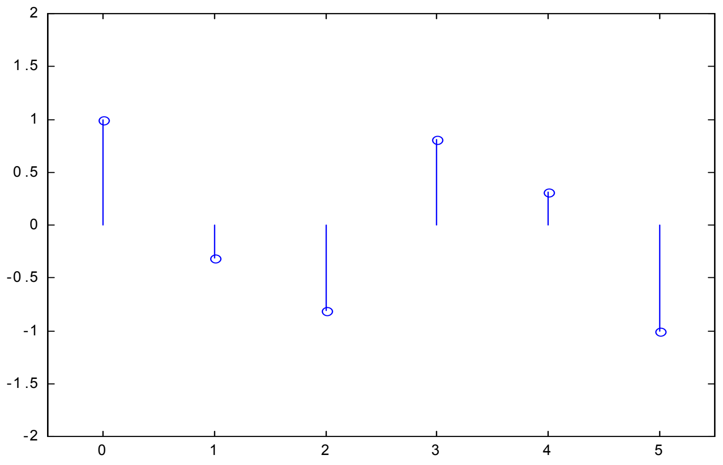


Figure 5.4. Original Signal, x[n].

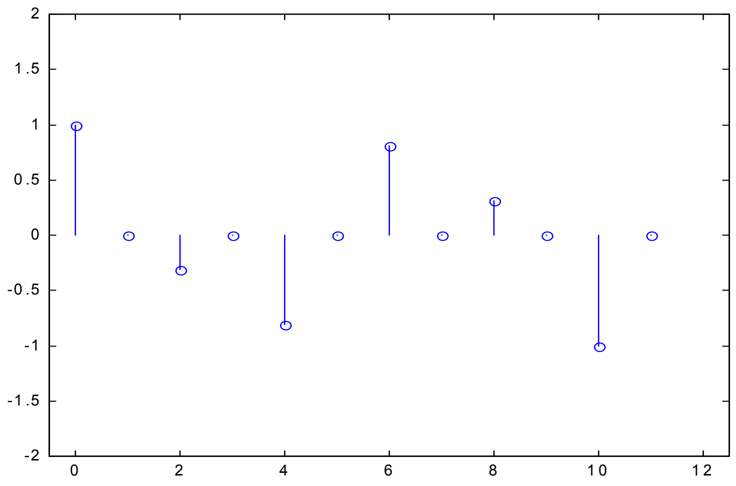


Figure5.4. Signal with added zeros, z[n].

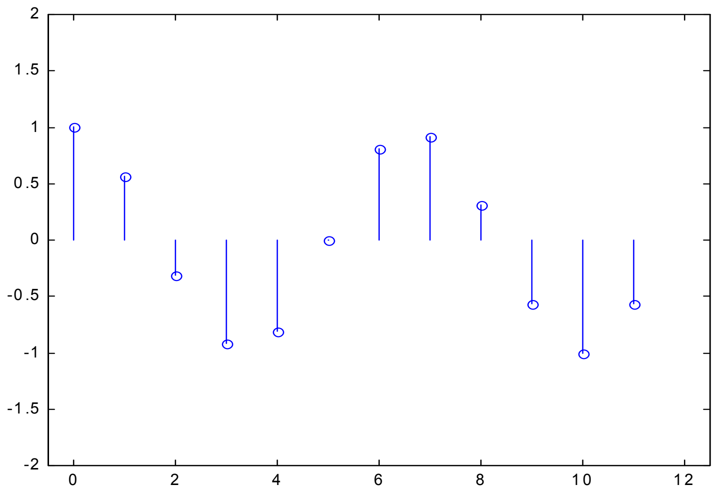


Figure 5.5. After lowpass filter, y[n].

To determine what lowpass filter to use, z[n] is studied.

$$z[n] = \begin{cases} x\left[\frac{n}{2}\right] & n \;\; even \\ 0 & n \;\; odd \end{cases}$$

and

$$Z(\omega) = \sum_{n=-\infty}^{\infty} z[n]e^{-j\omega n} = \sum_{n=even} x\left[\frac{n}{2}\right]e^{-j\omega n} + \sum_{n=odd} 0e^{-j\omega n} = \sum_{k=-\infty}^{\infty} x[k]e^{-j\omega 2k} = X(2\omega)$$

This means that Z(ω) repeats twice as fast as X(ω). Since x[n] is a discrete signal, X(ω) repeats every 2π. Also, since we are working with real signals, the magnitude of X(ω) is even, and the phase is odd. An example signal is shown in figure 5.6.

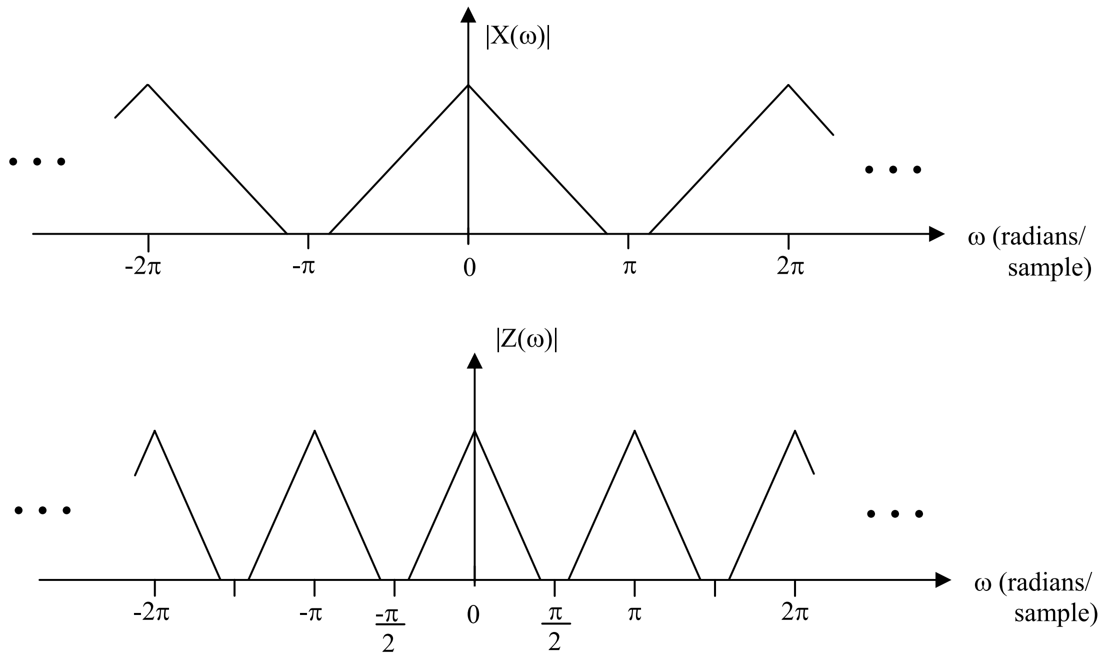


Figure 5.6. Magnitude of Frequency Response of x[n] and z[n].

Compare figure 5.6 with figure 5.3. We need to process z[n] so that the output is like $x_2[n]$. We need a lowpass filter with a cutoff frequency of π/2. The output is shown in figure 5.7.

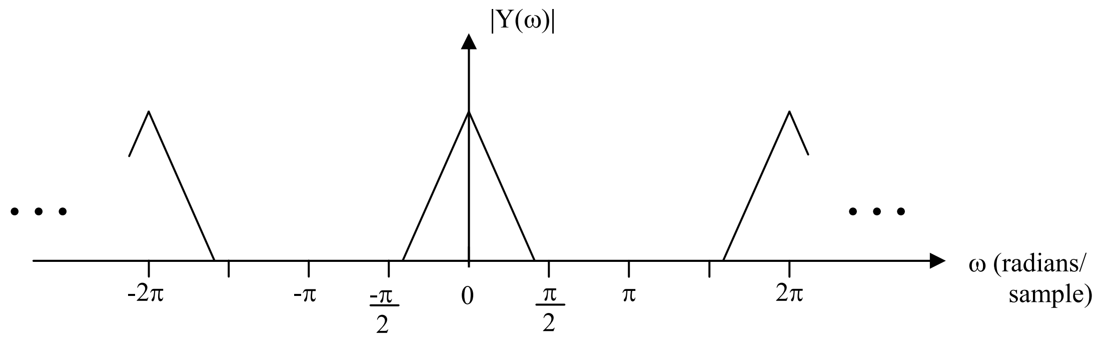


Figure 5.7. Magnitude of Frequency Response of y[n].

Although not obvious at first sight, the LPF must have a gain of *2* for y[n] to have the same amplitude as x[n].

When implementing this on the DSK, the frequencies for A/D and D/A are different. In this case, the D/A frequency is twice that of the A/D. The control register of the AIC must be set to asynchronous mode in order for the A/D and D/A frequencies to be different. This is the same register used in experiment 2 when including and removing the anti-aliasing filter in the AIC.

The anti-aliasing and reconstruction filters of the AIC can obscure the LP digital filter. To see how well the LP digital filter is working, remove the anti-aliasing filter from the input, and set the reconstruction filter as high as possible.

Experiments

Experiment 5.1
Modify *loopc.c* to output an input sample then three zeros. Set the AIC to asynchronous mode. Set the input sampling frequency close to 4kHz, and the output sampling frequency close to 16kHz. The output frequency must be **exactly** four times the input frequency. Set the output filter bandwidth as high as possible, and remove the input filter. Input a frequency of 1kHz and observe the output.

Experiment 5.2
Modify the program developed in experiment 5.1 to implement a multirate system that increases the sampling rate by four. Input a frequency of 1kHz and observe the output. Compare the output of this signal with the output of experiment 5.1.

Skills: After completing this experiment you should:

- Understand how multirate systems work
- Be able to implement a multirate system on the DSK

6. Audio Effects (Echo, Reverb, Chorus, Flanging, etc.)

Purpose

The purpose of this experiment is to play with audio effects.

Background

A short description of echo, reverb, chorus and flanging is given below, but there are many useful links on the web.

Echo

An echo is a delayed, attenuated version of a signal. The physical interpretation is shown in Figure 6.1.

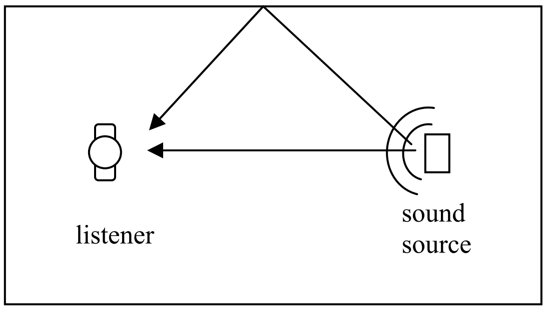
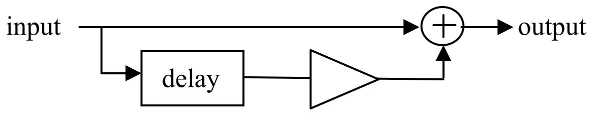


Figure 6.1. Direct and Reflected sound

The reflected sound has a longer path, so it reaches the listener later, and it bounces off a wall, which absorbs some sound. A block diagram of a system with echo is shown below



This can be implemented as an FIR filter with impulse response,

$$h[n] = \delta[n] + a\delta[n-m]$$

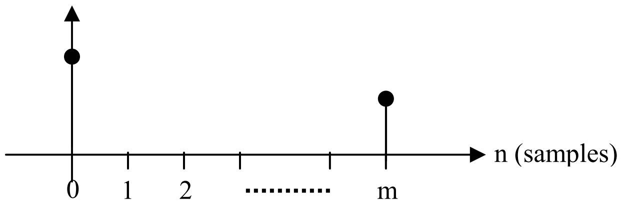


Figure 6.2. Impulse response of echo filter

The attenuation is *a* ($0<a<1$), and the delay is *m* samples. If the delay is very short, we do not hear it as an echo, but as part of the direct sound. To hear the delayed part as a separate signal, the delay should be at least 300ms.

Reverb

Reverberation, or reverb for short, is a simulation of sound in an enclosure. It can be thought of as many densely packed echoes. The impulse response looks something like this,

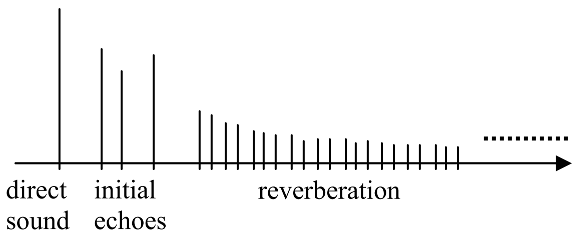
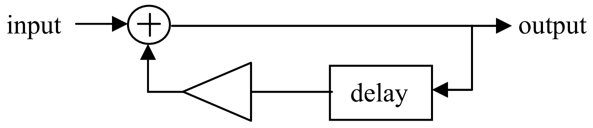


Figure 6.3. Impulse response of room showing echoes and reverb

Assume we are in a room as shown in Figure 6.1. The sound emitted by the source generally goes in all directions. This sound bounces off the room boundaries, and can take many paths in getting to the listener. These are the initial echoes. The sound does not stop when it gets to the listener, but keeps bouncing off the room boundaries. With each bounce the sound looses some power, and has a longer path to the listener.
It would take a very long FIR filter to implement a good sounding reverb, so it is usually implemented with IIR filters. A simple system is shown below.



The impulse response will be infinite, and if the feedback is less than one, exponentially decaying as shown below.

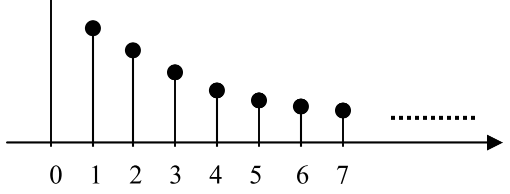


Figure 6.5. Impulse response of simple reverb filter

A general form for IIR filter transfer functions is

$$H(z)=\frac{b_0+b_1 z^{-1}+\cdots+b_M z^{-M}}{1+a_1 z-1+\cdots+a_N z^{-N}}$$

A block diagram of an implementation is,

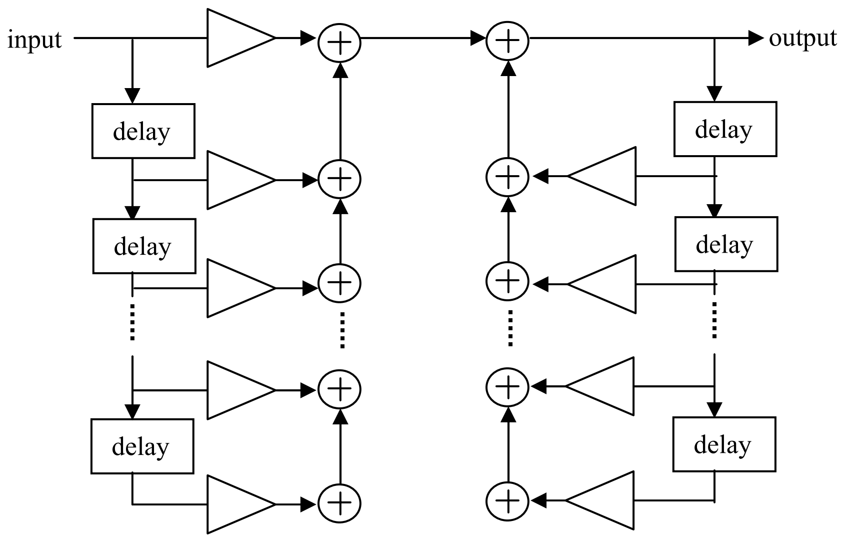
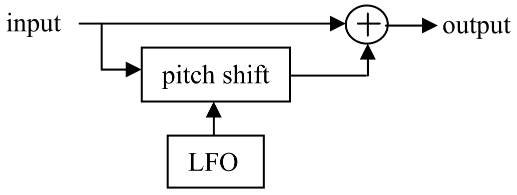


Figure 6.6. Block diagram of a general IIR filter

With this general model there can be feedback with different delays. The work is choosing the coefficients so that it sounds good.

Chorus

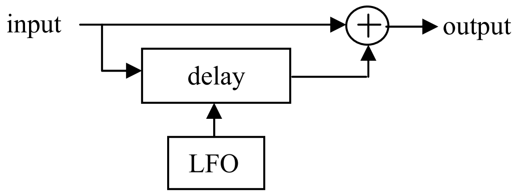
A block diagram of the chorus effect is shown below.



The LFO is a low frequency oscillator. The output is the direct sound plus a pitch shifted version, where the amount of pitch shift changes with time.

Flanging

Flanging is implemented by splitting a signal, delaying one of the signals with respect to the other, then adding them together. A block diagram is shown below.



The flanging effect was named after the way it was implemented with reel-to-reel machines. Two tapes of the same thing were played on two different machines. They were started at the same time, and the operator slowed one of them down by placing a thumb on the outside, or flange, of the reel. First one, then the other machine was slowed down, so that the relative delay was constantly changing.

The effects described above are not the only effects used in practice, there are many more. You can also implement an effect you make up - *let your imagination run wild.*

The main problem in implementing these effects will be the limited memory. These are time based effects, so for example, the echo will be more noticeable the longer the delay time. The DSK has 2k words of memory, and this has to be used for both the program and data. Say 1k is reserved for data, so that 1024 data can be stored at a time. If we have a sampling rate of 16kHz, 1024 samples is only 64m seconds. This is generally too short to be heard. With a lower sampling rate, the delay will be longer.

Experiment

a. Modify loopc.c to implement an echo effect. Set the sampling rate to 6kHz to increase the amount of delay. See how much you can increase the delay time. Input music from a CD into the DSK and listen to the output.
b. Select one other effect to implement.

Skills: Upon completing this experiment you should:

- Have knowledge of various audio effects
- Know how implement an echo effect

7. Signal Quantization and Coefficient Precision (Round-off) effects

Purpose

The purpose of this experiment is to see the effects of signal quantization and coefficient precision.

Background

Any implementation in hardware must take into account precision. Both the signal and the system are represented in binary numbers in the DSK and have finite precision. An in-depth study of precision requires a knowledge of stochastic processes, so we will only look at its effects here. How does the number of bits used to represent a signal effect the sound? What is the effect of using fewer and fewer bits for the coefficients in a filter? What effect does using fewer and fewer bits for the multiplications and additions have on the result? In this experiment, you will first listen to signal quantization, then study system precision.

Quantization

When a signal is converted from analog to digital, it is represented using bits. This representation by a finite number of bits is called *quantization*. The number of bits used for each sample determines the amount of quantization done. A sample represented with 8 bits has more precision than one represented with 4 bits. The precision needed depends on the application.

The process of going from an analog to a digital signal can be divided into two steps, as shown in figure 7.1.

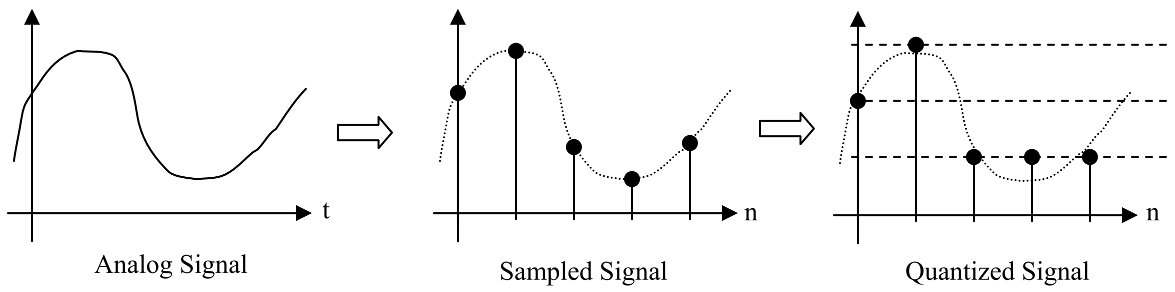


Figure 7.1. Sampling and Quantization Processes

The first step is sampling, the second is quantization. From the Nyquist theorem, we know that if a signal is sampled at more than twice the highest frequency, then the signal can be recovered exactly. This means that this step is reversible, and we could theoretically go back to the original signal exactly. The quantization step is rounding off, and this is irreversible. We are distorting the signal by doing quantization,so we cannot recover the original signal. One way of modeling quantization is as additive noise shown in figure 7.2.

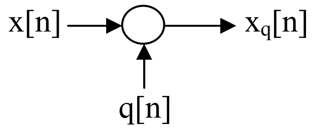


Figure 7.2. Model of Quantization

In this model, x[n] is the sampled signal, q[n] is the quantization noise, and $x_q[n]$ is the quantized signal.

Lets study this process of quantization within a mathematical framework. Assume each number is represented using *n* bits, and that the magnitude between quantizatin levels is Δ (see figure 7.3). With *n* bits, there are 2^n quantization levels. Twos complement binary is typically used for representing the levels, so with *n* bits and uniform quantization, the levels go from $(-2^{n-1})\Delta$ to $(2^{n-1}-1)\Delta$. (2^n total). For example, if *n*=2, then there are four quantization levels, and using twos complement the levels go from -2Δ to 1Δ.

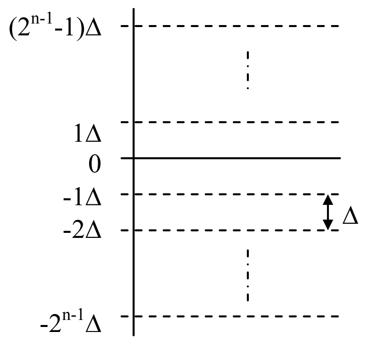


Figure 7.3. *n* quantization levels.

Since Δ is the magnitude between quantization levels, the maximum difference between x[n] and $x_q[n]$ within the quantization levels is Δ/2. In other words, *q[n]* is bounded,

$$-\frac{\Delta}{2} < q[n] < \frac{\Delta}{2}.$$

Since $x_q[n]$ can only take values between $(-2^{n-1})\Delta$ and $(2^{n-1}-1)\Delta$, it is also bounded. It is generally desirable for q[n] to be always bounded by Δ/2, not only within the quantization levels. This can be done by bounding the signal by

$$\left(-2^{n-1}\right)\Delta - \frac{\Delta}{2} < x[n] < \left(2^{n-1}-1\right)\Delta + \frac{\Delta}{2}.$$

This mathematical framework can be used to determine how many bits are needed depending on the signal quality desired. Quality in signals is usually measured as signal to noise ratio in dB. In this case, the quantization is the noise, so the signal to noise ratio is defined as

$$SNR = 10\log_{10}\frac{Power \quad in \quad Signal}{Power \quad in \quad Noise}.$$

If the signal is a sine wave, Acos(ωn), then its average power is $A^2/2$. For the example of figure 7.3, the maximum power in the signal is,

$$P_x = \frac{\left(\left(2^{n-1}-1\right)\Delta+\frac{\Delta}{2}\right)^2}{2} = \frac{\left(2^{n-1}\Delta-\Delta+\frac{\Delta}{2}\right)^2}{2} \approx \frac{\left(2^{n-1}\Delta\right)^2}{2} = 2^{2n-3}\Delta^2.$$

The quantization is noise, and cannot usually be expressed with an equation. It is generally modeled as a stochastic process. If we assume that the quantization is independent of the signal, white, and uniformly distributed from -Δ/2 to Δ/2, then the average power in the quantization is calculated from its variance.

$$Variance \quad of \quad q[n] = \sigma_q^2 = E\{q^2[n]\} - E^2\{q[n]\}$$

$$= \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} \alpha^2 f_q(\alpha)d\alpha = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} \alpha^2\frac{1}{\Delta}d\alpha = \left[\frac{\alpha^3}{3}\frac{1}{\Delta}\right]_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} = \frac{\left(\frac{\Delta}{2}\right)^3 - \left(-\frac{\Delta}{2}\right)^3}{3}\frac{1}{\Delta} = \frac{\Delta^2}{12}.$$

The signal to noise ratio after quantization is then,

$$SNR = 10\log_{10}\frac{2^{2n-3}\Delta^2}{\frac{\Delta^2}{12}} = 10\log_{10}\frac{2^{2n-3}}{\frac{1}{12}} = 10\log_{10}\left(12 * 2^{2n-3}\right)$$

The DSK quantizes using 14 bits, so it can have a maximum SNR of 86.05dB. This does not mean every signal will have a SNR of 86.05dB, this is the SNR of a sine wave using the maximum input amplitude. If we input a small signal, then some of the 14 bits will always be zero, and the SNR will be smaller.

The equation for SNR can be approximated as,

$$SNR = 10\log_{10}\left(12 * 2^{2n-3}\right) \approx 6.02n + 1.78.$$

This means that there is about a 6dB gain in SNR for every additional bit used. This gives an easy way to relate how many bits are needed with the desired signal quality. For example, it was determined that at least 45dB SNR was required for people to pay for telephone service. This was the basis for choosing 8 bits for the system. Another example is the CD. Taking into account the ambient noise in a home, and the maximum listening levels, 90-100dB was considered sufficient for music reproduction. A CD uses 16 bits and so has a SNR of about 98.1dB. In the new DVD-Audio, the signal can be represented from 16 to 24 bits. 24 bits means a theoretical SNR of 144dB. 144dB is a lot! There is no analog circuit that has noise this low. The thermal noise in one resistor is more than this, so this SNR is not implementable unless the system is run close to absolute zero. There is anecdotal evidence of this happenning to some students during dates, but no concrete proof.

System Precision

In this experiment the system is a filter, so the *system* precision is determined by the number of bits used to represent the filter coefficients and the number of bits used in arithmetic operations such as multiplication and addition. Again, how much precision is needed depends on the application. Find out what precision is used in MATLAB, and what precision is available on the DSK.
Addition or multiplication using finite registers means the result will be rounded off. This can be viewed similarly to quantization, where the rounding is modeled as additive noise.

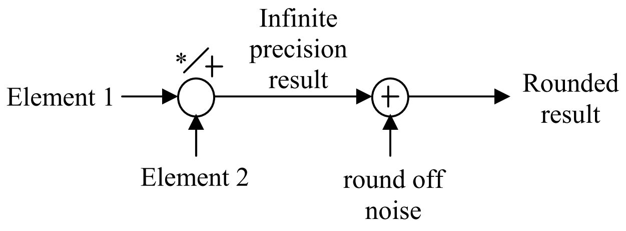


Figure 7.4. Model of round off noise in addition or multiplication

Experiments

Experiment 7.1

a. Calculate the maximum and minimum voltages the AIC can accept without clipping.
b. Calculate the voltage difference between quantization levels, Δ, for the AIC.
c. Connect the function generator to the input, and use either *loopi.asm* or *loopc.c* to listen to the output. Using the oscilloscope, make sure you are using the full input and output voltages of the AIC. Quantize an audio signal starting at 14 bits and gradually reduce the number of bits used. At what bit level does the quantization become audible?
d. Change the input to music and repeat the experiment. Is the quantization audible at the same bit level?

Experiment 7.2

a. Design a 10th order BP Chebychev type 1 filter with 0.5dB of ripple and a passband from .2 to .5 rad/sam in MATLAB. Reduce the coefficient precision to four decimal places using Direct Form I and Cascade Form. Plot the magnitude bode plot and pole-zero plots for the three filters. How do they compare? Which implementation is more sensitive to coefficient round off?

b. Implement the Direct Form I filter modifying loopc.c

c. Implement the Cascade form filter modifying loopc.c

Skills: Upon completing this experiment you should:

- Have a working knowledge of the relation between how many bits are used to represent a signal and how it sounds
- Know different methods of implementing filters and the advantages of each
- Know what effect implementation has on system performance
- Know what effect precision has on system performance

8. Adaptive filter and FFT implementation

Purpose

The purpose of this experiment is to see two advanced signal processing routines running in real time, and to use two inputs on the DSK.

Background

Adaptive Filters

Adaptive filter theory is extensive, and is not usually covered in an undergraduate class. We will look at one specific adaptive filter algorithm, the LMS algorithm, along with a FIR filter. The typical system implementation has two inputs, one containing the desired signal and noise, the other only noise. A block diagram of the system is shown in Figure 8.1.

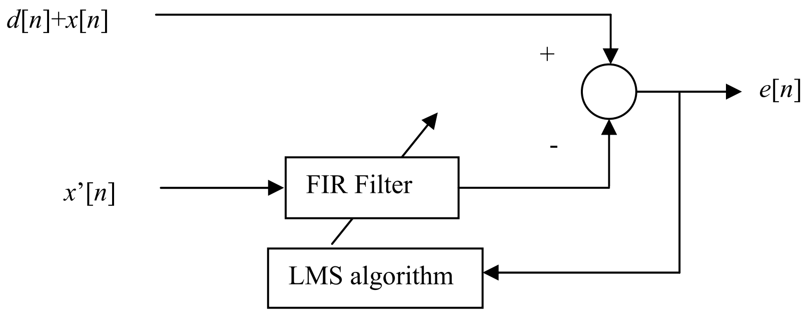


Figure 8.1. An adaptive filter to cancel noise.

The basic idea is to use $x'[n]$ to reduce $x[n]$. The LMS algorithm changes the FIR filter coefficients trying to make the output zero. For the algorithm to work, $x'[n]$ and $x[n]$ must be statistically related and orthogonal to the desired signal $d[n]$. The output of the FIR filter is a linear combination of $x'[n]$, and so it is also statistically related to $x[n]$. The output of the FIR filter can cancel part of $x[n]$ since they are statistically related, but cannot cancel $d[n]$ since they are orthogonal.

Putting this in a mathematical framework, we can proceed as follows.
Try and make the output error, $e[n]$, zero.
Assumptions: $d[n]$ is orthogonal to $x[n]$ and $x'[n]$.
$x[n]$ and $x'[n]$ have a nonzero correlation

The output error is,

$$e[n] = d[n] + x[n] - \sum_{k=0}^{N-1} w_n[k]x'[n-k]$$

We want to choose $w_n[k]$ so that $e[n]$ is zero. (Note that the weights have a subscript because they can change each iteration.) We cannot usually make $e[n]$ zero, so we need some way of determining how well we are doing. In this algorithm, the expected value of the squared error is used as a performance measure. Since we try to minimize the expected value, or mean, of the squarred error, the algorithm is called Least Mean Square (LMS). The Mean Squarred error (MSE) is:

$$E\left\{[e[n]]^2\right\} = E\left\{\left[d[n] + x[n] - \sum_{k=0}^{N-1} w_n[k]x'[n-k]\right]e[n]\right\}$$

$$= E\{d[n]e[n]\} + E\{x[n]e[n]\} - E\left\{\sum_{k=0}^{N-1} w_n[k]x'[n-k]e[n]\right\}.$$

The idea is to change the weights at each iteration to reduce the MSE. Since we are trying to minimize the MSE, we take the partial derivative with respect to the weights. To simplify the problem, we assume that $e[n]$ is not a function of $w_n[k]$. The derivative is

$$\frac{\partial E\left\{[e[n]]^2\right\}}{\partial w_n[q]} = -E\{x'[n-q]e[n]\}.$$

We will use the instantaneous value as an approximation to the expected value, giving

$$-E\{x'[n-q]e[n]\} \approx -x'[n-q]e[n].$$

An estimate of the gradient can be used to determine an equation for each weight. If the step size is small, then

$$\frac{\partial E\left\{[e[n]]^2\right\}}{\partial w_n[q]} \approx \frac{w_n[q] - w_{n+1}[q]}{\beta}$$

Solving for $w_{n+1}[k]$, and substituting from above we get

$$w_{n+1}[q] = w_n[q] - \beta\frac{\partial E\left\{[e[n]]^2\right\}}{\partial w_n[q]} = w_n[q] + \beta x'[n-q]e[n].$$

There will be one equation for each w that is updated each iteration.

This is the first implementation where we need two inputs. The AIC used by the DSK has two inputs, the main input connected to the RCA jack, and the secondary input on pin 3 of JP3. The input must be selected in the control register of the AIC before a sample can be taken. This means that the control register must be changed after every sample. The procedure is as follows: set the AIC to receive samples from the main (RCA) input, one input is taken, then set the AIC to receive input from the secondary input, one input is taken, and so forth. Since samples are taken alternately from the primary and secondary inputs, the sampling rate of each is half the set sampling rate. For example, if the sampling rate is set to 16kHz, each input will have a sampling rate of 8kHz.

The variable parameters are the *filter length*, and the *adaptation coefficient*, *β*. Longer filter lengths allow the filter to cancel more complex signals. The adaptation coefficient determines how fast the filter coefficients change. If the adaptation coefficient is close to zero, then the filter coefficients change slowly, and may not be able to cancel the signal if it is varying rapidly. If the adaptation coefficient is close to one, then the filter coefficients change fast, and the system may adapt not only to the signal, but to other disturbances. Selecting the coefficient is a tradeoff between convergence rate and disturbance rejection.

In this experiment, we will use two signal generators, one sine wave as signal, another sine wave as the noise to be cancelled. Connect the signal generators, and run the program. Change the frequency of the sine wave used as noise, and see how it is cancelled. Change the adaptation coefficient, and see how the time to cancel the signal changes. Change the order of the filter and see how the cancellation changes.

Fast Fourier Transform (FFT)

In this experiment you will implement a FFT in real time. The FFT is one of the most used algorithms in signal processing, and can be used for a wide variety of tasks such as implementing filters and determining the spectral content of signals. The FFT program you will implement is divided into two parts: a C program that reads and writes from the AIC and calculates the magnitude of the FFT, and an assembly subroutine that implements the FFT. The FFT was written in Assembly to speed up the implementation. It is easier to write programs in C, but programs in Assembly are faster. To avoid writing the whole program in Assembly, the main part of the program is in C, with computationally intensive parts in Assembly, where the difference between writing in C and Assembly is greatest.

To change the size of the FFT program, the memory use may have to change. The fft12c.cmd program defines the memory usage as follows:

A C program will have three memory areas, the *text*, the *stack* and the *heap*. The text area is where the program is located, the stack is where local variables are saved, and the heap is where global and static variables are saved. When the program is started, space for the global and static variables is allocated in the heap. When main() is started, space for its local variables is allocated on the stack. If a subroutine subr() is called, the arguments passed to the subroutine are stored on the stack and space for local variables from subr() is allocated on the stack. When the subroutine ends, space for the local variables from subr() is de-allocated, and the stack returns to its previous size.
When changing the size of the FFT, we must have enough space on the stack and heap to save the data. To determine the size of the stack and heap, determine the space needed for the global and local variables.

Experiments

Experiment 8.1
Assemble and run *adapter.asm*. Input a cosine as the signal, and another cosine as the noise. Change the adaptation coefficient and the filter length and see the effect.

Experiment 8.2
Compile, assemble, link and run *fft128c.c*. Input a sine wave.

a. Determine the relation between the signal amplitude and frequency and the fft output.
b. Look at the shape of the fft output. Slowly change the frequency of the input by small amounts and notice how the sidelobes change. Why does this happen?
c. Try to increase the size of the FFT. To do this you will have to change the memory allocation in the linker, have new twiddle factors, and change the length and stages variables of the main program. What is the largest FFT you can implement?

Skills: Upon completing this experiment you should:

- Have an idea of how LMS adaptive filters work
- Know what effect the adaptation coefficient has on the system
- Know what effect changing the filter length has on the system
- Know how to use the auxiliary input of the AIC
- Understand the FFT algorithm

9. Integrating Assembly Subroutines with programs in C

Purpose

The purpose of this experiment is to learn how to integrate subroutines written in Assembly with programs in C.

Background

In this experiment you will implement a mixed C/Assembly program that implements a filter. Part of the program will be written in C, the other part in Assembly. You will need to know how to call assembly subroutines from C, and how to define variables. The variables can be defined in C or in Assembly, but must be accessible to both parts of the program. This section is divided into three parts:

- Calling Assembly subroutines from C
- Using variables declared in C in the Assembly Subroutine
- Using variables declared in the Assembly Subroutine in C

Calling Assembly subroutines from C

The FFT program in the previous experiment is an excellent example of how to implement programs in C with subroutines in Assembly. Another simpler example is shown in Figures 8.1 and 8.2.

In this example, the input and output is done in C. The input value is passed to the subroutine, the subroutine calculates the absolute value and returns this new value to the C program. Let us go through the program step by step to see the main points of writing subroutines in assembly from C.

The subroutine must be declared as external in the C program, and as global in the Assembly program. This name must have an underscore (_asmab) in the Assembly subroutine because it is defined in the C program. In general, any variable in the C program can be made available in the Assembly subroutine if it is defined this way.

```
************************************************************************
    /* casm.c -  C PROGRAM with subroutine in assembly */

    #include "aiccom.c"                              /*AIC Communication routines */

    int AICSEC[4] = {0x0A14,0x1,0x3E7E,0x63};
    extern int asmab(int);                           /* Define the external Assembly*/
                                                     /* subroutine */

    main()
    {
        int  data_in, data_out;                      /*Initialize variables  */

        AICSET();                                    /*Function to config AIC*/

        while (1)                                    /*Create endless loop   */
        {
             data_in = UPDATE_SAMPLE(data_out);      /*Call function to update sample*/
             data_out=asmab(data_in);                /* Call assebly subroutine */
        }
    }
************************************************************************
```

Figure 8.1. C program that calls an Assembly Subroutine

```
************************************************************************
        ;Assembly language program called from casm.c.
        ;It returns the absolute value of the value passed from C.

    FP  .set     AR3
        .global  _asmab          ;declare external function

    _asmab:
        push     FP              ;save frame pointer on stack
        ldi      SP,FP           ;point to top of stack
        push     R6              ;save R6
        push     R7              ;save R7

        LDI      *-FP(2),R6      ;new sample -> R6
        ABSI     R6,R7           ;place absolute value of R6 in R7
        LDI      R7,R0           ;place output in R0

        pop      R7
        pop      R6
        pop      FP
        rets
************************************************************************
```

Figure 8.2. Assembly subroutine called from casm.c

The subroutine executes the following commands:

```
        push    FP              ;save frame pointer on stack
        ldi     SP,FP           ;point to top of stack
```

The frame pointer is pushed onto the stack, and is then used as the local frame pointer. SP is the stack pointer, and points to the top of the stack. AR3 is the frame pointer (defined here as FP), and points to the beginning of the local frame. See Figure 8.3 below.

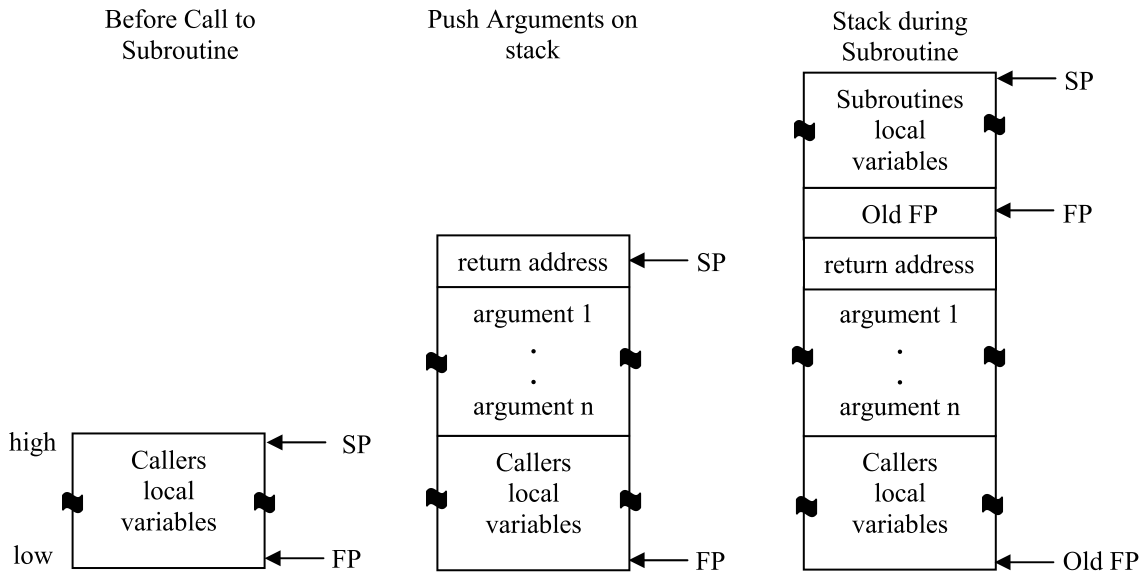


Figure 8.3. Stack use during a function call. (from TI *C Compiler User's Guide*, 1997)

```
        push    R6              ;save R6
        push    R7              ;save R7
```

Registers R6 and R7 are used in the subroutine, so their values are also saved to the stack before using. In general, registers R4-R7 and AR4-AR7 must be saved before using, and restored before returning. If other registers are used, they do not need to be saved.

```
        LDI     *-FP(2),R6      ;new sample -> R6
```

The sample passed from the C program to the subroutine is placed in R6. When calling asmab(), the C program writes data to the stack, including the values, or arguments, to be sent to the subroutine. These arguments can be accessed by the subroutine using a relative address, in this case, *-FP(2). If there is more than one argument sent, the first argument will be at *-FP(2), the second at *-FP(3), and so forth. See Figure 8.3 above.

```
        ABSI    R6,R7           ;place absolute value of R6 in R7
        LDI     R7,R0           ;place output in R0
```

The absolute value is calculated, and the result placed in R0. R0 is used for returning arguments from the subroutine to the C program. In the C program, data_out is assigned the value returned from the subroutine, so data_out = R0.

```
pop     R7
pop     R6
pop     FP
rets
```
The original values of R6, R7 and the frame pointer are read from the stack, and the subroutine ends with a return command.

Using variables declared in C in the Assembly Subroutine

There are various ways of having variables declared in C available to the Assembly subroutine. Some of these are:

- Sending variables as arguments to the subroutine
- Sending a pointer to an array
- Declaring the variables as global

The first method was shown in the example above. The FFT program from Experiment 8 uses a different approach. Since the FFT has length 128, it would not be practical to send the 128 data points as arguments to the subroutine. Instead, the FFT program sends a pointer to the array used. An example showing both how to pass a pointer to an array and how to declare variables as global is shown in figures 8.4 and 8.5.

```
**********************************************************************
/* c_prog.c -  C PROGRAM with subroutine in assembly */

#include "aiccom.c"                              /*AIC Communication routines */
int AICSEC[4] = {0x0A14,0x1,0x3E7E,0x63};
extern int asmarray(float *,int *);              /* Define the external Assembly
                                                 subroutine */
float x[20];                                     /* Define global
int    y[20],cvar;                                  variables */

main()
{
    int  data_in, data_out;                      /*Define local variables  */
    AICSET();                                    /*Function to config AIC*/
    x[2]=1.5;
    y[0]=2;

    while (1)                                    /*Create endless loop   */
    {
         data_in = UPDATE_SAMPLE(data_out);      /*Call function to update sample*/
         data_out=asmarray((float *)x,(int *)y); /* Call assebly subroutine */
    }
}
**********************************************************************
```

Figure 8.4. C program that calls an Assembly Subroutine

```
**********************************************************************
    ;Assembly language program called from c_prog.c.
    ;It multiplies x[2] with y[0], then adds cvar.

FP  .set      AR3
    .global  _asmarray        ;declare external function
    .global  _cvar            ;declare external variable

_asmarray:
    push           FP               ;save frame pointer on stack
    ldi            SP,FP            ;point to top of stack
    push           R6               ;save R6
    push           R7               ;save R7
    LDI            *-FP(2),AR0      ;pointer to x[0] -> AR0
    LDI            *-FP(3),AR1      ;pointer to y[0] -> AR1

    FLOAT          *AR1,R7          :convert y[0] from int to float -> R7
    MPYF3          *AR0(2),R7,R6    ;multiply x[2] with R7 -> R6
    FIX            R6,R0            ;convert from float to int, place in R0
    LDI            @_cvar,AR2       ;cvar -> AR2
    ADDI           AR2,R0           ;AR2 + R0 -> R0

    pop            R7
    pop            R6
    pop            FP
    rets
**********************************************************************
```

Figure 8.5. Assembly subroutine called from c_prog.c

Using variables declared in the Assembly Subroutine in C

Most variables will be defined in C and used in both C and Assembly. One situation where it is more convenient to define variables in Assembly is when we want to use circular addressing. The data must be aligned on a boundary so that the circular addressing will work correctly. This is easily done in Assembly.

As mentioned in the *TMS320C3x/C4x Optimizing C Compiler User's Guide*, there are three ways of accessing Assembly variables in C. Only one will be mentioned here, refer to the User's Guide for the other two. The example below is from the User's Guide.

```
*********************************************************************
        .global         _sine           ;Declare variable as global
        .usect          "sine_tab",4    ;Make a separate section for the sine variable
_sine:
        .float  0.0
        .float  0.015987
        .float  0.022145
*********************************************************************
```

Figure 8.6. Assembly subroutine

```
*********************************************************************
extern float sine[];            /* Define external object */
float   *sine_p = sine;         /* Declare pointer to sine */

f = sine_p[1];                  /* Access sine as normal array */
*********************************************************************
```

Figure 8.7. C Program

Note: The variable cannot be accessed using code like *sine[1]*.

As mentioned above, the main reason for using this type of declaration is to be able to put the variable at a segment boundary, and use circular addressing with no problem. Once the section is defined using *.usect*, space for this section must be separated in in the linker command file. Part of the cmd file is shown in Figure 8.8.

```
****************************************************
    MEMORY
    {
     VECS:      org = 0               len = 0x40       /*INTERRUPT VECTORS*/
     SRAM:      org = 0x40            len = 0x3FC0     /*USER STATIC RAM*/
     RAM0:      org = 0x809800        len = 0x750      /*INTERNAL RAM */
     RAM1:      org=0x809FB0          len=0x50         /*memory for sine_tab section*/
    }

    SECTIONS
    {
     .text:     {} > SRAM    /*CODE*/
     .cinit:    {} > RAM0    /*INITIALIZATION TABLES*/
     .stack:    {} > RAM0    /*SYSTEM STACK*/
     .bss:      {} > RAM0    /*BSS SECTION*/
     sine_tab   {} > RAM1    /*sine_tab section*/
     vecs:      {} > VECS    /*VECTOR SECTION*/
    }
****************************************************
```

Figure 8.8. cmd file to place sine_tab section where desired.

Experiment

Modify loopc.c to filter a signal with an assembly subroutine. The input and output must be done with updatesample in main(), and the rest of the processing done in the Assembly subroutine.

Skills: Upon completing this experiment you should:

- Know how to implement and call subroutines in Assembly from C using the DSK
- Be a better person
- Have good moves
- Dedicate your lives to things of importance
- Have lots of money