ECE 320 Spring 2004

Collection Editors:

Robert Morrison
Jason Laska

ECE 320 Spring 2004

Collection Editors:

Robert Morrison
Jason Laska

Authors:

Swaroop Appadwedula
Matthew Berry
Mark Butala
Richard Cantzler
Michael Frutiger
Mark Haun
Jake Janovetz
Douglas L. Jones
Matt Kleffner
Michael Kramer
Arjun Kulothungun
Jason Laska
Robert Morrison
Dima Moussa
Daniel Sachs
Brian Wade

Online:
< http://cnx.org/content/col10225/1.12/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

Weekly Labs

1.1 Lab 0

1.1.1 Lab 0: Hardware Introduction[1]

1.1.1.1 Introduction

This exercise introduces the hardware and software used in testing a simple DSP system. When you complete it, you should be comfortable with the basics of testing a simple real-time DSP system with the debugging environment you will use throughout the course. First, you will connect the laboratory equipment and test a real-time DSP system with pre-written code to implement an eight-tap (eight coefficient) **finite impulse response** (**FIR**) filter. With a working system available, you will then begin to explore the debugging software used for downloading, modifying, and testing code. Finally, exercises are included to refresh your familiarity with MATLAB.

1.1.1.2 Lab Equipment

This exercise assumes you have access to a laboratory station equipped with a Texas Instruments TMS320C549 digital signal processor chip mounted on a Spectrum Digital TMS320LC54x evaluation board. The DSP evaluation module should be connected to a PC running Windows and will be controlled using the PC application Code Composer Studio, a debugger and development environment. Mounted on top of each DSP evaluation board is a Spectrum Digital surround-sound module employing a Crystal Semiconductor CS4226 codec. This board provides two analog input channels and six analog output channels at the CD sample rate of 44.1 kHz. The DSP board can also communicate with user code or a terminal emulator running on the PC via a serial data interface.

In addition to the DSP board and PC, each laboratory station should also be equipped with a function generator to provide test signals and an oscilloscope to display the processed waveforms.

1.1.1.2.1 Step 1: Connect cables

Use the provided BNC cables to connect the output of the function generator to input channel 1 on the DSP evaluation board. Connect output channels 1 and 2 of the board to channels 1 and 2 of the oscilloscope. The input and output connections for the DSP board are shown in Figure 1.1 (Example Hardware Setup).

[1]This content is available online at <http://cnx.org/content/m11019/2.7/>.

Example Hardware Setup

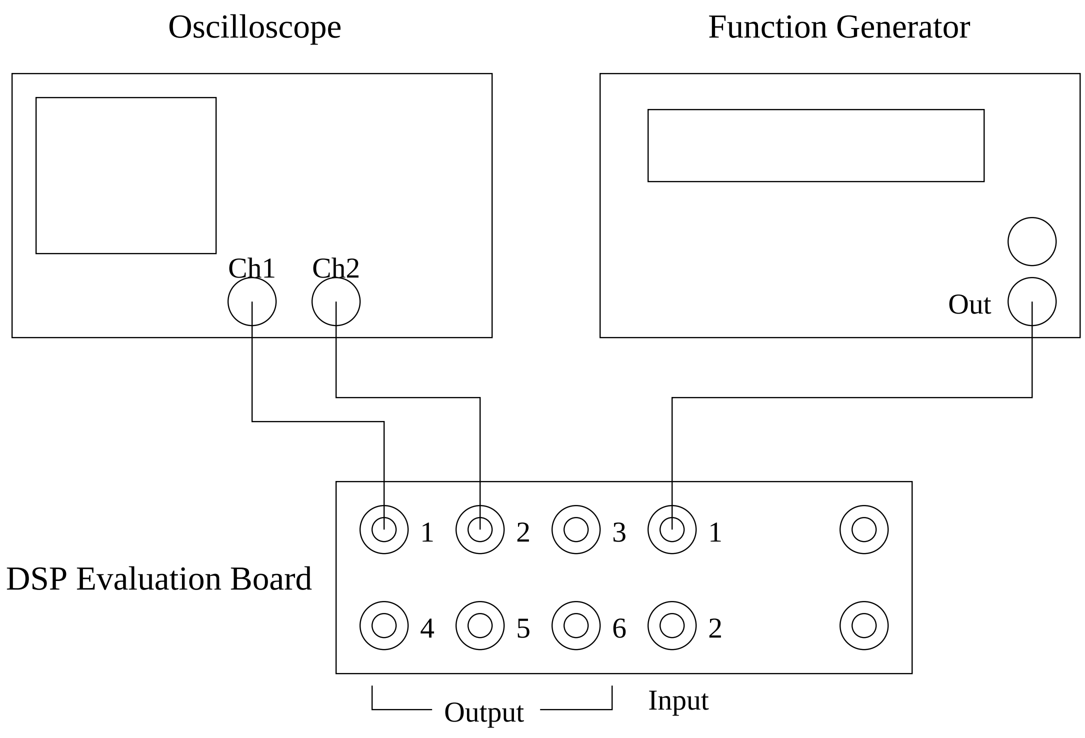


Figure 1.1

Note that with this configuration, you will have only one signal going into the DSP board and two signals coming out. The output on channel 1 is the filtered input signal, and the output on channel 2 is the unfiltered input signal. This allows you to view the raw input and filtered output simultaneously on the oscilloscope. Turn on the function generator and the oscilloscope.

1.1.1.2.2 Step 2: Log in

Use the network ID and password provided to log into the PC at your laboratory station.

When you log in, two shared networked drives should be mapped to the computer: the `W:` drive, which contains your own private network work directory, and the `V:` drive, where the necessary files for ECE 420 are stored. Be sure to save any files that you use for the course to the `W:` drive. Temporary files may be stored in the `C:\TEMP` directory; however, since files stored on the `C:` drive are accessible to any user, are local to each computer, and may be erased at any time, do not store course files on the `C:` drive. On the `V:` drive, the directories `v:\ece420\54kx\dsplib\` and `c:\ece420\54x\dsptools\` contain the files necessary to assemble and test code on the TI DSP evaluation boards.

Although you may want to work exclusively in one or the other of lab-partners' network account, you should be sure that both partners have copies of the lab assignment assembly code.

WARNING: Not having the assembly code during a quiz because "it's on my partner's account" is **NOT** a valid excuse!

For copying between partners' directory on `W:` or for working outside the lab, FTP access to your files is available at `ftp://elalpha.ece.uiuc.edu`.

1.1.1.3 The Development Environment

The evaluation board is controlled by the PC through the JTAG interface (XDS510PP) using the application Code Composer Studio. This development environment allows the user to download, run, and debug code assembled on the PC. Work through the steps below to familiarize yourself with the debugging environment and real-time system using the provided FIR filter code (Steps 3, 4 and 5), then verify the filter's frequency response with the subsequent MATLAB exercises (Steps 6 and 7).

1.1.1.3.1 Step 3: Assemble filter code

Before you can execute and test the provided FIR filter code, you must assemble the source file. First, bring up a `DOS` prompt window and create a new directory to hold the files, and then copy them into your directory:

- `w:`
- `mkdir lab0`
- `cd lab0`
- `copy v:\ece420\54x\dsplib\filter.asm .`
- `copy v:\ece420\54x\dsplib\coef.asm .`

Next, assemble the filter code by typing `asm filter` at the `DOS` prompt. The assembling process first includes the FIR filter coefficients (stored in `coef.asm`) into the assembly file `filter.asm`, then compiles the result to produce an output file containing the executable binary code, `filter.out`.

1.1.1.3.2 Step 4: Verify filter execution

With your filter code assembled, double-click on the Code Composer icon to open the debugging environment. Before loading your code, you must reset the DSP board and initialize the **processor mode status register** (**PMST**). To reset the board, select the `Reset` option from the `Debug` menu in the Code Composer application.

Once the board is reset, select the `CPU Registers` option from the `View` menu, then select `CPU Register`. This will open a sub-window at the bottom of the Code Composer application window that displays several of the DSP registers. Look for the `PMST` register; it must be set to the hexadecimal value `FFE0` to have the DSP evaluation board work correctly. If it is not set correctly, change the value of the `PMST` register by double-clicking on the value and making the appropriate change in the `Edit Register` window that comes up.

Now, load your assembled filter file onto the DSP by selecting `Load Program` from the `File` menu. Finally, reset the DSP again, and execute the code by selecting `Run` from the `Debug` menu.

The program you are running accepts input from input channel 1 and sends output waveforms to output channels 1 and 2 (the filtered signal and raw input, respectively). Note that the "raw input" on output channel 2 may differ from the actual input on input channel 1, because of distortions introduced in converting the analog input to a digital signal and then back to an analog signal. The A/D and D/A converters on the six-channel surround board operate at a sample rate of 44.1 kHz and have an **anti-aliasing filter** and an **anti-imaging filter**, respectively, that in the ideal case would eliminate frequency content above 22.05 kHz. The converters on the six-channel board are also **AC coupled** and cannot pass DC signals. On the basis of this information, what differences do you expect to see between the signals at input channel 1 and at output channel 2?

Set the amplitude on the function generator to 1.0 V peak-to-peak and the pulse shape to sinusoidal. Observe the frequency response of the filter by sweeping the input signal through the relevant frequency range. What is the relevant frequency range for a DSP system with a sample rate of 44.1 kHz?

Based on the frequency response you observe, characterize the filter in terms of its type (e.g., low-pass, high-pass, band-pass) and its -6 dB (half-amplitude) cutoff frequency (or frequencies). It may help to set the trigger on channel 2 of the oscilloscope since the signal on channel 1 may go to zero.

1.1.1.3.3 Step 5: Re-assemble and re-run with new filter

Once you have determined the type of filter the DSP is implementing, you are ready to repeat the process with a different filter by including different coefficients during the assembly process. Copy a second set of FIR coefficients over to your working directory with the following:

- `copy coef.asm coef1.asm`
- `copy v:\ece420\54x\dsplib\coef2.asm coef.asm`

You can now repeat the assembly and testing process with the new filter using the `asm` instruction at the `DOS` prompt and repeating the steps required to execute the code discussed in Step 4 (Section 1.1.1.3.2: Step 4: Verify filter execution).

Just as you did in Step 4 (Section 1.1.1.3.2: Step 4: Verify filter execution), determine the type of filter you are running and the filter's -6 dB point by testing the system at various frequencies.

1.1.1.3.4 Step 6: Check filter response in MATLAB

In this step, you will use MATLAB to verify the frequency response of your filter by copying the coefficients from the DSP to MATLAB and displaying the magnitude of the frequency response using the MATLAB command `freqz`.

The FIR filter coefficients included in the file `coef.asm` are stored in memory on the DSP starting at location (in hex) `0x1000`, and each filter you have assembled and run has eight coefficients. To view the filter coefficients as signed integers, select the `Memory` option from the `View` menu to bring up a `Memory Window Options` box. In the appropriate fields, set the starting address to `0x1000` and the format to `16-Bit Signed Int`. Click "OK" to open a memory window displaying the contents of the specified memory locations. The numbers along the left-hand side indicate the memory locations.

In this example, the filter coefficients are placed in memory in decreasing order; that is, the last coefficient, $h[7]$, is at location `0x1000` and the first coefficient, $h[0]$, is stored at `0x1007`.

Now that you can find the coefficients in memory, you are ready to use the MATLAB command `freqz` to view the filter's response. You must create a vector in MATLAB with the filter coefficients to use the `freqz` command. For example, if you want to view the response of the three-tap filter with coefficients -10, 20, -10 you can use the following commands in MATLAB:

- `h = [-10, 20, -10];`
- `plot(abs(freqz(h)))`

Note that you will have to enter eight values, the contents of memory locations `0x1000` through `0x1007`, into the coefficient vector, `h`.

Does the MATLAB response compare with your experimental results? What might account for any differences?

1.1.1.3.5 Step 7: Create new filter in MATLAB and verify

MATLAB scripts will be made available to you to aid in code development. For example, one of these scripts allows you to save filter coefficients created in MATLAB in a form that can be included as part of the assembly process without having to type them in by hand (a very useful tool for long filters). These scripts may already be installed on your computer; otherwise, download the files from the links as they are introduced.

First, have MATLAB generate a "random" eight-tap filter by typing `h = gen_filt;` at a MATLAB prompt. Then save this vector of filter coefficients by typing `save_coef('coef.asm',flipud(h));` Make sure you save the file in your own directory. (The scripts that perform these functions are available as gen_filt.m[2] and save_coef.m[3])

The `save_coef` MATLAB script will save the coefficients of the vector `h` into the named file, which in this case is `coef.asm`. Note that the coefficient vector is "flipped" prior to being saved; this is to make the coefficients in h fill DSP memory-locations `0x1000` through `0x1007` in reverse order, as before.

You may now re-assemble and re-run your new filter code as you did in Step 5 (Section 1.1.1.3.3: Step 5: Re-assemble and re-run with new filter).

Notice when you load your new filter that the contents of memory locations `0x1000` through `0x1007` update accordingly.

1.1.1.3.6 Step 8: Modify filter coefficients in memory

Not only can you view the contents of memory on the DSP using the debugger, you can change the contents at any memory location simply by double-clicking on the location and making the desired change in the pop-up window.

Change the contents of memory locations `0x1000` through `0x1007` such that the coefficients implement a scale and delay filter with impulse response:

$$h[n] = 8192\delta[n-4] \tag{1.1}$$

Note that the DSP interprets the integer value of 8192 as a fractional number by dividing the integer by 32,768 (the largest integer possible in a 16-bit two's complement register). The result is an output that is delayed by four samples and scaled by a factor of $\frac{1}{4}$. More information on the DSP's interpretation of numbers appears in Two's Complement and Fractional Arithmetic for 16-bit Processors[4].

> NOTE: A clear and complete understanding of how the DSP interprets numbers is absolutely necessary to effectively write programs for the DSP. Save yourself time later by learning this material now!

After you have made the changes to all eight coefficients, run your new filter and use the oscilloscope to measure the delay between the raw (input) and filtered (delayed) waveforms.

What happens to the output if you change either the scaling factor or the delay value? How many seconds long is a six-sample delay?

1.1.1.3.7 Step 9: Test-vector simulation

As a final exercise, you will find the output of the DSP for an input specified by a test vector. Then you will compare that output with the output of a MATLAB simulation of the same filter processing the same input; if the DSP implementation is correct, the two outputs should be almost identical. To do this, you will generate a waveform in MATLAB and save it as a test vector. You will then run your DSP filter using the test vector as input and import the results back into MATLAB for comparison with a MATLAB simulation of the filter.

The first step in using test vectors is to generate an appropriate input signal. One way to do this is to use the MATLAB function to generate a sinusoid that sweeps across a range of frequencies. The MATLAB function `save_test_vector` (available as save_test_vector.m[5] can then save the sinusoidal sweep to a file you will later include in the DSP code.

Generate a sinusoidal sweep and save it to a DSP test-vector file using the following MATLAB commands:

[2]See the file at <http://cnx.org/content/m11019/latest/gen_filt.m>
[3]See the file at <http://cnx.org/content/m11019/latest/save_coef.m>
[4]"Two's Complement and Fractional Arithmetic for 16-bit Processors" <http://cnx.org/content/m10808/latest/>
[5]See the file at <http://cnx.org/content/m11019/latest/save_test_vector.m>

```
≫ t=sweep(0.1*pi,0.9*pi,0.25,500);    % Generate a frequency sweep
≫ save_test_vector('testvect.asm',t); % Save the test vector
```

Next, use the MATLAB `conv` command to generate a simulated response by filtering the sweep with the filter h you generated using `gen_filt` above. Note that this operation will yield a vector of length 507 (which is $n+m-1$, where n is the length of the filter and m is the length of the input). You should keep only the first 500 elements of the resulting vector.

```
≫ out=conv(h,t)                 % Filter t with FIR filter h
≫ out=out(1:500)                % Keep first 500 elements of out
```

Now, modify the file `filter.asm` to use the alternative "test vector" core file, vectcore.asm[6]. Rather than accepting input from the A/D converters and sending output to the D/A, this core file takes its input from, and saves its output to, memory on the DSP. The test vector is stored in a block of memory on the DSP evaluation board that will not interfere with your program code or data.

> NOTE: The test vector is stored in the ".etext" section. See Core File: Introduction to Six-Channel Board for TI EVM320C54[7] for more information on the DSP memory sections, including a memory map.

The memory block that holds the test vector is large enough to hold a vector up to 4,000 elements long. The test vector stores data for both channels of input and from all six channels of output.

To run your program with test vectors, you will need to modify `filter.asm`. The assembly source is simply a text file and can be edited using the editor of your preference, including WordPad, Emacs, and VI. Replace the first line of the file with two lines. Instead of:

```
 .copy  "v:\ece420\54x\dsplib\core.asm"
```

use:

```
 .copy  "testvect.asm"
 .copy "v:\ece420\54x\dsplib\vectcore.asm"
```

Note that, as usual, the whitespace in front of the `.copy` directive is required.

These changes will copy in the test vector you created and use the alternative core file. After modifying your code, assemble it, then load and run the file using Code Composer as before. After a few seconds, halt the DSP (using the `Halt` command under the `Debug` menu) and verify that the DSP has halted at a branch statement that branches to itself. In the disassembly window, the following line should be highlighted: `0000:611F F073 B 611fh`.

Next, save the test output file and load it back into MATLAB. This can be done by first saving 3,000 memory elements (six channels times 500 samples) starting with location `0x8000` in program memory. Do

[6] See the file at <http://cnx.org/content/m11019/latest/vectcore.asm>
[7] "Core File: Introduction to Six-Channel Board for TI EVM320C54" <http://cnx.org/content/m10513/latest/>

this by choosing `File->Data->Save...` in Code Composer Studio, then entering the filename `output.dat` and pressing `Enter`. Next, enter `0x8000` in the Address field of the dialog box that pops up, `3000` in the Length field, and choose `Program` from the drop-down menu next to `Page`. Always make sure that you use the correct length (six times the length of the test vector) when you save your results.

Last, use the `read_vector` (available as read_vector.m[8]) function to read the saved result into MATLAB. Do this using the following MATLAB command:

```
≫ [ch1, ch2] = read_vector('output.dat');
```

Now, the MATLAB vector `ch1` corresponds to the filtered version of the test signal you generated. The MATLAB vector `ch2` should be nearly identical to the test vector you generated, as it was passed from the DSP system's input to its output unchanged.

> NOTE: Because of quantization error introduced in saving the test vector for the 16-bit memory of the DSP, the vector `ch2` will not be identical to the MATLAB generated test vector. Furthermore, a bug in our test vector environment sometimes causes blocks of samples to be dropped, so the test vector output signal may have gaps.

After loading the output of the filter into MATLAB, compare the expected output (calculated as `out` above) and the output of the filter (in `ch1` from above). This can be done graphically by simply plotting the two curves on the same axes; for example:

```
≫ plot(out,'r'); % Plot the expected curve in red
≫ hold on        % Plot the next plot on top of this one
≫ plot(ch1,'g'); % Plot the expected curve in green
≫ hold off
```

You should also ensure that the difference between the two outputs is near zero. This can be done by plotting the difference between the two vectors:

```
≫ plot(out(1:length(ch1))-ch1); % Plot error signal
```

You will observe that the two sequences are not exactly the same; this is due to the fact that the DSP computes its response to 16 bits precision, while MATLAB uses 64-bit floating point numbers for its arithmetic. Blocks of output samples may also be missing from the test vector output due to a bug in the test vector core. Nonetheless, the test vector environment allows one to run repeatable experiments using the same known test input for debugging.

[8]See the file at <http://cnx.org/content/m11019/latest/read_vector.m>

1.2 Lab 1

1.2.1 Lab 1: Prelab[9]

1.2.1.1 Assembly Exercise

Analyze the following lines of code. Refer to Two's Complement and Fractional Arithmetic for 16-bit Processors[10], Addressing Modes for TI TMS320C54x[11], and the *Mnemonic Instruction Set*[?] manual for help.

```
FIR_len .set    3

; Assume:
;   BK = FIR_len
;   AR0 = 1
;   AR2 = 1000h
;   AR3 = 1004h
;
;   FRCT = 1

     stl     A,*AR3+%
     rptz    A,(FIR_len-1)
     mac     *AR2+0%,*AR3+0%,A
```

Anything following a ";" is considered a comment. In this case, the comments indicate the contents of the auxiliary registers, the `BK` register, and the address registers before the execution of the first instruction, `stl`. The line `FIR_len .set 3` defines the name FIR_len as equal to 3. The `BK` register contains the length of the circular buffer we want to use. The `%` modifies the increment operator `+` so that it behaves as a circular buffer. This means that the address registers will be incremented until the (memory-address mod value-in-BK) = 0. When the increment operator `+` is followed by a `0`, it increments by the value specified in register `AR0`.

Note that any number followed by an "`h`" or preceded with a `0x` represents a **hexadecimal** value.

Example 1.1

1000h and 0x1000 both refer to the decimal number 4096.

Assume that the data memory is initialized as follows starting at location `1000h`.

[9] This content is available online at <http://cnx.org/content/m10022/2.22/>.
[10] "Two's Complement and Fractional Arithmetic for 16-bit Processors" <http://cnx.org/content/m10808/latest/>
[11] "Addressing Modes for TI TMS320C54x" <http://cnx.org/content/m10806/latest/>

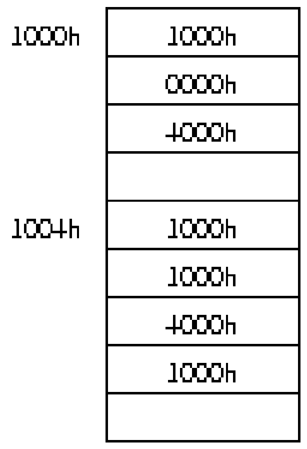


Figure 1.2: Data Memory Assignment (before execution)

After familiarizing yourself with the `stl`, `rptz`, and `mac` instructions, step through each line of code and record the values of the accumulator `A` and auxiliary registers `AR2` and `AR3` in the spaces provided in Figure 1.3. Additionally, record the value of the memory contents after all three instructions have been "executed" in the blank data memory table provided in Figure 1.4.

A	**AR2**	**AR3**	
`00 0000 8000h`	`1000h`	`1004h`	at start of code
			after `stl` instruction
			after `rptz` instruction
			after first `mac` instruction
			after second `mac` instruction
			after third `mac` instruction

Figure 1.3: Execution Results

When working through the exercise, take into account that the accumulator `A` is a 40-bit register, and that the multiplier is in the **fractional arithmetic mode**. In this mode, integers on the DSP are interpreted as fractions, and the multiplier will treat them accordingly. This is done by shifting the result of the integer multiplier in the ALU left one bit. (All the arithmetic is fractional in these examples.) Multiplies performed by the ALU (via the `mac` instruction) produce a result that is twice what you would expect if you just multiplied the two integers together. DSP numerical representation and arithmetic are described further in Two's Complement and Fractional Arithmetic for 16-bit Processors[12].

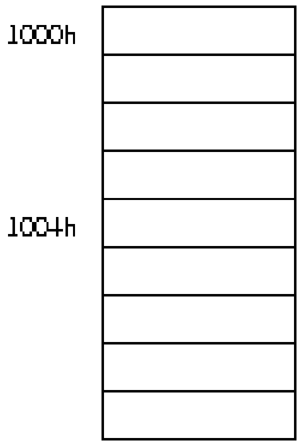


Figure 1.4: Data Memory Assignment (after execution)

[12]"Two's Complement and Fractional Arithmetic for 16-bit Processors" <http://cnx.org/content/m10808/latest/>

1.2.2 Lab 1: Lab[13]

1.2.2.1 Introduction

In this exercise, you will program in the DSP's assembly language to create FIR filters. Begin by studying the assembly code for the basic FIR filter filter.asm[14] .

[13]This content is available online at <http://cnx.org/content/m11020/2.6/>.
[14]http://cnx.rice.edu/content/m10017/latest/filter.asm

filter.asm

```
.copy "core.asm"   ; Copy in core file
   ; This initializes DSP and jumps to "main"

 FIR_len .set   8 ; This is an 8-tap filter.

         .sect ".data" ; Flag following as data declarations

        .align 16 ; Align to a multiple of 16
 coef ; assign label "coeff"
     .copy "coef.asm" ; Copy in coefficients

     .align 16
 firstate
      .space 16*8 ; Allocate 8 words of storage for
 ; filter state.

     .sect ".text" ; Flag the following as program code
  main
      ; Initialize various pointers
     stm    #FIR_len,BK ; initialize circular buffer length
     stm    #coef,AR2     ; initialize coefficient pointer
     stm    #firstate,AR3 ; initialize state pointer
     stm    #1,AR0 ; initialize AR0 for pointer increment

  loop
      ; Wait for a new block of 64 samples to come in
       WAITDATA

      ; BlockLen = the number of samples that come from WAITDATA (64)
         stm     #BlockLen-1, BRC ; Put repeat count into repeat counter
         rptb    endblock-1 ; Repeat between here and 'endblock'

         ld     *AR6,16, A ; Receive ch1 into A accumulator
         mar     *+AR6(2)            ; Rcv data is in every other channel
         ld      *AR6,16, B ; Receive ch2 into B accumulator
         mar     *+AR6(2)            ; Rcv data is in every other channel

         ld     A,B ; Transfer A into B for safekeeping

      ; The following code executes a single FIR filter.

         sth     A,*AR3+% ; store current input into state buffer
         rptz    A,(FIR_len-1) ; clear A and repeat
         mac     *AR2+0%,*AR3+0%,A ; multiply coef. by state & accumulate

         rnd     A ; Round off value in 'A' to 16 bits

     ; end of FIR filter code. Output is in the high part of 'A.'
```

```
         sth     A, *AR7+ ; Store filter output (from A) into ch1
         sth     B, *AR7+ ; Store saved input (from B) into ch2

         sth     B, *AR7+ ; Store saved input to ch3...ch6 also
```

filter.asm applies an FIR filter to the signal from input channel 1 and sends the resulting output to output channel 1. It also sends the original signal to output channel 2.

First, create a work directory on your network drive for the files in this exercise, and copy filter.asm from v:\ece320\54x\dsplib to your work directory (this is thesame file you worked with in Lab 0). Then, use MATLAB to generate two 20-tap FIR filters. The first filter should pass signals from 4 kHz to 8 kHz; the second filter should pass from 8 kHz to 12 kHz. For both filters, allow a 1 kHz transition band on each edge of the filter passband. To create these filters, first convert these band edges to digital frequencies based on the 44.1 kHz sample rate of the system, then use the MATLAB command remez to generate this filter; you can type help remez for more information. Use the save_coef command to save each of these filters into different files. (Make sure you reverse the vectors of filter coefficients before you save them.) Also save your filters as a MATLAB matrix, since you will need them later to generate test vectors. This can be done using the MATLAB save command. Once this is done, use the freqz command to plot the frequency response of each filter.

1.2.2.2 Part 1: Single-Channel FIR Filter

For now, you will implement only the filter with a 4 kHz to 8 kHz passband. Edit filter.asm to use the coefficients for this filter by making several changes.

First, the length of the FIR filter for this exercise is 20, not 8. Therefore, you need to change FIR_len to 20. FIR_len is set using the .set directive, which assigns a number to a symbolic name. You will need to change this to FIR_len .set 20.

Second, you will need to ensure that the .copy directive brings in the correct coefficients. Change the filename to point to the file that contains the coefficients for your first filter.

Third, you will need to modify the .align and .space directives appropriately. The TI TMS320C54x DSP requires that circular buffers, which are used for the FIR filter coefficient and state buffers, be aligned so that they begin at an address that is a multiple of a power of two greater than the length of the buffer. Since you are using a 20-tap filter (which uses 20-element state and coefficient buffers), the next greater power of two is 32. Therefore, you will need to align both the state and coefficient buffers to an address that is a multiple of 32. (16-element buffers would also require alignment to a multiple of 32.) This is done with the .align command. In addition, memory must be reserved for the state buffer. This is done using the .space directive, which takes as its input the number of **bits** of space to allocate. Therefore, to allocate 20 words of storage, use the directive .space 16*20 as shown below:

```
        .align 32             % Align to a multiple of 32
coef    .copy  "filter1.asm"  % Copy FIR filter coefficients

        .align 32             % Align to a multiple of 32
state   .space 16*20          % Allocate 20 words of data space
```

Assemble your code, set PMST to 0xFFE0, reset the DSP, and run. Ensure that it is has the correct frequency response. After you have verified that this code works properly, proceed to the next step.

1.2.2.3 Part 2: Dual-Channel FIR Filters

First, make a copy of your modified filter.asm file from Part 1 (Section 1.2.2.2: Part 1: Single-Channel FIR Filter). Work from this copy; do not modify your working filter from the previous part. You will use that code again later.

Next, modify your code so that in addition to sending the output of your first filter (with a 4 kHz to 8 kHz passband) to output channel 1 and the unfiltered input to output channel 2, it sends the output of your second filter (with a 8 kHz to 12 kHz passband) to output channel 3. To do this, you will need to use the

`.align` and `.copy` directives to load the second set of coefficients into data memory. You will also need to add instructions to initialize a pointer to the second set of coefficients and to perform the calculations for the second filter.

Exercise 1.2.2.1

Extra Credit Problem

One extra credit point will be awarded to you and your partner if you can implement the dual-channel system without using the auxiliary registers `AR4` and `AR5`? Why is this more difficult? Renaming `AR4` and `AR5` using the `.asg` directive does not count!

Using the techniques introduced in DSP Development Environment: Introductory Exercise for TI TMS320C54x[15], generate an appropriate test vector and expected outputs in MATLAB. Then, using the test-vector core file also introduced in DSP Development Environment: Introductory Exercise for TI TMS320C54x[16], find the system's output given this test vector. In MATLAB, plot the expected and actual outputs of the both filters and the difference between the expected and actual outputs. Why is the output from the DSP system not exactly the same as the output from MATLAB?

1.2.2.4 Part 3: Alternative Single-Channel FIR Implementation

An alternative method of implementing symmetric FIR filters uses the `firs` instruction. Modify your code from Part 1 (Section 1.2.2.2: Part 1: Single-Channel FIR Filter) to implement the filter with a 4 kHz to 8 kHz passband using the `firs`.

Two differences in implementation between your code from Part 1 (Section 1.2.2.2: Part 1: Single-Channel FIR Filter) and the code you will write for this part are that (1) the `firs` instruction expects coefficients to be located in program memory instead of data memory, and (2) `firs` requires the states to be broken up into two separate circular buffers. Refer to the `firs` instruction on *page 4-59* in the *Mnemonic Instruction Set*[?] manual, as well as a description and example of its use on *pages 4-5 through 4-8* of the *Applications Guide*[?] for more information (*Volumes 2 and 4* respectively of the *TMS320C54x DSP Reference Set*).

`AR0` needs to be set to -1 for this code to work properly. Why?

NOTE: `COEFF` is a label to the coefficients now expected to be in program memory. Refer to the `firs` description for more information).

```
mvdd *AR2,*AR3+0% ; write x(-N/2) over x(-N)
sth A,*AR2 ; write x(0) over x(-N/2)
add *AR2+0%,*AR3+0%,A  ; add x(0) and x(-(N-1))
;   (prepare for first multiply)

rptz B,#(FIR_len/2-1)
firs *AR2+0%,*AR3+0%,COEFF
mar ???????  ; Fill in these two instructions
mar ??????? ; They modify AR2 and AR3.

; note that the result is now in the
;  B accumulator
```

Because states and coefficients are now treated differently than in your previous FIR implementation, you will need to modify the pointer initializations to

[15]"DSP Development Environment: Introductory Exercise for TI TMS320C54x" <http://cnx.org/content/m10017/latest/>

[16]"DSP Development Environment: Introductory Exercise for TI TMS320C54x" <http://cnx.org/content/m10017/latest/>

```
stm #(FIR_len/2),BK ; initialize circular buffer length
stm #firstate_,AR2 ; initialize location containing first
;   half of states

stm #-1,AR0 ; Initialize AR0 to -1

stm #firstate2_,AR3         ; initialize location containing last half
```

Use the test-vector core file to find the output of this system given the same test vector you used to test the two-filter system. Compare the output of this code against the output of the same filter implemented using the `mac` instruction. Are the results the same? Why or why not? Ensure that the filtered output is sent to output channel 1, and that the unmodified output is still sent to output channel 2.

WARNING: You will lose credit if the unmodified output is not present or if the channels are reversed!

1.2.2.5 Quiz Information

The quiz for Lab 1 is broken down as follows:

- 1 point: Prelab (must be ready to show the TA the week before the quiz)
- 4 points: Working code: you must demonstrate that your code works using input from function generator and that it works using input from appropriate test vectors. Have an `.asm` file **ready** to demonstrate each. Of the 4 points, you get 0.5 points for a single 20-tap filter, 2 points for the two-filter system, and 1.5 points for the system using the `firs` opcode.
- 5 points: Oral quiz score.
- 1 extra credit point: As described above (p. 16).

The oral quiz may cover signal processing material relating to FIR filters, including, but not limited to, the delay through FIR filters, generalized linear phase, and the differences between ideal FIR filters and realizable FIR filters. You may also be asked questions about digital sampling theory, including, but not limited to, the Nyquist sampling theorem and the relationship between the analog frequency spectrum and the digital frequency spectrum of a continuous-time signal that has been sampled.

The oral quiz **will** cover the code that you have written during the lab. You are expected to understand, in detail, all of the code in the files you have worked on, even if your partner or a TA wrote it. (You are not expected to understand the core file in detail). The TA will ask you to explain various lines of code as part of the quiz. The TAs may also ask questions about 2's complement fractional arithmetic, circular buffers, alignment, and the mechanics of either of the two FIR filter implementations. You could be ready to trace through any of the code on paper and explain what each line of code does.

Use the TI documentation, specifically the *Mnemonic Instruction Set*[?] manual. Hard-copies of this manual can also be found in the lab. Also, feel free to ask the TAs to help explain the code that you have been given.

1.3 Lab 2

1.3.1 Lab 2: Theory[17]

1.3.1.1 Introduction

In the exercises that follow, you will explore some of the effects of **multirate processing** using the system in Figure 1.6. The **sample-rate compressor** ($\downarrow (D)$) in the block-diagram removes $D-1$ of every D input samples, while the **sample-rate expander** ($\uparrow (U)$) inserts $U-1$ zeros after every input sample. With the compression and expansion factors set to the same value ($D=U$), filters FIR 1 and FIR 3 operate at the sample rate F_s, while filter FIR 2 operates at the lower rate of $\frac{F_s}{D}$.

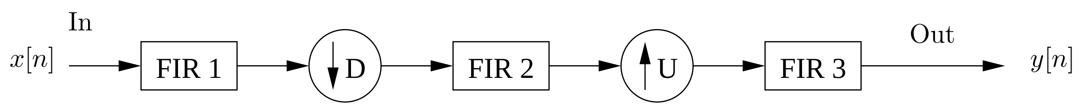


Figure 1.6: Net multirate system

Later, you will implement the system and control the compression and expansion factors at runtime with an interface provided for you. You will be able to disable any or all of the filters to investigate multirate effects. What purpose do FIR 1 and FIR 3 serve, and what would happen in their absence?

[17] This content is available online at <http://cnx.org/content/m10024/2.21/>.

1.3.2 Lab 2: Prelab (Part 1)[18]

1.3.2.1 Multirate Theory Exercise

Consider a sampled signal with the DTFT $X(\omega)$ shown in Figure 1.7.

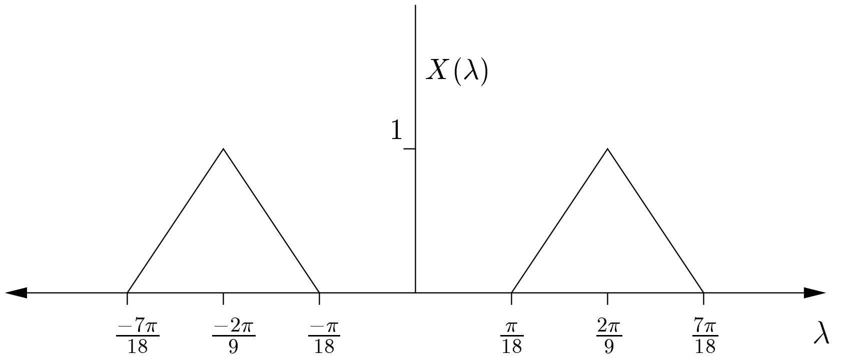


Figure 1.7: DTFT of the input signal.

Assuming $U = D = 3$, use the relations between the DTFT of a signal before and after sample-rate compression and expansion ((1.2) and (1.3)) to sketch the DTFT response of the signal as it passes through the multirate system of Figure 1.8 (without any filtering). Include both the intermediate response $W(\omega)$ and the final response $Y(\omega)$. It is important to be aware that the translation from digital frequency ω to analog frequency depends on the sampling rate. Therefore, the conversion is different for $X(\omega)$ and $W(\omega)$.

$$W(\omega) = \frac{1}{D} \sum_{k=0}^{D-1} X\left(\frac{\omega + 2\pi k}{D}\right) \quad (1.2)$$

$$Y(\omega) = W(U\omega) \quad (1.3)$$

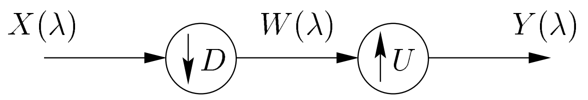


Figure 1.8: Multirate System

[18] This content is available online at <http://cnx.org/content/m10620/2.14/>.

1.3.3 Lab 2: Prelab (Part 2)[19]

1.3.3.1 Filter-Design Exercise

Using the zero-placement method, design the FIR filters for the multirate system in Multirate Filtering: Introduction (Figure 1.6). Recall that the z-transform of a length- N FIR filter is a polynomial in z^{-1}, and that this polynomial can be factored into $N-1$ roots.

$$\begin{aligned} H(z) &= h_0 + h_1 z^{-1} + h_2 z^{-2} + \cdots \\ &= \left(z_1 - z^{-1}\right)\left(z_2 - z^{-1}\right)\left(z_3 - z^{-1}\right)\cdots \end{aligned} \tag{1.4}$$

Use this relation to design a low-pass filter (for the anti-aliasing and anti-imaging filters of the multirate system) by placing twelve complex zeros on the unit circle at $\pm\left(\frac{3\pi}{8}\right)$, $\pm\left(\frac{\pi}{2}\right)$, $\pm\left(\frac{5\pi}{8}\right)$, $\pm\left(\frac{3\pi}{4}\right)$, $\pm\left(\frac{7\pi}{8}\right)$, and $\pm(\pi)$. This filter that you have just designed will serve for both FIR 1 and FIR 3. For filter FIR 2 (operating at the decimated rate), use four equally-spaced zeros on the unit circle located at $\pm\left(\frac{\pi}{4}\right)$ and $\pm\left(\frac{3\pi}{4}\right)$. Be sure to adjust the resulting filter coefficients to ensure that the gain does not exceed one at any frequency.

Design your filters by writing a MATLAB script to compute the filter coefficients from the given zero locations. The MATLAB function `poly` is very useful for this; type `help poly` in MATLAB for details.

Once you have determined the coefficients of the filters, use MATLAB function `freqz` to plot the frequency responses. You will find that the frequency response of these filters has a large gain. Adjust the resulting filter coefficients to ensure that the largest frequency gain is less than or equal to one by dividing the coefficients by an appropriate value. Do the frequency responses match your expectations based on the locations of the zeros in the z-plane?

[19]This content is available online at <http://cnx.org/content/m10815/2.6/>.

1.3.4 Lab 2: Lab[20]

1.3.4.1 Implementation

Before implementing the entire system shown in Multirate Processing: Introduction (Figure 1.6), we recommend you design a system that consists of a cascade of filters FIR 1 and FIR 2 without the sample-rate compressor or expander. After verifying that the response of your two-filter system is correct, proceed to implement the complete multirate system and verify its total response. At first, use fixed compression and expansion factors of $D = U = 4$. After you have verified that the multirate system works at a fixed rate, you should modify your code so that the rate can be changed easily. Later, you have the option of controlling this factor in real-time using a MATLAB interface. **Regardless of whether you choose to use the MATLAB interface, you must be able to quickly change the compression and expansion factors when you demo your code**.

1.3.4.1.1 Compressed-rate processing

In order to perform the processing at the lower sample rate, implement a counter in your code. Your counter will determine when the compressed-rate processing is to occur, and it can also be used to determine when to insert zeros into FIR 3 to implement the sample-rate expander.

Some instructions that may be useful for implementing your multirate structure are the `addm` (add to memory) and `bc` (branch conditional) instructions. You may also find the `banz` (branch on auxiliary register not zero) and the `b` (branch) instruction useful.

1.3.4.1.2 Real-time rate change and MATLAB interface (Optional)

A simple graphical user interface (GUI) is available (as mrategui.m[21] , which requires ser_snd.m[22]) that sends a number between 1 and 10 to the DSP via the serial port. This can be used to change the compression and expansion factor in real time.

Run the GUI by typing `mrategui` at the MATLAB prompt. A figure should automatically open up with a slider on it; adjusting the slider changes the compression and expansion factor sent to the DSP.

The assembly code for interacting with the serial port, provided in the handout Core File: Serial Port Communication Between MATLAB and TI TMS320C54x[23], stores the last number that the DSP has received from the computer in the memory location labeled `hold`. Therefore, unless you have changed the serial portion of the given code, you can find the last compression and expansion factor set by the GUI in this location. You need to modify your code so that each time a new number is received on the serial port, the compression and expansion factor is changed. If a "1" is received on the serial port, the entire system should run at the full rate; if a "10" is received, the system should discard nine samples between each sample processed at the lower rate.

Note that the `READSER` and `WRITSER` macros, which are used to read data from and send data to the serial port, overwrite `AR0`, `AR1`, `AR2`, and `AR3` registers, as well as `BK` and the condition flag `TC`. You must therefore ensure that these registers are not used by your code, or that you save and restore their values in memory before you call the `READSER` and `WRITSER` macros. This can be done using the `mvdm` and `mvmd` instructions. The serial macros set up the `AR1` and `AR3` each time they are called, so there is no need to change these registers before the macros are called.

More detail about the `READSER` and `WRITSER` macros can be found in Core File: Serial Port Communication Between MATLAB and TI TMS320C54x[24].

[20]This content is available online at <http://cnx.org/content/m11810/1.3/>.
[21]http://cnx.org/content/m11810/latest/mrategui.m
[22]http://cnx.org/content/m11810/latest/ser_snd.m
[23]"Core File: Serial Port Communication Between MATLAB and TI TMS320C54x" <http://cnx.org/content/m10821/latest/>
[24]"Core File: Serial Port Communication Between MATLAB and TI TMS320C54x" <http://cnx.org/content/m10821/latest/>

1.4 Lab 3

1.4.1 Lab 3: Theory[25]

1.4.1.1 Introduction

Like finite impulse-response (FIR) filters, **infinite impulse-response** (**IIR**) filters are **linear time-invariant** (**LTI**) systems that can recreate a large range of different frequency responses. Compared to FIR filters, IIR filters have both advantages and disadvantages. On one hand, implementing an IIR filter with certain stopband-attenuation and transition-band requirements typically requires far fewer filter taps than an FIR filter meeting the same specifications. This leads to a significant reduction in the computational complexity required to achieve a given frequency response. However, the poles in the transfer function require feedback to implement an IIR system. In addition to inducing nonlinear phase in the filter (delaying different frequency input signals by different amounts), the feedback introduces complications in implementing IIR filters on a fixed-point processor. Some of these complications are explored in IIR Filtering: Filter-Coefficient Quanitization Exercise in MATLAB (Section 1.4.3).

Later, in the processor exercise, you will explore the advantages and disadvantages of IIR filters by implementing and examining a fourth-order IIR system on a fixed-point DSP. The IIR filter should be implemented as a cascade of two second-order, Direct Form II sections. The data flow for a second-order, Direct-Form II section, or **bi-quad**, is shown in Figure 1.9. Note that in Direct Form II, the states (delayed samples) are neither the input nor the output samples, but are instead the intermediate values $w[n]$.

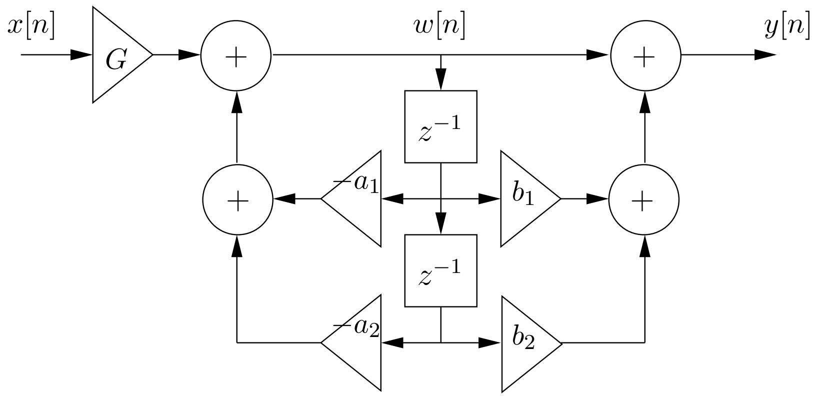


Figure 1.9: Second-order, Direct Form II section

[25]This content is available online at <http://cnx.org/content/m10025/2.22/>.

1.4.2 Lab 3: Prelab (Part 1)[26]

1.4.2.1

The transfer function for the second-order section shown in IIR Filtering: Introduction (Figure 1.9) is

$$H(z) = G\frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{1.5}$$

1.4.2.1.1 Exercise

First, derive the above transfer function. Begin by writing the **difference equations** for $w[n]$ in terms of the input and past values ($w[n-1]$ and $w[n-2]$). Then write the difference equation for $y[n]$ also in terms of the past samples of $w[n]$. After finding the two difference equations, compute the corresponding Z-transforms and use the relation $H(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)W(z)}{W(z)X(z)}$ to verify the IIR transfer function in (1.5).

Next, design the coefficients for a fourth-order filter implemented as the cascade of two bi-quad sections. Write a MATLAB script to compute the coefficients. Begin by designing the fourth-order filter and checking the response using the MATLAB commands

```
[B,A] = ellip(4,.25,10,.25)
freqz(B,A)
```

NOTE: MATLAB's `freqz` command displays the frequency responses of IIR filters and FIR filters. For more information about this, type `help freqz`. Be sure to look at MATLAB's definition of the transfer function.

NOTE: If you use the `freqz` command as shown above, without passing its returned data to another function, both the magnitude (in decibels) and the phase of the response will be shown.

Next you must find the roots of the numerator, **zeros**, and roots of the denominator, **poles**, so that you can group them to create two second-order sections. The MATLAB commands `roots` and `poly` will be useful for this task. Save the scripts you use to decompose your filter into second-order sections; they will probably be useful later.

Once you have obtained the coefficients for each of your two second-order sections, you are ready to choose a **gain** factor, G, for each section. As part of your MATLAB script, use `freqz` to compute the response $\frac{W(z)}{X(z)}$ with $G = 1$ for each of the sets of second-order coefficients. Recall that on the DSP we cannot represent numbers greater than or equal to 1.0. If the maximum value of $|\frac{W(z)}{X(z)}|$ is or exceeds 1.0, an input with magnitude less than one could produce $w[n]$ terms with magnitude greater than or equal to one; this is **overflow**. You must therefore select a gain values for each second-order section such that the response from the input to the states, $\frac{W(z)}{X(z)}$, is always less than one in magnitude. In other words, set the value of G to ensure that $|\frac{W(z)}{X(z)}| < 1$.

1.4.2.1.2 Preparing for processor implementation

As the processor exercises become more complex, it will become increasingly important to observe good programming practices. Of these, perhaps the most important is careful planning of your program flow,

[26]This content is available online at <http://cnx.org/content/m10623/2.11/>.

memory and register use, and testing procedure. Write out pseudo-code for the processor implementation of a bi-quad. Make sure you consider the way you will store coefficients and states in memory. Then, to prepare for testing, compute the values of $w[n]$ and $y[n]$ for both second-order sections at $n = \{0, 1, 2\}$ using the filter coefficients you calculated in MATLAB. Assume $x[n] = \delta[n]$ and all states are initialized to zero. You may also want to create a frequency sweep test-vector like the one in DSP Development Environment: Introductory Exercise for TI TMS320C54x[27] and use the filter command to find the outputs for that input. Later, you can recreate these input signals on the DSP and compare the output values it calculates with those you find now. If your program is working, the values will be almost identical, differing only slightly because of quantization effects, which are considered in IIR Filtering: Filter-Coefficient Quantization Exercise in MATLAB (Section 1.4.3).

[27]"DSP Development Environment: Introductory Exercise for TI TMS320C54x" <http://cnx.org/content/m10017/latest/>

1.4.3 Lab 3: Prelab (Part 2)[28]

1.4.3.1 Filter-Coefficient Quantization

One important issue that must be considered when IIR filters are implemented on a fixed-point processor is that the filter coefficients that are actually used are quantized from the "exact" (high-precision floating point) values computed by MATLAB. Although quantization was not a concern when we worked with FIR filters, it can cause significant deviations from the expected response of an IIR filter.

By default, MATLAB uses 64-bit floating point numbers in all of its computation. These floating point numbers can typically represent 15-16 digits of precision, far more than the DSP can represent internally. For this reason, when creating filters in MATLAB, we can generally regard the precision as "infinite," because it is high enough for any reasonable task.

NOTE: Not all IIR filters are necessarily "reasonable"!

The DSP, on the other hand, operates using 16-bit fixed-point numbers in the range of -1.0 to $1.0 - 2^{-15}$. This gives the DSP only 4-5 digits of precision and only if the input is properly scaled to occupy the full range from -1 to 1.

For this section exercise, you will examine how this difference in precision affects a **notch filter** generated using the `butter` command: `[B,A] = butter(2,[0.07 0.10],'stop')`.

1.4.3.1.1 Quantizing coefficients in MATLAB

It is not difficult to use MATLAB to **quantize** the filter coefficients to the 16-bit precision used on the DSP. To do this, first take each vector of filter coefficients (that is, the A and B vectors) and divide by the smallest power of two such that the resulting absolute value of the largest filter coefficient is less than or equal to one. This is an easy but fairly reasonable approximation of how numbers outside the range of -1 to 1 are actually handled on the DSP.

Next, quantize the resulting vectors to 16 bits of precision by first multiplying them by $2^{15} = 32768$, rounding to the nearest integer (use `round`), and then dividing the resulting vectors by 32768. Then multiply the resulting numbers, which will be in the range of -1 to 1, back by the power of two that you divided out.

1.4.3.1.2 Effects of quantization

Explore the effects of quantization by quantizing the filter coefficients for the notch filter. Use the `freqz` command to compare the response of the unquantized filter with two quantized versions: first, quantize the entire fourth-order filter at once, and second, quantize the second-order ("bi-quad") sections separately and recombine the resulting quantized sections using the `conv` function. Compare the response of the unquantized filter and the two quantized versions. Which one is "better?" Why do we always implement IIR filters using second-order sections instead of implementing fourth (or higher) order filters directly?

Be sure to create graphs showing the difference between the filter responses of the unquantized notch filter, the notch filter quantized as a single fourth-order section, and the notch filter quantized as two second-order sections. Save the MATLAB code you use to generate these graphs, and be prepared to reproduce and explain the graphs as part of your quiz. Make sure that in your comparisons, you rescale the resulting filters to ensure that the response is unity (one) at frequencies far outside the notch.

[28]This content is available online at <http://cnx.org/content/m10813/2.5/>.

1.4.4 Lab 3: Lab[29]

1.4.4.1 Implementation

On the DSP, you will implement the **elliptic low-pass filter** designed using the `ellip` command from IIR Filters: Filter-Design Exercise in MATLAB (Section 1.4.2). You should not try to implement the notch filter designed in IIR Filtering: Filter-Coefficient Quantization Exercise in MATLAB (Section 1.4.3), because it will not work correctly when implemented using Direct Form II. (Why not?)

To implement the fourth-order filter, start with a single set of second-order coefficients and implement a single second-order section. Make sure you write and review pseudo-code **before** you begin programming. Once your single second-order IIR is working properly you can then proceed to code the entire fourth-order filter.

1.4.4.1.1 Large coefficients

You may have noticed that some of the coefficients you have computed for the second-order sections are larger than 1.0 in magnitude. For any stable second-order IIR section, the magnitude of the "0" and "2" coefficients (a_0 and a_2, for example) will always be less than or equal to 1.0. However, the magnitude of the "1" coefficient can be as large as 2.0. To overcome this problem, you will have to divide the a_1 and b_1 coefficients by two prior to saving them for your DSP code. Then, in your implementation, you will have to compensate somehow for using half the coefficient value.

1.4.4.1.2 Repeating code

Rather than write separate code for each second-order section, you are encouraged first to write one section, then write code that cycles through the second-order section code twice using the repeat structure below. Because the IIR code will have to run inside the block I/O loop and this loop uses the **block repeat counter** (`BRC`), you must use another looping structure to avoid corrupting the `BRC`.

> NOTE: You will have to make sure that your code uses different coefficients and states during the second cycle of the repeat loop.

```
stm     (num_stages-1),AR1

start_stage

; IIR code goes here

banz    start_stage,*AR1-
```

1.4.4.1.3 Gain

It may be necessary to add gain to the output of the system. To do this, simply shift the output left (which can be done using the `ld` opcode with its optional `shift` parameter) before saving the output to memory.

1.4.4.2 Grading

Your grade on this lab will be split into three parts:

- 1 point: Prelab

[29]This content is available online at <http://cnx.org/content/m11021/2.4/>.

- 4 points: Code. Your DSP code implementing the fourth-order IIR filter is worth 3 points and the MATLAB exercise is worth 1 point.
- 5 points: Oral quiz. The quiz may cover differences between FIR and IIR filters, the prelab material, and the MATLAB exercise.

1.5 Lab 4

1.5.1 Lab 4: Theory[30]

1.5.1.1 Introduction

In this lab you are going to apply the **Fast Fourier Transform** (**FFT**) to analyze the spectral content of an input signal in real time. After computing the FFT of a 1024-sample block of input data, you will then compute the squared magnitude of the sampled spectrum and send it to the output for display on the oscilloscope. In contrast to the systems you have implemented in the previous labs, the FFT is an algorithm that operates on blocks of samples at a time. In order to operate on blocks of samples, you will need to use interrupts to halt processing so that samples can be transferred.

A second objective of this lab exercise is to introduce the TI-C549 C environment in a practical DSP application. In future labs, the benefits of using the C environment will become clear as larger systems are developed. The C environment provides a fast and convenient way to implement a DSP system using C and assembly modules.

The FFT can be used to analyze the spectral content of a signal. Recall that the FFT is an efficient algorithm for computing the **Discrete Fourier Transform** (**DFT**), a frequency-sampled version of the **DTFT**.

DFT:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-\left(i \frac{2\pi}{N} nk\right)} \tag{1.6}$$

where $n \wedge k \in \{0, 1, \dots, N-1\}$

Your implementation will include windowing of the input data prior to the FFT computation. This is simple a point-by-point multiplication of the input with an analysis window. As you will explore in the prelab exercises, the choice of window affects the shape of the resulting window.

A block diagram representation of the spectrum analyzer you will implement in the lab, including the required input and ouput locations, can be found depicted in Figure 1.10.

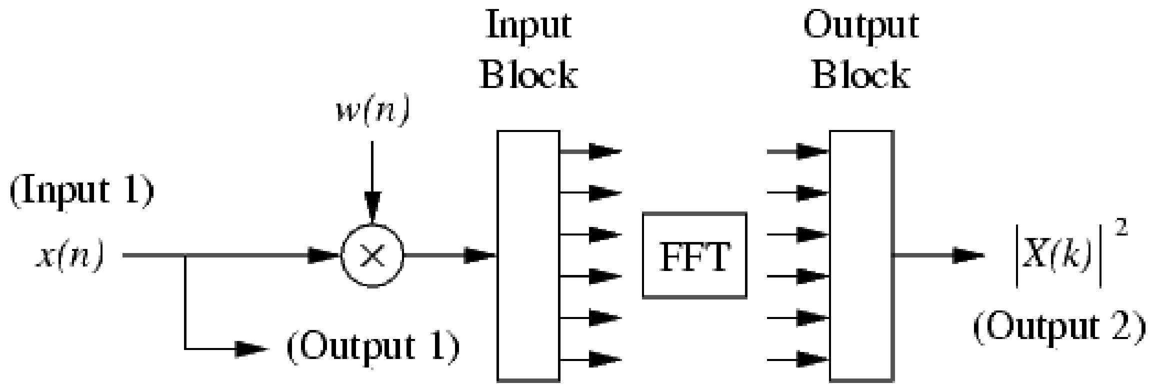


Figure 1.10: FFT-based spectrum analyzer

[30]This content is available online at <http://cnx.org/content/m11828/1.2/>.

1.5.2 Lab 4: Prelab[31]

1.5.2.1 MATLAB Exercise

Since the DFT is a sampled version of the spectrum of a digital signal, it has certain sampling effects. To explore these sampling effects more thoroughly, we consider the effect of multiplying the time signal by different window functions and the effect of using zero-padding to increase the length (and thus the number of sample points) of the DFT. Using the following MATLAB script as an example, plot the squared-magnitude response of the following test cases over the digital frequencies $\omega_c = \left[\frac{\pi}{8}, \frac{3\pi}{8}\right]$.

1. rectangular window with no zero-padding
2. hamming window with no zero-padding
3. rectangular window with zero-padding by factor of four (*i.e.*, 1024-point FFT)
4. hamming window window with zero-padding by factor of four

Window sequences can be generated in MATLAB by using the `boxcar` and `hamming` functions.

```
N = 256;                % length of test signals
num_freqs = 100;        % number of frequencies to test

% Generate vector of frequencies to test

omega = pi/8 + [0:num_freqs-1]'/num_freqs*pi/4;

S = zeros(N,num_freqs);                % matrix to hold FFT results


 for i=1:length(omega)                  % loop through freq. vector
    s = sin(omega(i)*[0:N-1]');         % generate test sine wave
    win = boxcar(N);                    % use rectangular window
    s = s.*win;                         % multiply input by window
    S(:,i) = (abs(fft(s))).^2;          % generate magnitude of FFT
                                        % and store as a column of S
 end

 clf;
 plot(S);                               % plot all spectra on same graph

```

Make sure you understand what every line in the script does. What signals are plotted?

You should be able to describe the tradeoff between mainlobe width and sidelobe behavior for the various window functions. Does zero-padding increase frequency resolution? Are we getting something for free? What is the relationship between the DFT, $X[k]$, and the DTFT, $X(\omega)$, of a sequence $x[n]$?

[31]This content is available online at <http://cnx.org/content/m10625/2.8/>.

1.5.3 Lab 4: Lab[32]

1.5.3.1 Implementation

As this is your first experience with the C environment, you will have the option to add most of the required code in C or assembly. A C skeleton will provide access to input samples, output samples, and interrupt handling code. You will add code to transfer the inputs and outputs (in blocks at a time), apply a hamming window, compute the magnitude-squared spectrum, and include a trigger pulse. After the hamming window is created, either an assembly or C module that bit reverses the input and performs an FFT calculation is called.

As your spectrum analyzer works on a block of samples at a time, you will need to use interrupts to pause your processing while samples are transferred from/to the CODEC (A/D and D/A) buffer. Fortunately, the interrupt handling routines have been written for you in a C shell program available at `v:\ece420\54x\dspclib\lab4main.c` and the core code.

1.5.3.1.1 Interrupt Basics

Interrupts are an essential part of the operation of any microprocessor. They are particularly important in embedded applications where DSPs are often used. Hardware interrupts provide a way for interacting with external devices while the processor executes code. For example, in a key entry system, a key press would generate a hardware interrupt. The system code would then jump to a specified location in program memory where a routine could process the key input. Interrupts provide an alternative to polling. Instead of checking for key presses at a predetermined rate (requires a clock), the system could be busy executing other code. On the TI-C54x DSP, interrupts provide a convenient way to transfer blocks of data to/from the CODEC in a timely fashion.

1.5.3.1.2 Interrupt Handling

The `lab4main.c` code and the core code are intended to make your interaction with the hardware much simpler. As there was a core file for working in the assembly environment (Labs 0-3), there is a core file for the C environment (V:\ece420\54x\dspclib\core.asm) which handles the interrupts from the CODEC (A/D and D/A) and the serial port. Here, we will describe the important aspects of the core code necessary to complete the assignment.

At the heart of the hardware interaction is the auto-buffering serial port. In the auto-buffering serial mode, the TI-C54x processor is able to do processing **uninterrupted** while samples are transferred to/from a buffer of length BlockLen = 64 samples. However, the spectrum analyzer to be implemented in this lab works over a block of $N = 1024$ samples. If it were possible to compute a 1024-point FFT in the sample time of one `BlockLen`, then no additional interrupt handling routines would be necessary. Samples could be collected in a 1024-length buffer and a 1024-point FFT could be computed uninterrupted while the auto-buffering buffer fills. Unfortunately, the DSP is not fast enough to accomplish this task.

We now provide an explanation of the shell C program `lab4main.c` listed in Appendix A (Section 1.5.3.3: Appendix A:). The `lab4main.c` file contains the function `interrupt void irq` and a main program. The main program is an infinite loop over blocks of $N = 1024$ samples. Note that while the DSP is executing instructions in this loop, interrupts occur every `BlockLen` samples. Inside the infinite loop, you will insert code to do the operations which follow. Although each of these operations may be performed in C or assembly, we suggest you follow the guidelines suggested.

1. Transfer inputs and outputs (C)
2. Apply a Hamming Window (C/assembly)
3. Bit-reverse the input (C and assembly)
4. Apply an N-point FFT (C and assembly)
5. Compute the magnitude-squared spectrum (C/assembly)

[32]This content is available online at <http://cnx.org/content/m11827/1.5/>.

6. Include a trigger pulse (C/assembly)

The function WaitAudio is an assembly function in the core code which handles the CODEC interrupts. An interrupt from the CODEC occurs every `BlockLen` samples. The `SetAudioInterrupt(irq)` call in the main program tells the core code to jump to the `irq` function when an interrupt occurs. In the `irq` function, `BlockLen` samples of the A/D input in `Rcvptr` (channel 1) are written to a length N `inputs` buffer, and `BlockLen` of the output samples in the `outputs` buffer are written to the D/A output via `Xmitptr` on channel 2. In C, pointers may be used as array names so that `Xmitptr[0]` is the first word pointed to by `Xmitptr`. As in the assembly core, the input samples are not in consecutive order. The right and left inputs are offset from `Rcvptr` respectively by $4i$ and $4i + 2$, $i = 0, \ldots,$ BlockLen $- 1$. The six output channels are accessed consecutively as offsets from `Xmitptr`. On channel 1 of the output, the input is echoed out. **You are to fill the buffer `outputs` with the windowed magnitude-squared FFT values by performing the operations listed above.**

In the main code, the `while(!input_full);` loop waits for N samples to collect in the `inputs` buffer. Next, the N inputs and outputs must be transferred. You are to write this portion of code. This portion of code is to be done first, within `BlockLen` sample times; otherwise the first `BlockLen` of samples of output would not be available on time. Once this loop is finished, the lengthy processing of the FFT can continue. During this processing, the DSP is interrupted every `BlockLen` samples to transfer samples. Once this processing is over, the infinite loop returns to `while(!input_full);` to wait for N samples to finish collecting.

The flow diagram in Figure 1.11 summarizes the operation of the interrupt handling routine

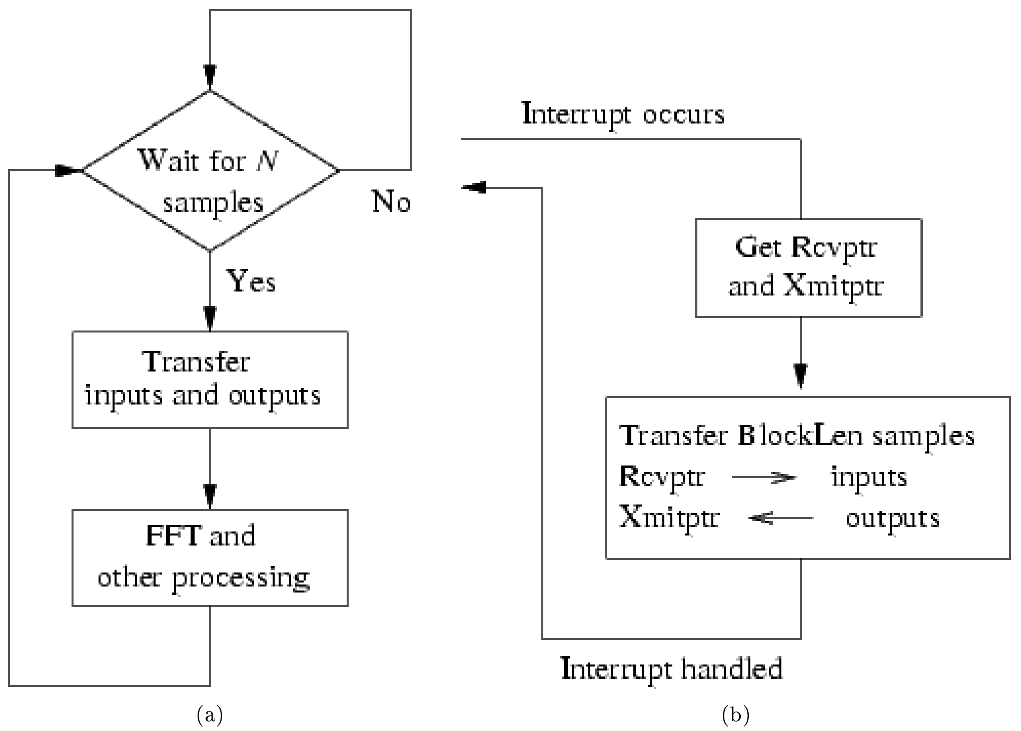


Figure 1.11: Overall program flow of the main function and the interrupt handling function. (a) main (b) interrupt handler

1.5.3.1.3 Assembly FFT Routine

As the list of operations indicates, bit-reversal and FFT computation are to be done in both C and assembly. For the assembly version, make sure that the line defining `C_FFT` is commented in `lab4main.c`. We are providing you with a shell assembly file, available at `v:\ece420\54x\dspclib\c_fft_given.asm` and shown in Appendix B (Section 1.5.3.4: Appendix B:), containing many useful declarations and some code. The code for performing bit-reversal and other declarations needed for the FFT routine are also provided in this section. **However, we would like you to enter this code manually, as you will be expected to understand its operation.**

The assembly file `c_fft_given.asm` contains two main parts, the data section starting with `.sect ".data"` and the program section starting with `.sect ".text"`. Every function and variable accessed in C must be preceded by a single underscore `_` in assembly and a `.global _name` must be placed in the assembly file for linking. In this example, `bit_rev_fft` is an assembly function called from the C program with a label `_bit_rev_fft` in the text portion of the assembly file and a `.global _bit_rev_fft` declaration. In each assembly function, the macro `ENTER_ASM` is called upon entering and `LEAVE_ASM` is called upon exiting. These macros are defined in `v:\ece420\54x\dspclib\core.inc`. The `ENTER_ASM` macro saves the status registers and `AR1`, `AR6`, and `AR7` when entering a function as required by the register use conventions. The `ENTER_ASM` macro also sets the status registers to the assembly conventions we have been using (i.e, `FRCT`=1 for fractional arithmetic and `CPL`=0 for `DP` referenced addressing). The `LEAVE_ASM` macro just restores the saved registers.

1.5.3.1.3.1 Parameter Passing

The parameter passing convention between assembly and C is simple for single input, single output assembly functions. From a C program, the input to an assembly program is in the low part of accumulator `A` with the output returned in the same place. When more than one parameter is passed to an assembly function, the parameters are passed on the stack (see the core file description for more information). We suggest that you avoid passing or returning more than one parameter. Instead, use global memory addresses to pass in or return more than one parameter. Another alternative is to pass a pointer to the start of a buffer intended for passing and returning parameters.

1.5.3.1.3.2 Registers Modified

When entering and leaving an assembly function, the `ENTER_ASM` and `LEAVE_ASM` macros ensure that certain registers are saved and restored. Since the C program may use any and all registers, the state of a register cannot be expected to remain the same between calls to assembly function(s). **Therefore, any information that needs to be preserved across calls to an assembly function must be saved to memory!**

Now, we explain how to use the FFT routine provided by TI for the C54x. The FFT routine `fft.asm` located in `v:\ece420\54x\dsplib\` computes an in-place, complex FFT. The length of the FFT is defined as a label `K_FFT_SIZE` and the algorithm assumes that the input starts at data memory location `_fft_data`. To have your code assemble for an N-point FFT, you will have to include the following label definitions in your assembly code.

```
N               .set      1024
K_FFT_SIZE      .set      N           ; size of FFT
K_LOGN          .set      10          ; number of stages (log_2(N))
```

In addition to defining these constants, you will have to include twiddle-factor tables for the FFT. These tables (twiddle1[33] and twiddle2[34]) are available in the shared directory `v:\ece420\54x\dsplib\`. Note that the tables are each N points long representing values from 0 to just shy of 180 degrees and must be accessed using a **circular pointer**. To include these tables at the proper location in memory with the appropriate labels referenced by the FFT, use the following

```
.sect  ".data"
.align  1024
sine         .copy "v:\ece420\54x\dsplib\twiddle1"
.align  1024
cosine       .copy "v:\ece420\54x\dsplib\twiddle2"
```

The FFT provided requires that the input be in bit-reversed order, with alternating real and imaginary components. Bit-reversed addressing is a convenient way to order input $x[n]$ into a FFT so that the output $X(k)$ is in sequential order (*i.e.* $X(0)$, $X(1)$, ..., $X(N-1)$ for an N-point FFT). The following table illustrates the bit-reversed order for an eight-point sequence.

Input Order	Binary Representation	Bit-Reversed Representation	Output Order
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1.1

The following routine performs the bit-reversed reordering of the input data. The routine assumes that the input is stored in data memory starting at the location labeled `_bit_rev_data`, which must be aligned to the least power of two greater than the input buffer length, and consists of alternating real and imaginary parts. Because our input data is going to be purely real in this lab, you will have to make sure that you set the imaginary parts to zero by zeroing out every other memory location.

```
bit_rev:
        STM     #_bit_rev_data,AR3          ; AR3 -> original input
        STM     #_fft_data,AR7              ; AR7 -> data processing buffer
        MVMM    AR7,AR2                     ; AR2 -> bit-reversed data
```

[33]http://cnx.org/content/m11827/latest/TWIDDLE1
[34]http://cnx.org/content/m11827/latest/TWIDDLE2

```
        STM     #K_FFT_SIZE-1,BRC
        RPTBD   bit_rev_end-1
        STM     #K_FFT_SIZE,AR0           ; AR0 = 1/2 size of circ buffer
        MVDD    *AR3+,*AR2+
        MVDD    *AR3-,*AR2+
         MAR     *AR3+0B
bit_rev_end:
         NOP
                RET
```

As mentioned, in the above code `_bit_rev_data` is a label indicating the start of the input data and `_fft_data` is a label indicating the start of a circular buffer where the bit-reversed data will be written. Note that although `AR7` is not used by the bit-reversed routine directly, it is used extensively in the FFT routine to keep track of the start of the FFT data space.

In general, to have a pointer index memory in bit-reversed order, the `AR0` register needs to be set to one-half the length of the circular buffer; a statement such as `ARx+0B` is used to move the `ARx` pointer to the next location. For more information regarding the bit-reversed addressing mode, refer to *page 5-18* in the *TI-54x CPU and Peripherals manual*[?]. Is it possible to bit-reverse a buffer in place? For a diagram of the ordering of the data expected by the FFT routine, see *Figure 4-10* in the *TI-54x Applications Guide*[?]. Note that the FFT code uses all the pointers available and does not restore the pointers to their original values.

1.5.3.1.4 C FFT Routine

A bit-reversing and FFT routine have also been provided in `lab4fft.c`, listed in Appendix C (Section 1.5.3.5: Appendix C:). **Again, make sure you understand how the bit reversal is taking place.** In `lab4main.c`, the line defining `C_FFT` must not be commented for use of the C FFT routine. The sine tables (twiddle factors) are located in sinetables.h[35] . This fft requires its inputs in two buffers, the real buffer `real` and the imaginary buffer `imag`, and the output is placed in the same buffers. The length of the FFT, `N`, and `logN` are defined in `lab4.h`, which is also listed in Appendix C (Section 1.5.3.5: Appendix C:). **When experimenting with the C FFT make sure you modify these length values instead of the ones in the assembly code and `lab4main.c`!**

1.5.3.1.5 Creating the Window

As mentioned, you will be using the FFT to compute the spectrum of a windowed input. For your implementation you will need to create a 1024-point Hamming window. First, create a Hamming window in matlab using the function `hamming`. For the assembly FFT, use `save_coef` to save the window to a file that can then be included in your code with the `.copy` directive. For the C FFT, use the matlab function `write_intvector_headerfile`[36] with `name` set to `'window'` and `elemperline` set to `8` to create the header file that is included in `lab4main.c`.

1.5.3.1.6 Displaying the Spectrum

Once the DFT has been computed, you must calculate the squared magnitude of the spectrum for display.

$$(|X(k)|)^2 = (\Re(X(k)))^2 + (\Im(X(k)))^2 \tag{1.7}$$

You may find the assembly instructions `squr` and `squra` useful in implementing (1.7).

[35]http://cnx.org/content/m11827/latest/sinetables.h
[36]http://cnx.org/content/m11827/latest/write_intvector_headerfile.m

Because the squared magnitude is always nonnegative, you can replace one of the magnitude values with a -1.0 as a trigger pulse for display on the oscilloscope. This is easily performed by replacing the DC term ($k = 0$) with a -1.0 when copying the magnitude values to the output buffer. The trigger pulse is necessary for the oscilloscope to lock to a specific point in the spectrum and keep the spectrum fixed on the scope.

1.5.3.1.7 Intrinsics

If you are planning on writing some of the code in C, then you may be forced to use intrinsics. Intrinsic instructions provide a way to use assembly instructions directly in C. An example of an intrinsic instruction is `bit_rev_data[0]=_smpyr(bit_rev_data[0],window[0])` which performs the assembly signed multiply round instruction. You may also find the `_lsmpy` instruction useful. For more information on intrinsics, see *page 6-22* of the *TI-C54x Optimizing C/C++ Compiler User's Guide*[?].

The following lines of code were borrowed from the C FFT to serve as an example of arithmetic operations in C. Save this code in a file called mathex.c and compile this file. Look at the resulting assembly file and investigate the differences between each block. Be sure to reference the compiler user's guide to find out what the state of the FRCT and OVM bits are. Does each block work properly for all possible values?

```
void main(void)
{
   int s1, s2;
   int t1, t2;
   int i1, i2;
   int n1 = 16383, n2 = 16382, n3 = 16381, n4 = 16380;

   /* Code for standard 32-bit hardware, */
   /* with x,y limited to 16 bits        */
   s1 = (n1*n2 + n3*n4) ≫ 15;
   s2 = (n1 + n2) ≫ 1;

   /* Code for TI TMSC5000 series */
   t1 = ((long int)(n1*n2) + (long int)(n3*n4)) ≫ 15;
   t2 = ((long int)n1 + (long int)n2) ≫ 1;

   /* Intrinsic code for TMS320C54X series */
   i1 = _sadd(_smpy(n1,n2), _smpy(n3,n4));
   i2 = _sshl(_sadd(n1, n2),-1);
}
```

1.5.3.1.8 Compiling and Linking

A working program can be produced by compiling the C code and linking assembly modules and the core module. The compiler translates C code to a relocatable assembly form. The linker assigns physical addresses on the DSP to the relocatable data and code segments, resolves `.global` references and links runtime libraries.

The procedure for compiling C code and linking assembly modules has been automated for you in the batch file `v:\ece420\54x\dsptools\c_asm.bat`. The name of the first file becomes the name of the executable. Once you have completed `lab4main.c` and `c_fft_given.asm`, type `c_asm lab4main.c c_fft_given.asm` to produce a `lab4main.out` file to be loaded onto the DSP. For the C FFT type `c_asm lab4main.c lab4fft.c` to produce `lab4main.out`. Load the output file onto the DSP as usual and confirm that valid FFTs are calculated. Once valid output is obtained, measure how many clock cycles it takes to compute both the assembly and C FFT.

1.5.3.2 Quiz Information

From your prelab experiments, you should be able to describe the effect of windowing and zero-padding on FFT spectral analysis. In your DSP system, experiment with different inputs, changing N and the type of window. Can you explain what happens as the input frequency is increased beyond the Nyquist rate? Does the $(|X(k)|)^2$ coincide with what you expect from Matlab? What is the relationship between the observed spectrum and the DTFT? What would happen if the FFT calculation takes longer than it takes to fill `inputs` with N samples? How long does it take to compute each FFT? What are the tradeoffs between writing code in C versus assembly?

1.5.3.3 Appendix A:

lab4main.c[37]

```
/* v:/ece420/54x/dspclib/lab4main.c */
/* dgs - 9/14/2001                  */
/* mdk - 2/10/2004   C FFT update   */

#include "v:/ece420/54x/dspclib/core.h"

/* comment the next line to use assembly fft */
#define C_FFT

 #ifdef C_FFT /* Use C FFT */

    #include "window.h"
    #include "lab4.h" /* Number of C FFT points defined here */

    /* function defined in lab4fft.c */
    void fft(void);

    /* FFT data buffers */
    int real[N]; /* Real part of data      */
    int imag[N]; /* Imaginary part of data */

 #else        /* Use assembly FFT */

    #define N 1024   /* Number of assembly FFT points */

    /* Function defined by c_fft_given.asm */
    void bit_rev_fft(void);

    /* FFT data buffers (declared in c_fft_given.asm) */
    extern int bit_rev_data[N*2]; /* Data input for bit-reverse function */
    extern int fft_data[N*2];     /* In-place FFT & Output array         */
    extern int window[N];         /* The Hamming window                  */
```

[37]http://cnx.org/content/m11827/latest/lab4main.c

```

#endif       /* C_FFT */


/* Our input/output buffers */
int inputs[N];
int outputs[N];

volatile int input_full = 0;  /* volatile means interrupt changes it */
int count = 0;


interrupt void irq(void)
{
  int *Xmitptr,*Rcvptr;                      /* pointers to Xmit & Rcv Bufs   */
  int i;

  static int in_irq = 0;           /* Flag to prevent reentrance */

  /* Make sure we're not in the interrupt (should never happen) */
  if( in_irq )
    return;

  /* Mark we're processing, and enable interrupts */
  in_irq = 1;
  enable_irq();

  /* The following waitaudio call is guaranteed not to
     actually wait; it will simply return the pointers. */
  WaitAudio(&Rcvptr,&Xmitptr);

  /* input_full should never be true... */
  if( !input_full )
  {
    for (i=0; i<BlockLen; i++)
    {
      /* Save input, and echo to channel 1 */
      inputs[count] = Xmitptr[6*i] = Rcvptr[4*i];

      /* Send FFT output to channel 2 */
      Xmitptr[6*i+1] = outputs[count];

      count++;
    }
  }

  /* Have we collected enough data yet? */
  if( count >= N )
    input_full = 1;

  /* We're not in the interrupt anymore... */
  disable_irq();
  in_irq = 0;
}


main()
{
  /* Initialize IRQ stuff */
  count = 0;
  input_full = 0;
  SetAudioInterrupt(irq);        /* Set up interrupts */

  while (1)
  {
    while( !input_full );     /* Wait for a data buffer to collect */

      /* From here until we clear input_full can only take *
       * BlockLen sample times, so don't do too much here. */

      /* First, transfer inputs and outputs */

 #ifdef C_FFT /* Use C FFT */
      /* I n s e r t   c o d e   t o   f i l l   */
      /* C   F F T   b u f f e r s               */

 #else        /* Use assembly FFT */
      /* I n s e r t   c o d e   t o   f i l l   */
      /* a s s e m b l y   F F T   b u f f e r s */

 #endif       /* C_FFT */

      /* Done with that... ready for new data collection */
      count = 0;      /* Need to reset the count                */
      input_full = 0; /* Mark we're ready to collect more data  */

      /*********************************************************/
      /* Now that we've gotten the data moved, we can do the    */
      /* more lengthy processing.                               */

 #ifdef C_FFT /* Use C FFT */

      /* Multiply the input signal by the Hamming window.       */
      /* . . . i n s e r t   C / a s m   code . . .             */

      /* Bit-reverse and compute FFT in C                       */
      fft();

      /* Now, take absolute value squared of FFT                */
      /* . . . i n s e r t   C / a s m   code . . .             */

      /* Last, set the DC coefficient to -1 for a trigger pulse */
    /* . . . i n s e r t   C / a s m   code . . .           */

    /* done, wait for next time around!                     */


#else       /* Use assembly FFT */

    /* Multiply the input signal by the Hamming window.     */
    /* . . . i n s e r t   C / a s m   code . . .           */

    /* Bit-reverse and compute FFT in assembly              */
    bit_rev_fft();

    /* Now, take absolute value squared of FFT              */
    /* . . . i n s e r t   C / a s m   code . . .           */

    /* Last, set the DC coefficient to -1 for a trigger pulse */
    /* . . . i n s e r t   C / a s m   code . . .           */

    /* done, wait for next time around!                     */


#endif      /* C_FFT */

  }
}
```

1.5.3.4 Appendix B:

c_fft_given.asm[38]

```
; v:\ece420\54x\dspclib\c_fft_given.asm
; dgs - 9/14/2001
    .copy "v:\ece420\54x\dspclib\core.inc"

    .global _bit_rev_data
    .global _fft_data
    .global _window

    .global _bit_rev_fft

     .sect ".data"

     .align 4*N
 _bit_rev_data .space 16*2*N ; Input to _bit_rev_fft

```

[38]http://cnx.org/content/m11827/latest/c_fft_given.asm

```
    .align 4*N
_fft_data .space 16*2*N ; FFT output buffer


; Copy in the Hamming window
_window ; The Hamming window
    .copy "window.asm"

    .sect ".text"

_bit_rev_fft
    ENTER_ASM

    call bit_rev                    ; Do the bit-reversal.

    call fft        ; Do the FFT

    LEAVE_ASM
    RET

bit_rev:
    STM     #_bit_rev_data,AR3          ; AR3 -> original input
    STM     #_fft_data,AR7              ; AR7 -> data processing buffer
    MVMM    AR7,AR2                     ; AR2 -> bit-reversed data
    STM     #K_FFT_SIZE-1,BRC
    RPTBD   bit_rev_end-1
    STM     #K_FFT_SIZE,AR0             ; AR0 = 1/2 size of circ buffer
    MVDD    *AR3+,*AR2+
    MVDD    *AR3-,*AR2+
    MAR     *AR3+0B
bit_rev_end:
    NOP
    RET

; Copy the actual FFT subroutine.
fft_data  .set _fft_data ; FFT code needs this.
    .copy  "v:\ece420\54x\dsplib\fft.asm"


; If you need any more assembly subroutines, make sure you name them
; _name, and include a ".global _name" directive at the top. Also,
; don't forget to use ENTER_ASM at the beginning, and LEAVE_ASM
; and RET at the end!
```

1.5.3.5 Appendix C:

lab4.h[39]

[39]http://cnx.org/content/m11827/latest/lab4.h

```
#define N 1024       /* Number of FFT points */
#define logN 10
```

lab4fft.c[40]

```
/***************************************************************/
/* lab4fft.c                                                   */
/* Douglas L. Jones                                            */
/* University of Illinois at Urbana-Champaign                  */
/* January 19, 1992                                            */
/* Changed for use w/ short integers and lookup table for ECE420 */
/* Matt Kleffner                                               */
/* February 10, 2004                                           */
/*                                                             */
 /*   fft: in-place radix-2 DIT DFT of a complex input           */
 /*                                                             */
 /*   Permission to copy and use this program is granted         */
 /*   as long as this header is included.                        */
 /*                                                             */
 /* WARNING:                                                    */
 /*   This file is intended for educational use only, since most */
 /*   manufacturers provide hand-tuned libraries which typically */
 /*   include the fastest fft routine for their DSP/processor    */
 /*   architectures. High-quality, open-source fft routines      */
 /*   written in C (and included in MATLAB) can be found at      */
 /*   http://www.fftw.org                                        */
 /*                                                             */
 /*   #defines expected in lab4.h                                */
 /*         N:   length of FFT: must be a power of two           */
 /*      logN:   N = 2**logN                                     */
 /*                                                             */
 /*   16-bit-limited input/output (must be defined elsewhere)    */
 /*   real:   integer array of length N with real part of data   */
 /*   imag:   integer array of length N with imag part of data   */
 /*                                                             */
 /*   sinetables.h must                                          */
 /*   1) #define Nt to an equal or greater power of two than N   */
 /*   2) contain the following integer arrays with               */
 /*      element magnitudes bounded by M = 2**15-1:              */
 /*         costable:   M*cos(-2*pi*n/Nt), n=0,1,...,Nt/2-1      */
 /*         sintable:   M*sin(-2*pi*n/Nt), n=0,1,...,Nt/2-1      */
 /*                                                             */
 /***************************************************************/

 #include "lab4.h"
```

[40]http://cnx.org/content/m11827/latest/lab4fft.c

```
#include "sinetables.h"

extern int real[N];
extern int imag[N];

void fft(void)
{
   int   i,j,k,n1,n2,n3;
   int   c,s,a,t,Wr,Wi;

   j = 0;            /* bit-reverse */
   n2 = N ≫ 1;
   for (i=1; i < N - 1; i++)
   {
      n1 = n2;
      while ( j >= n1 )
      {
         j = j - n1;
         n1 = n1 ≫ 1;
      }
      j = j + n1;

      if (i < j)
      {
         t = real[i];
         real[i] = real[j];
         real[j] = t;
         t = imag[i];
         imag[i] = imag[j];
         imag[j] = t;
      }
   }

   /* FFT */
   n2 = 1; n3 = Nt;

   for (i=0; i < logN; i++)
   {
      n1 = n2;      /* n1 = 2**i     */
      n2 = n2 + n2; /* n2 = 2**(i+1) */
      n3 = n3 ≫ 1; /* cos/sin arg of -6.283185307179586/n2 */
      a = 0;

      for (j=0; j < n1; j++)
      {
         c = costable[a];
         s = sintable[a];
         a = a + n3;

         for (k=j; k < N; k=k+n2)
         {
            /* Code for standard 32-bit hardware, */
            /* with real,imag limited to 16 bits  */
            /*
            Wr = (c*real[k+n1] - s*imag[k+n1]) ≫ 15;
            Wi = (s*real[k+n1] + c*imag[k+n1]) ≫ 15;
            real[k+n1] = (real[k] - Wr) ≫ 1;
            imag[k+n1] = (imag[k] - Wi) ≫ 1;
            real[k] = (real[k] + Wr) ≫ 1;
            imag[k] = (imag[k] + Wi) ≫ 1;
            */
            /* End standard 32-bit code */

            /* Code for TI TMS320C54X series */

            Wr = ((long int)(c*real[k+n1]) - (long int)(s*imag[k+n1])) ≫ 15;
            Wi = ((long int)(s*real[k+n1]) + (long int)(c*imag[k+n1])) ≫ 15;
            real[k+n1] = ((long int)real[k] - (long int)Wr) ≫ 1;
            imag[k+n1] = ((long int)imag[k] - (long int)Wi) ≫ 1;
            real[k] = ((long int)real[k] + (long int)Wr) ≫ 1;
            imag[k] = ((long int)imag[k] + (long int)Wi) ≫ 1;

            /* End code for TMS320C54X series */

            /* Intrinsic code for TMS320C54X series */
            /*
            Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
            Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
            real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
            imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
            real[k] = _sshl(_sadd(real[k], Wr),-1);
            imag[k] = _sshl(_sadd(imag[k], Wi),-1);
            */
            /* End intrinsic code for TMS320C54X series */
         }
      }
   }
   return;
}
```

1.6 Lab 5

1.6.1 Lab 5: Prelab[41]

1.6.1.1 Prelab: Matlab Preparation

We have made considerable use of Matlab in previous labs to design filters and determine frequency responses of systems. Matlab is also very useful as a simulation tool.

Use the following Matlab code skeleton to simulate your system and fill in the incomplete portions. Note that the code is not complete and will not execute properly as written. How does the spectrum of the transmitted signal change with T_{symb}?

lab_5_prelab.m[42]

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Matlab code skeleton for Digital Transmitter

close all;clear;

% Generate random bits
bits_per_symbol=2;
num_symbols=64;
numbits=bits_per_symbol*num_symbols;
 bits=rand(1,numbits)>0.5;

 Tsymb=32;           % samples per symbol


 % These are the 4 frequencies to choose from
 % Note that 32 samples per symbol does not correspond to
 % an integer number of periods at these frequencies
 omega1 =  9*pi/32;
 omega2 = 13*pi/32;
 omega3 = 17*pi/32;
 omega4 = 21*pi/32;


 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % Transmitter section

 % Initialize transmit sequence
 index=1; % Initialize bit index
 n=1; % Initialize sample index
 phi=0; % Initialize phase offset

 % Generate 64 32-sample symbols
 while (n<=num_symbols*Tsymb)

   if (bits(index:index+1) == [0 0])
      sig(n:n+Tsymb-1) = sin(omega1*[0:Tsymb-1]+phi);
      phi = omega1*Tsymb+phi; % Calculate phase offset for next symbol
      phi = mod(phi, 2*pi); % Restrict phi to [0,2*pi)
```

[41]This content is available online at <http://cnx.org/content/m10661/2.5/>.
[42]http://cnx.rice.edu/modules/m10661/latest/lab_5_prelab.m

```

     % -----------> Insert code here <-------------%

      end % end if-else statements

      index=index+2; % increment bit counter so we look at next 2 bits

      n=n+Tsymb;
    end   % end while


    % Show transmitted signal and its spectrum
    % ---------------> Insert code here <----------------%
```

1.6.2 Lab 5: Theory[43]

1.6.2.1 Frequency Shift Keying

Frequency Shift Keying (**FSK**) is a scheme to transmit digital information across an analog channel. Binary data bits are grouped into blocks of a fixed size, and each block is represented by a unique carrier frequency, called a **symbol**, to be sent across the channel. [44] This requires having a unique symbol for each possible combination of data bits in a block. In this laboratory exercise each symbol represents a two-bit block; therefore, there will be four different symbols.

The carrier frequency is kept constant over some number of samples known as the symbol period (T_{symb}). The symbol rate, defined as F_{symb}, is a fraction of the board's sampling rate, F_s. For our sampling rate of 44.1 kHz and a symbol period of 32, the symbol rate is 44.1k/32 symbols per second.

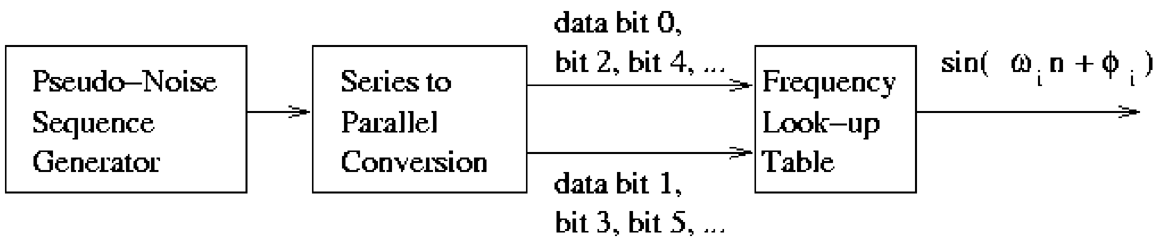

Pseudo–Noise Sequence Generator
Series to Parallel Conversion
data bit 0, bit 2, bit 4, ...
data bit 1, bit 3, bit 5, ...
Frequency Look–up Table
sin(ω_i n + φ_i)

Figure 1.12: Pseudo-noise sequence generator and FSK transmitter.

1.6.2.2 Pseudo-Noise Sequence Generator

The input bits to the transmitter are provided by the special shift-register, called a **pseudo-noise sequence generator** (**PN generator**), on the left side of Figure 1.12. A PN generator produces a sequence of bits that appears random. The PN sequence will repeat with period $2^B - 1$, where B is the width in bits of the shift register. A more detailed diagram of the PN generator alone appears in Figure 1.13.

[43]This content is available online at <http://cnx.org/content/m11849/1.3/>.

[44]The receiver then looks at the recovered symbol frequency to determine which block of bits was sent and converts it back to the appropriate binary data.

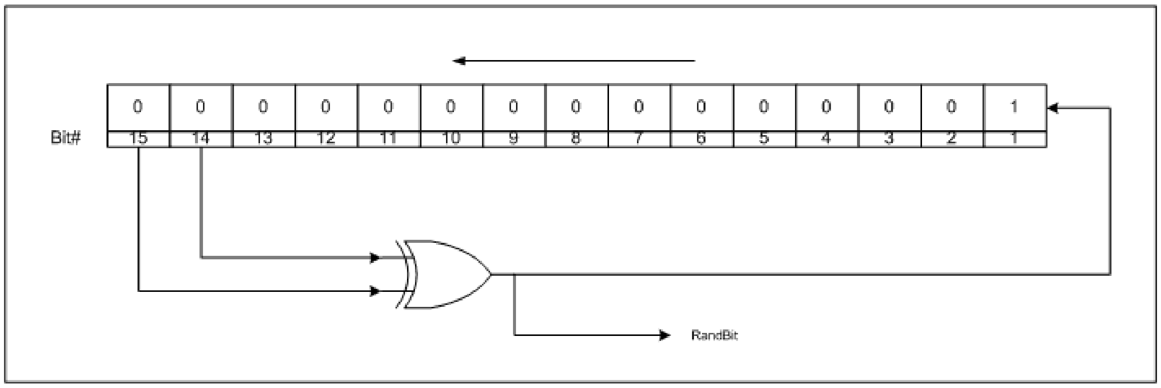


Figure 1.13: PN generator.

As shown in Figure 1.13, the PN generator is simply a shift-register and XOR gate. Bits 14 and 15 of the shift-register are XORed together and the result is shifted into the lowest bit of the register. This lowest bit is the output of the PN generator.

The PN generator is a useful source of random data bits for system testing. We can simulate the bit sequence that would be transmitted by a user as the random bits generated by the PN generator. Since communication systems tend to randomize the bits seen by the transmission scheme so that bandwidth can be efficiently utilized, the PN generator is a good data model.[45]

1.6.2.3 Series-to-Parallel Conversion

The shift-register produces one output bit at a time. Because each symbol the system transmits will encode two bits, we require the series-to-parallel conversion to group the output bits from the shift-register into blocks of two bits so that they can be mapped to a symbol.

1.6.2.4 Frequency Look-up Table

This is responsible for mapping blocks of bits to one of four frequencies as shown in Figure 1.12. Each possible two-bit block of data from the series-to-parallel conversion is mapped to a different carrier frequency ω_i

> NOTE: Note that the subscript i denotes a symbol's index in the transmitted signal; *i.e.*, the first symbol sent has index $i = 1$, the second symbol sent has index $i = 2$, and so on. Therefore, ω_i denotes the frequency and ϕ_i denotes the phase offset of the i^{th} transmitted symbol.

These frequencies are then used to generate the waveforms. The mappings for this assignment are given in Table 1.2.

Data Chunk	**Carrier Frequency ω_i**
00	$\frac{9\pi}{32}$
01	$\frac{13\pi}{32}$
11	$\frac{17\pi}{32}$
10	$\frac{21\pi}{32}$

[45]PN generators have other applications in communications, notably in the Code Division Multiple Access schemes used by cellular telephones.

Table 1.2

One way to implement this mapping is by using a look-up table. The two-bit data block can be interpreted as an offset into a frequency table where we have stored the possible transmission frequencies. Note that since each frequency mapping defines a symbol, this mapping is done at the symbol rate F_{symb}, or once for every T_{symb} DSP samples.

The symbol bit assignments are such that any two adjacent frequencies map to data blocks that differ by only one bit. This assignment is called **Gray coding** and helps reduce the number of bit errors made in the event of a received symbol error.

1.6.2.5 Phase Continuity

In order to minimize the bandwidth used by the transmitted signal, you should ensure that the phase of your transmitted waveform is continuous between symbols; i.e., the beginning phase of any symbol must be equal to the ending phase of the previous symbol. For instance, if a symbol of frequency $\frac{9\pi}{32}$ begins at phase 0, the symbol will end 31 output samples later at phase $31\frac{9\pi}{32}$. To preserve phase continuity, the next output sample must be at phase $32\frac{9\pi}{32}$, which is equivalent to phase π. Therefore, the next symbol, whatever its frequency, must begin at phase π. For each symbol, you must choose ϕ_i in the expression $\sin(\omega_i n + \phi_i)$ to create this continuity.

1.6.3 Lab 5: Lab[46]

In this lab you are to implement and optimize the **frequency shift keying** (**FSK**) digital transmitter and **pseudo-noise** (**PN**) sequence generator shown in this figure. For the lab grade, you will be judged on the execution time of your system (memory usage need not be minimized).

1.6.3.1 Implementation

You will implement and optimize the complete system shown in this figure. over the next two weeks. You may write in C, assembly, or any combination of the two; choose whatever will allow you to write the fastest code. The optimization process will probably be much easier if you plan for optimization before you begin any programming.

1.6.3.1.1 Reference Implementation

We provide for you in Appendix A a complete C implementation of this lab. It consists of the main C file fsk.c and a C-callable SINE function in SINE.asm. Compile and link this using the batch file C_ASM.bat by typing "C_ASM fsk SINE" on the command prompt. Load FSK.out onto the DSP, and run the code. Observe the output (Channel 1) on the scope.

After taking a look at the source code of this implementation, you'll realize that this is a rather inefficient implementation. It's there to show you what output is expected of your code, and the computational efficiency of your code will be judged against it. While the given code might serve as a starting point, you should do whatever you need to do to make your code as efficient as possible, while producing the same output as the given code.

1.6.3.1.2 PN Generator

Once you have planned your program strategy, implement the PN generator from Figure 2 (Section 1.6.2.2: Pseudo-Noise Sequence Generator) and verify that it is working. If you are programming in assembly, you may wish to refer to the description of assembly instructions for logical operations in *Section 2-2* of the *C54x Mnemonic Instruction Set* reference. Initialize the shift register to one.

In testing the PN generator, you may find the file `v:  \ece320\54x\dspclib\pn_output.mat`[47] helpful. To use it, type `load v:\ece320\54x\dspclib\pn_output` at the Matlab command prompt. This will load a vector `pn_output` into memory. The vector contains 500 elements, which are the first 500 output bits of the PN generator. Be prepared to prove to a TA that your PN generator works properly as part of your quiz.

1.6.3.1.3 Transmitter

For your transmitter implementation you are to use the data-block-to-carrier-frequency mapping in this table (Table 1.2) and a digital symbol period of $T_{\text{symb}} = 32$ samples.

Viewing the transmitted signal on the oscilloscope may help you determine whether your code works properly, but you should check it more carefully by setting breakpoints in Code Composer and using the `Memory` option from the `View` menu to view the contents of memory. The vector signal analyzer (VSA) provides another method of testing.

1.6.3.1.4 Testing with the VSA

The VSA is an instrument capable of demodulating digital signals. You may use the VSA to demodulate your FSK signal and display the symbols received.

[46]This content is available online at <http://cnx.org/content/m11848/1.3/>.
[47]http://cnx.rice.edu/modules/m11848/latest/pn_output.mat

1.6.3.1.4.1 Configuring the VSA

The VSA is the big HP unit on a cart in the front of the classroom. Plug the output from the DSP board into the "Channel 1" jack on the front of the vector signal analyzer, and then turn on the analyzer and follow these instructions to display your output:

After powering the signal analyzer up, the display will not be in the correct mode. Use the following sequence of keypresses to set it up properly:

> NOTE: If this doesn't work, hit "Save/Recall," F7 (Catalog), point at `ECE320.STA` with the wheel, and hit F5 (Recall State) and F1 (Enter).

- "Freq" button, followed by F1 (center), 11.025 (on the keypad), and F3 (KHz)
- F2 (span), 22, and F3 (KHz)
- "Range," then F5 (ch1 autorange up/down)
- "Instrument Mode," then F3 (demodulation)

1.6.3.1.4.2 Viewing the signal spectrum on the VSA

The VSA is also capable of displaying the spectrum of a signal. Hook up the output of your PN generator to the VSA and set it up properly to view the spectrum of the random sequence. Hit "Instrument Mode" and then F1 (Scalar) to see the spectrum. Note that you can also use your Lab 4 code for this purpose.

Does what you see match the Matlab simulations?

1.6.3.1.5 Optimization

One purpose of this lab is to teach optimization and efficient code techniques. For this reason, for your lab grade **you will be judged primarily on the total execution time of your system.** You are not required to optimize memory use. Note that by execution time we mean cycle count, not the number of instructions in your program. Remember that several of the TMS320C54xx instructions take more than one cycle. The multicycle instructions are primarily the multi-word instructions, including instructions that take immediates, like `stm`, and instructions using direct addressing of memory (such as `ld *(temp),A`). Branch and repeat statements also require several cycles to execute. Most C instructions take more than one cycle. The debugger can be used to determine the exact number of cycles used by your code; ask your TA to demonstrate. However, since the number of execution cycles used by an instruction is usually determined by the number of words in its encoding, the easiest way to estimate the number of cycles used by your code is to count the number of instruction words in the `.lst` file or the disassembly window in the debugger.

We will grade you based on the number of cycles used between the return from the `WAITDATA` call and the arrival at the next `WAITDATA` call in assembly, or the return from one WaitAudio call and the arrival at the next WaitAudio call in C. If the number of cycles between the two points is variable, the maximum possible number of cycles will be counted. You must use the `core.asm` file in `v:\ece320\54x\dsplib\core.asm` or the C core file in `v:\ece320\54x\dspclib\core.asm` as provided by the TAs; **these files may not be modified**. You explicitly may not change the number of samples read and written by each `WAITDATA` or WaitAudio call! We reserve the right to test your code by substituting the test vector core file.

1.6.3.2 Grading

This is a two-week lab. Your prelab is due a week after the quiz for Lab 4, and the quizzing occurs two weeks after the quiz for Lab 4.

Grading for this lab will be a bit different from past labs:

- 1 point: Prelab
- 2 points: Working code, implemented from scratch in assembly language or C.

- 5 points: Optimization. These points will be assigned based on your cycle counts and the optimizations you have made.
- 2 points: Oral quiz.

1.6.3.3 Appendix A:

fsk.c[48]
SINE.asm[49]
C_ASM.bat[50]

```
/* ECE320, Lab 5, Reference Implementation (Non-Optimized) */
/* Michael Frutiger 2/24/04 */

#include "v:/ece320/54x/dspclib/core.h"   /* Declarations for core file */

main()
{
    int *Rcvptr,*Xmitptr;                 /* pointers to Xmit & Rcv Bufs   */
    int r1,r2;      // temp random bit storage
     int symbol;           // 0 for [00], 1 for [01], 2 for [10], 3 for [11]
     int n;          // dummy variable

     int freqs[4] = {9, 13, 21, 17};     // 32*freqs
     int phase[32];
     int output[64];                     // temp output storage


     // Initial PN generator register contents
     int seed = 1;

     // Initial phase
     int prev_phase = 0;


     while( 1 )
     {
           /* Wait for a new block of samples */
           WaitAudio(&Rcvptr,&Xmitptr);

         // Get next two random bits
           r1 = randbit( &seed );
           r2 = randbit( &seed );
           // Convert 2 bit binary number to decimal
           symbol = series2parallel(r1,r2);

           for (n=0; n<32; n++)
           {
```

[48]http://cnx.org/content/m11848/latest/fsk.c
[49]http://cnx.org/content/m11848/latest/SINE.asm
[50]http://cnx.org/content/m11848/latest/C_ASM.bat

```
                        phase[n] = ( freqs[symbol]*n + prev_phase ) % 64;   // get into 0 to 64
                        if (phase[n] > 32) phase[n]=phase[n]-64;    // get into -32 to 32 range
                        phase[n] = phase[n] * 1024;    // 1024=2^15*1/32
                                                       // [-2^15 2^15] range for use with
                                                       // SINE.asm
                }
                sine(&phase[0], &output[0], 32);                // compute SINE, put result in o
31]
                prev_phase = ( freqs[symbol]*32 + prev_phase ) % 64;   // save current phase off

                        // Get next two random bits
                r1 = randbit( &seed );
                r2 = randbit( &seed );
                // Convert 2 bit binary number to decimal
                symbol = series2parallel(r1,r2);

                for (n=0; n<32; n++)
                {
                        phase[n] = ( freqs[symbol]*n + prev_phase ) % 64;
                        if (phase[n] > 32) phase[n]=phase[n]-64;
                        phase[n] = phase[n] * 1024;
                }
                sine(&phase[0], &output[32], 32);
                prev_phase = ( freqs[symbol]*32 + prev_phase ) % 64;


                // Transfer the two symbols to transmit buffer
                for (n=0; n<64; n++)
                {
                        Xmitptr[6*n] = output[n];
                }

        }
}


// Converts 2 bit binary number (r2r1) to decimal
int series2parallel(int r2, int r1)
{
  if ((r2==0)&&(r1==0)) return 0;
  else if ((r2==0)&&(r1==1)) return 1;
  else if ((r2==1)&&(r1==0)) return 2;
  else return 3;
}

//Returns as an integer a random bit, based on the 15 low-significance bits in iseed (whic
//modified for the next call).
int randbit(unsigned int *iseed)
{
  unsigned int newbit; // The accumulated XORs.
  newbit =  (*iseed >> 14) & 1 ^ (*iseed >> 13)  & 1;  // XOR bit 15 and bit 14
  // Leftshift the seed and put the result of the XORs in its bit 1.
  *iseed=(*iseed ≪ 1) | newbit;
  return (int) newbit;
}
```

Chapter 2

Project Labs

2.1 Adaptive Filtering

2.1.1 Adaptive Filtering: LMS Algorithm[1]

2.1.1.1 Introduction

Figure 2.1 is a block diagram of system identification using adaptive filtering. The objective is to change (adapt) the coefficients of an FIR filter, W, to match as closely as possible the response of an unknown system, H. The unknown system and the adapting filter process the same input signal $x[n]$ and have outputs $d[n]$(also referred to as the desired signal) and $y[n]$.

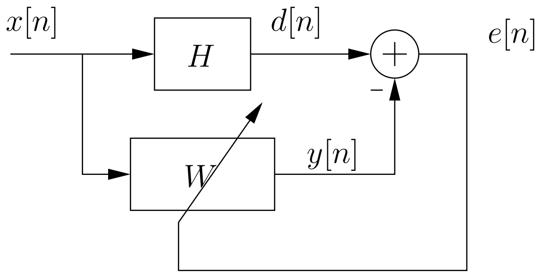


Figure 2.1: System identification block diagram.

2.1.1.1.1 Gradient-descent adaptation

The adaptive filter, W, is adapted using the least mean-square algorithm, which is the most widely used adaptive filtering algorithm. First the error signal, $e[n]$, is computed as $e[n] = d[n] - y[n]$, which measures the difference between the output of the adaptive filter and the output of the unknown system. On the

[1]This content is available online at <http://cnx.org/content/m10481/2.14/>.

basis of this measure, the adaptive filter will change its coefficients in an attempt to reduce the error. The coefficient update relation is a function of the error signal squared and is given by

$$h_{n+1}[i] = h_n[i] + \frac{\mu}{2}\left(-\frac{\partial(|e|)^2}{\partial h_n[i]}\right) \tag{2.1}$$

The term inside the parentheses represents the gradient of the squared-error with respect to the i^{th} coefficient. The gradient is a vector pointing in the direction of the change in filter coefficients that will cause the greatest increase in the error signal. Because the goal is to minimize the error, however, (2.1) updates the filter coefficients in the direction opposite the gradient; that is why the gradient term is negated. The constant μ is a step-size, which controls the amount of gradient information used to update each coefficient. After repeatedly adjusting each coefficient in the direction opposite to the gradient of the error, the adaptive filter should converge; that is, the difference between the unknown and adaptive systems should get smaller and smaller.

To express the gradient decent coefficient update equation in a more usable manner, we can rewrite the derivative of the squared-error term as

$$\begin{aligned}\frac{\partial(|e|)^2}{\partial h[i]} &= 2\frac{\partial e}{\partial h[i]}e \\ &= 2\frac{\partial(d-y)}{\partial h[i]}e \\ &= \left(2\frac{\partial\left(d-\sum_{i=0}^{N-1}h[i]x[n-i]\right)}{\partial h[i]}\right)(e)\end{aligned} \tag{2.2}$$

$$\frac{\partial(|e|)^2}{\partial h[i]} = 2(-x[n-i])e \tag{2.3}$$

which in turn gives us the final LMS coefficient update,

$$h_{\mathrm{n}+1}[i] = h_n[i] + \mu e x[n-i] \tag{2.4}$$

The step-size μ directly affects how quickly the adaptive filter will converge toward the unknown system. If μ is very small, then the coefficients change only a small amount at each update, and the filter converges slowly. With a larger step-size, more gradient information is included in each update, and the filter converges more quickly; however, when the step-size is too large, the coefficients may change too quickly and the filter will diverge. (It is possible in some cases to determine analytically the largest value of μ ensuring convergence.)

2.1.1.2 MATLAB Simulation

Simulate the system identification block diagram shown in Figure 2.1.

Previously in MATLAB, you used the `filter` command or the `conv` command to implement shift-invariant filters. Those commands will not work here because adaptive filters are shift-varying, since the coefficient update equation changes the filter's impulse response at every sample time. Therefore, implement the system identification block on a sample-by-sample basis with a `do` loop, similar to the way you might implement a time-domain FIR filter on a DSP. For the "unknown" system, use the fourth-order, low-pass, elliptical, IIR filter designed for the IIR Filtering: Filter-Design Exercise in MATLAB (Section 1.4.2).

Use Gaussian random noise as your input, which can be generated in MATLAB using the command `randn`. Random white noise provides signal at all digital frequencies to train the adaptive filter. Simulate the system with an adaptive filter of length 32 and a step-size of 0.02. Initialize all of the adaptive filter coefficients to zero. From your simulation, plot the error (or squared-error) as it evolves over time and plot the frequency response of the adaptive filter coefficients at the end of the simulation. How well does your adaptive filter match the "unknown" filter? How long does it take to converge?

Once your simulation is working, experiment with different step-sizes and adaptive filter lengths.

2.1.1.3 Processor Implementation

Use the same "unknown" filter as you used in the MATLAB simulation.

Although the coefficient update equation is relatively straightforward, consider using the `lms` instruction available on the TI processor, which is designed for this application and yields a very efficient implementation of the coefficient update equation.

To generate noise on the DSP, you can use the PN generator from the Digital Transmitter: Introduction to Quadrature Phase-Shift Keying[2], but shift the PN register contents up to make the sign bit random. (If the sign bit is always zero, then the noise will not be zero-mean and this will affect convergence.) Send the desired signal, $d[n]$, the output of the adaptive filter, $y[n]$, and the error to the D/A for display on the oscilloscope.

When using the step-size suggested in the MATLAB simulation section, you should notice that the error converges very quickly. Try an extremely small μ so that you can actually watch the amplitude of the error signal decrease towards zero.

2.1.1.4 Extensions

If your project requires some modifications to the implementation here, refer to ***Haykin*** [6] and consider some of the following questions regarding such modifications:

- How would the system in Figure 2.1 change for different applications? (noise cancellation, equalization, *etc.*)
- What happens to the error when the step-size is too large or too small?
- How does the length of an adaptive FIR filters affect convergence?
- What types of coefficient update relations are possible besides the described LMS algorithm?

2.2 Audio Effects

2.2.1 Audio Effects: Real-Time Control with the Serial Port[3]

2.2.1.1 Implementation

For this exercise, you will extend the system from Audio Effects: Using External Memory (Section 2.2.2) to generate a feedback-echo effect. You will then extend this echo effect to use the serial port on the DSP EVM. The serial interface will receive data from a MATLAB GUI that allows the two system gains and the echo delay to be changed using on-screen sliders.

[2]"Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" <http://cnx.org/content/m10042/latest/>
[3]This content is available online at <http://cnx.org/content/m10483/2.24/>.

2.2.1.1.1 Feedback system implementation

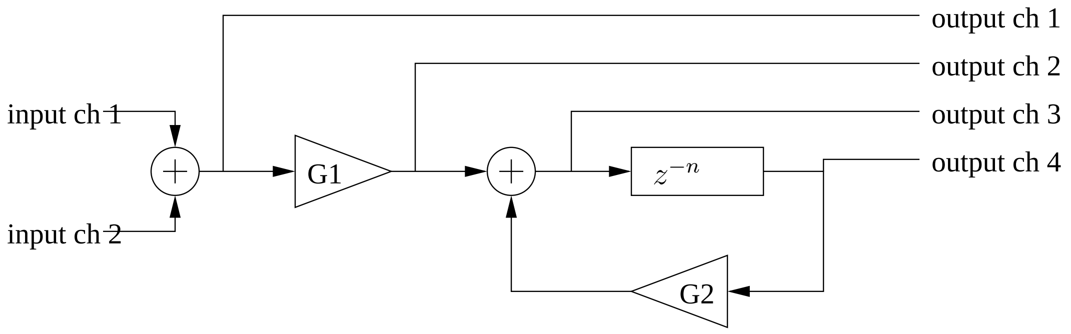


Figure 2.2: Feedback System with Test Points

First, modify code from Audio Effects: Using External Memory (Section 2.2.2) to create the feedback-echo system shown in Figure 2.2. A one-tap feedback-echo is a simple audio effect that sounds remarkably good. You will use both channels of input by summing the two inputs so that either or both may be used as an input to the system. Also, send several test signals to the six-channel board's D/A converters:

- The summed input signal
- The input signal after gain stage G_1
- The data going into the long delay
- The data coming out of the delay

You will also need to set both the input gain G_0 and the feedback gain G_1 to prevent overflow.

As you implement this code, ensure that the delay `n` and the gain values G_1 and G_2 are stored in memory and can be easily changed using the debugger. If you do this, it will be easier to extend your code to accept its parameters from MATLAB in MATLAB Interface Implementation (Section 2.2.1.1.2: MATLAB interface implementation).

To test your echo, connect a CD player or microphone to the input of the DSP EVM, and connect the output of the DSP EVM to a loudspeaker. Verify that an input signal echoes multiple times in the output and that the spacing between echoes matches the delay length you have chosen.

2.2.1.1.2 MATLAB interface implementation

After studying the MATLAB interface outlined at the end of Using the Serial Port with a MATLAB GUI[4], write MATLAB code to send commands to the serial interface based on three sliders: two gain sliders (for G_1 and G_2) and one delay slider (for `n`). Then modify your code to accept those commands and change the values for G_1, G_2 and `n`. Make sure that `n` can be set to values spanning the full range of 0 to 131,072, although it is not necessary that every number in that range be represented.

[4]"Using the Serial Port with a MATLAB GUI" <http://cnx.org/content/m12062/latest/>

2.2.2 Audio Effects: Using External Memory[5]

2.2.2.1 Introduction

Many audio effects require storing thousands of samples in memory on the DSP. Because there is not enough memory on the DSP microprocessor itself to store so many samples, external memory must be used.

In this exercise, you will use external memory to implement a long audio delay and an audio echo. Refer to Core File: Accessing External Memory on TI TMS320C54x[6] for a description and examples of accessing external memory.

2.2.2.2 Delay and Echo Implementation

You will implement three audio effects: a long, fixed-length delay, a variable-length delay, and a feedback-echo.

2.2.2.2.1 Fixed-length delay implementation

First, implement the 131,072-sample delay shown in Figure 2.3 using the `READPROG` and `WRITPROG` macros. Use memory locations `010000h-02ffffh` in external Program RAM to do this; you may also want to use the `dld` and `dst` opcodes to store and retrieve the 32-bit addresses for the accumulators. Note that these two operations store the words in memory in big-endian order, with the high-order word first.

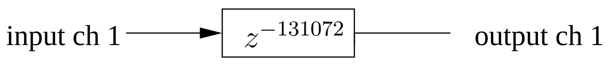


Figure 2.3: Fixed-Length Delay

Remember that arithmetic operations that act on the accumulators, such as the `add` instruction, operate on the complete 32- or 40-bit value. Also keep in mind that since 131,072 is a power of two, you can use masking (via the `and` instruction) to implement the circular buffer easily. This delay will be easy to verify on the oscilloscope. (How long, in seconds, do you expect this delay to be?)

2.2.2.2.2 Variable-delay implementation

Once you have your fixed-length delay working, make a copy and modify it so that the delay can be changed to any length between zero (or one) and 131,072 samples by changing the value stored in one double-word pair in memory. You should keep the buffer length equal to 131,072 and change only your addressing of the sample being read back; it is more difficult to change the buffer size to a length that is not a power of two.

Verify that your code works as expected by timing the delay from input to output and ensuring that it is approximately the correct length.

2.2.2.2.3 Feedback-echo implementation

Last, copy and modify your code so that the value taken from the end of the variable delay from Variable-delay implementation (Section 2.2.2.2.2: Variable-delay implementation) is multiplied by a gain factor and then added back into the input, and the result is both saved into the delay line and sent out to the digital-to-analog converters. Figure 2.4 shows the block diagram. (It may be necessary to multiply the input by

[5]This content is available online at <http://cnx.org/content/m10480/2.17/>.
[6]"Core File: Accessing External Memory on TI TMS320C54x" <http://cnx.org/content/m10823/latest/>

a gain as well to prevent overflow.) This will make a one-tap feedback echo, an simple audio effect that sounds remarkably good. To test the effect, connect the DSP EVM input to a CD player or microphone and connect the output to a loudspeaker. Verify that the echo can be heard multiple times, and that the spacing between echoes matches the delay length you have chosen.

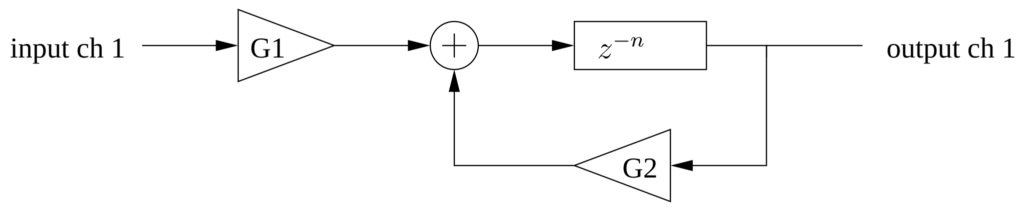


Figure 2.4: Feedback Echo

2.3 Communications

2.3.1 Communications: Using Direct Digital Synthesis[7]

2.3.1.1 Introduction

Direct Digital Synthesis (**DDS**) is a method for generating a desired waveform (such as a sine wave) by using the technique described in Figure 2.5 below.

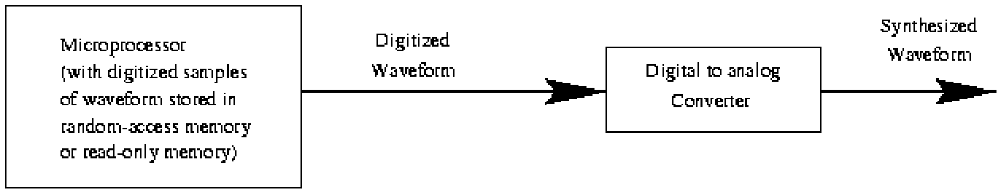


Figure 2.5: Direct digital synthesis (DDS) (*Couch*[7])

Quantized samples of a desired waveform are stored in the memory of the microprocessor system. This desired waveform can then be generated by "playing out" the stored words into the digital-to-analog converter. The frequency of this waveform is determined simply by how fast the stored words are read from memory, and is thus programmable. Likewise, the phase and amplitude of the generated waveform are programmable.

[7]This content is available online at <http://cnx.org/content/m10657/2.5/>.

The DDS technique is replacing analog circuits in many applications. For example, it is used in higher-priced communication receivers to generate local oscillator signals. It can also be used to generate sounds in electronic pipe organs and music synthesizers. Another application is its use by lab instrument manufacturers to generate output waveforms in function generators and arbitrary waveform generators (*Couch*[7]).

In this lab you will familiarize yourself with the capabilities of the Analog Devices AD9854 DDS. The DDS board is installed between the 6-channel card and the DSP card at some (not all) lab stations. You can tell which boxes have them by the way the 6-channel card sits higher inside the metal box.

2.3.1.2 Frequency Modulation (FM) Radio Exercise

To get your feet wet and see a demonstration of the DDS, perform the following exercise. Copy the files `FM.asm` (downloadable here (p. 67)) and `mod.asm` from the `v:\ece320\54x\dds\` directory. Assemble and run the frequency modulation (FM) program `FM.asm`. Next, plug an audio source into one of the two DSP input channels that you've been using all semester. If you have a CD on you, pop it into the computer and use that. If not, use a music web site on the Internet as your audio source. Connect the computer to the DSP by using a male-male audio cable and an audio-to-BNC converter box (little blue box), both of which are in the lab. The computer has three audio outputs on the back; use the middle jack. Ask your TA if you can't find the cable and/or box or don't see how to make the connection. Next, connect a dipole antenna to the output of the DDS (port \#1 on the back of the DDS board). A crude but effective dipole antenna can be formed by connecting together a few BNC and banana cables in the shape of a `T`. There should be one or two of these concoctions in the lab. Once the connections are made, turn on the black receiver in the lab, and tune it to 104.9 MHz (wide band FM). You should be able to hear your audio source!

> NOTE: If your audio sounds distorted, it's most likely due to the volume of your audio source being too loud and getting clipped by the DSP analog-to-digital converter.

2.3.1.2.1 Spectral Copies

Spectral Copies: The digital-to-analog converter on the DDS is unfiltered, which means that there is no anti-imaging filter to remove the spectral replicas. To see this, plug the output of the DDS board directly into the vector signal analyzer (VSA), and observe the spectrum. Use 104.9 MHz as the center frequency, and set the span wide enough so that you can see the spectra of the replicas to the left and right of the 104.9 MHz signal. Use the marker to find the peaks of the other replicas, and record their frequencies. Once you've done that, reattach the antenna to the DDS output, and tune the receiver to the frequencies you just recorded. You should be able to hear your audio on each of the other frequencies.

> NOTE: The clock rate of the DDS is 60MHz, which corresponds to 2π in digital frequency.

Therefore, the 104.9 MHz signal you just listened to is roughly equivalent to $\frac{7\pi}{2}$ in digital frequency. What are the digital frequencies of the other copies you saw on the VSA?

2.3.1.3 How to use the DDS

The DDS has several different modes of operation: single-tone, unramped **Frequency Shift Keying** (**FSK**), ramped FSK, chirp, and **Binary Phase Shift Keying** (**BPSK**). In this lab we will use the DDS in single-tone mode. Single-tone mode is easy to use, and is powerful enough to create many different kinds of waveforms, including FM and FSK.

2.3.1.3.1 FM code

The FM code you just ran (also listed here) is fairly straightforward. The program first calls the `radioinit` subroutine. This routine sets the DDS to single-tone mode and turns off an inverse-sinc filter to conserve power. Following `radioinit`, the `setcarrier` subroutine is called. This routine sets the frequency of the

DDS output by writing to the two most significant 8-bit frequency registers of the 48-bit frequency-tuning word on the DDS[8]. Although the frequency-tuning word on the DDS has 48 bits of resolution, the upper 16 bits provide us with enough resolution for the purposes of this lab, and so we will only be writing to the two most significant registers. See *page 26* in the DDS data sheet for a layout of the frequency-tuning word.

To set the carrier frequency, we first need to determine what frequency word has to be written to the frequency registers on the DDS. This can be done using (2.5):

$$\text{Frequency word} = \frac{\text{baseband frequency}}{60\text{ MHz}} 2^{48} \quad (2.5)$$

where **baseband frequency** corresponds to the desired frequency that lies in the range of 0-30 MHz. For example, to get the DDS to transmit at 104.9 MHz, you would choose the baseband frequency to be 15.1 MHz since 104.9 MHz is one of the unfiltered spectral replicas of 15.1 MHz. Then, using (2.5), the frequency word for 15.1 MHz (and 104.9 MHz) would be equal to `406D 3A06 D3D4h`. But since we only write to the two most significant registers of the frequency-tuning word, we only need the first 4 hexadecimal numbers of this result, i.e. `406Dh`. The first two of those, `40h`, need to get written to the most significant 8-bit frequency register, while the second two hex numbers, `6Dh`, need to get written to the second-most significant 8-bit frequency register. This is where the `40h` and `6Dh` in the `setcarrier` subroutine of the FM code come from.

Writing to the frequency registers is accomplished using the `portw` instruction. To write to the frequency or phase registers on the DDS, the second operand of the `portw` instruction must be `10xxxxxx`, where the lower six bits are the address of the specific register to be written to. The address of the most significant frequency register on the DDS is `04h`, and the address of the second most significant frequency register on the DDS is `05h` (see *page 26* in the data sheet). It is important to note that the way our DDS boards were built, you will not be allowed to make two consecutive writes. To solve this problem, a subroutine called `nullop` is called to waste some CPU time between writes. `nullop` does this by simply repeating the `nop` instruction 128 times.

After the program returns from the `setcarrier` subroutine, it enters an infinite loop in which it waits for a serial interrupt to occur. The serial interrupt occurs every time a new sample is acquired from one of the two input channels and is transmitted to the DSP via the serial port. When the interrupt occurs, an interrupt service routine called `ANALOG_IFC` (see `core_mod.asm` executes and calls the `handle_sample` subroutine. The `handle_sample` subroutine reads in the acquired sample from the serial port and scales that sample so that it can be "mapped" to a frequency in the range of ± 75 kHz[9]. The scaled sample therefore determines the frequency deviation and is added to `6Dh`. The last step is to write the result to the second most significant frequency register so that the frequency of the DDS output can be updated.

2.3.1.3.2 Programming the Phase

The process for changing the phase of the DDS output is the same as it was for changing the frequency of the DDS output. To change the phase, you need to write a phase word to a phase-adjust register on the DDS. The phase-adjust register is 14 bits wide and is split up into two smaller registers that you can write to (see *page 26* in the data sheet). The upper 6 bits have address `00h`, and the lower 8 bits have address `01h`. The phase word can be calculated using (2.6):

$$\text{Phase word} = \frac{\text{Carrier phase}}{2\pi} 2^{14} \quad (2.6)$$

Once you've calculated the phase word, you can write it to the DDS using the `portw` instruction as before. Just make sure you use the correct address for the phase register.

[8]Communication between the DSP and the DDS is done through a parallel bus.

[9]In FM radio, the amplitude of the message signal being transmitted determines the amount of frequency deviation from the carrier frequency of the passband signal. ± 75 kHz is the largest frequency deviation allowed.

2.3.1.3.3 Programming the Amplitude

The DDS also allows you to program the amplitude, but this functionality is not addressed in this lab. You will be able to implement a digital communication system in ECE320 without having to program the amplitude. Interested readers are referred to the data sheet.

2.3.1.4 FSK exercise

Now that you know how to use the DDS in single-tone mode, implement a simple FSK system that uses 2 frequencies: 120.005 MHz, and 120.011 MHz. You don't need to encode any data for this exercise. In other words, your DDS output should just continuously alternate between the two frequency symbols. Also, the DDS automatically ensures continuous phase, so you won't have to keep track of it. Use a symbol length of approximately 725μs (the same length as your lab 5 symbols). Timer interrupts are an elegant way to control the symbol lengths, but in this lab we will keep things simple and control the symbol lengths by creating a second (longer) `nullop` subroutine and calling it between writes to the DDS. The second `nullop` subroutine should waste approximately 725μs worth of time.

> NOTE: Since we're not using input or output from the DSP, you don't need to use the WaitData or WaitAudio macros.

2.3.1.4.1 Testing

Note that the corresponding baseband frequencies for 120.005 MHz and 120.011 MHz are 5 kHz, and 11 kHz, respectively. Since these baseband frequencies lie within the 22.05 kHz bandwidth of the DSP, you will be able to view your FSK signal in real time on the oscilloscope **without** the contribution from the spectral replicas. Just feed the output of the DDS into a second DSP (the anti-aliasing filter on the DSP will get rid of the spectral replicas), and pass it through to the output and the scope. You should be able to verify that there is continuous phase between frequency symbols, and that your symbol length is approximately 725μs. You should also view the spectrum of your DDS output on the VSA to verify that your symbols have the correct frequencies.

2.3.1.5 Appendix

FM.asm[10]

2.3.2 Digital Receiver: Carrier Recovery[11]

2.3.2.1 Introduction

After gaining a theoretical understanding of the carrier recovery sub-system of a digital receiver, you will simulate the sub-system in MATLAB and implement it on the DSP. The sub-system described is specifically tailored to a non-modulated carrier. A complete implementation will require modifications to the design presented.

The **phase-locked loop** (**PLL**) is a critical component in coherent communications receivers that is responsible for locking on to the carrier of a received modulated signal. Ideally, the transmitted carrier frequency is known exactly and we need only to know its phase to demodulate correctly. However, due to imperfections at the transmitter, the actual carrier frequency may be slightly different from the expected frequency. For example, in the QPSK transmitter of Digital Transmitter: Introduction to Quadrature Phase-Shift Keying[12], if the digital carrier frequency is $\frac{\pi}{2}$ and the D/A is operating at 44.1 kHz, then the expected analog carrier frequency is $f_c = \frac{\frac{\pi}{2}}{2\pi}44.1 = 11.25\text{kHz}$. If there is a slight change to the D/A sample

[10]http://cnx.rice.edu/modules/m10657/latest/FM.asm
[11]This content is available online at <http://cnx.org/content/m10478/2.16/>.
[12]"Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" <http://cnx.org/content/m10042/latest/>

rate (say $f_c = 44.05\text{kHz}$), then there will be a corresponding change in the actual analog carrier frequency ($f_c = 11.0125\text{kHz}$).

This difference between the expected and actual carrier frequencies can be modeled as a time-varying phase. Provided that the frequency mismatch is small relative to the carrier frequency, the feedback control of an appropriately calibrated PLL can track this time-varying phase, thereby locking on to both the correct frequency and the correct phase.

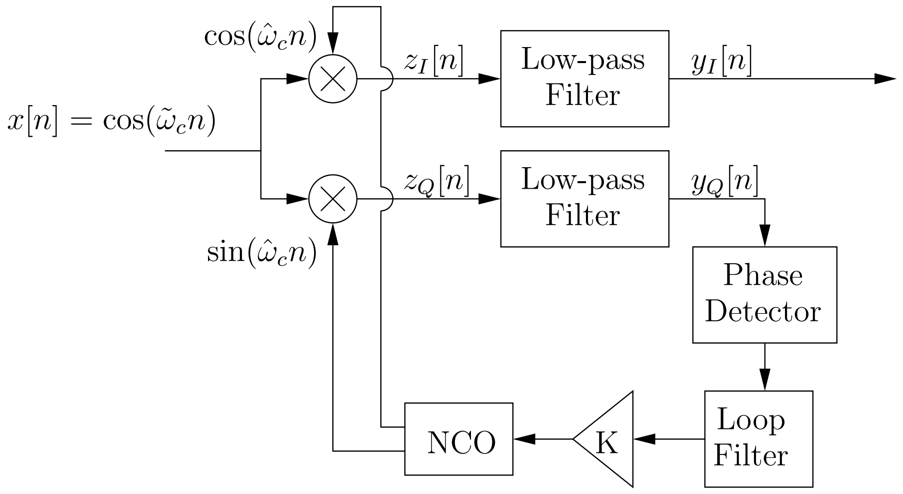


Figure 2.6: PLL Block Diagram

2.3.2.1.1 Numerically controlled oscillator

In a complete coherent receiver implementation, carrier recovery is required since the receiver typically does not know the exact phase and frequency of the transmitted carrier. In an analog system this recovery is often implemented with a **voltage-controlled oscillator** (**VCO**) that allows for precise adjustment of the carrier frequency based on the output of a phase-detecting circuit.

In our digital application, this adjustment is performed with a **numerically-controlled oscillator** (**NCO**) (see Figure 2.6). A simple scheme for implementing an NCO is based on the following re-expression of the carrier sinusoid:

$$\sin(\omega_c n + \theta_c) = \sin(\theta[n]) \tag{2.7}$$

where $\theta[n] = \omega_c n + \theta_c$ (ω_c and θ_c represent the carrier frequency and phase, respectively). Convince yourself that this time-varying phase term can be expressed as $\theta[n] = \sum_{m=0}^{n} \omega_c + \theta_c$ and then recursively as

$$\theta[n] = \theta[n-1] + \omega_c \tag{2.8}$$

The NCO can keep track of the phase, $\theta[n]$, and force a phase offset in the demodulating carrier by incorporating an extra term in this recursive update:

$$\theta[n] = \theta[n-1] + \omega_c + d_{\text{pd}}[n] \tag{2.9}$$

where $d_{\mathrm{pd}}[n]$ is the amount of desired phase offset at time n. (What would $d_{\mathrm{pd}}[n]$ look like to generate a frequency offset?)

2.3.2.1.2 Phase detector

The goal of the PLL is to maintain a demodulating sine and cosine that match the incoming carrier. Suppose ω_c is the believed digital carrier frequency. We can then represent the actual received carrier frequency as the expected carrier frequency with some offset, $\tilde{\omega_c} = \omega_c + \tilde{\theta}[n]$. The NCO generates the demodulating sine and cosine with the expected digital frequency ω_c and offsets this frequency with the output of the loop filter. The NCO frequency can then be modeled as $\widehat{\omega_c} = \omega_c + \widehat{\theta}[n]$. Using the appropriate trigonometric identities [13], the in-phase and quadrature signals can be expressed as

$$z_0[n] = 1/2\left(\cos\left(\tilde{\theta}[n] - \widehat{\theta}[n]\right) + \cos\left(2\omega_c + \tilde{\theta}[n] + \widehat{\theta}[n]\right)\right) \tag{2.10}$$

$$z_Q[n] = 1/2\left(\sin\left(\tilde{\theta}[n] - \widehat{\theta}[n]\right) + \sin\left(2\omega_c + \tilde{\theta}[n] + \widehat{\theta}[n]\right)\right) \tag{2.11}$$

After applying a low-pass filter to remove the double frequency terms, we have

$$y_1[n] = 1/2\cos\left(\tilde{\theta}[n] - \widehat{\theta}[n]\right) \tag{2.12}$$

$$y_Q[n] = 1/2\sin\left(\tilde{\theta}[n] - \widehat{\theta}[n]\right) \tag{2.13}$$

Note that the quadrature signal, $z_Q[n]$, is zero when the received carrier and internally generated waves are exactly matched in frequency and phase. When the phases are only slightly mismatched we can use the relation

$$\forall\theta, \text{small} : (\sin(\theta) \simeq \theta) \tag{2.14}$$

and let the current value of the quadrature channel approximate the phase difference: $z_Q[n] \simeq \tilde{\theta}[n] - \widehat{\theta}[n]$. With the exception of the sign error, this difference is essentially how much we need to offset our NCO frequency[14]. To make sure that the sign of the phase estimate is right, in this example the phase detector is simply negative one times the value of the quadrature signal. In a more advanced receiver, information from both the in-phase and quadrature branches is used to generate an estimate of the phase error.[15]

2.3.2.1.3 Loop filter

The estimated phase mismatch estimate is fed to the NCO via a loop filter, often a simple low-pass filter. For this exercise you can use a one-tap IIR filter,

$$y[n] = \beta x[n] + \alpha y[n-1] \tag{2.15}$$

To ensure unity gain at DC, we select $\beta = 1 - \alpha$

It is suggested that you start by choosing $\alpha = 0.6$ and $K = 0.15$ for the loop gain. Once you have a working system, investigate the effects of modifying these values.

[13] $\cos(A)\cos(B) = 1/2(\cos(A-B) + \cos(A+B))$ and $\cos(A)\sin(B) = 1/2(\sin(B-A) + \sin(A+B))$.

[14] If $\tilde{\theta}[n] - \widehat{\theta}[n] > 0$ then $\widehat{\theta}[n]$ is too large and we want to decrease our NCO phase.

[15] What should the relationship between the I and Q branches be for a digital QPSK signal?

2.3.2.2 MATLAB Simulation

Simulate the PLL system shown in Figure 2.6 using MATLAB. As with the DLL simulation, you will have to simulate the PLL on a sample-by-sample basis.

Use (2.9) to implement your NCO in MATLAB. However, to ensure that the phase term does not grow to infinity, you should use addition modulo 2π in the phase update relation. This can be done by setting $\theta[n] = \theta[n] - 2\pi$ whenever $\theta[n] > 2\pi$.

Figure 2.7 illustrates how the proposed PLL will behave when given a modulated BPSK waveform. In this case the transmitted carrier frequency was set to $\tilde{\omega_c} = \frac{\pi}{2} + \frac{\pi}{1024}$ to simulate a frequency offset.

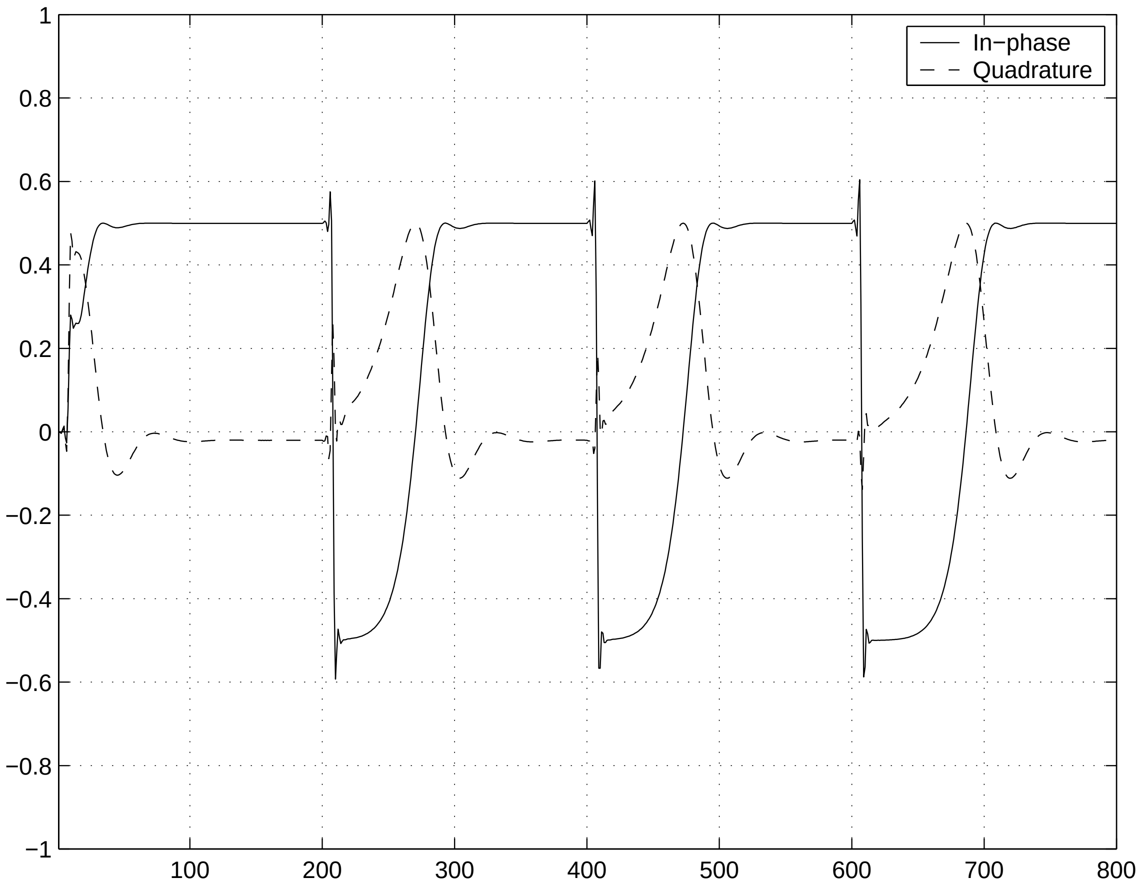


Figure 2.7: Output of PLL sub-system for BPSK modulated carrier.

Note that an amplitude transition in the BPSK waveform is equivalent to a phase shift of the carrier by $\frac{\pi}{2}$. Immediately after this phase change occurs, the PLL begins to adjust the phase to force the quadrature component to zero (and the in-phase component to 1/2). Why would this phase detector not work in a real BPSK environment? How could it be changed to work?

2.3.2.3 DSP Implementation

As you begin to implement your PLL on the DSP, it is highly recommended that you implement and test your NCO block first before completing the rest of your phase-locked loop.

2.3.2.3.1 Sine-table interpolation

Your NCO must be able to produce a sinusoid with continuously variable frequency. Computing values of $\sin(\theta[n])$ on the fly would require a prohibitive amount of computation and program complexity; a look-up table is a better alternative.

Suppose a sine table stores N samples from one cycle of the waveform: $\forall k, k = \{0, \dots, N-1\} : \left(\sin\left(\frac{2\pi k}{N}\right)\right)$. Sine waves with discrete frequencies $\omega = \frac{2\pi}{N}p$ are easily obtained by outputting every p^{th} value in the table (and using circular addressing). The continuously variable frequency of your NCO will require **non-integer** increments, however. This raises two issues: First, what sort of interpolation should be used to get the in-between sine samples, and second, how to maintain a non-integer pointer into the sine table.

You may simplify the interpolation problem by using "lower-neighbor" interpolation, i.e., by using the integer part of your pointer. Note that the full-precision, non-integer pointer must be maintained in memory so that the fractional part is allowed to accumulate and carry over into the integer part; otherwise, your phase will not be accurate over long periods. For a long enough sine table, this approximation will adjust the NCO frequency with sufficient precision.[16]

Maintaining a non-integer pointer is more difficult. In earlier exercises, you have used the auxiliary registers (`ARx`) to manage pointers with integer increments. The auxiliary registers are not suited for the non-integer pointers needed in this exercise, however, so another method is required. One possibility is to perform addition in the accumulator with a modified decimal point. For example, with $N = 256$, you need eight bits to represent the integer portion of your pointer. Interpret the low 16 bits of the accumulator to have a decimal point seven bits up from the bottom; this leaves nine bits to store the integer part above the decimal point. To increment the pointer by one step, add a 15-bit value to the low part of the accumulator, then zero the top bit to ensure that the value in the accumulator is greater than or equal to zero and less than 256.[17] To use the integer part of this pointer, shift the accumulator contents seven bits to the right, add the starting address of the sine table, and store the low part into an `ARx` register. The auxiliary register now points to the correct sample in the sine table.

As an example, for a nominal carrier frequency $\omega = \frac{\pi}{8}$ and sine table length $N = 256$, the nominal step size is an integer $p = \frac{\pi}{8}N\frac{1}{2\pi} = 16$. Interpret the 16-bit pointer as having nine bits for the integer part, followed by a decimal point and seven bits for the fractional part. The corresponding literal (integer) value added to the accumulator would be $16 \times 2^7 = 2048$.[18]

2.3.2.3.2 Extensions

You may want to refer to *Proakis* [10] and *Blahut* [1]. These references may help you think about the following questions:

- How does the noise affect the described carrier recovery method?
- What should the phase-detector look like for a BPSK modulated carrier? (Hint: You would need to consider both the in-phase and quadrature channels.)
- How does α affect the bandwidth of the loop filter?
- How do the loop gain and the bandwidth of the loop filter affect the PLL's ability to lock on to a carrier frequency mismatch?

[16]Of course, nearest-neighbor interpolation could be implemented with a small amount of extra code.
[17]How is this similar to the addition modulo 2π discussed in the MATLAB Simulation (Section 2.3.2.2: MATLAB Simulation)?
[18]If this value were 2049, what would be the output frequency of the NCO?

2.3.3 Digital Receivers: Symbol-Timing Recovery for QPSK[19]

2.3.3.1 Introduction

This receiver exercise introduces the primary components of a QPSK receiver with specific focus on symbol-timing recovery. In a receiver, the received signal is first coherently demodulated and low-pass filtered (see Digital Receivers: Carrier Recovery for QPSK (Section 2.3.2)) to recover the message signals (in-phase and quadrature channels). The next step for the receiver is to sample the message signals at the symbol rate and decide which symbols were sent. Although the symbol rate is typically known to the receiver, the receiver does not know when to sample the signal for the best noise performance. The objective of the symbol-timing recovery loop is to find the best time to sample the received signal.

Figure 2.8 illustrates the digital receiver system. The transmitted signal coherently demodulated with both a sine and cosine, then low-pass filtered to remove the double-frequency terms, yielding the recovered in-phase and quadrature signals, $\hat{s_I}\ [n]$ and $\hat{s_Q}\ [n]$. These operations are explained in Digital Receivers: Carrier Recovery for QPSK (Section 2.3.2). The remaining operations are explained in this module. Both branches are fed through a **matched filter** and re-sampled at the symbol rate. The matched filter is simply an FIR filter with an impulse response matched to the transmitted pulse. It aids in timing recovery and helps suppress the effects of noise.

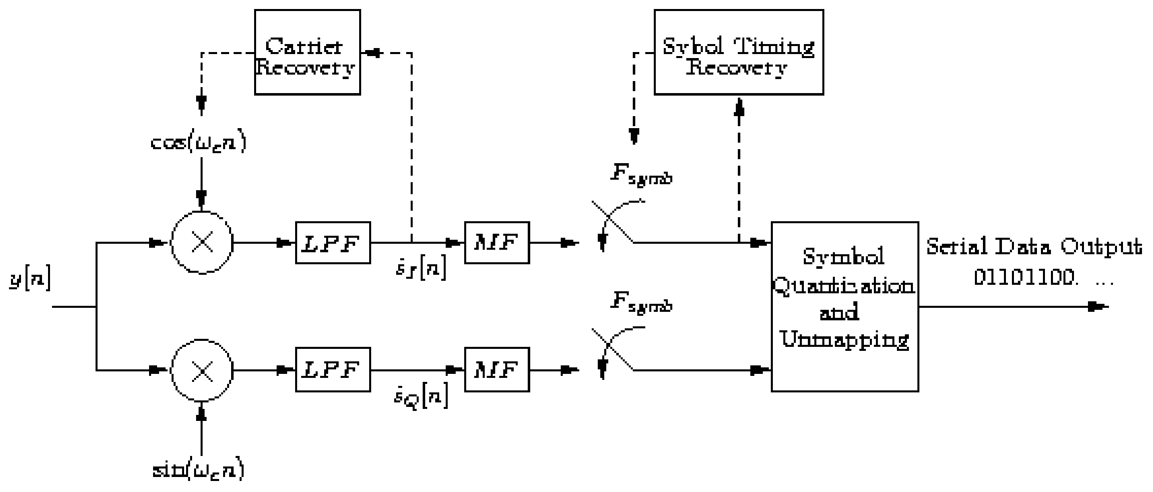


Figure 2.8: Digital receiver system

If we consider the square wave shown in Figure 2.9 as a potential recovered in-phase (or quadrature) signal (*i.e.*, we sent the data $[+1, -1, +1, -1, \ldots]$) then sampling at any point other than the symbol transitions will result in the correct data.

[19]This content is available online at <http://cnx.org/content/m10485/2.14/>.

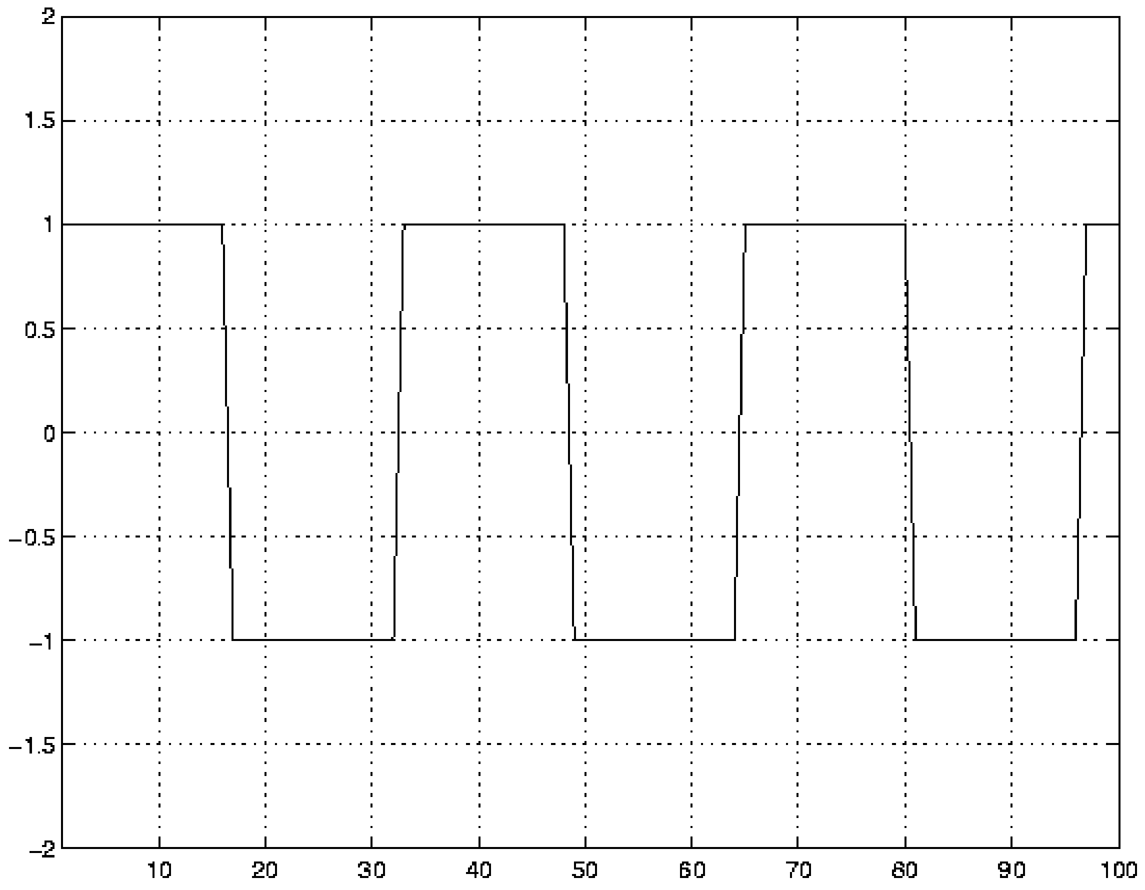


Figure 2.9: Clean BPSK waveform.

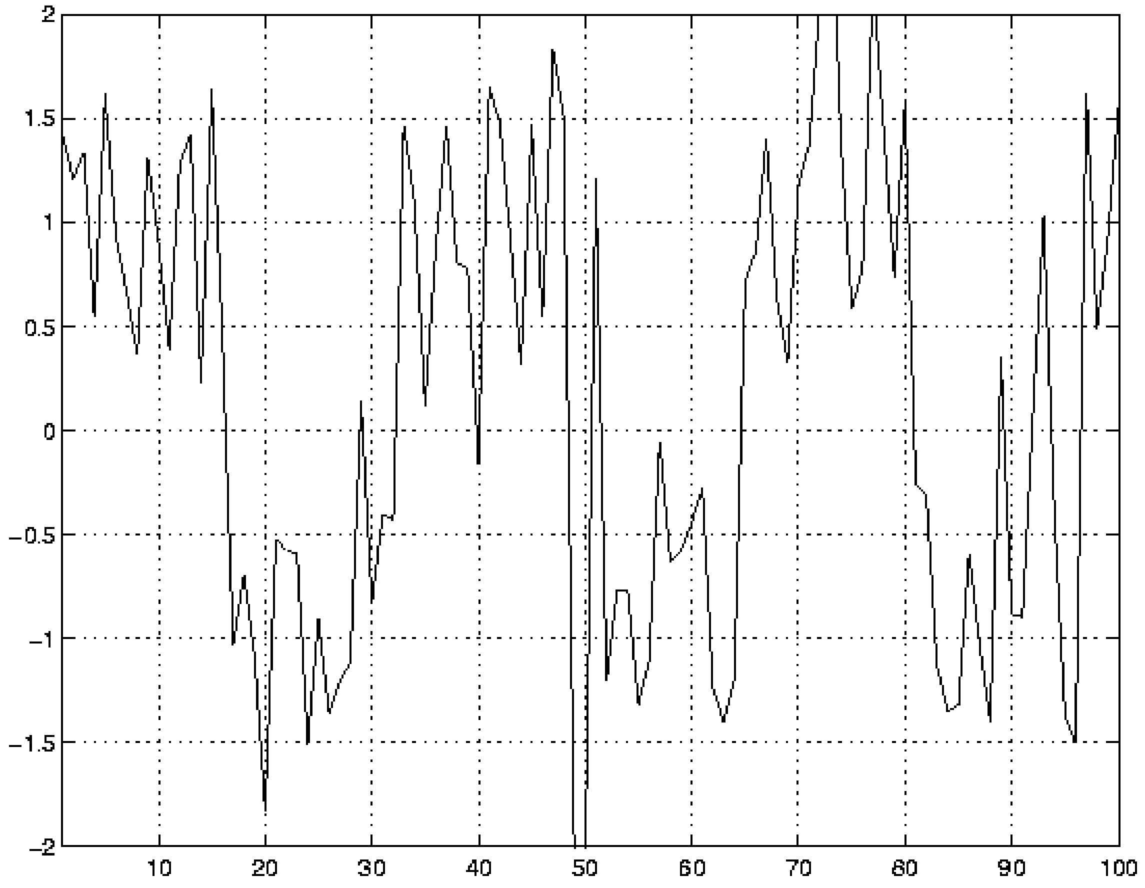


Figure 2.10: Noisy BPSK waveform.

However, in the presence of noise, the received waveform may look like that shown in Figure 2.10. In this case, sampling at any point other than the symbol transitions does not guarantee a correct data decision. By averaging over the symbol duration we can obtain a better estimate of the true data bit being sent (+1 or −1). The best averaging filter is the matched filter, which has the impulse response $u[n] - u[n - T_{\text{symb}}]$, where $u[n]$ is the unit step function, for the simple rectangular pulse shape used in Digital Transmitter: Introduction to Quadrature Phase-Shift Keying[20]. [21]Figure 2.11 and Figure 2.12 show the result of passing both the clean and noisy signal through the matched filter.

[20]"Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" <http://cnx.org/content/m10042/latest/>

[21]For digital communications schemes involving different pulse shapes, the form of the matched filter will be different. Refer to the listed references for more information on symbol timing and matched filters for different symbol waveforms.

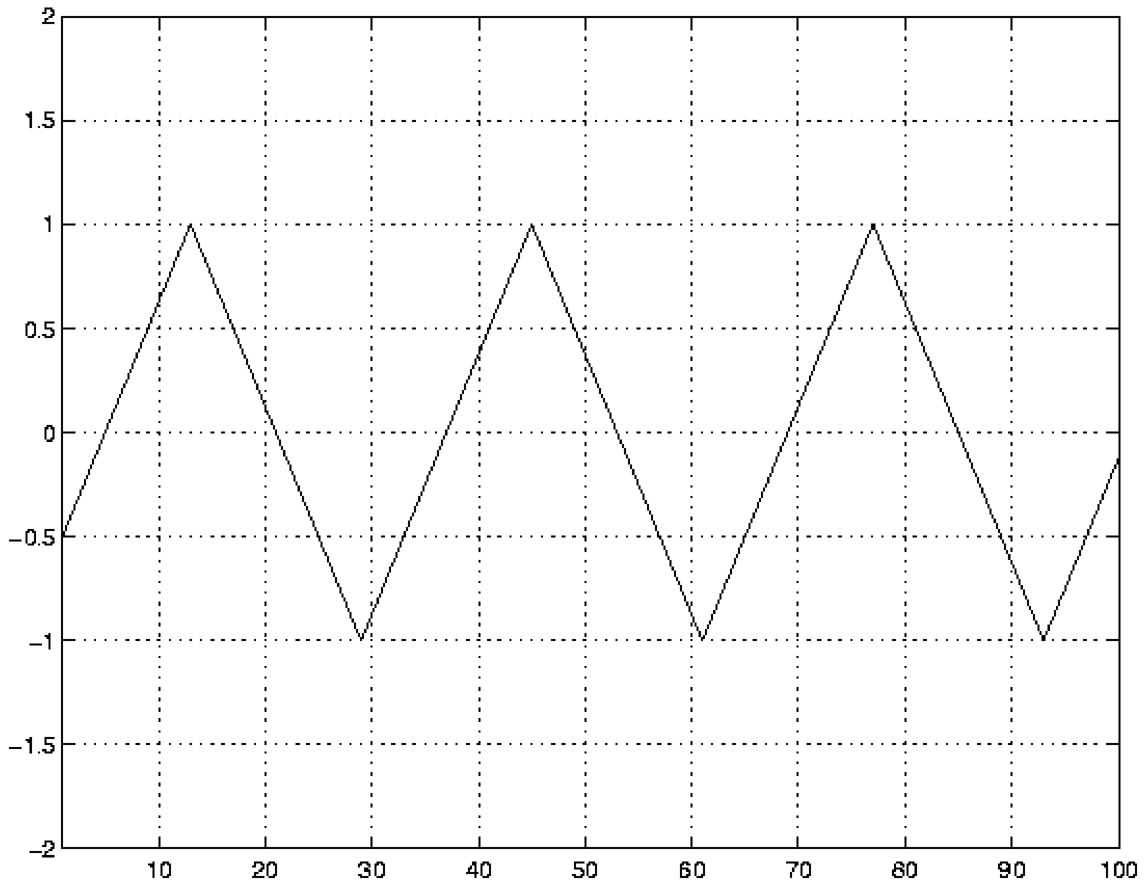


Figure 2.11: Averaging filter output for clean input.

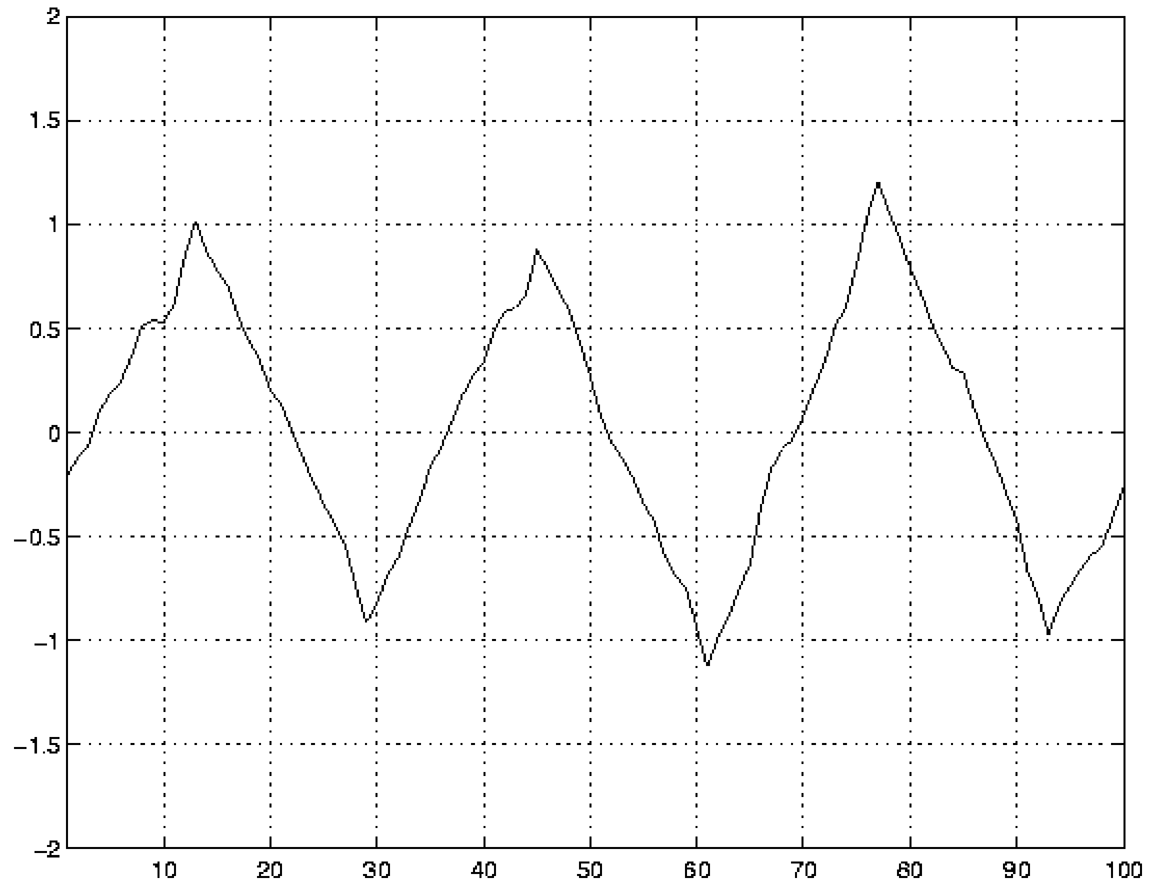


Figure 2.12: Averaging filter output for noisy input.

Note that in both cases the output of the matched filter has peaks where the matched filter exactly lines up with the symbol, and a positive peak indicates a +1 was sent; likewise, a negative peak indicates a −1 was sent. Although there is still some noise in second figure, the peaks are relatively easy to distinguish and yield considerably more accurate estimation of the data (+1 or −1) than we could get by sampling the original noisy signal in Figure 2.10.

The remainder of this handout describes a symbol-timing recovery loop for a BPSK signal (equivalent to a QPSK signal where only the in-phase signal is used). As with the above examples, a symbol period of $T_s = 16$ samples is assumed.

2.3.3.1.1 Early/late sampling

One simple method for recovering symbol timing is performed using a **delay-locked loop** (**DLL**). Figure 2.13 is a block diagram of the necessary components.

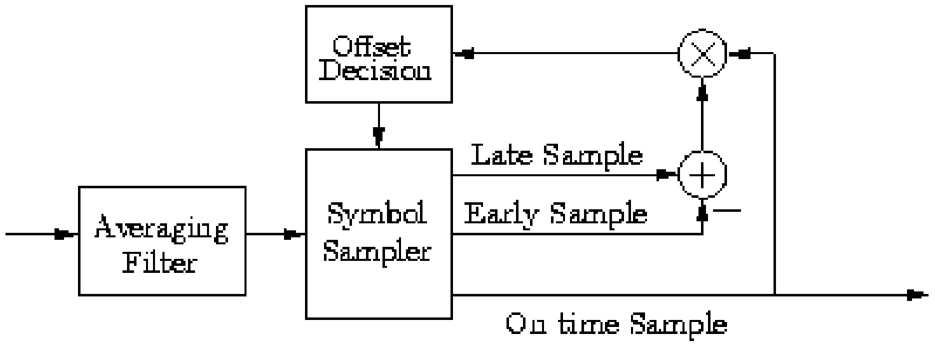


Figure 2.13: DLL block diagram.

Consider the sawtooth waveform shown in Figure 2.11, the output of the matched filter with a square wave as input. The goal of the DLL is to sample this waveform at the peaks in order to obtain the best performance in the presence of noise. If it is not sampling at the peaks, we say it is sampling too early or too late.

The DLL will find peaks without assistance from the user. When it begins running, it arbitrarily selects a sample, called the **on-time sample**, from the matched filter output. The sample from the time-index one greater than that of the on-time sample is the **late sample**, and the sample from the time-index one less than that of the on-time sample is the **early sample**. Figure 2.14 shows an example of the on-time, late, and early samples. Note in this case that the on-time sample happens to be at a peak in the waveform. Figure 2.15 and Figure 2.16 show examples in which the on-time sample comes before a peak and after the peak.

The on-time sample is the output of the DLL and will be used to decide the data bit sent. To achieve the best performance in the presence of noise, the DLL must adjust the timing of on-time samples to coincide with peaks in the waveform. It does this by changing the number of time-indices between on-time samples. There are three cases:

1. In Figure 2.14, the on-time sample is already at the peak, and the receiver knows that peaks are spaced by T_{symb} samples. If it then takes the next on-time sample T_{symb} samples after this on-time sample, it will be at another peak.
2. In Figure 2.15, the on-time sample is too early. Taking an on-time sample T_{symb} samples later will be too early for the next peak. To move closer to the next peak, the next on-time sample is taken $T_{\text{symb}}+1$ samples after the current on-time sample.
3. In Figure 2.16, the on-time sample is too late. Taking an on-time sample T_{symb} samples later will be too late for the next peak. To move closer to the next peak, the next on-time sample is taken $T_{\text{symb}}-1$ samples after the current on-time sample.

The offset decision block uses the on-time, early, and late samples to determine whether sampling is at a peak, too early, or too late. It then sets the time at which the next on-time sample is taken.

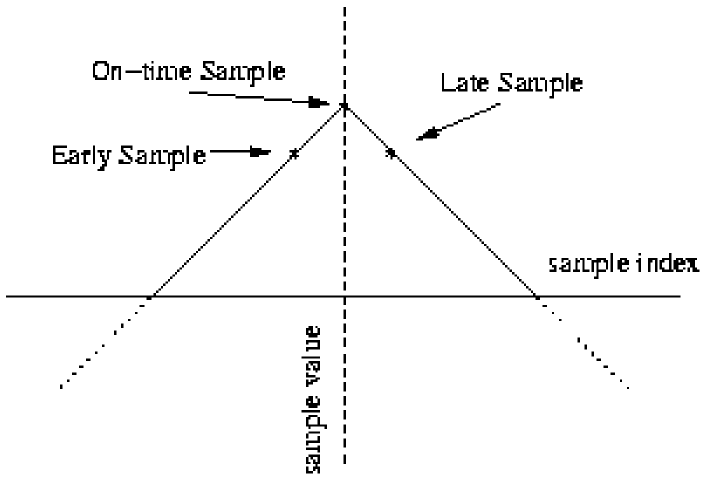


Figure 2.14: Sampling at a peak.

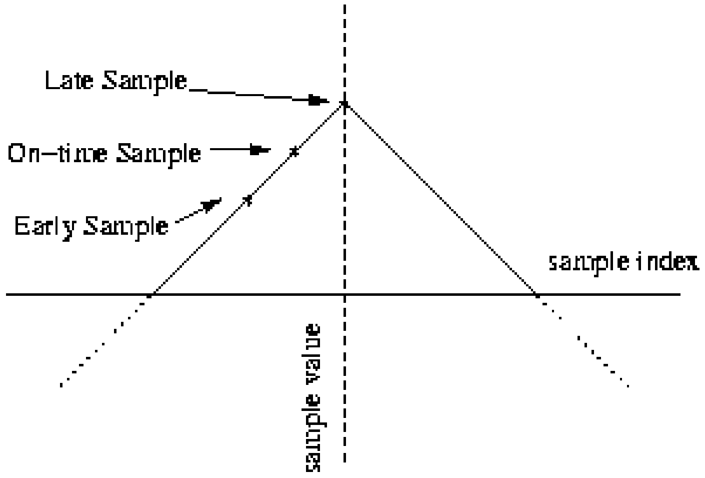


Figure 2.15: Sampling too early.

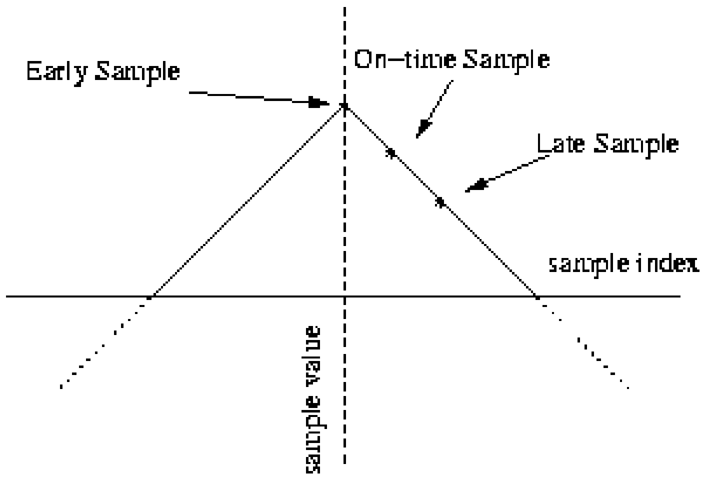


Figure 2.16: Sampling too late.

The input to the offset decision block is $\text{on} - \text{time}\,(\text{late} - \text{early})$, called the **decision statistic**. Convince yourself that when the decision statistic is positive, the on-time sample is too early, when it is zero, the on-time sample is at a peak, and when it is negative, the on-time sample is too late. It may help to refer to Figure 2.14, Figure 2.15, and Figure 2.16. Can you see why it is necessary to multiply by the on-time sample?

The offset decision block could adjust the time at which the next on-time sample is taken based only on the decision statistic. However, in the presence of noise, the decision statistic becomes a less reliable indicator. For that reason, the DLL adds many successive decision statistics and corrects timing only if the sum exceeds a threshold; otherwise, the next on-time sample is taken T_{symb} samples after the current on-time sample. The assumption is that errors in the decision statistic caused by noise, some positive and some negative, will tend to cancel each other out in the sum, and the sum will not exceed the threshold because of noise alone. On the other hand, if the on-time sample is consistently too early or too late, the magnitude of the added decision statistics will continue to grow and exceed the threshold. When that happens, the offset decision block will correct the timing and reset the sum to zero.

2.3.3.1.2 Sampling counter

The symbol sampler maintains a counter that decrements every time a new sample arrives at the output of the matched filter. When the counter reaches three, the matched-filter output is saved as the late sample, when the counter reaches two, the matched-filter output is saved as the on-time sample, and when the counter reaches one, the matched-filter output is saved as the early sample. After saving the early sample, the counter is reset to either $T_{\text{symb}} - 1$, T_{symb}, or $T_{\text{symb}} + 1$, according to the offset decision block.

2.3.3.2 MATLAB Simulation

Because the DLL requires a feedback loop, you will have to simulate it on a sample-by-sample basis in MATLAB.

Using a square wave of period 32 samples as input, simulate the DLL system shown in Figure 2.13. Your input should be several hundred periods long. What does it model? Set the decision-statistic sum-threshold to 1.0; later, you can experiment with different values. How do you expect different thresholds to affect the DLL?

Figure 2.17 and Figure 2.18 show the matched filter output and the on-time sampling times (indicated by the impulses) for the beginning of the input, before the DLL has locked on, as well as after 1000 samples (about 63 symbols' worth), when symbol-timing lock has been achieved. For each case, note the distance between the on-time sampling times and the peaks of the matched filter output.

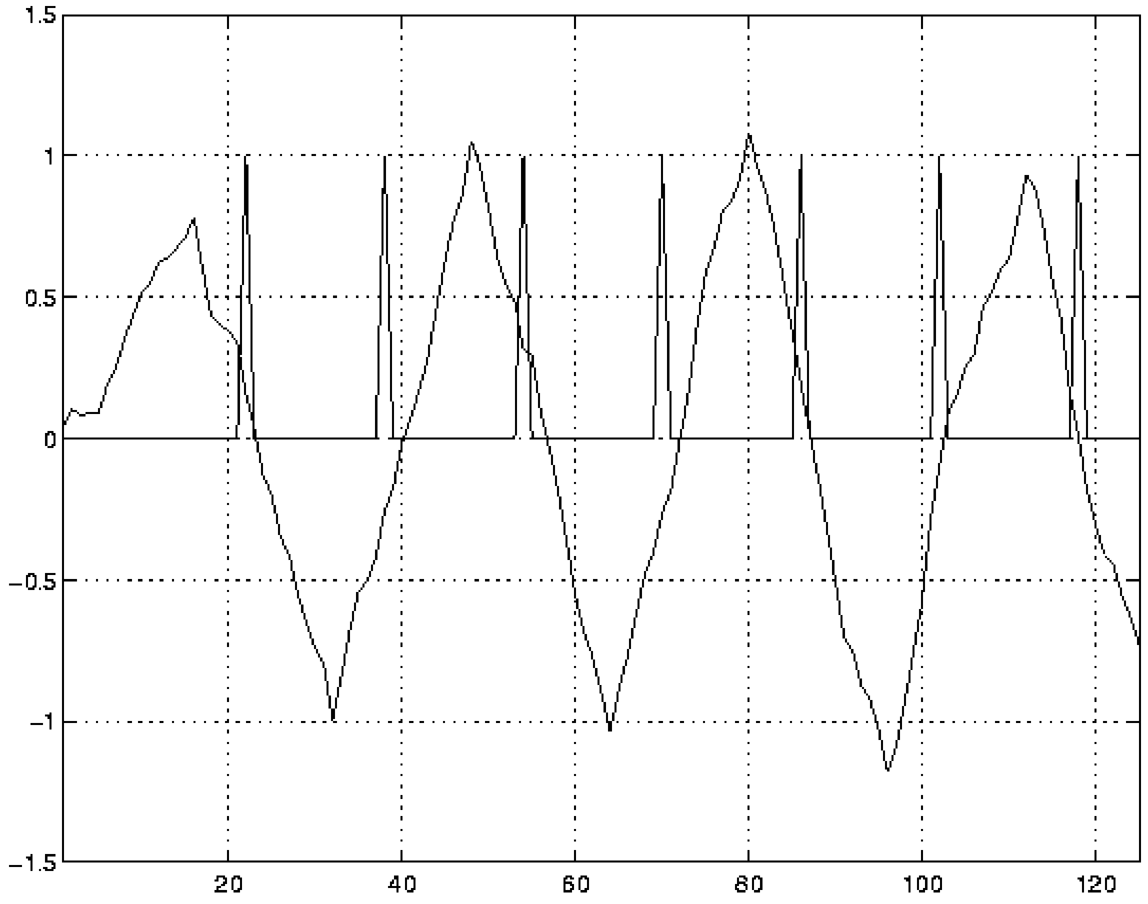


Figure 2.17: Symbol sampling before DLL lock.

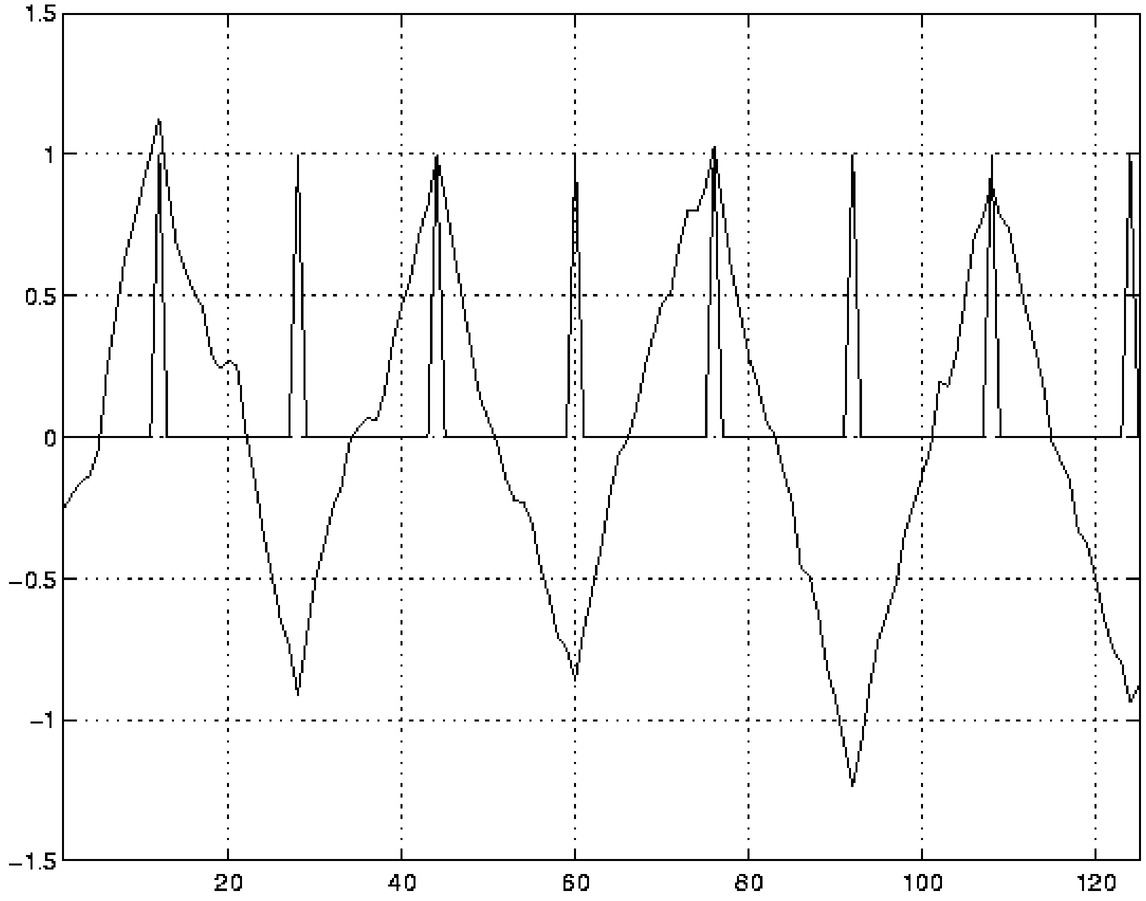


Figure 2.18: Symbol sampling after DLL lock.

2.3.3.3 DSP Implementation

Once your MATLAB simulation works, DSP implementation is relatively straightforward. To test your implementation, you can use the function generator to simulate a BPSK waveform by setting it to a square wave of the correct frequency for your symbol period. You should send the on-time sample and the matched-filter output to the D/A to verify that your system is working.

2.3.3.4 Extensions

As your final project will require some modification to the discussed BPSK signaling, you will want to refer to the listed references, (see *Proakis*[11] and *Blahut*[2], and consider some of the following questions regarding such modifications:

- How much noise is necessary to disrupt the DLL?
- What happens when the symbol sequence is random (not simply $[+1, -1, +1, -1, \dots]$)?
- What would the matched filter look like for different symbol shapes?

- What other methods of symbol-timing recovery are available for your application?
- How does adding decision statistics help suppress the effects of noise?

2.4 Video Processing

2.4.1 Video Processing Manuals[22]

2.4.1.1 Essential documentation for the 6000 series TI DSP

The following documentation will certainly prove useful:

- The IDK Programmer's Guide[23]
- The IDK User's Guide[24]
- The IDK Video Device Drivers User's Guide[25]

NOTE: Other manuals may be found on TI's website[26] by searching for TMS320C6000 IDK

2.4.2 Video Processing Part 1: Introductory Exercise[27]

2.4.2.1 Introduction

The purpose of this lab is to acquaint you with the TI Image Developers Kit (IDK). The IDK contains a floating point C6711 DSP, and other hardware that enables real time video/image processing. In addition to the IDK, the video processing lab bench is equipped with an NTSC camera and a standard color computer monitor.

You will complete an introductory exercise to gain familiarity with the IDK programming environment. In the exercise, you will modify a C skeleton to horizontally flip and invert video input (black and white) from the camera. The output of your video processing algorithm will appear in the top right quadrant of the monitor.

In addition, you will analyze existing C code that implements filtering and edge detection algorithms to gain insight into IDK programming methods. The output of these "canned" algorithms, along with the unprocessed input, appears in the other quadrants of the monitor.

Finally, you will create an auto contrast function. And will also work with a color video feed and create a basic user interface, which uses the input to control some aspect of the display.

An additional goal of this lab is to give you the opportunity to discover tools for developing an original project using the IDK.

2.4.2.1.1 Important Documentation

The following documentation will certainly prove useful:

- The IDK User's Guide[28] . Section 2 is the most important.
- The IDK Video Device Drivers User's Guide[29] . The sections on timing are not too important, but pay attention to the Display and Capture systems and have a good idea of how they work.

[22]This content is available online at <http://cnx.org/content/m10889/2.5/>.
[23]http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf
[24]http://www-s.ti.com/sc/psheets/spru494a/spru494a.pdf
[25]http://www-s.ti.com/sc/psheets/spru499/spru499.pdf
[26]http://www.ti.com
[27]This content is available online at <http://cnx.org/content/m11987/1.2/>.
[28]http://www-s.ti.com/sc/psheets/spru494a/spru494a.pdf
[29]http://www-s.ti.com/sc/psheets/spru499/spru499.pdf

- The IDK Programmer's Guide[30] . Sections 2 and 5 are the ones needed. Section 2 is very, very important in Project Lab 2. It is also useful in understanding "streams" in project lab 1.

NOTE: Other manuals may be found on TI's website[31] by searching for TMS320C6000 IDK

2.4.2.2 Video Processing - The Basics

The camera on the video processing lab bench generates a video signal in NTSC format. NTSC is a standard for transmitting and displaying video that is used in television. The signal from the camera is connected to the "composite input" on the IDK board (the yellow plug). This is illustrated in Figure 2-1 on page 2-3 of the IDK User's Guide. Notice that the IDK board is actually two boards stacked on top of each other. The bottom board contains the C6711 DSP, where your image processing algorithms will run. The daughterboard is on top, it contains the hardware for interfacing with the camera input and monitor output. For future video processing projects, you may connect a video input other than the camera, such as the output from a DVD player. The output signal from the IDK is in RGB format, so that it may be displayed on a computer monitor.

At this point, a description of the essential terminology of the IDK environment is in order. The video input is first decoded and then sent to the FPGA, which resides on the daughterboard. The FPGA is responsible for video capture and for the filling of the input frame buffer (whose contents we will read). For a detailed description of the FPGA and its functionality, we advise you to read Chapter 2 of the IDK User's Guide.

The Chip Support Library (CSL) is an abstraction layer that allows the IDK daughterboard to be used with the entire family of TI C6000 DSPs (not just the C6711 that we're using); it takes care of what is different from chip to chip.

The Image Data Manager (IDM) is a set of routines responsible for moving data between on-chip internal memory, and external memory on the board, during processing. The IDM helps the programmer by taking care of the pointer updates and buffer management involved in transferring data. Your DSP algorithms will read and write to internal memory, and the IDM will transfer this data to and from external memory. Examples of external memory include temporary "scratch pad" buffers, the input buffer containing data from the camera, and the output buffer with data destined for the RGB output.

The two different memory units exist to provide rapid access to a larger memory capacity. The external memory is very large in size – around 16 MB, but is slow to access. But the internal is only about 25 KB or so and offers very fast access times. Thus we often store large pieces of data, such as the entire input frame, in the external memory. We then bring it in to internal memory, one small portion at a time, as needed. A portion could be a line or part of a line of the frame. We then process the data in internal memory and then repeat in reverse, by outputting the results line by line (or part of) to external memory. This is full explained in Project Lab 2, and this manipulation of memory is important in designing efficient systems.

The TI C6711 DSP uses a different instruction set than the 5400 DSP's you are familiar with in lab. The IDK environment was designed with high level programming in mind, so that programmers would be isolated from the intricacies of assembly programming. Therefore, we strongly suggest that you do all your programming in C. Programs on the IDK typically consist of a main program that calls an image processing routine.

The main program serves to setup the memory spaces needed and store the pointers to these in objects for easy access. It also sets up the input and output channels and the hardware modes (color/grayscale ...). In short it prepares the system for our image processing algorithm.

The image processing routine may make several calls to specialized functions. These specialized functions consist of an outer wrapper and an inner component. The wrapper oversees the processing of the entire image, while the component function works on parts of an image at a time. And the IDM moves data back and forth between internal and external memory.

[30]http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf
[31]http://www.ti.com

As it brings in one line in from external memory, the component function performs the processing on this one line. Results are sent back to the wrapper. And finally the wrapper contains the IDM instructions to pass the output to external memory or wherever else it may be needed.

Please note that this is a good methodology used in programming for the IDK. However it is very flexible too, the "wrapper" and "component functions" are C functions and return values, take in parameters and so on too. And it is possible to extract/output multiple lines or block etc. as later shown.

In this lab, you will modify a component to implement the flipping and inverting algorithm. And you will perform some simple auto-contrasting as well as work with color.

In addition, the version of Code Composer that the IDK uses is different from the one you have used previously. The IDK uses Code Composer Studio v2.1. It is similar to the other version, but the process of loading code is slightly different.

2.4.2.3 Code Description

2.4.2.3.1 Overview and I/O

The next few sections describe the code used. First please copy the files needed by following the instructions in the "Part 1" section of this document. This will help you easily follow the next few parts.

The program flow for image processing applications may be a bit different from your previous experiences in C programming. In most C programs, the main function is where program execution starts and ends. In this real-time application, the main function serves only to setup initializations for the cache, the CSL, and the DMA (memory access) channel. When it exits, the main task, tskMainFunc(), will execute automatically, starting the DSP/BIOS. It will loop continuously calling functions to operate on new frames and this is where our image processing application begins.

The tskMainFunc(), in main.c, opens the handles to the board for image capture (VCAP_open()) and to the display (VCAP_open()) and calls the grayscale function. Here, several data structures are instantiated that are defined in the file img_proc.h. The IMAGE structures will point to the data that is captured by the FPGA and the data that will be output to the display. The SCRATCH_PAD structure points to our internal and external memory buffers used for temporary storage during processing. LPF_PARAMS is used to store filter coefficients for the low pass filter.

The call to img_proc() takes us to the file img_proc.c. First, several variables are declared and defined. The variable quadrant will denote on which quadrant of the screen we currently want output; out_ptr will point to the current output spot in the output image; and pitch refers to the byte offset (distance) between two lines. This function is the high level control for our image-processing algorithm. See algorithm flow.

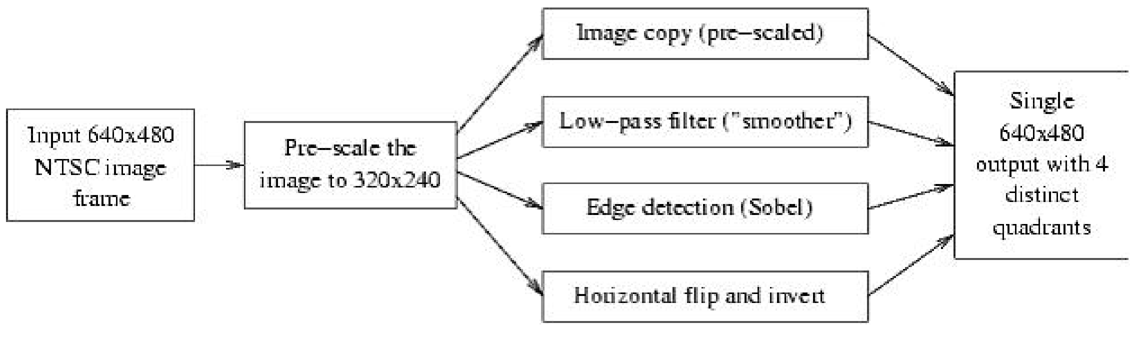

Input 640x480 NTSC image frame
Pre-scale the image to 320x240
Image copy (pre-scaled)
Low-pass filter ("smoother")
Edge detection (Sobel)
Horizontal flip and invert
Single 640x480 output with 4 distinct quadrants

Figure 2.19: Algorithm Flow

The first function called is the pre_scale_image function in the file pre_scale_image.c. The purpose of this function is to take the 640x480 image and scale it down to a quarter of its size by first downsampling the input rows by two and then averaging every two pixels horizontally. The internal and external memory spaces, pointers to which are in the scratch pad, are used for this task. The vertical downsampling occurs when every other line is read into the internal memory from the input image. Within internal memory, we will operate on two lines of data (640 columns/line) at a time, averaging every two pixels (horizontal neighbors) and producing two lines of output (320 columns/line) that are stored in the external memory.

To accomplish this, we will need to take advantage of the IDM by initializing the input and output streams. At the start of the function, two instantiations of a new structure dstr_t are declared. You can view the structure contents of dstr_t on p. 2-11 of the IDK Programmer's Guide. These structures are stream "objects". They give us access to the data when using the dstr_open() command. In this case dstr_i is an input stream as specified in the really long command dstr_open(). Thus after opening this stream we can use the get_data command to get data one line at a time. Streams and memory usage are described in greater detail in the second project lab. This data flow for the pre-scale is shown in data flow.

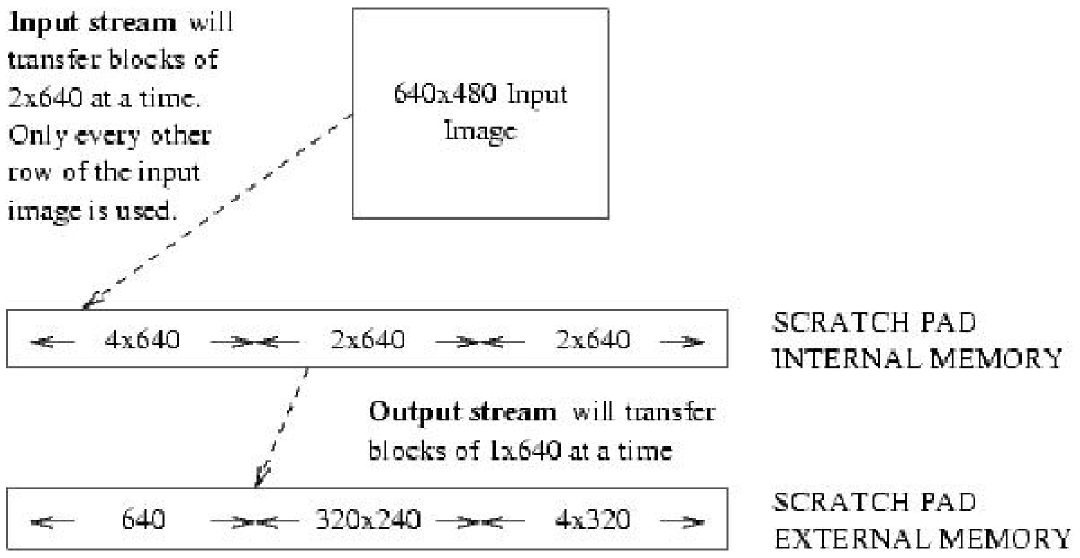

Input stream will transfer blocks of 2x640 at a time. Only every other row of the input image is used.
640x480 Input Image
4x640
2x640
2x640
SCRATCH PAD INTERNAL MEMORY
Output stream will transfer blocks of 1x640 at a time
640
320x240
4x320
SCRATCH PAD EXTERNAL MEMORY

Figure 2.20: Data flow of input and output streams.

To give you a better understanding of how these streams are created, let's analyze the parameters passed in the first call to dstr_open() which opens an input stream.

External address: in_image->data This is a pointer to the place in external memory serving as the source of our input data (it's the source because the last function parameter is set to DSTR_INPUT). We're going to bring in data from external to internal memory so that we can work on it. This external data represents a frame of camera input. It was captured in the main function using the VCAP_getframe() command.

External size: (rows + num_lines) * cols = (240 + 2) * 640 This is the total size of the input data which we will bring in. We will only be taking two lines at a time from in_image->data, so only 240 rows. The "plus 2" represents two extra rows of input data which represent a buffer of two lines - used when filtering, which is explained later.

Internal address: int_mem This is a pointer to an 8x640 array, pointed to by scratchpad->int_data. This is where we will be putting the data on each call to dstr_get(). We only need part of it, as seen in the next parameter, as space to bring in data.

Internal size: 2 * num_lines * cols = 2 * 2 * 640 The size of space available for data to be input into int_mem from in_image->data. We pull in two lines of the input frame so it num_lines * cols. We have the multiply by 2 as we are using double buffering for bringing in the data. We need double the space in internal memory than the minimum needed, the reason is fully explained in IDK Programmer's Guide.

Number of bytes/line: cols = 640, Number of lines: num_lines = 2 Each time dstr_get_2D() is called, it will return a pointer to 2 new lines of data, 640 bytes in length. We use the function dstr_get_2D(), since we are pulling in two lines of data. If instead we were only bringing in one line, we would use dstr_get() statements.

External memory increment/line: stride*cols = 1*640 The IDM increments the pointer to the external memory by this amount after each dstr_get() call.

Window size: 1 for double buffered single line of data (Look at the three documentation pdfs for a full explanation of double buffering) The need for the window size is not really apparent here. It will become apparent when we do the 3x3 block convolution. Then, the window size will be set to 3 (indicating three lines of buffered data). This tells the IDM to send a pointer to extract 3 lines of data when dstr_get() is called, but only increment the stream's internal pointer by 1 (instead of 3) the next time dstr_get() is called. Thus you will get overlapping sets of 3 lines on each dstr_get() call. This is not a useful parameter when setting up an output stream.

Direction of input: DSTR_INPUT Sets the direction of data flow. If it had been set to DSTR_OUTPUT (as done in the next call to dstr_open()), we would be setting the data to flow from the Internal Address to the External Address.

We then setup our output stream to write data to a location in external memory which we had previously created.

Once our data streams are setup, we can begin processing by first extracting a portion of input data using dstr_get_2D(). This command pulls the data in and we setup a pointer (in_data) to point to this internal memory spot. We also get a pointer to a space where we can write the output data (out_data) when using dstr_put(). Then we call the component function pre_scale() (in pre_scale.c) to operate on the input data and write to the output data space, using these pointers.

The prescaling function will perform the horizontal scaling by averaging every two pixels. This algorithm operates on four pixels at a time. The entire function is iterated within pre_scale_image() 240 times, which results in 240 * 2 rows of data being processed – but only half of that is output.

Upon returning to the wrapper function, pre_scale_image, a new line is extracted; the pointers are updated to show the location of the new lines and the output we had placed in internal memory is then transferred out. This actually happens in the dstr_put() function – thus is serves a dual purpose; to give us a pointer to internal memory which we can write to, and the transferring of its contents to external memory.

Before pre_scale_image() exits, the data streams are closed, and one line is added to the top and bottom of the image to provide context necessary for the next processing steps (The extra two lines - remember?). Also note, it is VERY important to close streams after they have been used. If not done, unusual things such as random crashing and so may occur which are very hard to track down.

Now that the input image has been scaled to a quarter of its initial size, we will proceed with the four image processing algorithms. In img_proc.c, the set_ptr() function is called to set the variable out_ptr to point to the correct quadrant on the 640x480 output image. Then copy_image(), copy_image.c, is called, performing a direct copy of the scaled input image into the lower right quadrant of the output.

Next we will set the out_ptr to point to the upper right quadrant of the output image and call conv3x3_image() in conv3x3_image.c. As with pre_scale_image(), the _image indicates this is only the wrapper function for the ImageLIB (library functions) component, conv3x3(). As before, we must setup our input and output streams. This time, however, data will be read from the external memory (where we have the pre-scaled image) and into internal memory for processing, and then be written to the output image. Iterating over each row, we compute one line of data by calling the component function conv3x3() in conv3x3.c.

In conv3x3(), you will see that we perform a 3x3 block convolution, computing one line of data with the low pass filter mask. Note here that the variables IN1[i], IN2[i], and IN3[i] all grab only one pixel at a time.

This is in contrast to the operation of pre_scale() where the variable in_ptr[i] grabbed 4 pixels at a time. This is because in_ptr was of type unsigned int, which implies that it points to four bytes (the size of an unsigned int is 4 bytes) of data at a time. IN1, IN2, and IN3 are all of type unsigned char, which implies they point to a single byte of data. In block convolution, we are computing the value of one pixel by placing weights on a 3x3 block of pixels in the input image and computing the sum. What happens when we are trying to compute the rightmost pixel in a row? The computation is now bogus. That is why the wrapper function copies the last good column of data into the two rightmost columns. You should also note that the component function ensures output pixels will lie between 0 and 255. For the same reason we provided the two extra "copied" lines when performing the prescale.

Back in img_proc.c, we can begin the edge detection algorithm, sobel_image(), for the lower left quadrant of the output image. This wrapper function, located in sobel_image.c, performs edge detection by utilizing the assembly written component function sobel() in sobel.asm. The wrapper function is very similar to the others you have seen and should be straightforward to understand. Understanding the assembly file is considerably more difficult since you are not familiar with the assembly language for the c6711 DSP. As you'll see in the assembly file, the comments are very helpful since an "equivalent" C program is given there.

The Sobel algorithm convolves two masks with a 3x3 block of data and sums the results to produce a single pixel of output. One mask has a preference for vertical edges while the other mask for horizontal ones. This algorithm approximates a 3x3 nonlinear edge enhancement operator. The brightest edges in the result represent a rapid transition (well-defined features), and darker edges represent smoother transitions (blurred or blended features).

2.4.2.4 Part One

This section provides a hands-on introduction to the IDK environment that will prepare you for the lab exercise. First, connect the power supply to the IDK module. Two green lights on the IDK board should be illuminated when the power is connected properly.

You will need to create a directory img_proc for this project in your home directory. Enter this new directory, and then copy the following files as follows (again, be sure you're in the directory img_proc when you do this):

- copy V:\ece320\idk\c6000\IDK\Examples\NTSC\img_proc
- copy V:\ece320\idk\c6000\IDK\Drivers\include
- copy V:\ece320\idk\c6000\IDK\Drivers\lib

After the IDK is powered on, open Code Composer 2 by clicking on the "CCS 2" icon on the desktop. From the "Project" menu, select "Open," and then open img_proc.pjt. You should see a new icon appear at the menu on the left side of the Code Composer window with the label img_proc.pjt. Double click on this icon to see a list of folders. There should be a folder labeled "Source." Open this folder to see a list of program files.

The main.c program calls the img_proc.c function that displays the output of four image processing routines in four quadrants on the monitor. The other files are associated with the four image processing routines. If you open the "Include" folder, you will see a list of header files. To inspect the main program, double click on the main.c icon. A window with the C code will appear to the right.

Scroll down to the tskMainFunc() in the main.c code. A few lines into this function, you will see the line LOG_printf(&trace,"Hello\n"). This line prints a message to the message log, which can be useful for debugging. Change the message "Hello\n" to "Your Name\n" (the "\n" is a carriage return). Save the file by clicking the little floppy disk icon at the top left corner of the Code Composer window.

To compile all of the files when the ".out" file has not yet been generated, you need to use the "Rebuild All" command. The rebuild all command is accomplished by clicking the button displaying three little red arrows pointing down on a rectangular box. This will compile every file the main.c program uses. If you've only changed one file, you only need to do a "Incremental Build," which is accomplished by clicking on the button with two little blue arrows pointing into a box (immediately to the left of the "Rebuild All" button).

Click the "Rebuild All" button to compile all of the code. A window at the bottom of Code Composer will tell you the status of the compiling (i.e., whether there were any errors or warnings). You might notice some warnings after compilation - don't worry about these.

Click on the "DSP/BIOS" menu, and select "Message Log." A new window should appear at the bottom of Code Composer. Assuming the code has compiled correctly, select "File" -> "Load Program" and load img_proc.out (the same procedure as on the other version of Code Composer). Now select "Debug" -> "Run" to run the program (if you have problems, you may need to select "Debug" -> "Go Main" before running). You should see image processing routines running on the four quadrants of the monitor. The upper left quadrant (quadrant 0) displays a low pass filtered version of the input. The low pass filter "passes" the detail in the image, and attenuates the smooth features, resulting in a "grainy" image. The operation of the low pass filter code, and how data is moved to and from the filtering routine, was described in detail in the previous section. The lower left quadrant (quadrant 2) displays the output of an edge detection algorithm. The top right and bottom right quadrants (quadrants 1 and 3, respectively), show the original input displayed unprocessed. At this point, you should notice your name displayed in the message log.

2.4.2.4.1 Implementation

You will create the component code flip_invert.c to implement an algorithm that horizontally flips and inverts the input image. The code in flip_invert.c will operate on one line of the image at a time. The copyim.c wrapper will call flip_invert.c once for each row of the prescaled input image. The flip_invert function call should appear as follows:

flip_invert(in_data, out_data, cols);

where in_data and out_data are pointers to the input and output buffers in internal memory, and cols is the length of each column of the prescaled image.

The img_proc.c function should call the copyim.c wrapper so that the flipped and inverted image appears in the top right (first) quadrant. The call to copyim is as follows: copyim(scratch_pad, out_img, out_ptr, pitch);

This call is commented out in the im_proc.c code. The algorithm that copies the image (unprocessed) to the screen is currently displayed in quadrant 1, so you will need to comment out its call and replace it with the call to copyim.

Your algorithm should flip the input picture horizontally, such that someone on the left side of the screen looking left in quadrant 3 will appear on the right side of the screen looking right. This is similar to putting a slide in a slide projector backwards. The algorithm should also invert the picture, so that something white appears black and vice versa. The inversion portion of the algorithm is like looking at the negative for a black and white picture. Thus, the total effect of your algorithm will be that of looking at the wrong side of the negative of a picture.

NOTE: Pixel values are represented as integers between 0 and 255.

To create a new component file, write your code in a file called "flip_invert.c". You may find the component code for the low pass filter in "conv3x3_c.c" helpful in giving you an idea of how to get started. To compile this code, you must include it in the "img_proc" project, so that it appears as an icon in Code Composer. To include your new file, right click on the "img_proc.pjt" icon in the left window of Code Composer, and select "Add Files." Compile and run!

2.4.3 Video Processing Part 2: Grayscale and Color[32]

2.4.3.1 Introduction

The purpose of this project lab is to introduce how to further manipulate data acquired in grayscale mode and then expand this to the realm of color. This lab is meant as a follow-up to "Video Processing Part 1: Introductory Exercise,". This lab will implement a grayscale auto-contrast and color image manipulation.

[32]This content is available online at <http://cnx.org/content/m11988/1.2/>.

You will complete an introductory exercise to demonstrate your familiarity with the IDK programming environment. You will then complete an introductory exercise in how to use color; and modify a C skeleton to apply simple color masks to video input from the camera.

After this lab, you should be able to effectively and efficiently manipulate grayscale images, as well as modify color images.

You may want to refer to the following TI manuals:

- IDK User's Guide[33]
- IDK Video Device Drivers User's Guide[34]
- IDK Programmer's Guide[35] . Section 2 is very, very important in this lab.

2.4.3.2 Prelab

Having familiarized yourself with grayscale images in the previous project lab, the first part of the prelab will require you to code a function similar to the flip_invert function you have already designed, while the second part of the prelab will introduce how to use and access color images.

2.4.3.2.1 Grayscale

In this part of the prelab exercise, you will develop an algorithm to find the maximum and minimum values of a grayscale input image. Create a function that will process one row of the image at a time and find the overall minimum and maximum intensities in the image.

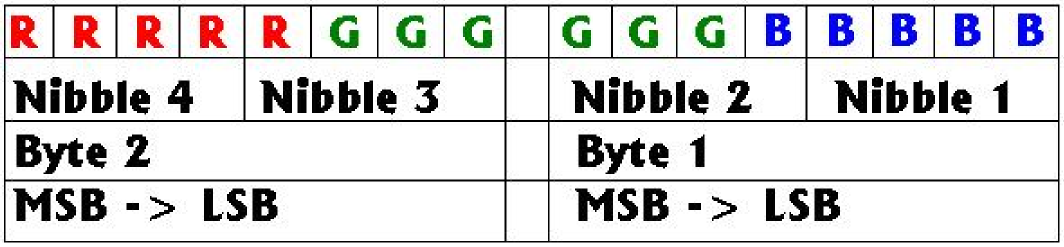
auto_contrast_find_extrema(in_data, min, max, col)

2.4.3.2.2 Color

The NTSC camera acquires images in the color format YCbCr, where Y represents luminosity, Cb the blue component, and Cr the red component. Each image must be converted to 16-bit RGB for output on a standard color computer monitor. The function "ycbcr422pl_to_rgb565" performs this conversion. Knowing how this function converts each pixel to RGB is relatively unimportant, however, knowing the packed (5:6:5) RBG format is essential.

Before we ignore the ycbcr422pl_to_rgb565 function completely, it is useful to look at how it operates. Find the run time of the function by examining the file "ycbcr422pl_to_rgb565.c" and note that it must convert an even number of pixels at a time. If it were possible to have this function process the whole color image at in one function call, how many clock cycles would the function take? Since we are limited in the number of rows we can modify at a time, how many clock cycles should it take to process the whole image one row at a time? To demonstrate the overhead needed for this function, note how many clock cycles the function would take if it converted the whole image two pixels at a time.

[33]http://www-s.ti.com/sc/psheets/spru494a/spru494a.pdf
[34]http://www-s.ti.com/sc/psheets/spru499/spru499.pdf
[35]http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf



R	R	R	R	R	G	G	G		G	G	G	B	B	B	B	B
Nibble 4				Nibble 3					Nibble 2				Nibble 1			
Byte 2									Byte 1							
MSB -> LSB									MSB -> LSB							

Figure 2.21: RGB (5:6:5). A packed RGB pixel holds 5 bits for red, 6 bits for green, and 5 bits for blue.

Since each color is not individually addressable in the packed RGB format (e.g. bits representing red and blue are stored in the same byte), being able to modify different bits of each byte is necessary. To help clarify what bits are being set/cleared/toggled, numbers can be represented in hex format. For example, the integer 58 can be represented by "00111010" in binary or by "3A" in hex. In C, hex numbers are indicated with the prefix "0x."

Example:

- int black = 0x00; // black = 0
- int foo_h = 0xF0; // foo_h = 240
- int foo_l = 0x0D; // foo_l = 13

Another thing to note is that each pixel requires two bytes of memory, requiring two memory access operations to alter each pixel. Also NOTE that in a row of input color data, the indexing starts at 1. Thus RGB[1] contains red/green data and then RGB[2] contains the green/blue data – both for the first pixel.

What is the packed RGB value for the highest intensity green? What is the value of the first addressable byte of this 'hi-green' pixel? What is the value of the second byte?

Now, say you are given the declaration of a pixel as follows:

int pixel;

Write a simple (one line is sufficient) section of code to add a blue tint to a pixel. Do the same for adding a red tint, and for a green tint (may require more than one line). Use the and (represented by an ampersand) operator to apply a mask.

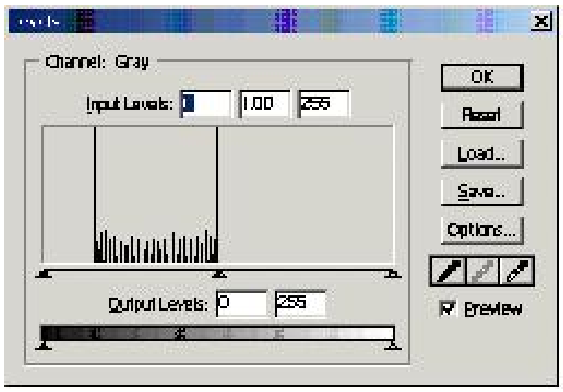
2.4.3.3 Implementation

The first part of this lab will require you to write a function to perform auto-contrasting. You should use your function from prelab 2.1 to obtain the maximum and minimum values of the image, and then create another function to do the appropriate scaling.

The second part of this lab will involve implementing some simple, and hopefully cool, color effects.

2.4.3.3.1 Grayscale

Use the function you designed in prelab 2.1 to create an algorithm to auto-contrast the image. Auto-contrast is accomplished by scaling the pixel value from the min-to-max range to the full range. This effect is seen below:



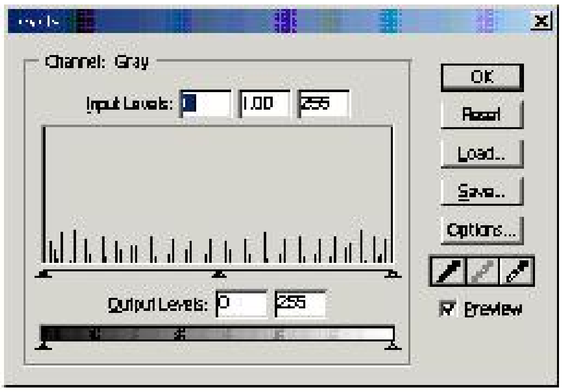


Figure 2.22: (left) Frequency of a grayscale image with pixel intensities ranging in value from 32 to 128, and (right) Frequency of the same grayscale image after performing an auto-contrast.

Recall from “Introduction to the IDK” that the DSP has a floating point unit; the DSP will perform floating point instructions much faster than integer division, quare-root, etc.

Example:

- int opposite, adjacent;
- float tan;
- tan = ((float) opposite) / ((float) adjacent);

This function should be called similarly to the flip_invert function in the previous lab. Once you have implemented your function, look for ways to optimize it. Notice that you must loop through the image twice: once to find the minimum and maximum values, and then again to apply the scaling. (Hint: the function dstr_rewind rewinds the image buffer).

Use the same core files for this part of the lab as were used in the previous lab. You may simply make a copy of the previous lab’s folder and develop the necessary code from there.

2.4.3.3.2 Color

In this part of the lab, you will use the concepts from the prelab to implement certain effects.

Copy the directory “V:\ece320\projects\colorcool” to your W: drive.

We want to use a certain area of the screen as a "control surface". For example, the fingers held up on a hand placed within that area can be used as a parameter, to control the image on the screen. Specifically, we will use the total brightness of this control surface to control the color tint of the screen.

You are given a shell program which takes in a color input frame in YcbCr format and converts it to RGB. You will modify this shell to

- 1. Calculate the total brightness
- 2. Calculate the tint for each color component R, G and B.
- 3. Apply the tint to the image

2.4.3.3.2.1 Code Briefing

The code provided merely performs a color conversion required to go from the input NTSC image to the output RGB image. The relevant streams of data are brought in using the in_luma, in_cr, in_cb odd and even streams

The odd, even is done because the input YcbCr data is interlaced and the different "color" components Y(luminance), Cr, and Cb are stored in different arrays, unlike RGB where the data is packed together for each pixel. Thus the streams are accessed inside the color_conv_image wrapper function. We then pass a line at a time to the color_conv component function which converts and flips one line at a time.

We will need to modify the code here, in color_conv to achieve your goals. The control surface will be a square block 100 by 100 pixels in the bottom left corner of the screen. The brightness will be calculated by summing all the R, G and B values of all the pixels in this portion of the screen. We then apply the tint effect as such:

- if the total brightness is below a certain level 'X': use a red tint,
- if the total brightness is above 'X' and below 'Y' : use a green tint,
- if above 'Y' : use a blue tint

The tint has to be scaled too. For example, if brightness is less than X but close to it we need a high blue. But if it's closer to zero we need a darker blue and so on. The scaling need not be linear. In fact if you did the auto-contrast function you will have noticed that the floating point operations are expensive, they tend to slow the system. This is more so in the color case, as we have more data to scale. So try to use simple bit shifts to achieve the needed effect.

- Right Shift : $\gg$
- Left Shift : $\ll$
- Masking : Use a single ampersand, so to extract the first red component: `RGB[1] &amp; 0xF8`

2.4.3.3.2.2 Tips and Tricks

You're on your own now! But some things to remember and to watch out for are presented here, as well as ideas for improvement. Remember:

- The input is two bytes per pixel. Keep the packed RGB format in mind.
- Also we process one line at a time from top to bottom. We cannot go back to previous lines to change them. So we can only modify the tint of the screen below the control surface. What you could do however is keep global variables for the different scalings in main. Then pass these to color_conv by reference, and update it when converting colors. But perform the update after using the existing scale values to scale the screen region above the control surface. This will introduce a delay from scaling change to screen update. This can be solved by copying the entire input to memory before outputting it but this is quite expensive, and we'll deal with memory in the next section.
- Be careful when performing masking, shifting and separting. Bring things down to least significant set of bits (within a byte) to simplify thinking of the scaling. Also be careful not to overlap masks, especially during shifting and adding

Here are a few recommendations:

- Try to use the Y data passed to the color_con funtion to compute the brightness – much faster.
- Also poke around and find out how to use the Cr, Cb data and scale those. It's far less expensive and may produce neater results.
- If something doesn't work, think things through again. Or better still take a break and then come back to the problem.

2.4.4 Video Processing Part 3: Memory Management[36]

2.4.4.1 Introduction

In this project, you will learn how to combine the use of the external and internal memory systems of the IDK, as well as how to use the TI-supplied library functions. It may seem daunting, but fear not, there are only a few commands to learn. The key is to know how to use them well.

The project assignment will involve copying a portion of the input image and displaying it in a different area of the screen. The area copied to should be quickly and easily adjustable in the code. In addition to this, we will filter this copied portion and display it as well.

And you must refer to the following TI manuals available on the class website under the Projects section. The sections mentioned in Video Processing Lab 1 are also important.

- IDK Video Device Drivers User's Guide[37] The Display and Capture systems are important – the figures on pages 2-7 and 3-8 are useful too.
- IDK Programmer's Guide[38] . Sections 2 and 5 are the ones needed. Section 2 is very important here. Keep a printout if necessary, it is useful as a reference.

2.4.4.2 Memory - The Basics

As explained in the previous lab, there are two sections of memory, internal and external. The internal is small but fast, whereas the external is large but slow. An estimate of the sizes: 25K for the internal, 16M for the external, in bytes.

As seen earlier, this necessitates a system of transferring memory contents between the two memory systems. For example, an input color screen is in YCbCr format. This consists of 640 X 480 pixels with 8 bits per pixel. This results in 300 Kbytes, which cannot be stored in internal memory. This same problem applies for the output buffer.

Thus it is best to use the external memory for storage of large chunks of data, and the internal memory for processing of smaller chunks. An example of this, as seen in the previous lab, was color conversion. In that system, we brought in the input frame line-by-line into internal memory. We then converted the color space and stored the results in internal memory as well. Following this, we transferred the results to external memory.

This is the basic overview of the need for the two memory systems. Next we will discuss the setup and use of memory spaces, explaining the workings of the color conversion program

2.4.4.3 Memory - Setup

Firstly, please copy the directory below to your account so you can follow the code as we go along.

`V:\ece320\projects\colorcool`

The program in this directory is a basic color conversion program which outputs the input frame to the display.

2.4.4.3.1 Allocating Memory Space

The first step in using memory is to declare it, i.e. tell the compiler to setup some space for it. This is done at the very beginning of the 'main.c' file.

- 1. Declare the type of memory space and it's name. Use the `#pragma DATA_SECTION` command. There are two parameters :
    - a) the name of the memory spaces

[36]This content is available online at <http://cnx.org/content/m11989/1.2/>.
[37]http://www-s.ti.com/sc/psheets/spru499/spru499.pdf
[38]http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf

  - b) and the type – internal or external

- 2. Then specify the byte alignment using the #pragma DATA_ALIGN command. This is similar to the byte alignment in the C54x. So, to store black and white images, you would use 8 bits. But for RGB, you would use 16 bits.

```
// specifies name of mem space -- ext_mem
// and type as internal memory -- ".image:ext_sect"
// the data_align specification is the byte alignment -- ours is
// 8 bits
#pragma DATA_SECTION(ext_mem,".image:ext_sect");
#pragma DATA_ALIGN(ext_mem,8);

// specifies name of mem space -- int_mem
// and type as internal memory -- ".image:int_sect"
// the data_align specification is the byte alignment -- ours is
// 8 bits
#pragma DATA_SECTION(int_mem,".chip_image:int_sect");
#pragma DATA_ALIGN(int_mem, 16);
```

- We then specify the size of the memory space. We use a variable for the basic unit size (e.g. unsigned char for 1 byte) and a length for the number of basic units needed. Please note, the memory space is not delineated by 'image' rows or columns, The system thinks it is one long array of data, it is up to us to process this as separate lines of 'image' data.

```
// specify size as  width 640
//     height 480
//     and 8 bytes per pixel
// which could represent an RGB screen of 640 X 480 with
// 2 bytes per pixel. Unsigned char = 8 bytes
unsigned char ext_mem[640 * 480 * 2];

// here we create 6 lines of RGB data of 640 columns each,
// 2 bytes per pixel
unsigned char int_mem[6 * 2 * 640];
```

Now have a look at the main.c file and take note of the memory spaces used. The internal memory is of size 12 * 640. This single memory space is going to be used to store both the input lines from the camera image and also the results of the color conversion, thus explaining its large size. Basically the internal memory is partitioned by us for different buffers. The output data buffer needs only 4*640 bytes thus it's space starts at

`int_mem + (8 * cols); //cols = 640`

and ends at 12*cols – which gives us 4*cols of space. Though it is useful to partition internal memory in such a way, it is recommended not to. It is very easy to mess up the other data too, so simple, so our solution would have been to create a separate memory space of size 4*cols.

The external memory, though declared here, will not be used in the program, however you may need to allocate some external memory for this project lab assignment.

2.4.4.3.2 The INPUT and OUTPUT buffers and Main.c Details

Good examples of the external memory use are the input buffer (captured image) and output buffer (to be placed onto the screen). There are a few steps in obtaining these buffers:

- 1. First, we open the capture and display devices in tskMainFunc() using

  ```
  VDIS_open();
  VCAP_open();
  ```

- 2. If the open calls are successful, we then call the color function to process the video feed using

  ```
  color(VCAP_NTSC, VDIS_640X480X16, numFrames);
  ```

  This specifies:

  · the capture image format – NTSC
  · display image format and size
  · numFrames to run the system for – in our case one day to be passed on to the color function. Please note, we merely specify the formats but do not configure the system to use these formats, yet.

  We then move on to the color(...) function within main.c
- 3. First we declare some useful pointers which we will use for the various images and their components and so forth. The IMAGE structure holds a pointer to the image array (img_data). In addition, it holds integers for the number of image rows (img_rows) and number of image columns (img_cols).(Implementation Details in img_proc.h) Declare more of these structures as needed for any memory spaces you create yourself. Furthermore, "scratch_pad" structures hold information about the location and size of internal and external memories. This is another use of pointers being used to hold the locations of the different memory spaces. (Implementation Details in img_proc.h) We also configure the display and capture formats using

  ```
  VDIS_config(displayMode);
  VCAP_config(captureMode);
  ```

- Following this we enter the loop :

  ```
          for (frameCnt=0; frameCnt<numFrames; frameCnt++)
  ```

  This loop iterates for a set number of frames and processes them one at a time. And the lines following this :

  ```
  input  = VCAP_getFrame(SYS_FOREVER);
  output = (Uint16*)VDIS_toggleBuffs(0);
  ```

  are used to obtain the capture and output frames. After this statement, 'input' will hold a pointer to external memory where the captured frame is stored. The 'input' pointer holds pointers 'y1', 'c1' etc to the different color component of the image. These color components are in external memory as well. And 'output' will hold a pointer to a buffer in external memory, to which we will write whatever we need to output to the screen. Basically the buffer is the size of the output frame (640 X 480 X 2 bytes/pixel), and we can write what we wish to it. And, the next time togglebufs(0) is called, everything we placed in that buffer will be put on the screen. And a new buffer will be allocated, the pointer 'output' will be updated and we can now write to the next frame. The next line

```
out_image.img_data = (unsigned char *) output;
```

updates the pointers we had setup. We then move on to the color_convert(..) routine. We pass the memory pointers we had created so that our color_conv program can process the input frame we obtained. In color_conv, we begin by setting up streams to bring in data and streams to send out data. After that we begin the color-space conversion.

2.4.4.4 Memory Streams

Memory streams are structures used to facilitate the transfer of data between internal and external memory. But why do we need a structure? Can't we just do it manually?

You could, but you'd spend two months to do the same work as a single stream, which only takes a few minutes (hopefully). So to cut a long story short, streams are your friends. They help remove much of the complexity associated with internal/external memory transfers.

First, please make sure you've read the manual sections mentioned on page 1. There are two basic types of streams : input and output. Input is a transfer from external to internal. Output is the opposite. Think of bringing in and putting out.

For each type we need to specify various parameters, such source and destination addresses, increments, size of transfer chunks and so forth. This specification is done once for each transfer session (say, once for each image transfer), using the dstr_open command. We then use dstr_get and dstr_put commands to tell the stream to bring in or put out data one chunk at a time.

2.4.4.4.1 Creating and Destroying Streams

Streams are dstr_t objects. You can create a dstr_t object and then initialize it using the dstr_open() command. Basically, start with,

```
dstr_t   o_dstr;
```

Then use the

```
dstr_open (...);
```

The dstr_open () specification is given in the manual. Some clarifications are made here. As an example we will consider the output stream o_dstr in color_convert(). This stream is an output stream. This stream is used to transfer data from internal memory to the screen output data buffer. (we captured the buffer's memory location in the previous section using togglebufs(), it's memory address is stored in the pointer out_image->img_data)

Arguments (note : out_rows = 480, out_cols = 640):

-
  ```
  dstr_t  *dstr
  ```
  needs a pointer to the data stream object we wish to use. In our case this would be o_dstr.

-
  ```
  void *x_data
  ```

takes a pointer to the location in external memory which we are using. In our program this is specified in out_image->img_data. And since we are using an output stream, this argument specifies the Destination of the stream. (This argument is the Source for an input stream)

-
  ```
  int x_size
  ```

  takes in the size of the external data buffer to which we are writing to. This specifies the actual number of bytes of external memory we will be traversing. So this is NOT necessarily the full size of the output buffer (i.e. NOT always 640 X 480 X 2) For our example we are writing to the full screen hence we use

  ```
  (2 * out_rows * out_cols)
  ```

  which results in 640 X 480 X 2 bytes of data. An example of the exception is when we write to only, say, the first 10 rows of the screen. In this case we would only traverse: 10 X 640 X 2 bytes. One more thing to note is that if you need to only write to the first 40 columns of the first 10 rows, you would still need to traverse the same amount of space and you would use 10 X 640 X 2 bytes again for this argument. In this case however, you will be skipping some of the data, as shown later.

-
  ```
  void *i_data
  ```

  takes a pointer to the location in internal memory we are using. In our program this is specified as out_data. And since we are using an output stream, this argument specifies the Source of our stream. (This argument is the Destination for an input stream).

-
  ```
  unsigned short i_size
  ```

  is used to specify the total size of the internal memory we will be using. In our case we will be writing one line of the output screen - (4 * out_cols) This is the amount we allocated earlier. This evaluates to 640 * 2 * 2 bytes. The extra '2' is needed for double-buffering, which is a system used by the IDK for transferring data into internal memory. Basically, the IDM (image data manger) needs twice the amount of internal memory as data transferred. i.e. one line is worth only 640 * 2 bytes, but because of double buffering we allocate twice that for the IDM's use. Remember this when allocating memory space for internal memory.

-
  ```
  unsigned short quantum
  ```

  specifies the amount of data transferred in a single dstr_get or dstr_put statement. In our case it would be (2 * out_cols). This evaluates to 640 * 2 bytes – one line of the output screen each time we use dstr_put Now, if we were transferring only part of a line, let's take the first 40 columns of the first 10 rows example. With each dstr_put, we will output only the first forty columns of each row. Thus we are transferring 40 * 2 bytes in each call. But this can be extended further. By use of the 'dstr_get_2D' we can transfer multiple lines of data. So we can, say, transfer two full rows of the output screen (4 * cols) or in our mini-example this would mean 2 * 40 * 2 bytes. Transferring of multiple lines is very useful, especially when using filters which work on 2-D 'regions' of data.

-
  ```
  unsigned short multiple
  ```

specifies the number of lines we are transferring with each call. Now this is not the conceptual number of lines. It is the physical multiple of argument 6 that we are transferring. It is best to leave this at one and modify argument 6 above.

-
  ```
  unsigned short stride
  ```

  needs the amount by which to move the external memory pointer. This gives us control over how the lines are extracted. In our case, it being the simplest, we move one line at a time : 2*out_cols The stride pointer is especially useful when creating input streams. For example you can pull in overlapping lines of input. So you can pull in lines 1 and 2 in the first dstr_get(). The next dstr_get() can pull in lines 2 and 3 or you can setup it up to pull lines 3 and 4 or 4 and 5 or ..... depending on the stride. In particular, this is useful in Sobel (edge-detect) filtering, where you need data above and below a pixel to evaluate the output.

-
  ```
  unsigned short w_size
  ```

  is the window size. For transferring a single line at a time we would use '1' here, and the system will recognize this is as one line double-buffered. But if we needed to transfer two lines we would merely submit '2' as the argument.

-
  ```
  dstr_t dir
  ```

  specifies the type of stream. Use DSTR_OUTPUT for output stream and DSTR_INPUT for input stream.

Once a stream is created, you can use the get and put commands in a loop, to bring in or put out line/s of data. Calling dstr_get on an input stream will give you a buffer where data is present to be read off. And calling an output stream will give you a buffer to which you can write data (which will be transported out on the next dstr_put call).

Remember, you have to be careful how many times you call these functions as you so not want to overflow. For example in our output example, we could call the dstr_put() upto 480 times – the number of single row transfers. Anymore, and the system may crash.

Also please remember to close the stream once you are done with it, i.e after all iterations. See the color_convert function to see when we close the streams using dstr_close(...). This is VERY important, since not closing a stream will cause random crashing of your system. The system may seem to run as you expected, but it will crash, if not after 1 second, then after 1 minute or 1 hour. This problem is one of the first you should look for when debugging such symptoms.

Also take a look at the streams for the input color components YCbCr to see how they are setup. You will find the figure on Device Driver Paper page 3-8 very useful in deciphering these streams. Understand them and you are set!

Quick-Test: Write a stream to obtain one-line buffers for columns 31 through 50 (20 columns) of the output buffer, with 50 rows. This rectangular region should start at pixel (100, 200). So each transfer should give a buffer of 20 * 2 bytes worth of information. Think of how you'd setup the stream.

2.4.4.4.2 Memory Tricks and Tips

Some simple memory tips are given here, you can come up with your own too.

- Know how data flows in your system, this will help you increse efficiency and possibly eliminate complex stream use as well.

- The dstr_get_2D and dstr_put_2D are used for multiple line transfers. Use these to your advantage.
- You can use a simple memory ping-pong system to lessen memory use. If you need to use, say 200 X 300 rectangular region and filter it repeatedly. Then keep two memory 200 X 300 memory spaces. Write to the first, filter out to the second. Then filter the second out to the first, and so on until you're done.

2.4.4.4.3 Limitations

- Space is a always a factor, especially with internal memory.
- It's harder to extract columns of data as opposed to rows. To transfer a column, you need to setup a different stream, one that skips a whole 'row-1' of data with each dstr_get statement. Then you will need to iterate this to get the pixel on each row of that column. Multiple get's are necessary because the data is not contiguous in memory.

2.4.4.5 IDK Libraries

To make your life easier, the IDK has some libraries which you can use for common image processing tasks. One such function is the Sobel (edge-detect) filter. These functions are usually hand coded in assembly and are extremely efficient, so it's best not to try to beat them.

The Sobel filter is contained in the file 'sobel_h.asm' and the header file needed is 'sobel_h.h'. You must add the program file and it's header in the project to use them. Next you will need to create a wrapper function and use the

```
#include "sobel_h.h"
```

directive in the wrapper function at the top. Don't forget to create a header function for your wrapper as well and add it to your project.

Next you will need to setup the streams and provide the assembly function the needed parameters. Namely, it needs a pointer to 3 lines worth of input data to be processed, one line of output data, the number of columns and number of rows. The library Sobel filter works on 3 lines of input and produces 1 line of output with each call. Look at the 'sobel_h.asm' to get a better understanding of the parameters

This material should be familiar from the previous lab where we explored wrapper and component functions. Now time for the assignment!

2.4.4.6 The Assignment

Your assignment, should you choose to accept it is to build a simple filter system. You will start with the basic color conversion program given to you in:

```
V:\ece320\projects\colorcool
```

The system will copy the red-component of a 100 by 100 area of the screen (let's call this area M). It will place this in a different area of the screen. Also you will need to place a Sobel filtered version of this red-area to the screen as well. The locations where the copied and filtered images are placed must be quickly modifiable on request (use variable position as parameters to wrapper functions rather than fixed coordinates)

2.4.4.6.1 Tips, Tricks and Treats

- Plan the system before hand to make efficient use of modular functions and memory
- For example, you only need just one "output area if size M" function to screen.
- Keep handy pointers to the different memory spaces.

- Use wrapper functions for the filter and copy_to_screen operations.
- Write the modules so that they can be tested independently.
- Be careful with color conversion. For example when copying the red-component of M, you need only 8 bits per pixel.
- Keep the previous lab in mind when deciding when/where to extract the area M.

2.5 Surround Sound

2.5.1 Surround Sound: Passive Encoding and Decoding[39]

2.5.1.1 Introduction

To begin understanding how to decode the Dolby Pro Logic Surround Sound standard, you will implement a Pro Logic encoder and a passive surround sound decoder. This decoder operates on many of the same principles as the more sophisticated commercial systems. Significantly more technical information regarding Dolby Pro Logic can be found at *Gundry* [5].

2.5.1.2 Encoder

You will create a MATLAB implementation of the passive encoder given by the block diagram in Figure 2.23.

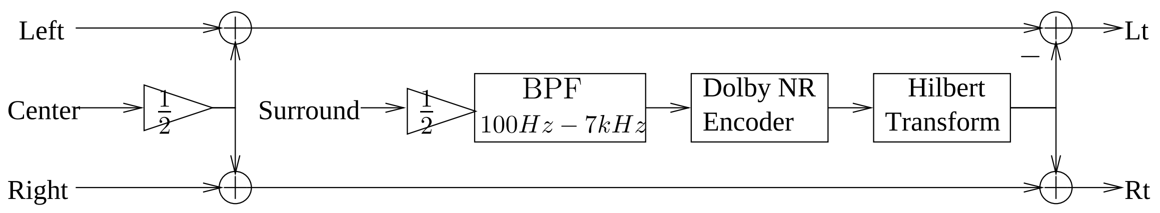


Figure 2.23: Dolby Pro Logic Encoder

The encoder block diagram shows four input signals: Left, Center, Right, and Surround. These are audio signals created by a sound designer during movie production that are intended to play back from speakers positioned at the left side, at the front-center, at the right side, and at the rear of a home theater. The system in the block diagram encodes these four channels of audio on two output channels, `Lt` and `Rt`, in such a way that an appropriately designed decoder can approximately recover the original four channels. Additionally, to accommodate those who do not use a surround sound receiver, the encoder outputs are listenable when played back on a stereo (two-channel) system, even retaining the correct left-right balance.

The basic components of the encoder are multipliers, adders, a Hilbert transform, a band-pass filter, and a Dolby Noise Reduction encoder. If you wish to implement Dolby Noise Reduction, refer to *Dressler* [4]. The other components are discussed below.

The transfer function of the Hilbert Transform is shown in Figure 2.24. The Hilbert Transform is an ideal (unrealizable) all-pass filter with a phase shift of -90°. Observe that a cosine input becomes a sine and a sine input becomes a negative cosine. In MATLAB, generate a cosine and sine signal of some frequency and use the `hilbert` function to perform on each signal an approximation to the Hilbert Transform. (Why is the Hilbert Transform unrealizable?) The imaginary part of the Hilbert Transform output (*i.e.*,

[39]This content is available online at <http://cnx.org/content/m10484/2.13/>.

`imag(hilbert(signal))`) will be the -90° phase-shifted version of the original signal. Plot each signal to confirm your expectations.

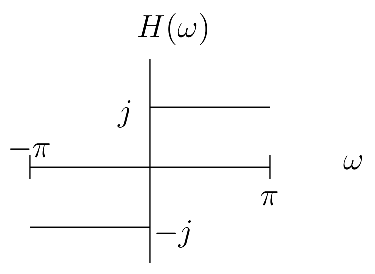


Figure 2.24: Hilbert transform transfer function

For the band-pass filter, design a second-order Butterworth filter using the `butter` function in MATLAB.

2.5.1.2.1 Generating a surround signal

Create four channels of audio to encode as a Pro Logic Surround Signal. Use simple mixing techniques to generate the four channels. For example, use a voice signal for the center channel and fade a roaming sound such as a helicopter from left to right and front to back. In MATLAB, use the `wavread` and `auread` functions to read `.wav` and `.au` audio files which can be found on the Internet.

2.5.1.3 Decoder

Implement the passive decoder shown in Figure 2.25 on the DSP. Use an appropriate time delay based on the distance between the front and back speakers and the speed of sound.

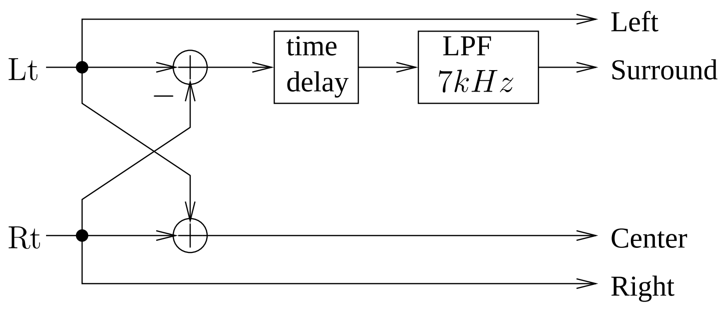


Figure 2.25: Dolby Pro Logic Passive Decoder

Is there significant crosstalk between the front and surround speakers? Do you get good separation between left and right speakers? Can you explain how the decoder recovers approximations to the original four channels?

2.5.1.4 Extensions

Differences in power levels between channels are used to enhance the directional effect in what is called "active decoding." One way to find the power level in a signal is to square it and pass the squared signal through a very narrow-band low-pass filter ($f \leq 80$Hz). How is the low-frequency content of the squared signal related to the power of the original signal? Remember that squaring a signal in the time domain is equivalent to convolving the signal with itself in the frequency domain.

To implement a very narrow-band low-pass filter, you may consider using the Chamberlin filter topology, described in Surround Sound: Chamberlin Filters (Section 2.5.2).

2.5.2 Surround Sound: Chamberlin Filters[40]

2.5.2.1 Introduction

Chamberlin filter topology is frequently used in music applications where very narrow-band, low-pass filters are necessary. Chamberlin implementations do not suffer from some stability problems that arise in direct-form implementations of very narrow-band responses. For more information about IIR/FIR filter design for DSPs, refer to the *Motorola Application Note* [8].

2.5.2.2 Filter Topology

A Chamberlin filter is a simple two-pole IIR filter with the transfer function given in (2.16):

$$H(z) = \frac{{F_z}^2 z^{-1}}{1 - \left(2 - \left(F_c Q_c - {F_c}^2\right)\right) z^{-1} - 1z^{-2}} \qquad (2.16)$$

where $F(c)$ determines the frequency where the filter peaks, and $Q_c\left(\frac{1}{Q}\right)$ determines the rolloff. **Q** is defined as the positive ratio of the center frequency to the bandwidth. A derivation and more detailed explanation is given in *Dattorro* [3]. The topology of the filter is shown in Figure 2.26. Note that the final feedback stage puts a pole just inside the unit circle on the real axis. For a response with smaller bandwidth, move the pole closer to the unit circle, but do not move it so far that the filter becomes unstable. Multiple second-order sections can be cascaded to yield a sharper rolloff.

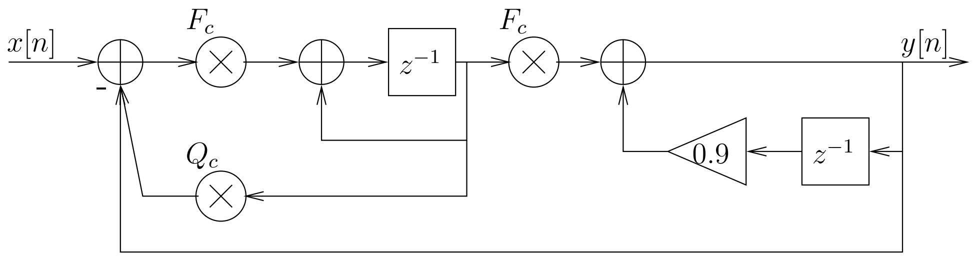


Figure 2.26: Chamberlin Filter Topology

[40]This content is available online at <http://cnx.org/content/m10479/2.15/>.

Figure 2.27 and Figure 2.28 show how the response of the filter varies with Q_c and F_c.

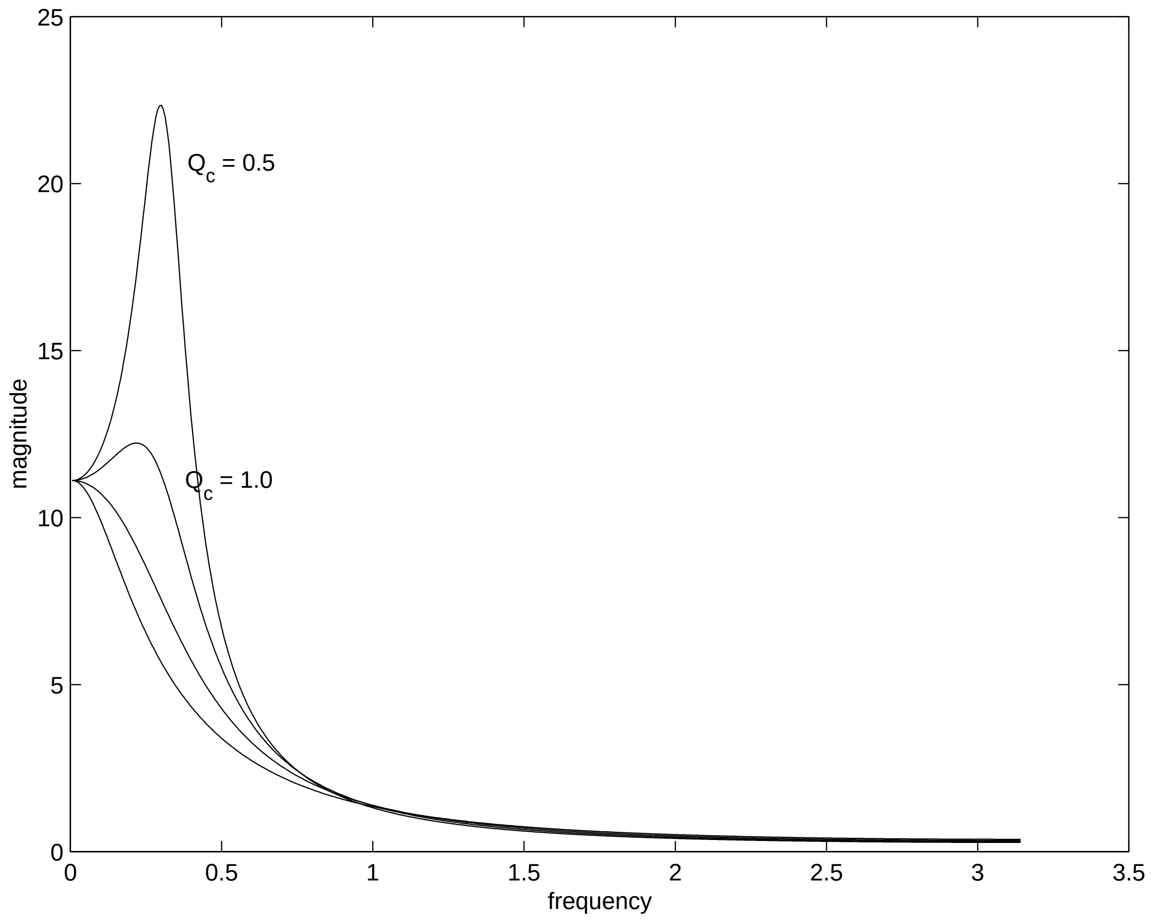


Figure 2.27: Chamberlin filter responses for various Q_c ($F_c = .3$)

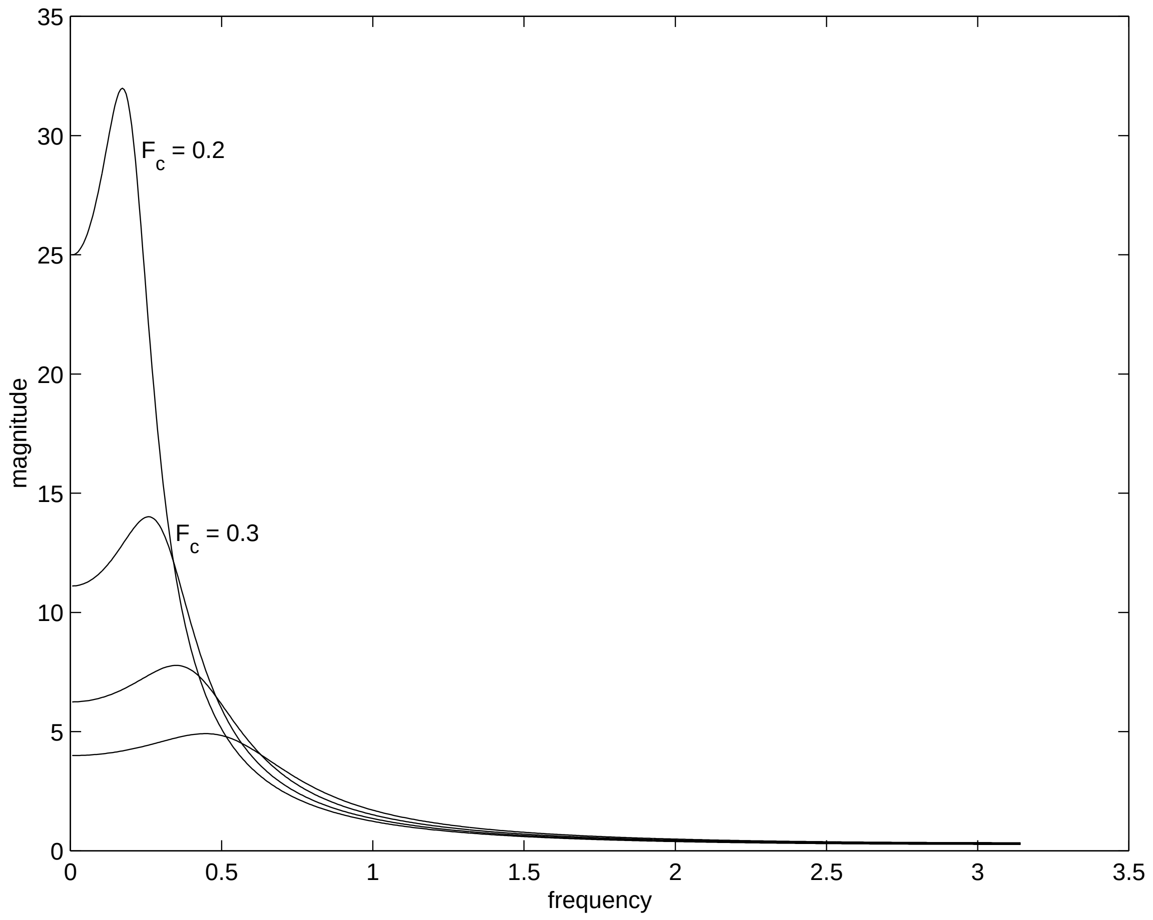


Figure 2.28: Chamberlin filter responses for various F_c ($Q_c = .8333$)

2.5.2.3 Exercise

First, create a MATLAB script that takes two parameters, Q_c and F_c, and plots the frequency response of a filter with a transfer function given in (2.16). Then implement a Chamberlin filter on the DSP and compare its performance with that of your MATLAB simulation for the same values of Q_c and F_c. What do you observe?

2.6 Speech

2.6.1 Speech Processing: LPC Exercise in MATLAB[41]

2.6.1.1 MATLAB Exercises

First, take a simple signal (*e.g.*, one period of a sinusoid at some frequency) and plot its autocorrelation sequence for appropriate values of l. You may wish to use the `xcorr` MATLAB function to compare with your own version of this function. At what time shift l is $r_{ss}[l]$ maximized and why? Is there any symmetry in $r_{ss}[l]$? What does $r_{ss}[l]$ look like for periodic signals?

Next, write your own version of the Levinson-Durbin algorithm in MATLAB. Note that MATLAB uses indexing from 1 rather than 0. One way to resolve this problem is to start the loop with $i = 2$, then shift the variables k, E, α, and r_{ss} to start at $i = 1$ and $j = 1$. Be careful with indices such as $i - j$, since these could still be 0.

Apply your algorithm to a 20- 30 ms segment of a speech signal. Use a microphone to record `.wav` audio files on the PC using Sound Recorder or a similar application. Typically, a sample rate of 8 kHz is a good choice for voice signals, which are approximately bandlimited to 4 kHz. You will use these audio files to test algorithms in MATLAB. The functions `wavread`, `wavwrite`, `sound` will help you read, write and play audio files in MATLAB:

The output of the algorithm is the prediction coefficients a_k (usually about $P = 10$ coefficients is sufficient), which represent the speech segment containing significantly more samples. The LPC coefficients are thus a compressed representation of the original speech segment, and we take advantage of this by saving or transmitting the LPC coefficients instead of the speech samples. Compare the coefficients generated by your function with those generated by the `levinson` or `lpc` functions available in the MATLAB toolbox. Next, plot the frequency response of the IIR model represented by the LPC coefficients (see Speech Processing: Theory of LPC Analysis and Synthesis (2.21)). What is the fundamental frequency of the speech segment? Is there any similarity in the prediction coefficients for different 20- 30 ms segments of the same vowel sound? How could the prediction coefficients be used for recognition?

2.6.2 Speech Processing: Theory of LPC Analysis and Synthesis[42]

2.6.2.1 Introduction

Linear predictive coding (**LPC**) is a popular technique for speech compression and speech synthesis. The theoretical foundations of both are described below.

2.6.2.1.1 Correlation coefficients

Correlation, a measure of similarity between two signals, is frequently used in the analysis of speech and other signals. The cross-correlation between two discrete-time signals $x[n]$ and $y[n]$ is defined as

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n]\, y[n-l] \tag{2.17}$$

where n is the sample index, and l is the lag or time shift between the two signals *Proakis and Manolakis* [9] (*pg. 120*). Since speech signals are not stationary, we are typically interested in the similarities between signals only over a short time duration (30 ms). In this case, the cross-correlation is computed only over a window of time samples and for only a few time delays $l = \{0, 1, \dots, P\}$.

Now consider the autocorrelation sequence $r_{ss}[l]$, which describes the redundancy in the signal $s[n]$.

$$r_{ss}[l] = \left(\frac{l}{N}\sum_{n=0}^{N-1} s[n]\, s[n-l]\right) \tag{2.18}$$

[41]This content is available online at <http://cnx.org/content/m10824/2.5/>.
[42]This content is available online at <http://cnx.org/content/m10482/2.19/>.

where $s[n]$, $n = \{-\mathrm{P}, -\mathrm{P}+1, \dots, N-1\}$ are the known samples (see Figure 2.29) and the $\frac{1}{N}$ is a normalizing factor.

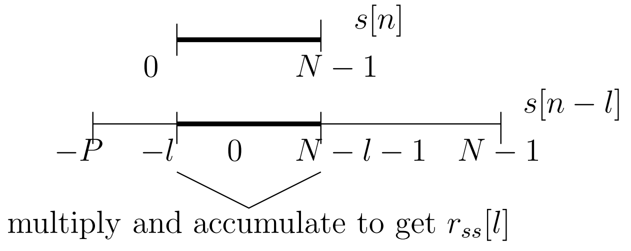


Figure 2.29: Computing the autocorrelation coefficients

Another related method of measuring the redundancy in a signal is to compute its autocovariance

$$r_{ss}[l] = \left(\frac{1}{N-1}\sum_{n=l}^{N-1} s[n]\,s[n-l]\right) \tag{2.19}$$

where the summation is over $N-l$ products (the samples $\{s[-\mathrm{P}], \dots, s[-1]\}$ are ignored).

2.6.2.1.2 Linear prediction model

Linear prediction is a good tool for analysis of speech signals. Linear prediction models the human vocal tract as an **infinite impulse response** (**IIR**) system that produces the speech signal. For vowel sounds and other voiced regions of speech, which have a resonant structure and high degree of similarity over time shifts that are multiples of their pitch period, this modeling produces an efficient representation of the sound. Figure 2.30 shows how the resonant structure of a vowel could be captured by an IIR system.

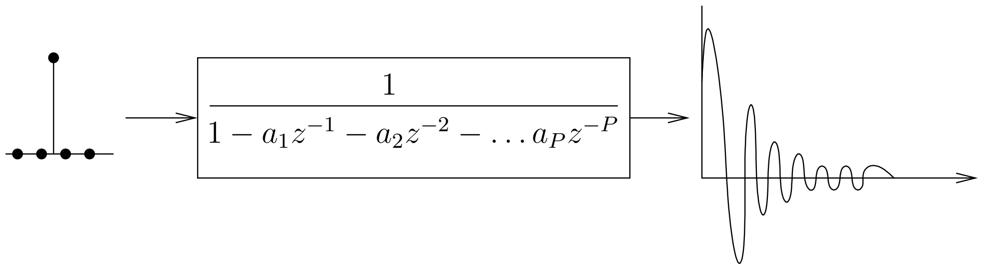


Figure 2.30: Linear Prediction (IIR) Model of Speech

The linear prediction problem can be stated as finding the coefficients a_k which result in the best prediction (which minimizes mean-squared prediction error) of the speech sample $s[n]$ in terms of the past samples $s[n-k]$, $k = \{1, \dots, P\}$. The predicted sample $\hat{s}[n]$ is then given by *Rabiner and Juang* [12]

$$\hat{s}[n] = \sum_{k=1}^{P} a_k s[n-k] \tag{2.20}$$

where P is the number of past samples of $s[n]$ which we wish to examine.

Next we derive the frequency response of the system in terms of the prediction coefficients a_k. In (2.20), when the predicted sample equals the actual signal (*i.e.*, $\hat{s}[n] = s[n]$), we have

$$s[n] = \sum_{k=1}^{P} a_k s[n-k]$$

$$s(z) = \sum_{k=1}^{P} a_k s(z) z^{-k}$$

$$s(z) = \frac{1}{1 - \sum_{k=1}^{P} a_k z^{-k}} \tag{2.21}$$

The optimal solution to this problem is *Rabiner and Juang* [12]

$$a = \begin{pmatrix} a_1 & a_2 & \dots & a_P \end{pmatrix}$$

$$r = \begin{pmatrix} r_{ss}[1] & r_{ss}[2] & \dots & r_{ss}[P] \end{pmatrix}^T$$

$$R = \begin{pmatrix} r_{ss}[0] & r_{ss}[1] & \dots & r_{ss}[P-1] \\ r_{ss}[1] & r_{ss}[0] & \dots & r_{ss}[P-2] \\ \vdots & \vdots & \vdots & \vdots \\ r_{ss}[P-1] & r_{ss}[P-2] & \dots & r_{ss}[0] \end{pmatrix}$$

$$a = R^{-1} r \tag{2.22}$$

Due to the Toeplitz property of the R matrix (it is symmetric with equal diagonal elements), an efficient algorithm is available for computing a without the computational expense of finding R^{-1}. The **Levinson-Durbin algorithm** is an iterative method of computing the predictor coefficients a *Rabiner and Juang* [12] (*p.115*).

Initial Step: $E_0 = r_{ss}[0]$, $i = 1$

for $i = 1$ to P.

Steps

1. $k_i = \frac{1}{E_{i-1}} \left(r_{ss}[i] - \sum_{j=1}^{i-1} \alpha_{j,i-1} r_{ss}[|i-j|] \right)$
2.
   - $\alpha_{j,i} = \alpha_{j,i-1} - k_i \alpha_{i-j,i-1}$ $j = \{1, \dots, i-1\}$
   - $\alpha_{i,i} = k_i$
3. $E_i = \left(1 - {k_i}^2\right) E_{i-1}$

2.6.2.1.3 LPC-based synthesis

It is possible to use the prediction coefficients to synthesize the original sound by applying $\delta[n]$, the unit impulse, to the IIR system with lattice coefficients k_i , $i = \{1, \ldots, P\}$ as shown in Figure 2.31. Applying $\delta[n]$ to consecutive IIR systems (which represent consecutive speech segments) yields a longer segment of synthesized speech.

In this application, lattice filters are used rather than direct-form filters since the lattice filter coefficients have magnitude less than one and, conveniently, are available directly as a result of the Levinson-Durbin algorithm. If a direct-form implementation is desired instead, the α coefficients must be factored into second-order stages with very small gains to yield a more stable implementation.

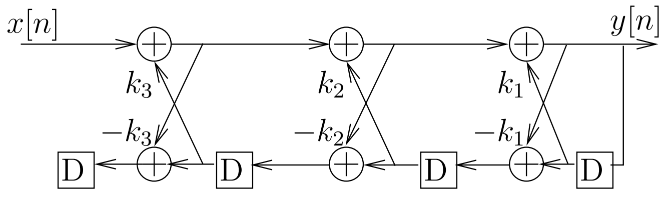


Figure 2.31: IIR lattice filter implementation.

When each segment of speech is synthesized in this manner, two problems occur. First, the synthesized speech is monotonous, containing no changes in pitch, because the $\delta[n]$'s, which represent pulses of air from the vocal chords, occur with fixed periodicity equal to the analysis segment length; in normal speech, we vary the frequency of air pulses from our vocal chords to change pitch. Second, the states of the lattice filter (*i.e.*, past samples stored in the delay boxes) are cleared at the beginning of each segment, causing discontinuity in the output.

To estimate the pitch, we look at the autocorrelation coefficients of each segment. A large peak in the autocorrelation coefficient at lag $l \neq 0$ implies the speech segment is periodic (or, more often, approximately periodic) with period l. In synthesizing these segments, we recreate the periodicity by using an impulse train as input and varying the delay between impulses according to the pitch period. If the speech segment does not have a large peak in the autocorrelation coefficients, then the segment is an unvoiced signal which has no periodicity. Unvoiced segments such as consonants are best reconstructed by using noise instead of an impulse train as input.

To reduce the discontinuity between segments, do not clear the states of the IIR model from one segment to the next. Instead, load the new set of reflection coefficients, k_i, and continue with the lattice filter computation.

2.6.2.2 Additional Issues

- Spanish vowels (m**o**p, **a**ce, **ea**sy, g**o**, b**u**t) are easier to recognize using LPC.
- Error can be computed as $a^T Ra$, where R is the autocovariance or autocorrelation matrix of a test segment and a is the vector of prediction coefficients of a template segment.
- A pre-emphasis filter before LPC, emphasizing frequencies of interest in the recognition or synthesis, can improve performance.
- The pitch period for males (80- 150 kHz) is different from the pitch period for females.
- For voiced segments, $\frac{r_{ss}[T]}{r_{ss}[0]} \simeq 0.25$, where T is the pitch period.

2.6.3 Speech Processing: LPC Exercise on TI TMS320C54x[43]

2.6.3.1 Implementation

The sample rate on the 6-channel DSP boards is fixed at 44.1 kHz, so decimate by a factor of 5 to achieve the sample rate of 8.82 kHz, which is more appropriate for speech processing.

Compute the autocorrelation or autocovariance coefficients of 256-sample blocks of input samples from a function generator for time shifts $l = \{0, 1, \dots, 15\}$ (*i.e.*, for $P = 15$) and display these on the oscilloscope with a trigger. (You may zero out the other 240 output samples to fill up the 256-sample block). For computing the autocorrelation, you will have to use memory to record the last 15 samples of the input due to the overlap between adjacent blocks. Compare the output on the oscilloscope with simulation results from MATLAB.

The next step is to use a speech signal as the input to your system. Use a microphone as input to the original thru6.asm[44] code and adjust the gains in your system until the output uses most of the dynamic range of the system without saturating. Now, to capture and analyze a small segment of speech, write code that determines the start of a speech signal in the microphone input, records a few seconds of speech, and computes the autocorrelation or autocovariance coefficients. The start of a speech signal can be determined by comparing the input to some noise threshold; experiment to find a good value. For recording large segments of speech, you may need to use external memory. Refer to Core File: Accessing External Memory on TI TMS320C54x[45] for more information.

Finally, incorporate your code which computes autocorrelation or autocovariance coefficients with the code which takes speech input and compare the results seen on the oscilloscope to those generated by MATLAB.

2.6.3.1.1 Integer division (optional)

In order to implement the Levinson-Durbin algorithm, you will need to use integer division to do Step 1 (p. 107) of the algorithm. Refer to the *Applications Guide*[?] and the `subc` instruction for a routine that performs integer division.

[43]This content is available online at <http://cnx.org/content/m10825/2.6/>.
[44]http://cnx.org/content/m10825/latest/thru6.asm
[45]"Core File: Accessing External Memory on TI TMS320C54x" <http://cnx.org/content/m10823/latest/>

Bibliography

[1] R. Blahut. *Digital Transmission of Information.* Addison-Wesley, 1990.

[2] R. Blahut. *Digital Transmission of Information.* Addison-Wesley, 1990.

[3] J. Dattorro. Effect design part 1: Reverberator and other filters. *Journal Audio Engineering Society,* vol. 45:660–684, September 1996.

[4] R. Dressler. Dolby prologic surround decoder principles of operation. http://www.dolby.com/tech/whtppr.html.

[5] K. Gundry. An introduction to noise reduction. http://www.dolby.com/ken/.

[6] S. Haykin. *Adaptive Filter Theory.* Prentice Hall, 3rd edition edition, 1996.

[7] II Leon W. Couch. *Digital and Analog Communication Systems.* Prentice Hall, Upper \\ Saddle River, New Jersey, 07458, 3rd edition edition, 1995.

[8] Motorola. *Implementing IIR/FIR Filters with Motorola's DSP56000/SPS/DSP56001, Digital Signal Processors.* http://merchant.hibbertco.com/mtrlext/fs22/pdf-docs/motorola/apr7.rev2.pdf.

[9] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications.* Prentice-Hall, Upper Saddle River, NJ, 1996.

[10] J.G. Proakis. *Digital Communications.* McGraw-Hill, 3rd edition edition, 1995.

[11] J.G. Proakis. *Digital Communications.* McGraw-Hill, 3rd edition edition, 1995.

[12] L. Rabiner and B. H. Juang. *Fundamentals of Speech Recognition.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *ECE 320 Spring 2004*
Edited by: Robert Morrison, Jason Laska
URL: http://cnx.org/content/col10225/1.12/
License: http://creativecommons.org/licenses/by/1.0

Module: "DSP Development Environment: Introductory Exercise for TI TMS320C54x (ECE 420 Specific)"
Used here as: "Lab 0: Hardware Introduction"
By: Mark Butala, Jason Laska
URL: http://cnx.org/content/m11019/2.7/
Pages: 3-9
Copyright: Mark Butala, Jason Laska
License: http://creativecommons.org/licenses/by/1.0

Module: "FIR Filtering: Basic Assembly Exercise for TI TMS320C54x"
Used here as: "Lab 1: Prelab"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade, Jason Laska
URL: http://cnx.org/content/m10022/2.22/
Pages: 11-13
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade, Jason Laska
License: http://creativecommons.org/licenses/by/1.0

Module: "FIR Filtering: Exercise for TI TMS320C54x (ECE 320 specific)"
Used here as: "Lab 1: Lab"
By: Mark Butala
URL: http://cnx.org/content/m11020/2.6/
Pages: 14-18
Copyright: Mark Butala
License: http://creativecommons.org/licenses/by/1.0

Module: "Multirate Filtering: Introduction"
Used here as: "Lab 2: Theory"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10024/2.21/
Page: 20
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Multirate Filtering: Theory Exercise"
Used here as: "Lab 2: Prelab (Part 1)"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10620/2.14/
Page: 21
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Multirate Filtering: Filter-Design Exercise in MATLAB"
Used here as: "Lab 2: Prelab (Part 2)"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10815/2.6/
Page: 22
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Multirate Filtering: Implementation on TI TMS320C54x"
Used here as: "Lab 2: Lab"
By: Robert Morrison
URL: http://cnx.org/content/m11810/1.3/
Page: 23
Copyright: Robert Morrison
License: http://creativecommons.org/licenses/by/1.0
Based on: Multirate Filtering: Implementation on TI TMS320C54x
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10621/2.8/

Module: "IIR Filtering: Introduction"
Used here as: "Lab 3: Theory"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10025/2.22/
Page: 25
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "IIR Filtering: Filter-Design Exercise in MATLAB"
Used here as: "Lab 3: Prelab (Part 1)"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10623/2.11/
Pages: 26-27
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "IIR Filtering: Filter-Coefficient Quantization Exercise in MATLAB"
Used here as: "Lab 3: Prelab (Part 2)"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10813/2.5/
Page: 28
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Surround Sound: Chamberlin Filters"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10479/2.15/
Pages: 102-104
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Speech Processing: LPC Exercise in MATLAB"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10824/2.5/
Page: 105
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Speech Processing: Theory of LPC Analysis and Synthesis"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10482/2.19/
Pages: 105-108
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

Module: "Speech Processing: LPC Exercise on TI TMS320C54x"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10825/2.6/
Pages: 108-109
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
License: http://creativecommons.org/licenses/by/1.0

ECE 320 Spring 2004
Development of real-time digital signal processing (DSP) systems using a DSP microprocessor; several structured laboratory exercises, such as sampling and digital filtering, followed by an extensive DSP project of the student's choice.