DSP Laboratory with TI TMS320C54x

Collection Editor:
Douglas L. Jones

DSP Laboratory with TI TMS320C54x

Collection Editor:
Douglas L. Jones

Authors:

Swaroop Appadwedula
Matthew Berry
Mark Haun
Jake Janovetz
Douglas L. Jones
Michael Kramer
Jason Laska
Dima Moussa
Daniel Sachs
Brian Wade

Online:
< http://cnx.org/content/col10078/1.2/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: January 22, 2004
PDF generated: October 25, 2012
For copyright and attribution information for the modules contained in this collection, see p. 107.

Table of Contents

Preface for U of I DSP Laboratory[1]

This text builds on over fourteen years of DSP laboratory instruction and over ten years of collaborative development of instructional laboratory materials. The content has evolved in tandem with ECE 320: Digital Signal Processing Laboratory, a senior-level, two-credit-hour elective laboratory course at the University of Illinois at Urbana-Champaign, and to a large extent reflects its goals and structure. The material is nonetheless well suited for a variety of course organizations, and earlier versions of the material have been used with success at the University of Washington and elsewhere.

This text could be effectively used with several types of course structures, including

- a semester-long project-oriented DSP laboratory,
- a quarter- or semester-long DSP laboratory structured around weekly laboratory exercises,
- a hands-on laboratory supplement as part of a signal processing theory course,
- a self-study course in DSP implementation.

ECE 320 at the University of Illinois represents the first type of course. It consists of roughly two equal parts: a series of weekly laboratory assignments, including introduction to the Texas Instruments TMS320C549 microprocessor and DSP development environment, real-time FIR, IIR, and multirate filtering, spectral analysis using the FFT, and a digital communications transmitter. Students work together in pairs on these laboratory assignments and are orally quizzed individually after completing each weekly laboratory assignment. The materials for each week are a semi-self-paced tutorial with three major parts: a review of the signal processing concepts, a design or familiarization exercise (often MATLAB-based), and a real-time implementation assignment using the TMS320C549 microprocessor. After completion of these common modules in mid-semester, student teams conceive of a substantial real-time DSP project of their choice and spend the remainder of the semester designing, simulating, implementing, and testing it. Supplementary modules introducing students to the basics of digital communication (including phase-locked loops and delay-locked loops), adaptive filtering, speech processing, and audio signal processing accelerate students' progress on projects in these areas.

A course emphasizing signal processing algorithms might forgo a major project and instead use the supplementary modules to complete a quarter or semester of weekly laboratory assignments. A one-hour hands-on laboratory supplement to a signal processing lecture course could stretch the first few units (*e.g.*, through spectral analysis) over a semester, thereby reinforcing and enhancing students' understanding of the core signal processing theory and algorithms. Due to the self-paced, tutorial nature of the materials, a student can independently learn the aspects of real-time DSP implementation that interest them; students in our senior independent design course at the University of Illinois have successfully used the materials in this manner.

The laboratory materials and assignments reflect our belief that a thorough instruction in signal processing implementation requires exposure to assembly-language programming of fixed-point DSP microprocessors, as this represents an important component of current and at least near-future industrial practice. Instructors with other goals or perspectives may find most of the tutorial, design material, and assignments relevant even if they choose compiler-based or non-real-time implementation. Laboratories using different development systems or different DSP microprocessors will likely find almost all of the material well suited

[1] This content is available online at <http://cnx.org/content/m10681/2.12/>.

for their needs; only the hardware-specific language and instructions need be modified. Earlier versions of this material have been used with several different DSP microprocessors and development boards based on the Motorola DSP56000 and the Texas Instruments TMS320 families.

Connexions is an ideal venue for this text for several reasons. DSP hardware and development tools are evolving very rapidly, so a textbook produced through conventional publishers is likely to be almost obsolete before it is printed. Every university has a unique set of equipment, curriculum, and students, necessitating site-specific specialization of laboratory instructional material; conventional publishing is unable to produce textbooks cost-effectively with the rapid turnaround and low volumes thus required. We have always made our materials open, available, and free to other institutions to use in their own laboratory course development, so the open-source spirit of the Connexions project reflects our own philosophy and should more easily enable others to build on our experience. Finally, this material was created, modified, rewritten, and enhanced by a large and changing group of authors over a period of years in response to new ideas and evolving needs, goals, and equipment; its development thus embodies the Connexions philosophy.

The development of these materials would not have been possible without the active support and encouragement of many people and organizations. First, we express our gratitude to the corporations, particularly Texas Instruments, Motorola, and Hewlett-Packard/Agilent, whose generosity has equipped our instructional laboratory with state-of-the-art DSP development systems and instruments; our laboratory course would not be possible without their support. It would also have been impossible without the active support of the departmental leadership and the staff of the Electrical and Computer Engineering department, and particularly Dan Mast, for supporting, designing, equipping, and maintaining our instructional laboratory. We thank the Connexions team for their very substantial help in "connexifying" our materials, including conversion of the majority of the material into CNXML and MathML format; without their efforts, the text in this form would not exist. Support from the National Science Foundation in recent years enables continuing development of the course in response to student and industry needs. Most importantly, we are grateful to the generations of teaching assistants and students who have taught and learned from these materials over the past decade or more; it is their hard work, creative input, and dynamic interaction that have yielded this result.

Chapter 1

Required 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/m10017/2.24/>.

Example Hardware Setup

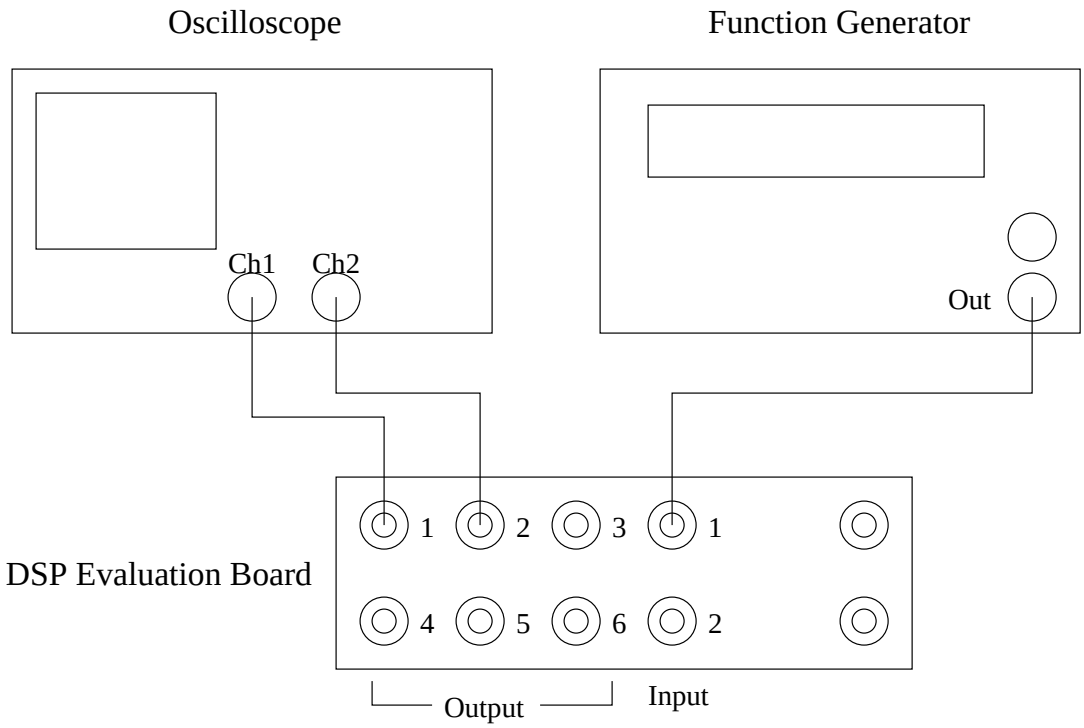

Figure 1.1

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.

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 (Section 1.1.1.3.1: Step 3: Assemble filter code)), then verify the filter's frequency response with the subsequent MATLAB exercises (Steps 6 and 7 (Section 1.1.1.3.4: Step 6: Check filter response in MATLAB)).

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, then copy filter.asm[2], coef1.asm[3], coef2.asm[4], core.asm[5], and vectcore.asm[6] into your directory.

Next, make a copy of `coef1.asm` called "coef.asm" and 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. The different coefficients are in the file `coef2.asm`. Make a copy of `coef2.asm` and call it `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).

[2]See the file at <http://cnx.org/content/m10017/latest/filter.asm>
[3]See the file at <http://cnx.org/content/m10017/latest/coef1.asm>
[4]See the file at <http://cnx.org/content/m10017/latest/coef2.asm>
[5]See the file at <http://cnx.org/content/m10017/latest/core.asm>
[6]See the file at <http://cnx.org/content/m10017/latest/vectcore.asm>

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 we want to view the response of the three-tap filter with coefficients -10, 20, -10 we 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[7] and save_coef.m[8].)

The 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.

[7] See the file at <http://cnx.org/content/m10017/latest/gen_filt.m>
[8] See the file at <http://cnx.org/content/m10017/latest/save_coef.m>

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 filter

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

creating a scaled and delayed version of the input. 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 1/4. More information on the DSP's interpretation of numbers appears in Two's Complement and Fractional Arithmetic for 16-bit Processors (Section 3.1.1).

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 `sweep` (available as sweep.m[9]) to generate a sinusoid that sweeps across a range of frequencies. The MATLAB function `save_test_vector` (available as save_test_vector.m[10] 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:

```
≫ 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[11]. Rather than

[9] See the file at <http://cnx.org/content/m10017/latest/sweep.m>
[10] See the file at <http://cnx.org/content/m10017/latest/save_test_vector.m>
[11] See the file at <http://cnx.org/content/m10017/latest/vectcore.asm>

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 (Section 3.2.1) 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  "core.asm"
```

use:

```
.copy  "testvect.asm"
.copy "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 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[12]) function to read the saved result into MATLAB. Do this using the following MATLAB command:

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

[12] See the file at <http://cnx.org/content/m10017/latest/read_vector.m>

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.

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-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.

Note that to compare two vectors in this way, the two vectors must be exactly the same length, which is ensured after using the MATLAB command `out=out(1:500)` above.

1.2 Lab 1

1.2.1 Lab 1: Prelab[13]

1.2.1.1 Assembly Exercise

Analyze the following lines of code. Refer to Two's Complement and Fractional Arithmetic for 16-bit Processors (Section 3.1.1), Addressing Modes for TI TMS320C54x (Section 3.1.2), and the *Mnemonic Instruction Set*[?] manual for help.

```
1  FIR_len .set    3
2
3  ; Assume:
4  ;   BK = FIR_len
5  ;   AR0 = 1
6  ;   AR2 = 1000h
7  ;   AR3 = 1004h
8  ;
9  ;   FRCT = 1
10
11      stl     A,*AR3+%
12      rptz    A,(FIR_len-1)
13      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.

[13] This content is available online at <http://cnx.org/content/m10022/2.22/>.

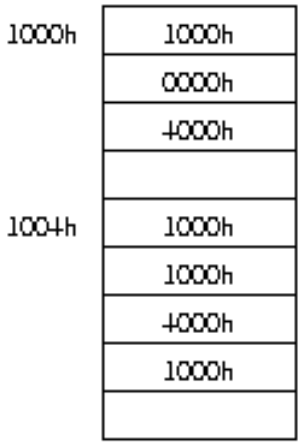

Figure 1.2: Data Memory Assignment (before execution)

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 (Section 3.1.1).

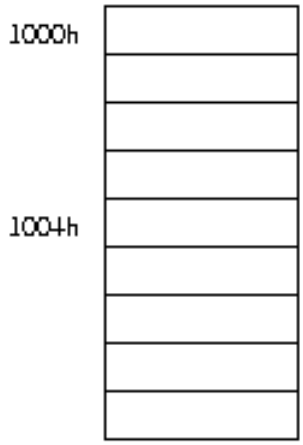

Figure 1.4: Data Memory Assignment (after execution)

1.2.2 Lab 1: Lab[14]

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[15] .

[14] This content is available online at <http://cnx.org/content/m10023/2.19/>.
[15] http://cnx.rice.edu/content/m10017/latest/filter.asm

filter.asm

```
 1 .copy "core.asm"   ; Copy in core file
2    ; This initializes DSP and jumps to "main"
3
4  FIR_len .set   8 ; This is an 8-tap filter.
5
6           .sect ".data" ; Flag following as data declarations
7
8         .align 16 ; Align to a multiple of 16
9  coef ; assign label "coeff"
10     .copy "coef.asm" ; Copy in coefficients
11
12     .align 16
13 firstate
14      .space 16*8 ; Allocate 8 words of storage for
15 ; filter state.
16
17     .sect ".text" ; Flag the following as program code
18  main
19      ; Initialize various pointers
20     stm    #FIR_len,BK ; initialize circular buffer length
21     stm    #coef,AR2     ; initialize coefficient pointer
22     stm    #firstate,AR3 ; initialize state pointer
23     stm    #1,AR0 ; initialize AR0 for pointer increment
24
25  loop
26      ; Wait for a new block of 64 samples to come in
27       WAITDATA
28
29      ; BlockLen = the number of samples that come from WAITDATA (64)
30         stm     #BlockLen-1, BRC ; Put repeat count into repeat counter
31         rptb    endblock-1 ; Repeat between here and 'endblock'
32
33         ld     *AR6,16, A ; Receive ch1 into A accumulator
34         mar     *+AR6(2)            ; Rcv data is in every other channel
35         ld      *AR6,16, B ; Receive ch2 into B accumulator
36         mar     *+AR6(2)            ; Rcv data is in every other channel
37
38         ld     A,B ; Transfer A into B for safekeeping
39
40      ; The following code executes a single FIR filter.
41
42         sth     A,*AR3+% ; store current input into state buffer
43         rptz    A,(FIR_len-1) ; clear A and repeat
44         mac     *AR2+0%,*AR3+0%,A ; multiply coef. by state & accumulate
45
46         rnd     A ; Round off value in 'A' to 16 bits
47
48     ; end of FIR filter code. Output is in the high part of 'A.'
49
50         sth     A, *AR7+ ; Store filter output (from A) into ch1
51         sth     B, *AR7+ ; Store saved input (from B) into ch2
52
53         sth     B, *AR7+ ; Store saved input to ch3...ch6 also
54         sth     B, *AR7+ ; ch4
```

`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[16] and core.asm[17] into it. 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:

```
1         .align 32             % Align to a multiple of 32
2  coef   .copy  "filter1.asm"  % Copy FIR filter coefficients
3
4         .align 32             % Align to a multiple of 32
5  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

[16] http://cnx.rice.edu/content/m10017/latest/filter.asm
[17] http://cnx.rice.edu/content/m10017/latest/core.asm

`.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
Challenge Problem
Can you 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 (Section 1.1.1), 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 (Section 1.1.1), 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).

```
1 mvdd *AR2,*AR3+0% ; write x(-N/2) over x(-N)
2 sth A,*AR2 ; write x(0) over x(-N/2)
3 add *AR2+0%,*AR3+0%,A  ; add x(0) and x(-(N-1))
4 ;   (prepare for first multiply)
5
6 rptz B,#(FIR_len/2-1)
7 firs *AR2+0%,*AR3+0%,COEFF
8 mar ???????  ; Fill in these two instructions
9 mar ??????? ; They modify AR2 and AR3.
10
11 ; note that the result is now in the
12 ;  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

```
1 stm #(FIR_len/2),BK ; initialize circular buffer length
2 stm #firstate_,AR2 ; initialize location containing first
3 ;   half of states
4
5 stm #-1,AR0 ; Initialize AR0 to -1
6
7 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.

1.3 Lab 2

1.3.1 Lab 2: Theory[18]

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 $\frac{F_s}{D}$.

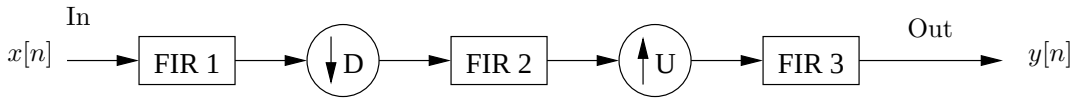

Figure 1.6: Net multirate system

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?

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

1.3.2 Lab 2: Prelab (part 1)[19]

1.3.2.1 Multirate Theory Exercise

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

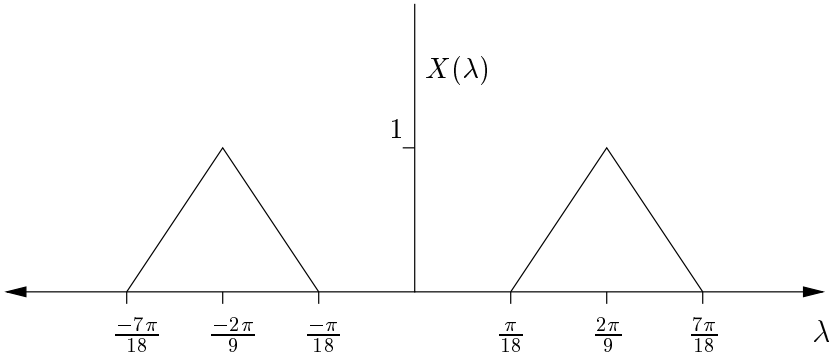

Figure 1.7: DTFT of the input signal.

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 $W(\omega)$.

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

$$Y(\omega) = W(U\omega) \tag{1.3}$$

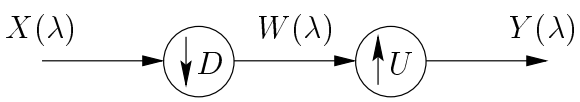

Figure 1.8: Multirate System

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

1.3.3 Lab 2: Prelab (part 2)[20]

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?

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

1.3.4 Lab 2: Lab[21]

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 $D = U = 4$. Later, you control this factor using a MATLAB interface; be sure to keep this in mind as you write 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

A simple graphical user interface (GUI) is available (as mrategui.m[22] , which requires ser_snd.m[23]) 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 that you have been given 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 (Section 3.2.3).

[21] This content is available online at <http://cnx.org/content/m10621/2.9/>.
[22] http://cnx.org/content/m10621/latest/mrategui.m
[23] http://cnx.org/content/m10621/latest/ser_snd.m

1.4 Lab 3

1.4.1 Lab 3: Theory[24]

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 $w[n]$.

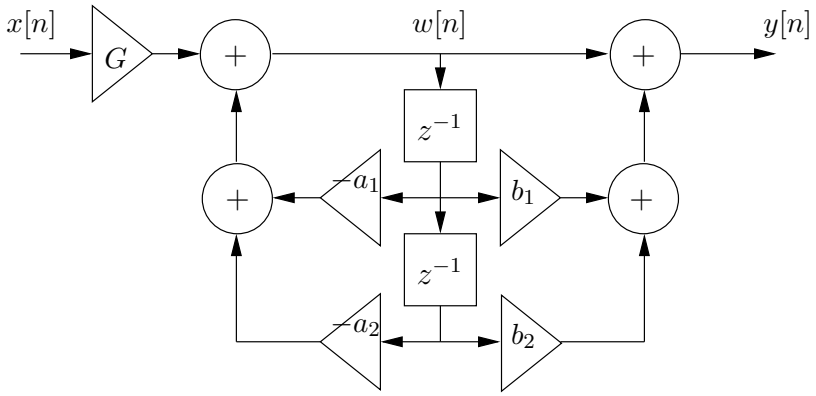

Figure 1.9: Second-order, Direct Form II section

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

1.4.2 Lab 3: Prelab (part 1)[25]

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,

[25] 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 (Section 1.1.1) 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).

1.4.3 Lab 3: Prelab (part 2)[26]

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.

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

1.4.4 Lab 3: Lab[27]

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.

[27] This content is available online at <http://cnx.org/content/m10624/2.8/>.

1.5 Lab 4

1.5.1 Lab 4: Theory[28]

1.5.1.1 Introduction

The **Fast Fourier Transform** (**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**.

1.5.1.1.1 DFT

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

where n *and* $k \in \{0, 1, \ldots, N-1\}$

Because the FFT is a block-based algorithm, its computation is performed at the block I/O rate, in contrast to other exercises in which processing occurred on a sample-by-sample basis.

[28] This content is available online at <http://cnx.org/content/m10027/2.19/>.

1.5.2 Lab 4: Prelab[29]

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.

```
1  N = 256;                % length of test signals
2  num_freqs = 100;        % number of frequencies to test
3
4  % Generate vector of frequencies to test
5
6  omega = pi/8 + [0:num_freqs-1]'/num_freqs*pi/4;
7
8  S = zeros(N,num_freqs);                % matrix to hold FFT results
9
10
11  for i=1:length(omega)                  % loop through freq. vector
12     s = sin(omega(i)*[0:N-1]');         % generate test sine wave
13     win = boxcar(N);                    % use rectangular window
14     s = s.*win;                         % multiply input by window
15     S(:,i) = (abs(fft(s))).^2;          % generate magnitude of FFT
16                                         % and store as a column of S
17  end
18
19  clf;
20  plot(S);                               % plot all spectra on same graph
21
```

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 $x[n]$?

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

1.5.3 Lab 4: Lab[30]

1.5.3.1 Implementation

You will use the FFT to compute the spectrum of a windowed input. For your implementation, use a 64-point Hamming window. You may use the MATLAB function `save_coef` (available as save_coef.m[31] to save the window to a file that you can then include in your code with the `.copy` directive.

1.5.3.1.1 FFT usage

The FFT routine `fft.asm` 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 a 64-point FFT, you will have to include the following label definitions in your code.

```
K_FFT_SIZE .set 64 ; size of FFT
K_LOGN .set 6 ; number of stages (log_2(N))
```

In addition to defining these constants, you will have to include **twiddle-factor** tables for the FFT. Copy these tables (`twiddle1` and `twiddle2`) into your work directory. Note that the tables are each 512 points long, representing values from 0 to just shy of 180 degrees, and must be accessed as a circular buffer. 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 "twiddle1"
.align 1024
cosine .copy "twiddle2"
```

To run the FFT code, use the instruction `call fft` where `fft` is a label at the beginning of the available `fft.asm` code.

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 table (Table 1.1) below illustrates the bit-reversed order for an eight-point sequence.

[30] This content is available online at <http://cnx.org/content/m10626/2.8/>.
[31] http://cnx.rice.edu/author/workgroups/90/m10017/save_coef.m

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 `input_data` and consists of alternating real and imaginary parts. Because our input data in this exercise is purely real, you will have to set the imaginary parts to zero by zeroing out every other memory location.

```
1 bit_rev:
2         SSBX    FRCT                          ; fractional mode is on
3         STM     #input_data,AR3               ; AR3 -> 1 st original input
4         STM     #fft_data,AR7                 ; AR7 -> data processing buffer
5         MVMM    AR7,AR2                       ; AR2 -> 1st bit1 reversed data
6         STM     #K_FFT_SIZE-1,BRC
7         RPTBD   bit_rev_end-1
8         STM     #K_FFT_SIZE,AR0               ; AR0 = 1/2 size of circ buffer
9         MVDD    *AR3+,*AR2+
10         MVDD    *AR3-,*AR2+
11         MAR     *AR3+0B
12 bit_rev_end:
13         RET                                    ; return to Real FFT main module
```

As mentioned, in the above code `input_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. Include the `bit_rev` routine in your code and call it using the `call bit_rev` command in the appropriate location. 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 *CPU and Peripherals*[?] manual. See *Figure 4-10* in the *Applications Guide*[?] to view the ordering of the data expected by the FFT routine.

NOTE: The FFT code uses all the pointers available. Additionally, it does not restore the pointers it uses to their original values, so you will have to re-initialize any pointers you are using after the `fft` call.

1.5.3.1.2 Displaying the spectrum

Once the DFT has been computed, 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). Why do we display the squared-magnitude instead of the magnitude itself?

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.

1.6 Lab 5

1.6.1 Lab 5: Theory[32]

1.6.1.1 Introduction

The **quadrature phase-shift keying** (**QPSK**) digital transmitter of Figure 1.10 (QPSK Transmitter) is one of many DSP systems used in the communications industry. The following sections describe the transmitter in detail.

1.6.1.1.1 Quadrature phase shift keying (QPSK)

QPSK is a method for transmitting digital information across an analog channel. Data bits are grouped into pairs, and each pair is represented by a particular waveform, called a symbol, to be sent across the channel after modulating the carrier. (The receiver will demodulate the signal and look at the recovered symbol to determine which pair of bits was sent.) This requires having a unique symbol for each possible combination of data bits in a pair. Because there are four possible combinations of data bits in a pair, QPSK creates four different symbols, one for each pair, by changing the I gain and Q gain for the cosine and sine modulators in Figure 1.10 (QPSK Transmitter). To transmit each pair of bits in the source data, the gains are kept constant over a fixed number of output samples known as the **symbol period**, T_{symb}. The **symbol rate**, F_{symb}, is a fraction of the board's sample rate, F_s. For our sample rate of 44.1 kHz and a symbol period of 16, the symbol rate is $F_{\text{symb}} = \frac{44100}{16}$ symbols per second.

The QPSK transmitter system uses both the sine and cosine at the carrier frequency to transmit two separate message signals, $s_I[n]$ and $s_Q[n]$, referred to as the **in-phase** and **quadrature** signals. Provided that a coherent receiver system is employed, both the in-phase and quadrature signals can be recovered exactly, allowing us to transmit twice the amount of signal information at the same carrier frequency as we could with a single oscillator.

QPSK Transmitter

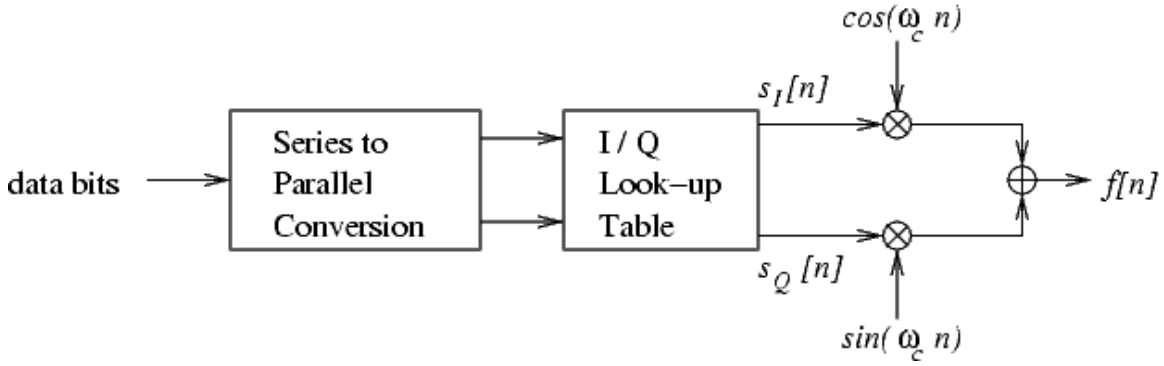

Figure 1.10

1.6.1.1.2 Pseudo-noise generation

The input bits to the transmitter are provided by a special shift-register, called a **pseudo-noise generator** (**PN generator**), in Figure 1.11 (Pseudo-Noise Generator). A PN generator produces a sequence of bits

[32] This content is available online at <http://cnx.org/content/m10042/2.19/>.

that appears random. The PN sequence will repeat with period $2^B - 1$, where B is the width in bits of the shift register.

As shown in Figure 1.11 (Pseudo-Noise Generator), the PN generator is simply a shift-register and XOR gate. Bits 1, 5, 6, and 7 of the shift-register are XORed together and the result is shifted into the highest bit of the register. The lowest bit, which is shifted out, is the output of the PN generator.

The PN generator is a useful source of random data bits for system testing. We can use the output of a PN generator as a "typical" sequence that could be transmitted by a user. The sequence is a good data model because communications systems tend to randomize the bits transmitted for efficient use of bandwidth. PN generators have other applications in communications, notably in the Code Division Multiple Access schemes used by cellular telephones.

Pseudo-Noise Generator

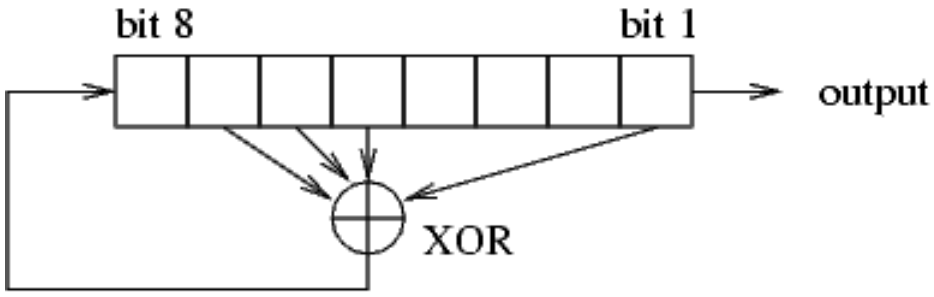

Figure 1.11

1.6.1.1.3 Series-to-parallel conversion

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

1.6.1.1.4 I/Q look-up table

This block is responsible for mapping pairs of bits to in-phase and quadrature gains. Such a mapping is often described by a signal constellation. Figure 1.12 (QPSK Constellation) shows the data mapping constellation for the QPSK system. In this case the data are grouped into pairs and each pair maps to a separate in-phase (I) and quadrature (Q) gain. These I and Q gains are then used to generate the in-phase and quadrature message signals, $s_I[n]$ and $s_Q[n]$.

QPSK Constellation

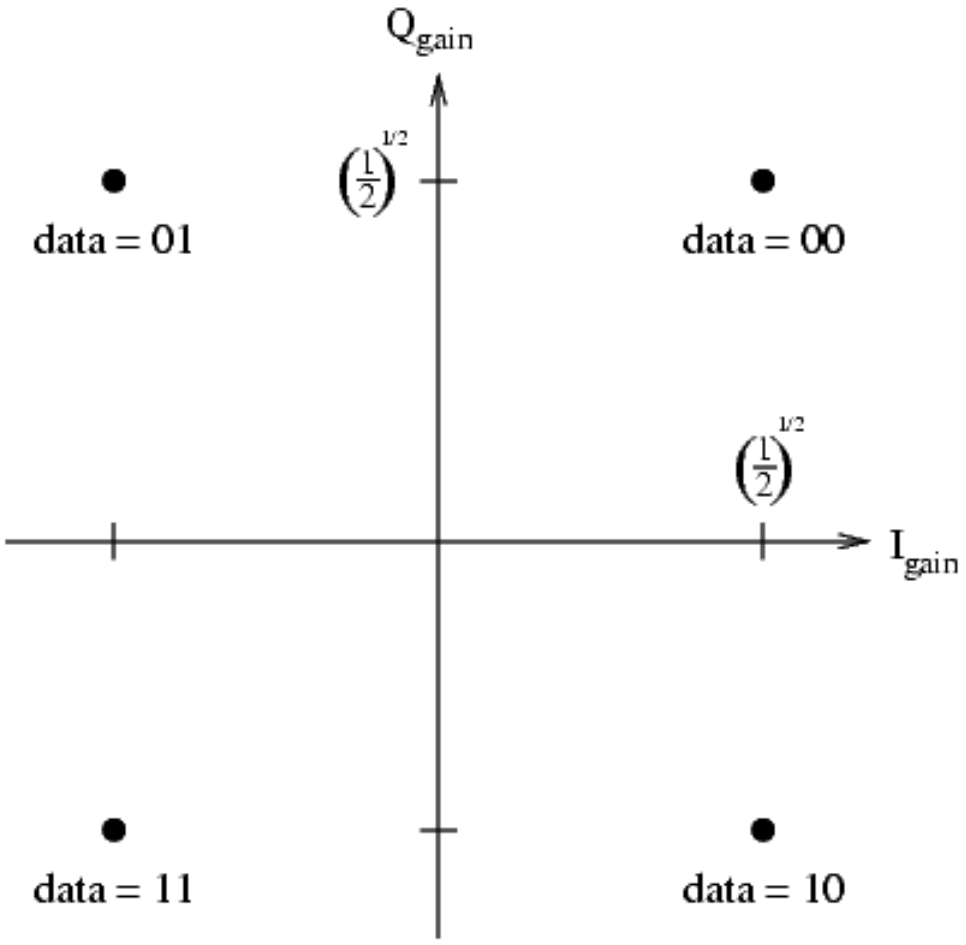

Figure 1.12

One way to implement this mapping is by using a look-up table. A pair of data bits can be interpreted as an offset into an $I/\ Q$ table that stores the in-phase and quadrature gains. Note that since each $I/\ Q$ mapping defines a symbol, this mapping is done at the symbol rate F_{symb}, or once for every T_{symb} DSP samples. [33]

The constellation bit-assignments are such that any two adjacent constellation points 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.

[33] The I and Q gains of $\pm\left(\frac{1}{\sqrt{2}}\right)$ have been chosen to ensure that the magnitude of the transmitted signal never exceeds 1.0.

1.6.2 Lab 5: Prelab[34]

1.6.2.1 MATLAB Simulation

MATLAB is commonly used to design filters and determine frequency responses of systems, but it is also very useful as a simulation tool.

Use the following MATLAB code skeleton to simulate the QPSK transmitter from Digital Transmitter: Introduction to Quadrature Phase-Shift Keying (Figure 1.10: QPSK Transmitter) 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}? How do you interpret the figure created by `plot(rI,rQ)`?

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB Code Skeleton for QPSK Digital Transmitter

% Generate random bits
bits_per_symbol=2;
num_symbols=128;
numbits=bits_per_symbol*num_symbols;
bits=rand(1,numbits)>0.5;

Tsymb = 16;                     % symbol length
omega = pi/2;                   % carrier frequency

%%%%%%%%%%%%%%%%%%%%%%%%
% Transmitter section
                              % initialize transmit sequence
t = zeros(1,num_symbols*Tsymb);
i = 1;                          % initialize bit index
n = 1;                          % initialize time index

while (n <= num_symbols*Tsymb)
 if ( bits(i:i+1) == [ 0 0])
     Igain = 1/sqrt(2);
     Qgain = 1/sqrt(2);
 % ------>Insert code here<-------

 end;
 i = i+2;                       % next 2 bits

 % Generate symbol to be transmitted
 t(n:n+Tsymb-1) =    %------>Insert code here<-------

 n = n+Tsymb;                   % next symbol
end;

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

% Show the transmitted signal constellation
```

[34] This content is available online at <http://cnx.org/content/m10627/2.9/>.

```
39  rI = t.*cos(omega*[1:num_symbols*Tsymb]);
40  rQ = t.*sin(omega*[1:num_symbols*Tsymb]);
41
42  % Filter out the double-frequency term
43  low_pass=fir1(512,0.5);
44  rI=conv(rI,low_pass);
45  rQ=conv(rQ,low_pass);
46  figure;
47  plot(rI,rQ);
```

1.6.3 Lab 5: Lab[35]

1.6.3.1 Implementation

You will implement the complete system shown in Digital Transmitter: Introduction to Quadrature Phase-Shift Keying (Figure 1.10: QPSK Transmitter). Then you will optimize the system for execution time. The optimization process will probably be much easier if you plan for optimization before you begin any programming.

1.6.3.1.1 PN generator

Once you have planned your program strategy, implement the PN generator from Digital Transmitter: Introduction to Quadrature Phase-Shift Keying (Figure 1.11: Pseudo-Noise Generator) and verify that it is working. You may wish to refer to the description of assembly instructions for logical operations in *Section 2-2* of the *Mnemonic Instruction Set*[**?**] reference. Initialize the shift register to one.

1.6.3.1.2 Transmitter

For your transmitter implementation, use the signal constellation shown in Digital Transmitter: Introduction to Quadrature Phase-Shift Keying, a digital carrier frequency ω_c of $\frac{\pi}{2}$ and a digital symbol period of $T_{\text{symb}} = 16$ 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. A **vector signal analyzer** (**VSA**) provides another method of testing, which is described in Vector Signal Analyzer: Testing a QPSK Transmitter (Section 1.6.4).

1.6.3.1.3 Grading

One objective of this exercise is to teach optimization and efficient code techniques. For this reason, your performance will be judged primarily on the total execution time of your system. Note that by execution time we mean cycle count, not the number of instructions in your program. Remember that several of the TI 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. The *Mnemonic Instruction Set*[**?**] reference will tell you how many cycles required for each instruction; make sure you look at the cycle count for the syntax you are using. It is also possible to use the debugger to determine the number of cycles used by your code.

You will be graded based on the number of cycles used between the return from one `WAITDATA` call and the arrival at the next `WAITDATA` call. If the number of cycles between one `WAITDATA` and the next is variable, the maximum possible number of cycles will be used. You must use the core.asm[36] file as provided; this file may not be modified. You explicitly may not change the number of samples read and written by each `WAITDATA` call! We reserve the right to test your code by substituting the test vector core file[37] .

[35] This content is available online at <http://cnx.org/content/m10628/2.8/>.
[36] http://cnx.rice.edu/author/workgroups/90/m10017/core.asm
[37] http://cnx.rice.edu/author/workgroups/90/m10017/vectcore.asm

1.6.4 Lab 5: Testing[38]

1.6.4.1 Introduction

The **vector signal analyzer** (**VSA**) is an instrument capable of demodulating digital signals. You may use the VSA to demodulate a QPSK signal and display the symbol constellation. The VSA will also display the signal spectrum.

Plug the output from the DSP board into the "Channel 1" jack on the front of the vector signal analyzer and turn on the analyzer. The next two sections explain how to set up the VSA to display the constellation or the signal spectrum.

1.6.4.1.1 Displaying the constellation

To display your QPSK constellation, enter the following key sequence:

- "`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)

1.6.4.1.2 Displaying the signal spectrum

The VSA is capable of displaying the spectrum of a signal. Connect the output of your transmitter output to the VSA. Enter "`Instrument Mode`" and then F1 (Scalar) to see the spectrum.

[38] This content is available online at <http://cnx.org/content/m10667/2.6/>.

Chapter 2

Project Labs

2.1 Digital Receiver

2.1.1 Digital Receivers: Symbol-Timing Recovery for QPSK[1]

2.1.1.1 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 2.1.2)) 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 2.1 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 2.1.2). 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.

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

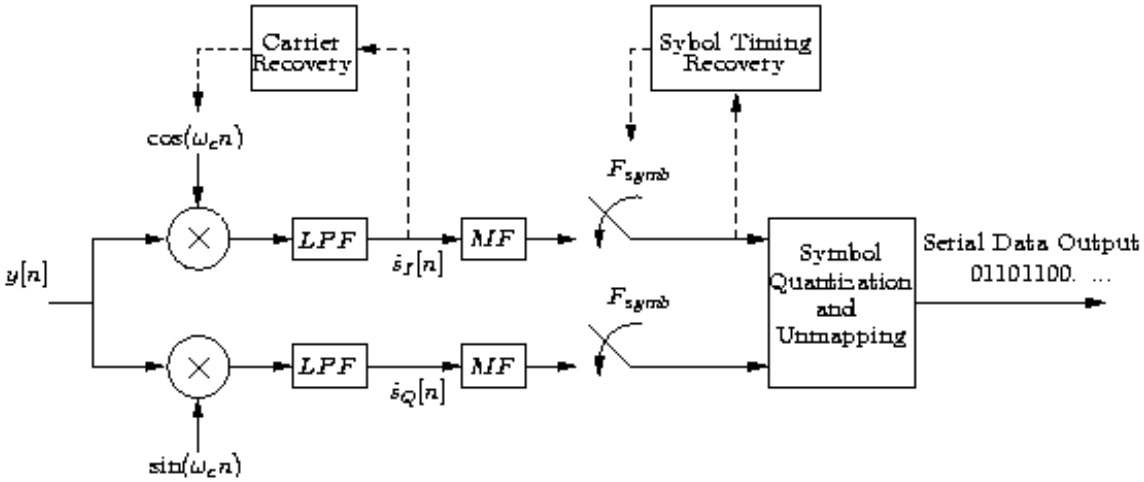

Figure 2.1: Digital receiver system

If we consider the square wave shown in Figure 2.2 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.

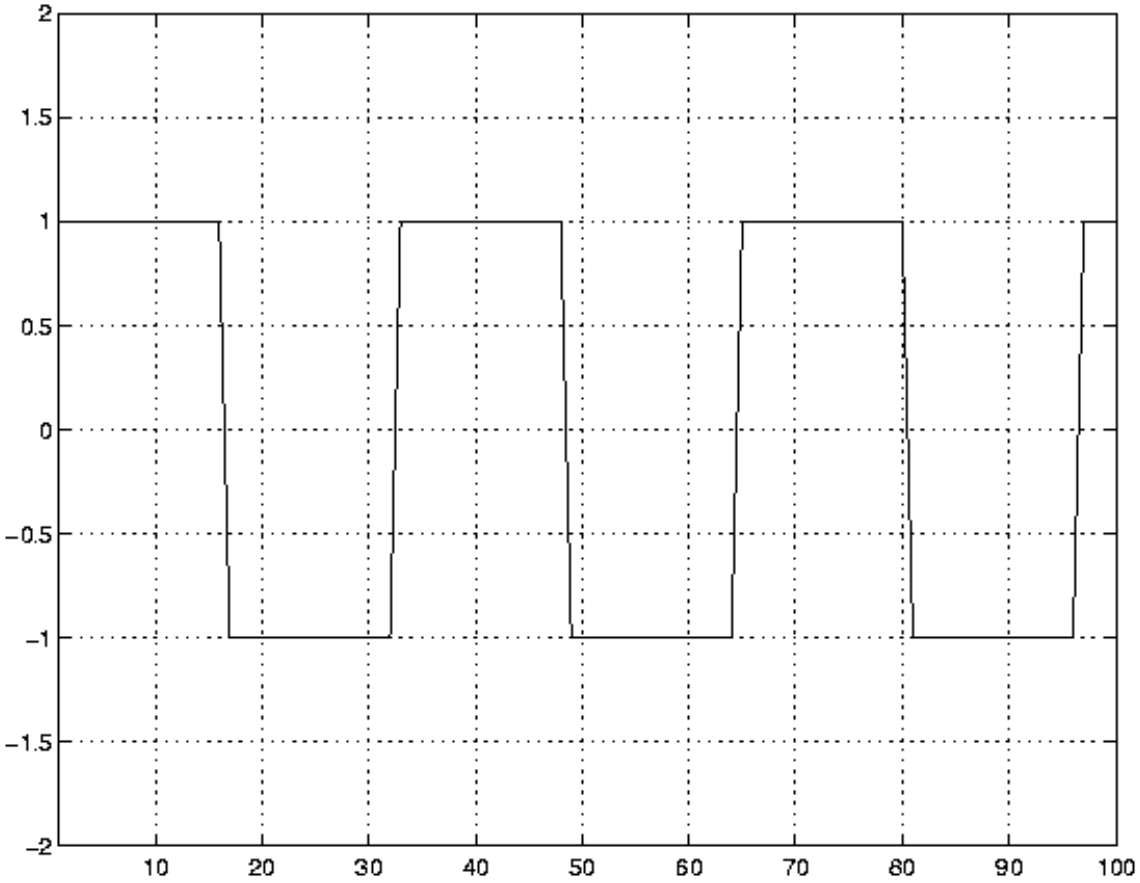

Figure 2.2: Clean BPSK waveform.

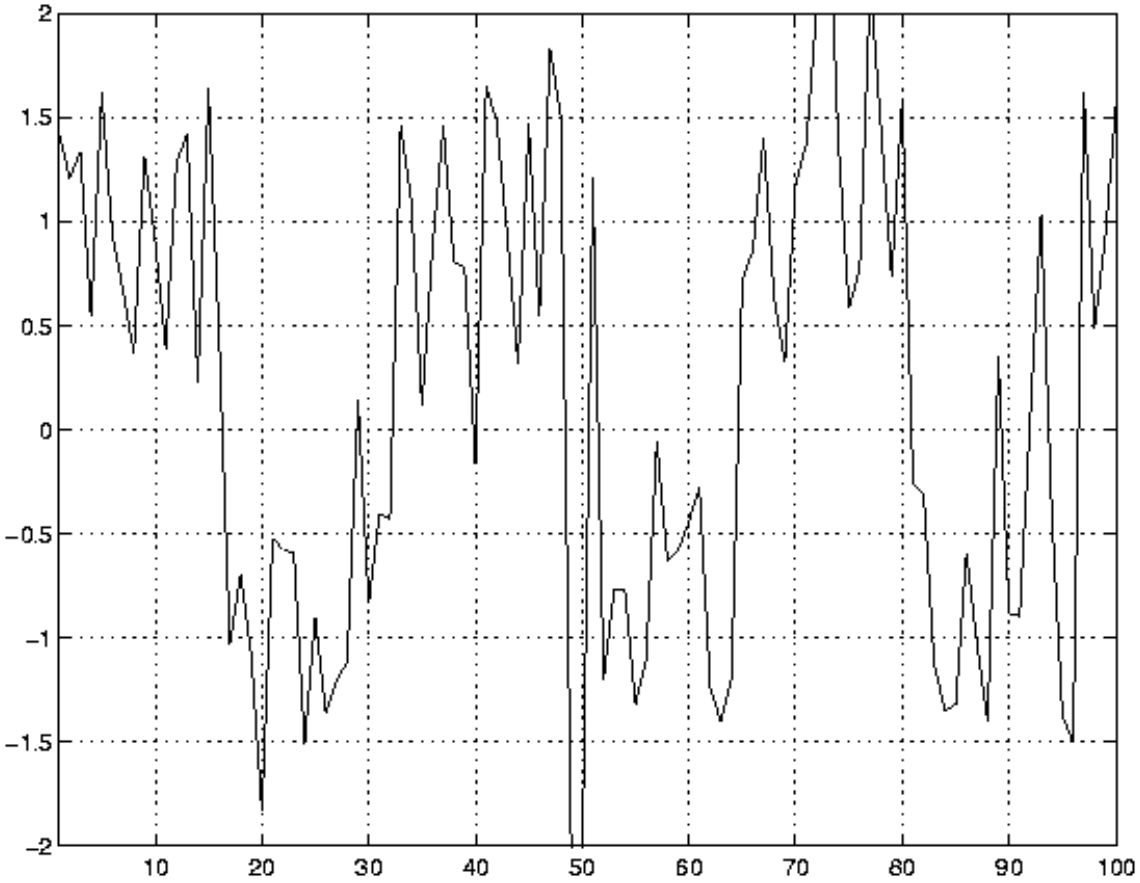

Figure 2.3: Noisy BPSK waveform.

However, in the presence of noise, the received waveform may look like that shown in Figure 2.3. 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 (Section 1.6.1). [2]Figure 2.4 and Figure 2.5 show the result of passing both the clean and noisy signal through the matched filter.

[2]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.

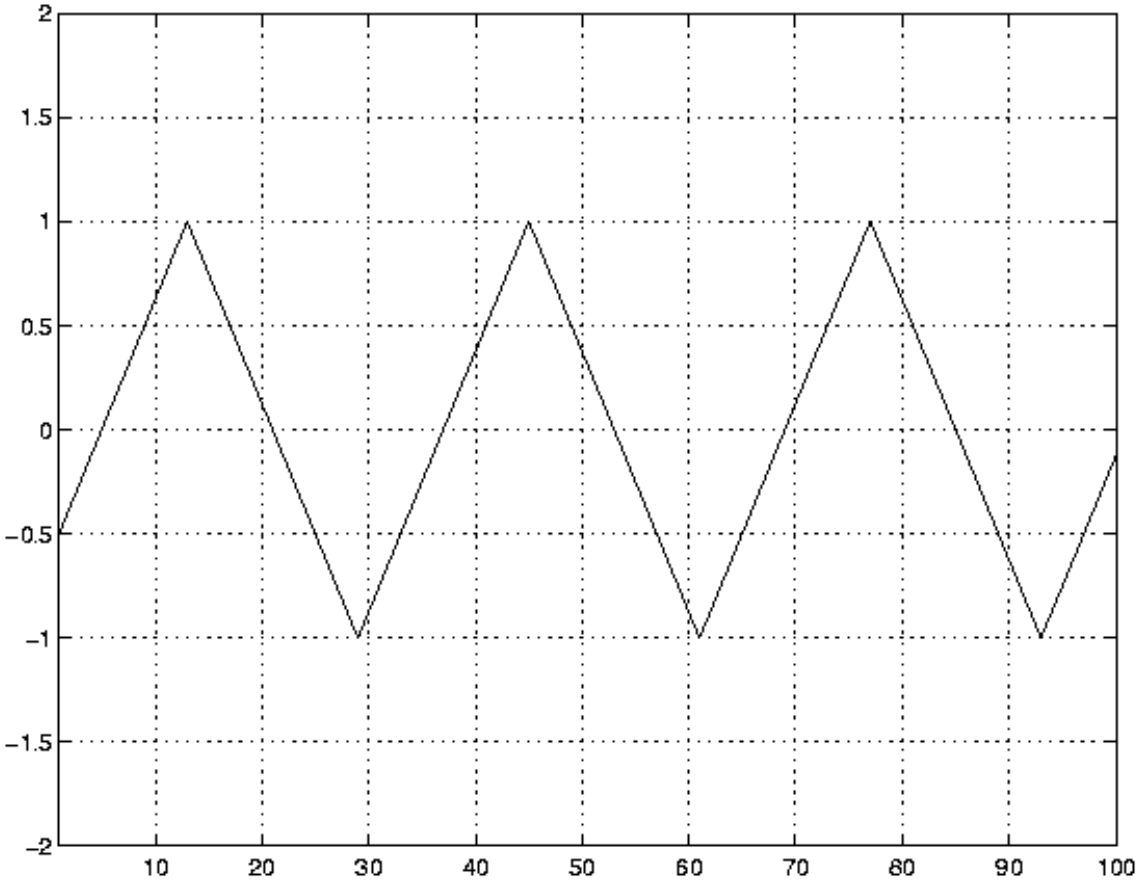

Figure 2.4: Averaging filter output for clean input.

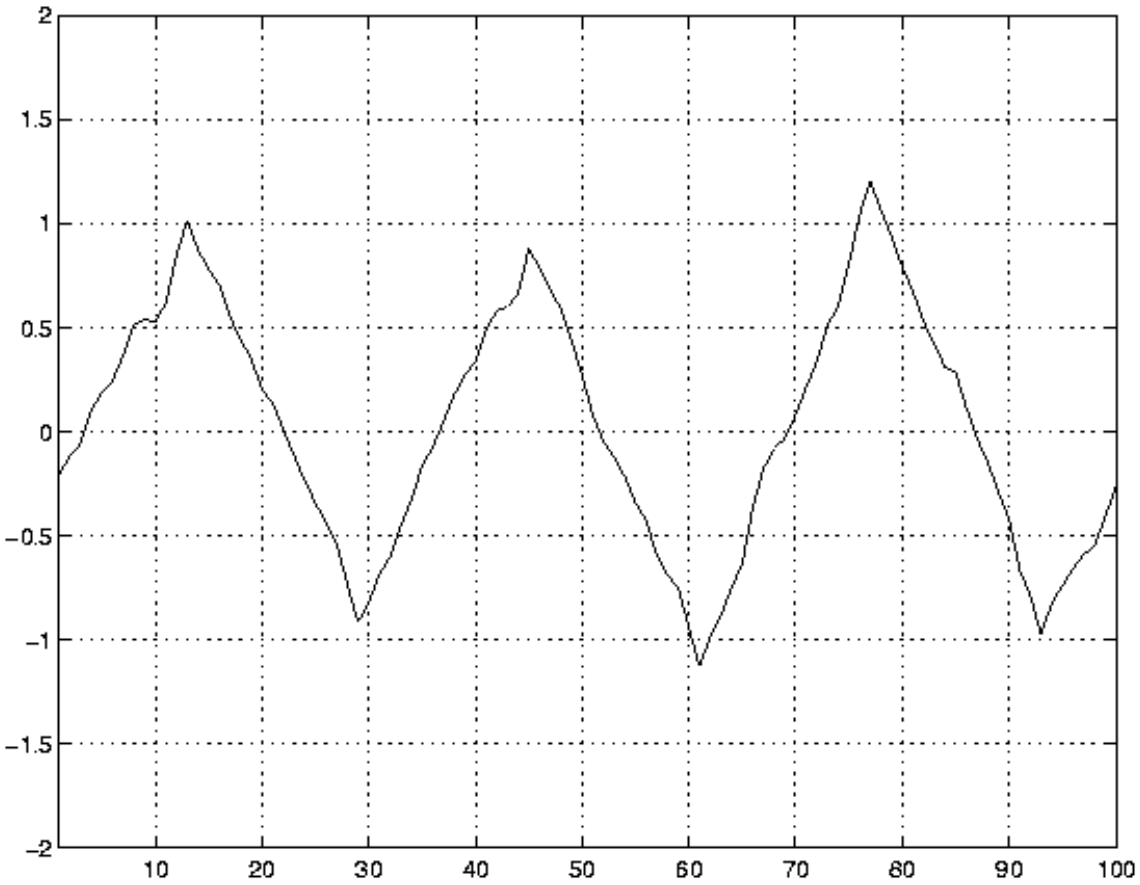

Figure 2.5: Averaging filter output for noisy input.

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 2.3.

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.

2.1.1.1.1 Early/late sampling

One simple method for recovering symbol timing is performed using a **delay-locked loop** (**DLL**). Figure 2.6 is a block diagram of the necessary components.

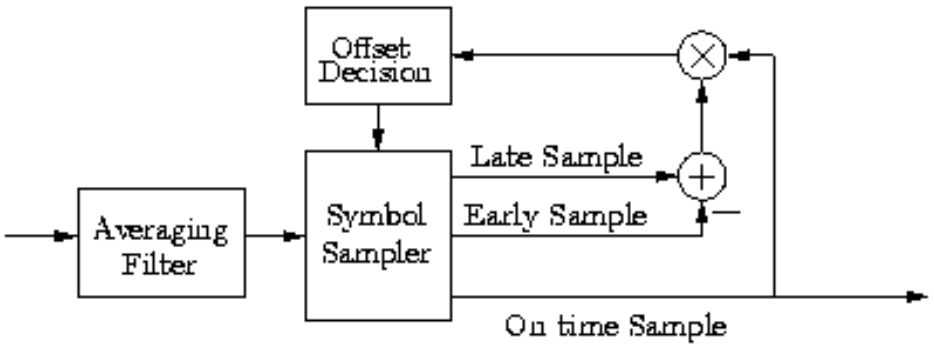

Figure 2.6: DLL block diagram.

Consider the sawtooth waveform shown in Figure 2.4, 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 2.7 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 2.8 and Figure 2.9 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 2.7, 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 2.8, 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 2.9, 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.

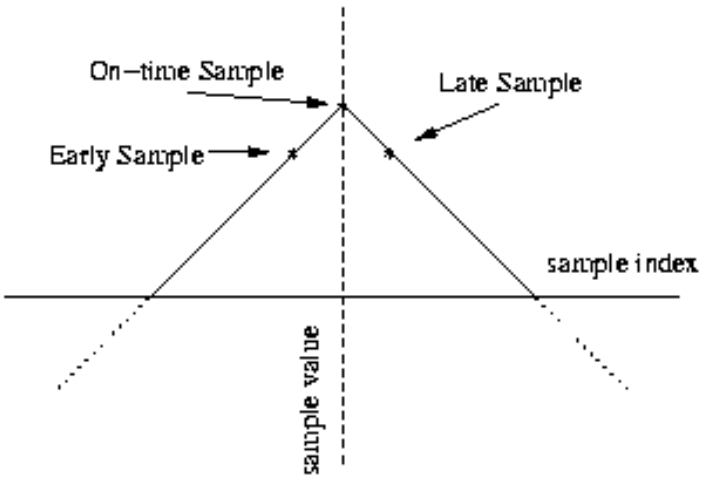

Figure 2.7: Sampling at a peak.

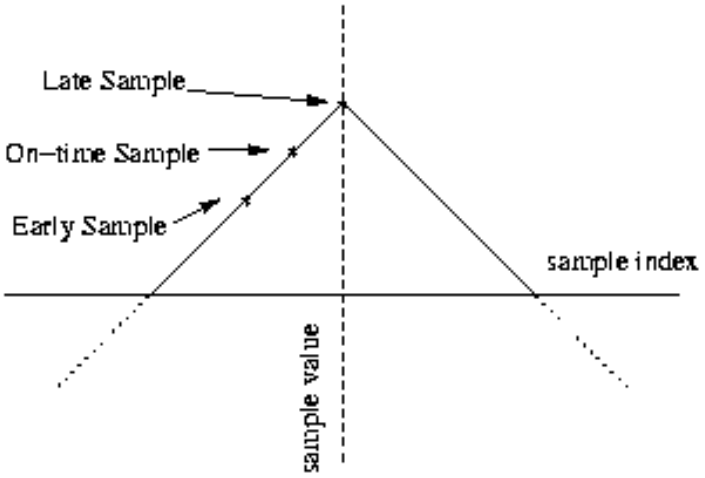

Figure 2.8: Sampling too early.

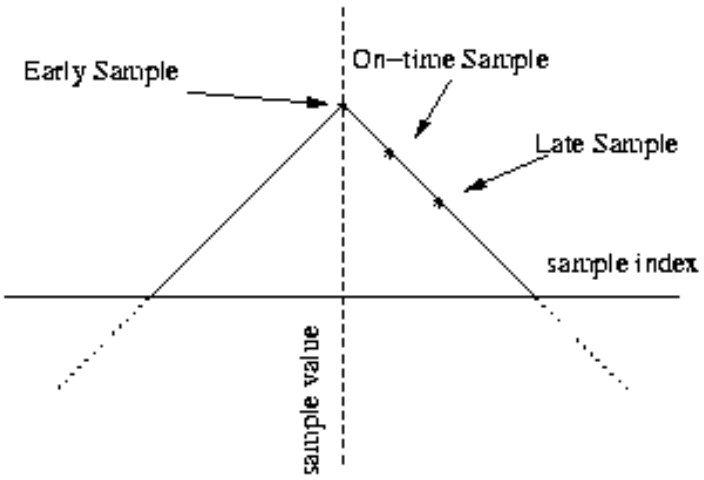

Figure 2.9: Sampling too late.

The input to the offset decision block is on − time (late − 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 2.7, Figure 2.8, and Figure 2.9. 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.

2.1.1.1.2 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.

2.1.1.2 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 2.6. 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 2.10 and Figure 2.11 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.

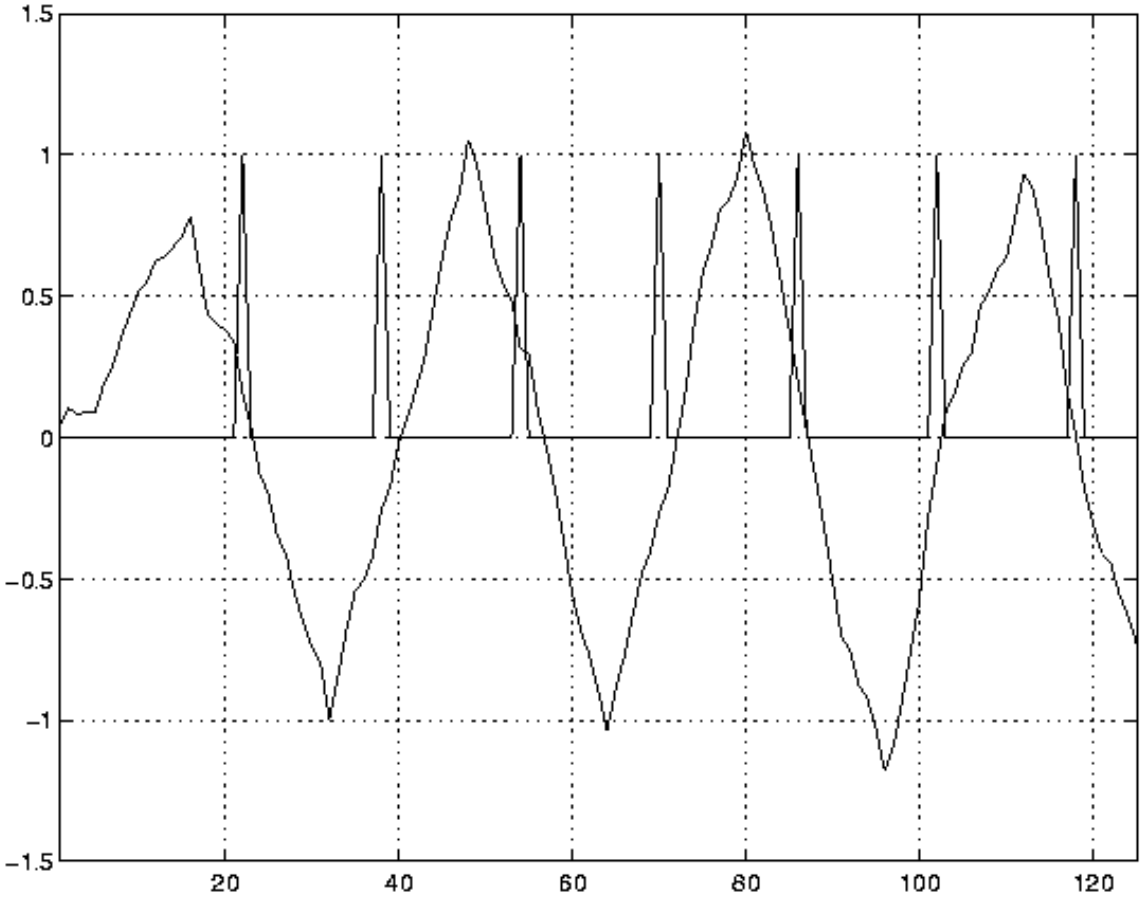

Figure 2.10: Symbol sampling before DLL lock.

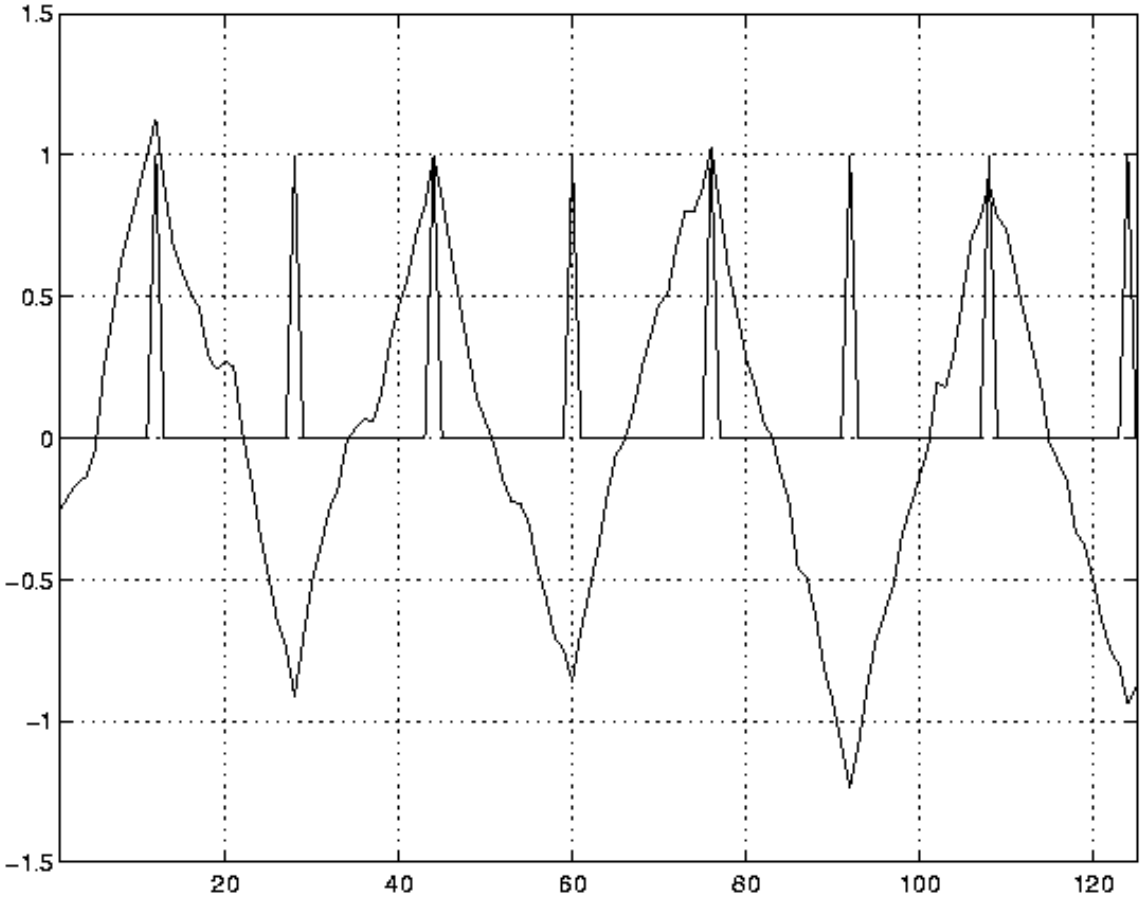

Figure 2.11: Symbol sampling after DLL lock.

2.1.1.3 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.

2.1.1.4 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*[9] and *Blahut*[1], 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 $[+1, -1, +1, -1, \ldots]$)?
- 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?

2.1.2 Digital Receiver: Carrier Recovery[3]

2.1.2.1 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 (Section 1.6.1), 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 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.

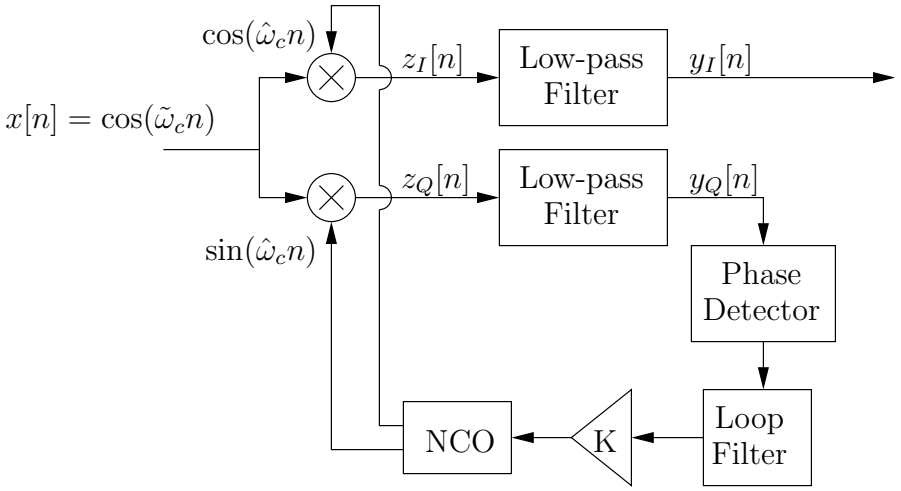

Figure 2.12: PLL Block Diagram

[3]This content is available online at <http://cnx.org/content/m10478/2.16/>.

2.1.2.1.1 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 2.12). A simple scheme for implementing an NCO is based on the following re-expression of the carrier sinusoid:

$$\sin(\omega_c n + \theta_c) = \sin(\theta[n]) \tag{2.1}$$

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

$$\theta[n] = \theta[n-1] + \omega_c \tag{2.2}$$

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:

$$\theta[n] = \theta[n-1] + \omega_c + d_{\mathrm{pd}}[n] \tag{2.3}$$

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?)

2.1.2.1.2 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 $\hat{\omega_c} = \omega_c + \hat{\theta}[n]$. Using the appropriate trigonometric identities [4], the in-phase and quadrature signals can be expressed as

$$z_0[n] = 1/2\left(\cos\left(\tilde{\theta}[n] - \hat{\theta}[n]\right) + \cos\left(2\omega_c + \tilde{\theta}[n] + \hat{\theta}[n]\right)\right) \tag{2.4}$$

$$z_Q[n] = 1/2\left(\sin\left(\tilde{\theta}[n] - \hat{\theta}[n]\right) + \sin\left(2\omega_c + \tilde{\theta}[n] + \hat{\theta}[n]\right)\right) \tag{2.5}$$

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

$$y_1[n] = 1/2\cos\left(\tilde{\theta}[n] - \hat{\theta}[n]\right) \tag{2.6}$$

$$y_Q[n] = 1/2\sin\left(\tilde{\theta}[n] - \hat{\theta}[n]\right) \tag{2.7}$$

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

$$\sin(\theta) \simeq \theta \quad , \quad \text{small} \tag{2.8}$$

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

and let the current value of the quadrature channel approximate the phase difference: $z_Q[n] \simeq \tilde{\theta}[n] - \hat{\theta}[n]$. With the exception of the sign error, this difference is essentially how much we need to offset our NCO frequency[5]. 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.[6]

2.1.2.1.3 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,

$$y[n] = \beta x[n] + \alpha y[n-1] \tag{2.9}$$

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.

2.1.2.2 MATLAB Simulation

Simulate the PLL system shown in Figure 2.12 using MATLAB. As with the DLL simulation, you will have to simulate the PLL on a sample-by-sample basis.

Use (2.3) 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 2.13 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.

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

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

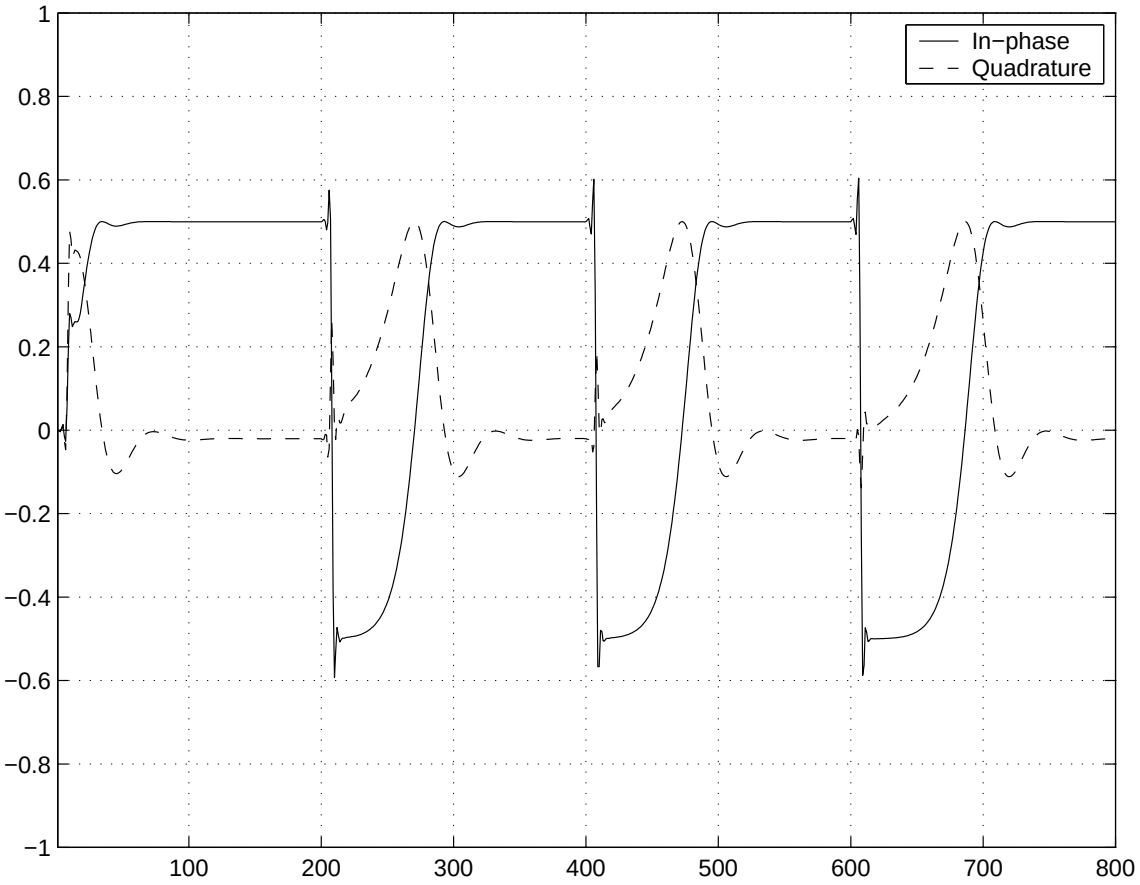

Figure 2.13: Output of PLL sub-system for BPSK modulated carrier.

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?

2.1.2.3 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.

2.1.2.3.1 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: $\sin\left(\frac{2\pi k}{N}\right)$, $k = \{0, \ldots, N-1\}$. 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.[7]

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.[8] 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 $16 \times 2^7 = 2048$.[9]

2.1.2.3.2 Extensions

You may want to refer to *Proakis* [10] and *Blahut* [2]. 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?

2.2 Audio Effects

2.2.1 Audio Effects: Using External Memory[10]

2.2.1.1 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 (Section 3.2.2) for a description and examples of accessing external memory.

[7] Of course, nearest-neighbor interpolation could be implemented with a small amount of extra code.
[8] How is this similar to the addition modulo 2π discussed in the MATLAB Simulation (Section 2.1.2.2: MATLAB Simulation)?
[9] If this value were 2049, what would be the output frequency of the NCO?
[10] This content is available online at <http://cnx.org/content/m10480/2.17/>.

2.2.1.2 Delay and Echo Implementation

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

2.2.1.2.1 Fixed-length delay implementation

First, implement the 131,072-sample delay shown in Figure 2.14 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.

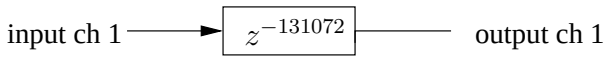

Figure 2.14: Fixed-Length Delay

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?)

2.2.1.2.2 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.

2.2.1.2.3 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 2.2.1.2.2: 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 2.15 shows the block diagram. (It may be necessary to multiply the input by 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.

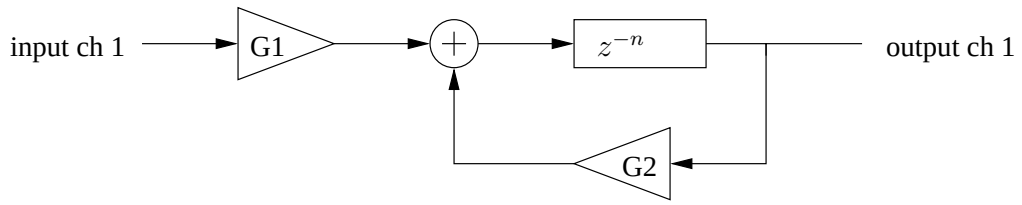

Figure 2.15: Feedback Echo

2.2.2 Audio Effects: Real-Time Control with the Serial Port[11]

2.2.2.1 Implementation

For this exercise, you will extend the system from Audio Effects: Using External Memory (Section 2.2.1) 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.

2.2.2.1.1 Feedback system implementation

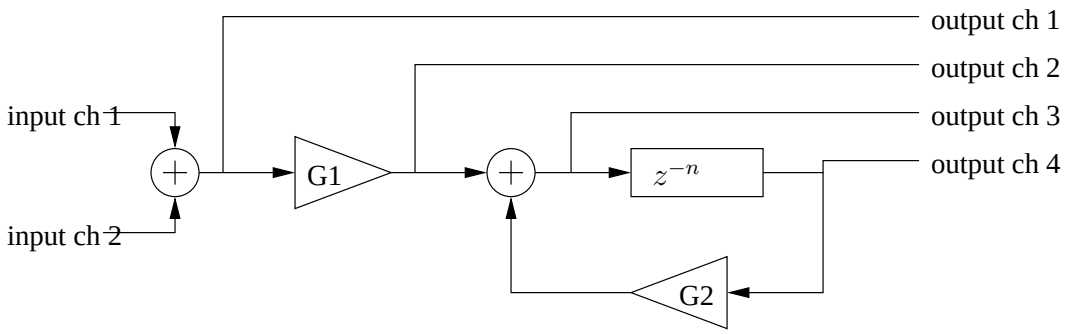

Figure 2.16: Feedback System with Test Points

First, modify code from Audio Effects: Using External Memory (Section 2.2.1) to create the feedback-echo system shown in Figure 2.16. 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

[11] This content is available online at <http://cnx.org/content/m10483/2.24/>.

- 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 2.2.2.1.2: 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.

2.2.2.1.2 MATLAB interface implementation

After studying the MATLAB interface outlined at the end of Using the Serial Port with a MATLAB GUI[12], 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.

2.3 Surround Sound

2.3.1 Surround Sound: Passive Encoding and Decoding[13]

2.3.1.1 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].

2.3.1.2 Encoder

You will create a MATLAB implementation of the passive encoder given by the block diagram in Figure 2.17.

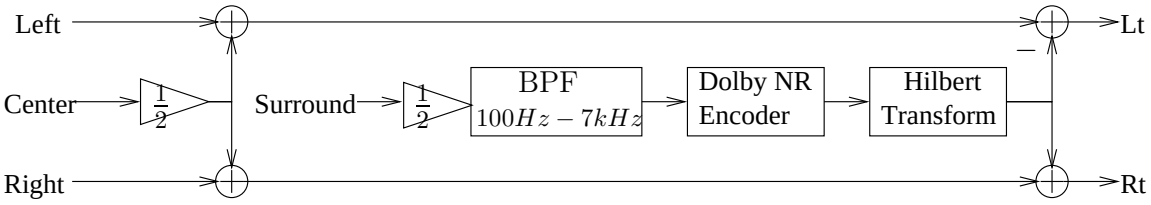

Figure 2.17: Dolby Pro Logic Encoder

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

[12] "Using the Serial Port with a MATLAB GUI" <http://cnx.org/content/m12062/latest/>
[13] This content is available online at <http://cnx.org/content/m10484/2.13/>.

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

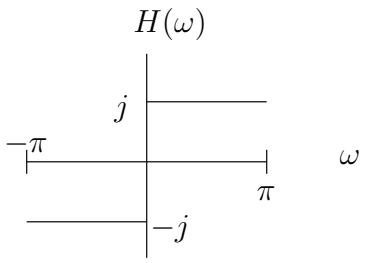

Figure 2.18: Hilbert transform transfer function

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

2.3.1.2.1 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.

2.3.1.3 Decoder

Implement the passive decoder shown in Figure 2.19 on the DSP. Use an appropriate time delay based on the distance between the front and back speakers and the speed of sound.

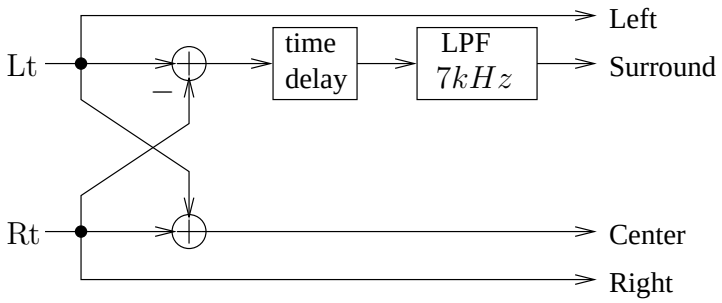

Figure 2.19: Dolby Pro Logic Passive Decoder

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?

2.3.1.4 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 2.3.2).

2.3.2 Surround Sound: Chamberlin Filters[14]

2.3.2.1 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* [7].

2.3.2.2 Filter Topology

A Chamberlin filter is a simple two-pole IIR filter with the transfer function given in (2.10):

$$H(z) = \frac{{F_z}^2 z^{-1}}{1 - \left(2 - \left(F_c Q_c - {F_c}^2\right)\right) z^{-1} - 1z^{-2}} \tag{2.10}$$

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 2.20. 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

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

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.

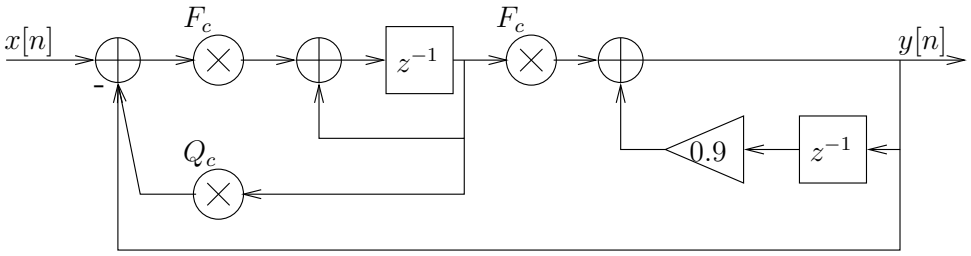

Figure 2.20: Chamberlin Filter Topology

Figure 2.21 and Figure 2.22 show how the response of the filter varies with Q_c and F_c.

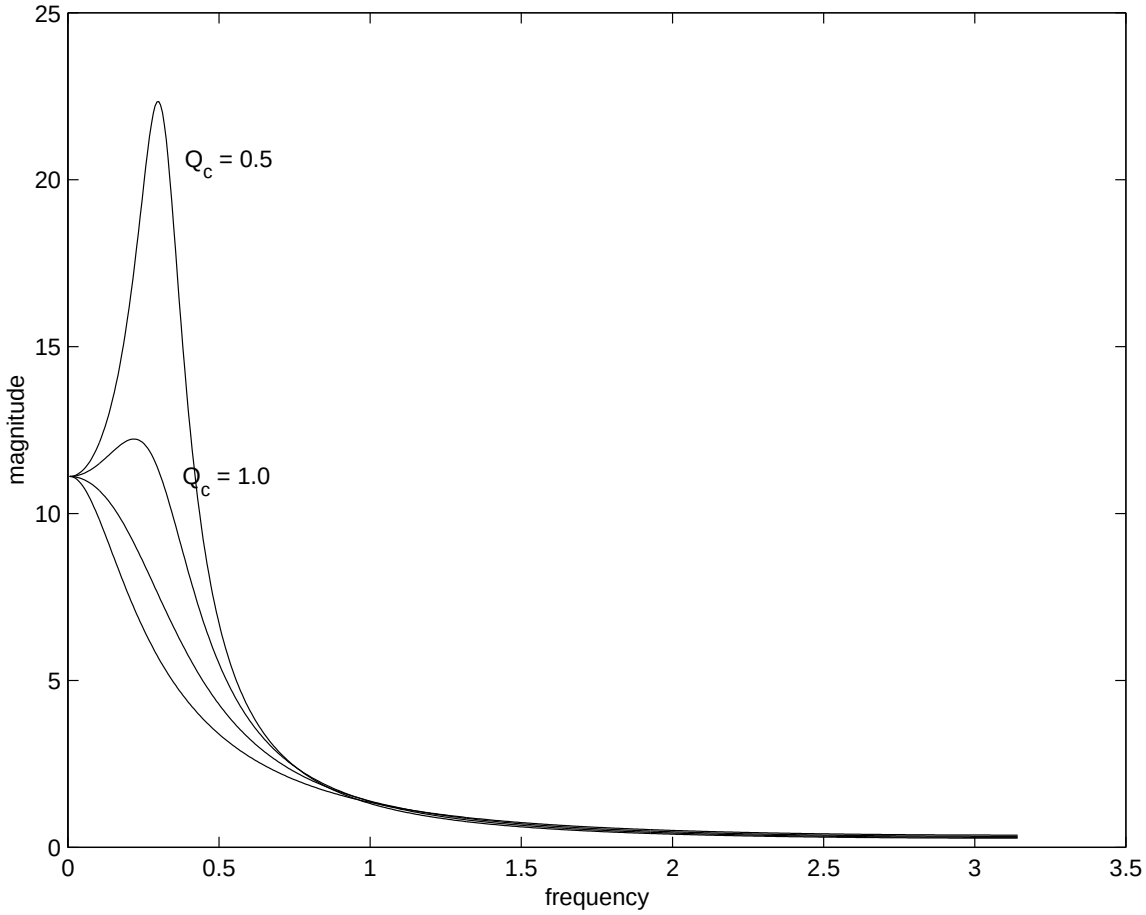

Figure 2.21: Chamberlin filter responses for various Q_c ($F_c = .3$)

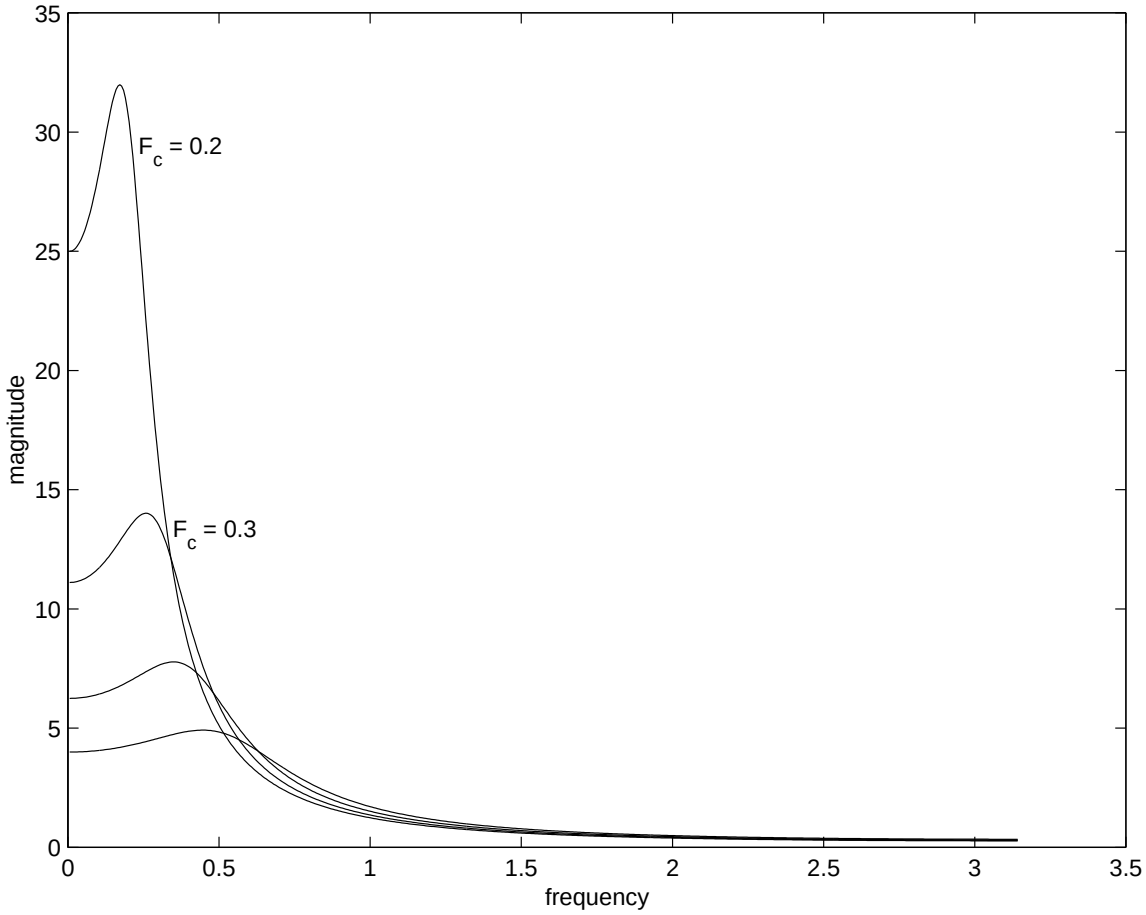

Figure 2.22: Chamberlin filter responses for various F_c ($Q_c = .8333$)

2.3.2.3 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 (2.10). 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?

2.4 Adaptive Filtering

2.4.1 Adaptive Filtering: LMS Algorithm[15]

2.4.1.1 Introduction

Figure 2.23 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 $y[n]$.

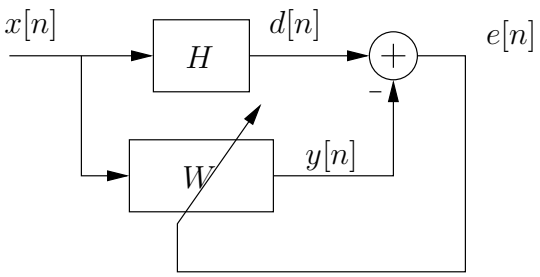

Figure 2.23: System identification block diagram.

2.4.1.1.1 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 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

$$h_{n+1}[i] = h_n[i] + \frac{\mu}{2}\left(-\frac{\partial(|e|)^2}{\partial h_n[i]}\right) \tag{2.11}$$

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, (2.11) 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.

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

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

$$\begin{aligned} \frac{\partial(|e|)^2}{\partial h[i]} &= 2\frac{\partial e}{\partial h[i]}e \\ &= 2\frac{\partial(d-y)}{\partial h[i]}e \\ &= \left(2\frac{\partial\left(d-\sum_{i=0}^{N-1}h[i]x[n-i]\right)}{\partial h[i]}\right)(e) \end{aligned} \tag{2.12}$$

$$\frac{\partial(|e|)^2}{\partial h[i]} = 2(-x[n-i])e \tag{2.13}$$

which in turn gives us the final LMS coefficient update,

$$h_{n+1}[i] = h_n[i] + \mu e x[n-i] \tag{2.14}$$

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.)

2.4.1.2 MATLAB Simulation

Simulate the system identification block diagram shown in Figure 2.23.

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.

2.4.1.3 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 (Section 1.6.1), 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.

2.4.1.4 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 2.23 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?

2.5 Speech Processing

2.5.1 Speech Processing: Theory of LPC Analysis and Synthesis[16]

2.5.1.1 Introduction

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

2.5.1.1.1 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

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n]\, y[n-l] \tag{2.15}$$

where n is the sample index, and l is the lag or time shift between the two signals *Proakis and Manolakis* [8] (*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 $l = \{0, 1, \ldots, P\}$.

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

$$r_{ss}[l] = \left(\frac{l}{N} \sum_{n=0}^{N-1} s[n]\, s[n-l] \right) \tag{2.16}$$

where $s[n]$, $n = \{-\mathrm{P}, -\mathrm{P}+1, \ldots, N-1\}$ are the known samples (see Figure 2.24) and the $\frac{1}{N}$ is a normalizing factor.

[16] This content is available online at <http://cnx.org/content/m10482/2.19/>.

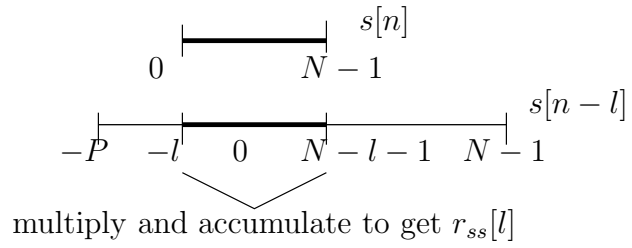

Figure 2.24: Computing the autocorrelation coefficients

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

$$r_{ss}[l] = \left(\frac{1}{N-1} \sum_{n=l}^{N-1} s[n]\, s[n-l] \right) \tag{2.17}$$

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

2.5.1.1.2 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 2.25 shows how the resonant structure of a vowel could be captured by an IIR system.

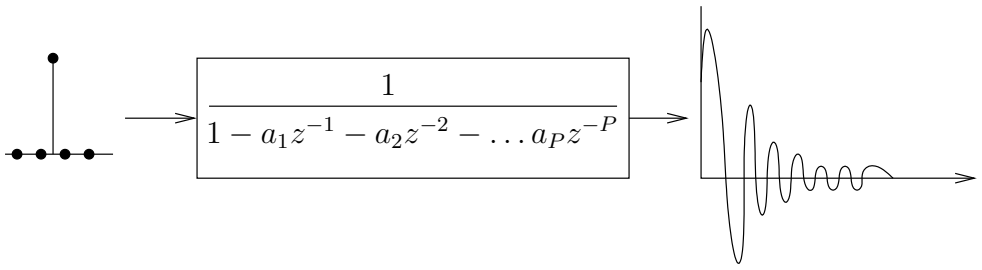

Figure 2.25: Linear Prediction (IIR) Model of Speech

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

$s[n-k]$, $k = \{1, \ldots, P\}$. The predicted sample $\hat{s}[n]$ is then given by *Rabiner and Juang* [11]

$$\hat{s}[n] = \sum_{k=1}^{P} a_k s[n-k] \tag{2.18}$$

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 (2.18), when the predicted sample equals the actual signal (*i.e.*, $\hat{s}[n] = s[n]$), we have

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

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

$$s(z) = \frac{1}{1 - \sum_{k=1}^{P} a_k z^{-k}} \tag{2.19}$$

The optimal solution to this problem is *Rabiner and Juang* [11]

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

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

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

$$a = R^{-1} r \tag{2.20}$$

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* [11] (*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. - $\alpha_{j,i} = \alpha_{j,i-1} - k_i \alpha_{i-j,i-1}$ $j = \{1, \ldots, i-1\}$
   - $\alpha_{i,i} = k_i$
3. $E_i = \left(1 - {k_i}^2\right) E_{i-1}$

2.5.1.1.3 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 2.26. 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.

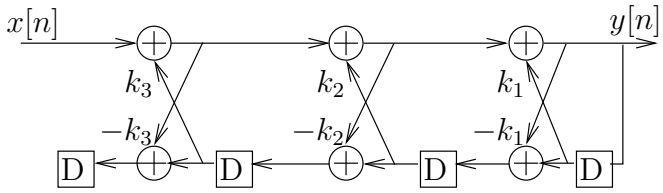

Figure 2.26: IIR lattice filter implementation.

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.

2.5.1.2 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.

2.5.2 Speech Processing: LPC Exercise in MATLAB[17]

2.5.2.1 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 (2.19)). 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?

2.5.3 Speech Processing: LPC Exercise on TI TMS320C54x[18]

2.5.3.1 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 $l = \{0, 1, \ldots, 15\}$ (*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[19] 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 (Section 3.2.2) 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.

[17] This content is available online at <http://cnx.org/content/m10824/2.5/>.
[18] This content is available online at <http://cnx.org/content/m10825/2.6/>.
[19] http://cnx.org/content/m10825/latest/thru6.asm

2.5.3.1.1 Integer division (optional)

In order to implement the Levinson-Durbin algorithm, you will need to use integer division to do Step 1 (p. 75) of the algorithm. Refer to the *Applications Guide*[?] and the `subc` instruction for a routine that performs integer division.

Chapter 3

General References

3.1 Processor

3.1.1 Two's Complement and Fractional Arithmetic for 16-bit Processors[1]

3.1.1.1 Two's-complement notation

Two's-complement notation is an efficient way of representing signed numbers in microprocessors. It offers the advantage that addition and subtraction can be done with ordinary unsigned operations. When a number is written in two's complement notation, the most significant bit of the number represents its sign: 0 means that the number is positive, and 1 means the number is negative. A positive number written in two's-complement notation is the same as the number written in unsigned notation (although the most significant bit must be zero). A negative number can be written in two's complement notation by inverting all of the bits of its absolute value, then adding one to the result.

Example 3.1
Consider the following four-bit two's complement numbers (in binary form):

$1 = 0001$	$-1 = 1110 + 1 = 1111$
$2 = 0010$	$-2 = 1101 + 1 = 1110$
$6 = 0110$	$-6 = 1001 + 1 = 1010$
$8 = 1000$	$-8 = 0111 + 1 = 1000$

Table 3.1

NOTE: 1000 represents -8, not 8. This is because the topmost bit (the sign bit) is 1, indicating that the number is negative.

The maximum number that can be represented with a k-bit two's-complement notation is $2^{k-1} - 1$, and the minimum number that can be represented is -2^{k-1}. The maximum integer that can be represented in a 16-bit memory register is 32767, and the minimum integer is -32768.

[1] This content is available online at <http://cnx.org/content/m10808/2.9/>.

3.1.1.2 Fractional arithmetic

The DSP microprocessor is a 16-bit integer processor with some extra support for **fractional arithmetic.** Fractional arithmetic turns out to be very useful for DSP programming, since it frees us from worries about overflow on multiplies. (Two 16-bit numbers, multiplied together, can require 32 bits for the result. Two 16-bit fixed-point fractional numbers also require 32 bits for the result, but the 32-bit result can be rounded into 16 bits while only introducing an error of approximately 2^{-16}.) For this reason, we will be using fixed-point fractional representation to describe filter taps and inputs throughout this course.

Unfortunately, the assembler and debugger we are using do not recognize this fractional fixed-point representation. For this reason, when you are using the assembler or debugger, you will see decimal values (ranging from -32768 to 32767) on screen instead of the fraction being represented. The conversion is simple; the fractional number being represented is simply the decimal value shown divided by 32768. This allows us to represent numbers between -1 and $1 - 2^{-15}$.

NOTE: 1 cannot be represented exactly.

When we multiply using this representation, an extra shift left is required. Consider the two examples below:

Example 3.2

fractional	$0.5 \times 0.5 = 0.25$
decimal	$16384 \times 16384 = 4096 \times 2^{16} : 4096/327681/8$
hex	$4000 \times 4000 = 1000 \times 2^{16}$

Table 3.2

fractional	$0.125 \times 0.75 = 0.093750$
decimal	$4096 \times 24576 = 1536 \times 2^{16} : 1536/327680.046875$
hex	$1000 \times 6000 = 0600 \times 2^{16}$

Table 3.3

You may wish touse the MATLAB commands `hex2dec` and `dec2hex`. When we do the multiplication, we are primarily interested in the top 16 bits of the result, since these are the data that are actually used when we store the result back into memory and send it out to the digital-to-analog converter. (The entire result is actually stored in the accumulator, so rounding errors do not accumulate when we do a sequence of multiply-accumulate operations in the accumulators.) As the example above shows, the top 16 bits of the result of multiplying the fixed point fractional numbers together is half the expected fractional result. The extra left shift multiplies the result by two, giving us the correct final product.

The left-shift requirement can alternatively be explained by way of decimal place alignment. Remember that when we multiply decimal numbers, we first multiply them ignoring the decimal points, then put the decimal point back in the last step. The decimal point is placed so that the total number of digits right of the decimal point in the multiplier and multiplicand is equal to the number of digits right of the decimal point in their product. The same applies here; the "decimal point" is to the right of the leftmost (sign) bit, and there are 15 bits (digits) to the right of this point. So there are a total of 30 bits to the right of the decimal in the source. But if we do not shift the result, there are 31 bits to the right of the decimal in the 32-bit result. So we shift the number to the left by one bit, which effectively reduces the number of bits right of the decimal to 30.

Before the numbers are multiplied by the ALU, each term is **sign-extended** generating a 17-bit number from the 16-bit input. Because the examples presented above are all positive, the effect of this sign extension is simply adding an extra "0" bit at the top of the register (*i.e.*, positive numbers are not affected by the sign extension). As the following example illustrates, not including this sign-bit for negative numbers produces erroneous results.

fractional	$-0.5 \times 0.5 = -0.25$
decimal	$49152 \times 16384 = 12288 \times 2^{16} : 12288/326780.375$
hex	$C000 \times 4000 = 30000000 = 3000 \times 2^{16}$

Table 3.4

Note that even after the result is left-shifted by one bit following the multiply, the top bit of the result is still "0", implying that the result is incorrectly interpreted as a positive number.

To correct this problem, the ALU sign-extends negative multipliers and multiplicands by placing a "1" instead of a "0" in the added bit. This is called **sign extension** because the sign bit is "extended" to the left another place, adding an extra bit to the left of the number without changing the number's value.

fractional	$-0.5 \times 0.5 = -0.25$
hex	$1C000 \times 4000 = 70000000 = 7000 \times 2^{16}$

Table 3.5

Although the top bit of this result is still "0", after the final 1-bit left-shift the result is `E000 000h` which is a negative number (the top bit is "1"). To check the final answer, we can negate the product using the two's complement method described above. After flipping all of the bits we have `1FFF FFFFh`, and adding one yields `2000 0000h`, which equals 0.25 when interpreted as an 32 bit fractional number.

3.1.2 Addressing Modes for TI TMS320C54x[2]

Microprocessors provide a number of ways to specify the location of data to be used in calculations. For example, one of the data values to be used in an `add` instruction may be encoded as part of that instruction's **opcode**, the raw machine language produced by the assembler as it parses your assembly language program. This is known as **immediate addressing**. Alternatively, perhaps the opcode will instead contain a memory address which holds the data (**direct addressing**). More commonly, the instruction will specify that an auxiliary register holds the memory address which in turn holds the data (**indirect addressing**). The processor knows which addressing mode is being used by examining special bit fields in the instruction opcode.

Knowing the basic addressing modes of your microprocessor is important because they map directly into assembly language syntax. Many annoying and sometimes hard-to-find bugs are caused by inadvertently using the wrong addressing mode in an instruction. Also, in any assembly language, the need to use a particular addressing mode often dictates which instruction one picks for a given task.

Chapter five, **Data Addressing**, in the *CPU and Peripherals*[**?**] reference contains extended descriptions of most of the addressing modes described below.

[2]This content is available online at <http://cnx.org/content/m10806/2.7/>.

3.1.2.1 Accumulators: src, dst

Whenever the abbreviations `src` or `dst` are used in the assembly language syntax description for an instruction, it means that only the accumulators `A` and `B` may be used for that particular operand. These are seen everywhere, but two classic examples are `ld`, which always loads data into an accumulator from somewhere else, and `sth/stl`, which always store data from an accumulator to somewhere else.

Examples:

```
ld     *AR5,A      ; sets A = (contents of memory location pointed to by AR5)
sth    B,*AR7+     ; sets (contents of memory location pointed to be AR7) = B,
;    and then increments AR7 by one
```

3.1.2.2 Memory-mapped Registers: MMR, MMRx, MMRy

Many of the TMS320C54x registers are memory-mapped, meaning that they occupy real addresses at the low end of data memory space. The most commonly used of these are the auxiliary registers `AR0` through `AR7`. Whenever the abbreviation `MMR` is used in the assembly language syntax description for an instruction, it means that any memory-mapped register may be used for that particular operand. Only eight instructions use memory-mapped register addressing: `ldm`, `mvdm`, `mvmd`, `mvmm`, `popm`, `pshm`, `stlm`, and `stm`. With `mvmm`, since the instruction accepts two memory-mapped register operands, `MMRx` and `MMRy`, only `AR0-AR7` and `SP` may be used.

Do not use an asterisk in front of `ARx` variables here, since this is not indirect addressing.

Examples:

```
mvmm     AR3,AR5    ; sets AR5 = AR3
stm      #5,AR2     ; sets AR2 = 5
ldm      AR0,A      ; sets A = AR0
```

3.1.2.3 Immediate Addressing: #k3, #k5, K, #k9, #lk

Immediate addressing means that the numerical value of the data is itself provided within the assembly instruction. Various TMS320C54x instructions allow immediate data of 3, 5, 8, 9, or 16 bits in length, which are signified in the assembly language syntax descriptions with one of the above symbols. The 16-bit form is the most common and is signified by `#lk`. 16-bit immediate values always require an extra instruction word and therefore take an extra machine cycle to execute.

An immediate data operand is almost always specified in assembler syntax by prepending a pound sign (#) to the data. Depending on the context, the assembler may assume that you meant immediate addressing anyway.

Examples:

```
ld       #0,A       ; sets A = 0
cmpm     AR1,#1     ; sets flag TC = 1 if AR1 == 1; else TC = 0
```

Labels make this more complicated. Recall that a label in your assembly code is nothing more than shorthand for the memory address where the labeled code or data is stored. So does an instruction like

```
stm     coef,AR2   ; sets AR2 = memory address of label coef
```

mean to store the contents of memory location `coef` in `AR2`, or does it mean to store the memory address `coef` itself in `AR2`? The second interpretation is correct. Because the `stm` instruction has only one form, expecting a `#lk` immediate operand, the assembler does not care whether the label is prefixed with a pound sign or not. Still, it would have been better for us to include the pound sign in the above example for clarity.

Many instructions have several versions allowing the use of different addressing modes (see `ld` for a good example of this). With these instructions, including the pound sign is not optional when specifying immediate addressing. The only safe rule, then, is always to prefix the label with a pound sign if you wish to specify the memory address of the label and not the contents of that address.

If you are not sure how a particular instruction has been assembled, you can always examine the `.lst` file produced by the assembler, and compare the hexadecimal opcodes listed to the left of the assembly instructions with the assembly opcodes given in the assembly language manual (Chapter 4 of the *Mnemonic Instruction Set*[?] reference).

3.1.2.4 Direct Addressing: Smem and others

In the modes called **direct addressing** by TI, the instruction opcode contains a memory offset (see the "dma" bits on page 5-8 of the *CPU and Peripherals*[?] reference) seven bits long, which is combined with either the `DP` (data pointer) or `SP` (stack pointer) register to obtain a complete 16-bit data-memory address. This divides the data memory into pages of 128 words each.

`SP` is initialized for you in the core file and should not need to be modified. `SP`-referenced direct addressing is used by the `pshd`, `pshm`, `popd`, and `popm` instructions for stack manipulation, as well as by all subroutine calls and returns, which save program addresses on the stack.

`DP`-referenced direct addressing is available wherever you see the `Smem` abbreviation in an assembly syntax description. The advantage of `DP`-referenced addressing over the `*(lk)` form described in the next section is that `DP`-referenced addressing will not add an extra instruction word (and corresponding extra machine cycle). The disadvantage is that it is limited to 128 words of contiguous memory, and you have to make sure that `DP` points to the right 128 words. `DP` may be changed with the `ld` instruction as needed.

Examples:

```
ld     10,A       ; sets A = (contents of memory location DP + 10)
add    6,B        ; sets B = B + (contents of memory location DP + 6)
```

> NOTE: Make sure you understand that the numbers 10 and 6 above are interpreted as memory addresses, **not** data values. To get data values, you would need to use a pound sign in front of the numbers.

3.1.2.5 Absolute Addressing: dmad, pmad, *(lk)/Smem

This seems to be TI's term for all the forms of direct addressing which it does not call direct addressing! It is represented in assembly-instruction syntax-definitions using one of the above abbreviations (`*(lk)` addressing is available when the syntax definition says `Smem`).

3.1.2.5.1 dmad

`dmad` (Data Memory ADdress) operands are used by `mvxx` data move instructions and represent 16-bit memory addresses in data memory whose contents are used in the instruction.

Example:

```
f3ptr   .word    0          ; reserve one word of storage; initialize to 0
. . . .
mvdm    f3ptr,AR4  ; set AR4 = memory address of f3ptr
```

3.1.2.5.2 pmad

`pmad` (Program Memory ADdress) operands are used by the `firs`, `macd`, `macp`, `mvdp`, and `mvpd` instructions, as well as all subroutine calls and branching instructions. They represent 16-bit addresses in program memory whose contents are used in the instruction, or jumped to in the case of branch instructions. Other than subroutine calls and branches, the most common use of a `pmad` is for the `firs` instruction.

Example:

```
firs    *AR3+,*AR4+,coefs
```

NOTE: `coefs` is a label in the program section of the code, **not** the data section.

3.1.2.5.3 *(lk)

`*(lk)` addressing is a syntactic oddity. The asterisk symbol generally means that indirect addressing is being used (see below), but this is actually direct addressing with a 16-bit data memory address encoded in the instruction's last word. The reason for the asterisk is that TI **does** set the "I" bit in the opcode, usually denoting indirect addressing, and this form can only be used when an `Smem` is called for in the assembly syntax. Other bits in the low byte of the first instruction word tell the processor that the "`*(lk)` exception" is to be used, and to fetch the memory address in the next word (see the MOD bits on page 5-10 of the *CPU and Peripherals*[?] reference). You can easily recognize this addressing mode in `.lst` files because the low byte of the first instruction word always equals `F8h`.

Examples:

```
hold    .word    1           ; reserve one word of storage and initialize to 1
count   .word    0           ; reserve one word of storage and initialize to 0
. . . .
ld       *(count),B  ; sets B = 0 (assuming memory was not changed)
st       T,*(hold)   ; sets (storage location at address hold) = T
```

3.1.2.6 Indirect Addressing: Smem, Xmem, Ymem

Indirect addressing on the TMS320C54x always uses the auxiliary registers AR0 through AR7 and comes in two basic flavors. These are easily recognized from the assembly language syntax descriptions as either Smem or Xmem/Ymem.

3.1.2.6.1 Smem

In Smem indirect addressing, only one indirect address is used in the instruction and a number of variations is possible (see the table on page 5-13 of the *CPU and Peripherals*[?] reference). An asterisk is always used, which signifies indirect addressing. Any of the registers AR0-AR7 may be used, with optional modifications: automatic post-decrement by one, pre- and post-increment by one, post-increment and post-decrement by n (n being stored in AR0), and more, including many options for circular addressing (which automatically implements circular buffers) and bit-reversed addressing (which is useful for FFTs).

3.1.2.6.2 Xmem/Ymem

Xmem/Ymem indirect addressing is generally used in instructions that need two different indirect addresses, although there are a few instances where an Xmem by itself is specified in order to save bits in the opcode for other options. In Xmem/Ymem indirect addressing, fewer bits are used to encode the option modifiers in the opcode; hence, fewer options are available: post-increment by one, post-decrement by one, and post-increment by AR0 with circular addressing.

Examples:

```
stl    B,*AR6     ; sets (contents of location pointed to by AR6) = low word of B
stl    B,*AR6+0%  ; sets (contents of location pointed to by AR6) = low word of B,
;      then increments AR6 with circular addressing
mar    *+AR3(-6)  ; decrements AR3 by 6 (increment by -6)
```

NOTE: The mar (modify address register) instruction is unusual in the sense that it takes an Smem operand but does nothing with the data pointed to by the ARx register. Its purpose is to perform any of the allowed register modifications discussed above without having to do anything else. This is often handy when you are using an Xmem/Ymem-type instruction but need to do an ARx modification that is only allowed with an Smem-type operand.

3.1.2.7 Summary

The `ld` instruction is illustrative of the many possible addressing modes which can be selected with the proper choice of assembly language syntax:

```
ld      #0,A          ; immediate data:  sets A = 0
ld      0,A           ; DP-referenced direct:  sets A = (contents of the address DP + 0)
ld      mydata,A      ; DP-referenced direct:  sets A = (contents of the address
;       DP + lower seven bits of mydata)
ld      #mydata,A     ; immediate data:  sets A = 16 bit address mydata
ld      *(mydata),A   ; *(lk) direct:  sets A = (contents of the 16 bit address mydata)
ld      B,A           ; accumulator:  sets A = B
ld      *AR1+,A       ; indirect:  sets A = (contents of address pointed to by AR1),
;       and afterwards increments AR1 by one
ldm     AR2,A         ; memory-mapped register:  sets A = AR2
```

3.2 Core File

3.2.1 Core File: Introduction to Six-Channel Board for TI EVM320C54[3]

3.2.1.1 The Six Channel Surround Sound Board

The six-channel board attaches to the top of the DSP evaluation module and replaces its onboard, one-channel A/D and D/A with six channels of D/A and two channels of A/D running at a sample rate of 44.1 kHz. Alternatively, the A/D can be disabled and a SP/DIF digital input enabled, allowing PCM digital data from a CD or DVD to be sent directly into the DSP for processing. The two input channels and six output channels all sample at the same time; clock skew between channels is not an issue. By default, the core code reads and writes blocks of 64 samples for each channel of input and output; however, this aggregation can be changed to any value between 1 and 80 samples[4]. If your code needs a larger aggregation of samples - for instance, for a 256 point FFT - you will need to do this aggregation yourself.

Other features include buffered serial communication code, which allows you to send and receive data from the serial port. This can be used to control your DSP program with a graphical user-interface on the PC; it can also be used to report status back to the PC for applications such as speech recognition.

The core code, core.asm[5] (which requires globals.inc[6] , ioregs.inc[7] , and misc.inc[8]) also initializes the DSP itself. It enables the fractional arithmetic mode for the ALU, programs the wait states for the external memory, and sets the DSP clock to 80 MHz[9].

3.2.1.1.1 Testing the six-channel sample code

We will start with a sample application, which simply sends the inputs into the outputs–relaying both the audio inputs from the A/D converters to the D/A converters, and any data that comes in on the serial

[3] This content is available online at <http://cnx.org/content/m10513/2.13/>.
[4] The upper bound is determined by the amount of memory available to the auto-buffering unit.
[5] http://cnx.rice.edu/author/workgroups/90/m10017/core.asm
[6] http://cnx.org/content/m10513/latest/globals.inc
[7] http://cnx.org/content/m10513/latest/ioregs.inc
[8] http://cnx.org/content/m10513/latest/misc.inc
[9] The DSP is rated to run at 100 MHz; however, the serial port does not work reliably when the DSP clock speed is greater than 80 MHz.

port back to the PC. To familiarize yourself with this sample application, locate a copy of thru6.asm[10] and assemble it.

Once you have done that, start Code Composer. Since we are using the on-chip RAM on the TMS320C549 to hold program code, we need to map that RAM into program space before we can load our code. This can be done by opening the CPU Registers window (the same one you use to look at the `ARx` registers and the accumulators) and changing the `PMST` register to `0xFFE0`. This sets the `OVLY` bit to 1, switching the internal RAM into the DSP's program memory space.

Finally, load the `thru6.out` file, use Code Composer's `Reset DSP` menu option to reset the DSP, and run the code. Probe at the connections with the function generator and the oscilloscope; inputs and outputs are shown in Figure 3.1. Note that output channels 1-3 come from input channel 1, and output channels 4-6 come from input channel 2. Figure 3.1 shows the six-channel board's connector configuration.

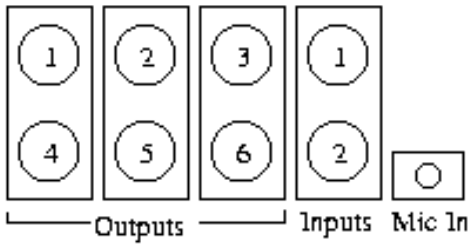

Figure 3.1: Six-Channel Board Analog Inputs and Outputs

Also test the serial communications portion of the thru6.asm[11] application. This can be done by starting a provided terminal emulator package (such as Teraterm Pro or HyperTerminal), configuring it to communicate at 38400 bps, with no parity, eight data bits, and one stop bit, and attaching the correct serial port on the computer to the TI TMS320C54x EVM. A serial port is a 9-pin D-shell connector; it is located on the DSP EVM next to the power connector. Typically, there will be two matching D-shell connectors on your computer, often labeled COM1 and COM2; make sure you connect your serial cable to the right one!

Once you have started the terminal emulator, and the emulator has been correctly set to communicate with the DSP board, reload and rerun the thru6.asm[12] application. Once it is running, you should be able to communicate with the DSP by typing text into the terminal emulator's window. Characters that you type should be echoed back; this indicates that the DSP has received and retransmitted the characters. If the DSP is not connected properly or not running, nothing will be displayed as you type. If this happens, check the connections and the terminal emulator configuration, and try again. Due to a terminal emulation quirk, the "Enter" key does not work properly.

After you have verified that the EVM is communicating with the PC, close the terminal window.

3.2.1.2 Memory Maps and the Linker

Because the DSP has separate program and data spaces, you would expect for the program and data memory to be independent. However, for the DSP to run at its maximum efficiency, it needs to read its code from on-chip RAM instead of going off-chip; off-chip access requires a one- or two-cycle delay whenever an instruction is read. The 32K words of on-chip RAM, however, are a single memory block; we cannot map one part of it into program space and another part of it into data space. It is possible to configure the DSP so that the on-chip RAM is mapped into both program space and data space, allowing code to be executed from the

[10] http://cnx.rice.edu/modules/m10825/latest/thru6.asm
[11] http://cnx.rice.edu/modules/m10825/latest/thru6.asm
[12] http://cnx.rice.edu/modules/m10825/latest/thru6.asm

onboard memory and avoiding the extra delay. Figure 3.2 shows the DSP's memory map with the DSP's on-chip memory mapped into program space.

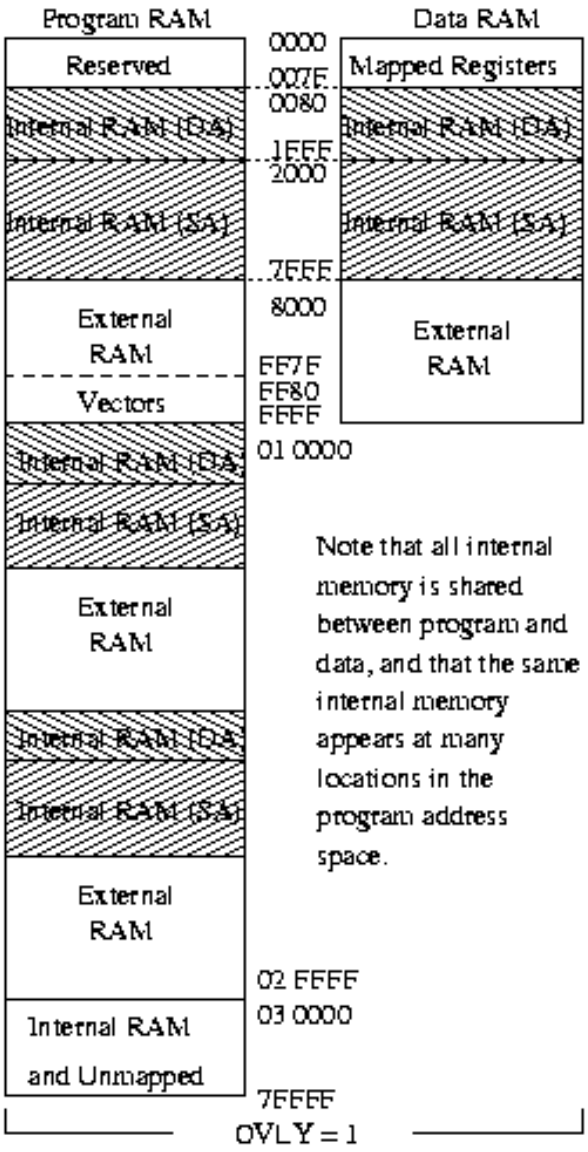

Figure 3.2: Hardware Memory Map

Because the same on-chip RAM is mapped into both program and data space, you must be careful not to overwrite your code with data or vice versa. To help you, the linker will place the code and data in different parts of the memory map. If you use the `.word` or `.space` directives to inform the linker of all of your usage of data memory, you will not accidentally try to reuse memory that has been used to store code or other data. (Remember that `.word` allocates one memory location and initializes it to the value given as its parameter. `.space 16*<words>` allocates `<words>` words of uninitialized storage.) Avoid using

syntaxes like `stm #2000h,AR3` to point auxiliary registers to specific memory locations directly, as you may accidentally overwrite important code or data. Instead, use syntaxes like `stm #hold,AR3`, where `hold` is a label for memory explicitly declared by `.word` or `.space` directives.

There are two types of internal memory on the TI TMS320C549 DSP: SARAM (Single Access RAM) and DARAM (Dual Access RAM). The first 8K of internal memory is DARAM; the next 24K is SARAM. The difference between these two types of memory is that while SARAM can only be read or written once in a cycle, DARAM can be read or written twice in a cycle. This is relevant because the TMS320C549 DSP core can access memory up to three times in each cycle: two accesses in Data RAM to read or write operands, and one access in Program RAM to fetch the next instruction. Both DARAM and SARAM are divided into "pages"; access to memory located in different "pages" will never conflict. If, however, two operands are fetched from the same "page" of SARAM (which is divided into 8K word pages: `2000h-3FFFh`, `4000h-5FFFFh`, and `6000h-7FFFh`) in the same cycle, a one-cycle stall will occur while the second memory location is accessed. Due to the pipeline, two memory accesses in the same instruction execute in different cycles. However, if two successive instructions access the same area of SARAM, a stall can occur.

Part of the SARAM (from `6000h` to `7FFFh`) is used for storing your program code; a small amount of SARAM below `6000h` is also used for storing the DSP's stack. Part of the DARAM (from `0800h` to `0FFFh`) is used for the input and output buffers and is also unavailable. Ensure that any code you write does not use any of these reserved sections of data memory. In addition, the core file reserves six locations in scratch-pad RAM (`060h` to `065h`); do not use these locations in your program code.

3.2.1.3 Sections and the Linker

You can use the `.text` directive to declare program code, and the `.data` directive to declare data. However, there are many more sections defined by the linker control file. Note that the core file uses memory in some of these sections.

You can place program code in the following sections using the `.sect` directive:

- `.text`: (`.sect ".text"`) SARAM between `6000h` and `7FFFh` (8192 words)
- `.etext`: (`.sect ".etext"`) External RAM between `8000h` and `FEFFh` (32,512 words) The test-vector version of the DSP core stores the test vectors in the `.etext` section.

You can place data in the following sections:

- `.data`: (`.sect ".data"`) DARAM between `1000h` and `1FFFh` (4096 words)
- `.sdata`: (`.sect ".sdata"`) SARAM between `2000h` and `5EFFh` (16,128 words)
- `.ldata`: (`.sect ".ldata"`) DARAM between `0080h` and `07FFh` (1,920 words)
- `.scratch`: (`.sect ".scratch"`) Scratchpad RAM between `0060h` and `007Fh` (32 words)
- `.edata`: (`.sect ".edata"`) External RAM between `8000h` and `FFFFh` (32,768 words) (Requires special initialization; if you need to use this memory, load and run the thru6.asm[13] application before you load your application to initialize the EVM properly.)

If you always use these sections to allocate data storage regions instead of setting pointers to arbitrary locations in memory, you will greatly reduce the chances of overwriting your program code or important data stored at other locations in memory. However, the linker cannot prevent your pointers from being incremented past the end of the memory areas you have allocated.

Figure 3.3 shows the memory regions and sections defined by the linker control file. Note that the sections defined in the linker control file but not listed above are reserved by the core file and should not be used.

[13] http://cnx.rice.edu/modules/m10825/latest/thru6.asm

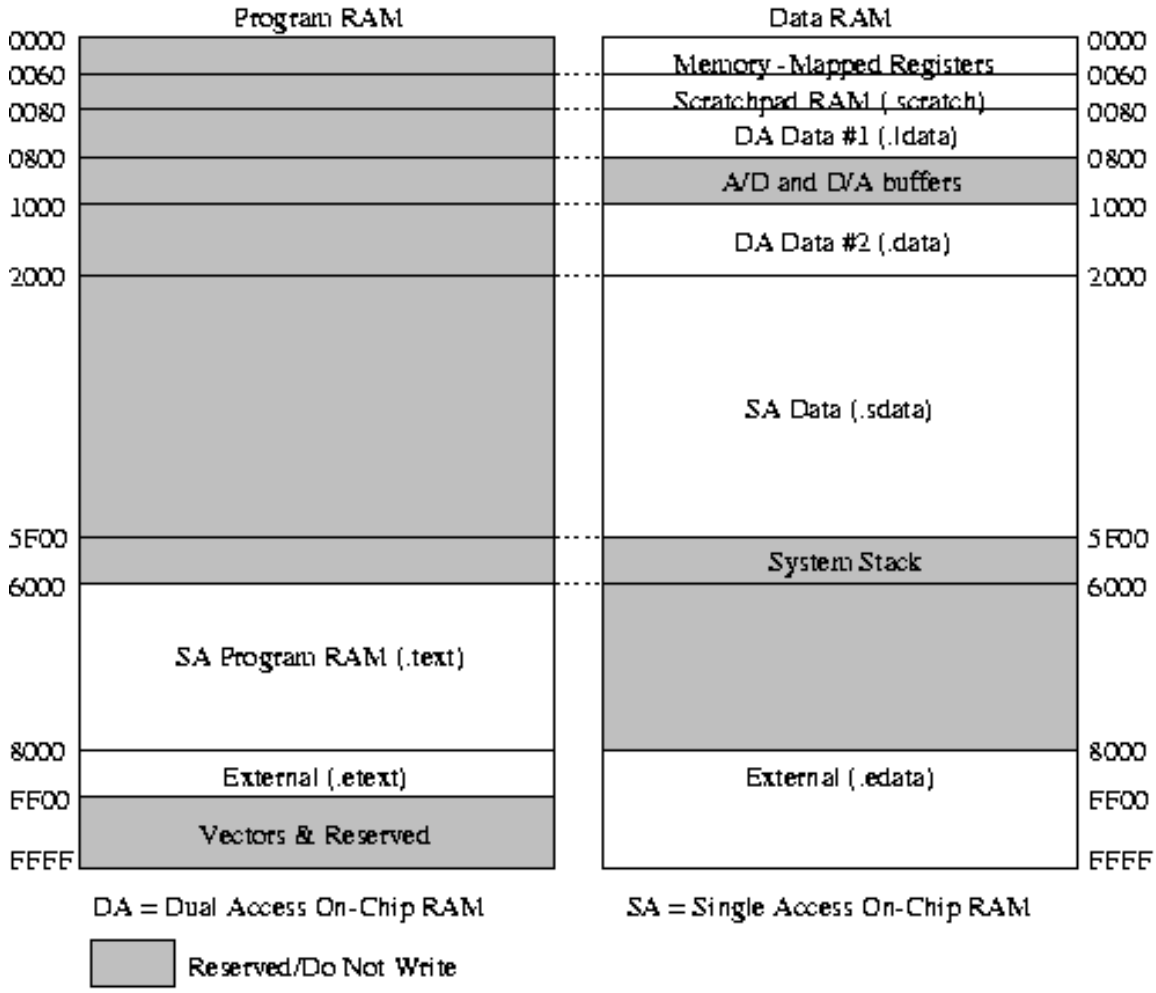

Figure 3.3: Linker Memory Map and Section Names

3.2.1.4 Using the Core File

To simplify discussion, we have split up the thru6.asm[14] file into two separate files for discussion. One, thru6a.asm[15] contains only the code for using the A/D and D/A converters on the six-channel surround board; the other, ser_echo.asm[16] contains only the code to send and receive data from the serial port.

[14] http://cnx.rice.edu/modules/m10825/latest/thru6.asm
[15] http://cnx.org/content/m10513/latest/thru6a.asm
[16] http://cnx.rice.edu/modules/m10821/latest/ser_echo.asm

3.2.1.4.1 Using the D/A and A/D converters

Here we will discuss thru6a.asm[17] , which is shown below. ser_echo.asm[18] is discussed in Core File: Serial Port Communication Between MATLAB and TI TMS320C54x (Section 3.2.3).

```
1 .copy "core.asm"
2
3 .sect ".text"
4 main
5 ; Your initialization goes here.
6
7 loop
8 ; Wait for a new block of 64 samples to come in
9 WAITDATA
10
11 ; BlockLen = the number of samples that come from WAITDATA (64)
12 stm #BlockLen-1, BRC ; Repeat BlockLen=64 times
13 rptb block-1 ; ...from here to the "block" label
14
15 ld      *AR6,16, A ; Receive ch1
16 mar *+AR6(2)               ; Rcv data is in every other word
17 ld      *AR6,16, B ; Receive ch2
18 mar *+AR6(2)               ; Rcv data is in every other word
19
20 ; Code to process samples goes here.
21
22 sth A, *AR7+ ; Store into output buffer, ch1
23 sth A, *AR7+ ; ch2
24 sth A, *AR7+ ; ch3
25
26 sth B, *AR7+ ; Store into output buffer, ch4
27 sth B, *AR7+ ; ch5
28 sth B, *AR7+ ; ch6
29
30 block
31 b loop
```

Line 1 copies in the core code, which initializes the six-channel board and the serial interface, provides the interface macros, and then jumps to "main" in your code. *Line 3* declares that what follows should be placed in the program-code area in internal memory.

On *Line 4*, we find the label "main". This is the entry point for your code; once the DSP has finished initializing the six-channel board and the serial port, the core file jumps to this label.

On *Line 9*, there is a call to `WAITDATA`. `WAITDATA` waits for the next block of 64 samples to arrive from the A/D. When it returns, a pointer to the samples captured by the A/D is returned in `AR6` (which can also be referred to as `pINBUF`); a pointer to the start of the output buffer is returned in `AR7` (also `pOUTBUF`). Note

[17] http://cnx.org/content/m10513/latest/thru6a.asm
[18] http://cnx.rice.edu/modules/m10821/latest/ser_echo.asm

that `WAITDATA` simply calls the `wait_fill` subroutine in the core file, which uses the `B` register internally, along with the `DP` register and the `TC` flag; therefore, do not expect the value of `B` to be preserved across the `WAITDATA` call.

Lines 12 and *13* set up a block repeat. `BlockLen` is set by the core code as the length of a block; the repeat instruction therefore repeats for every sample time. *Lines 15-18* retrieve one sample from each of the two channels; note that the received data is placed in every other memory location. *Lines 22-24* place the first input channel into the first three output channels, and *lines 26-28* place the second input channel into the last three output channels. Figure 3.1 shows the relationship between the channel numbers shown in the code and the inputs and outputs on the six-channel board.

Line 31 branches back to the top, where we wait for the next block to arrive and start over.

3.2.1.4.2 Using test vectors

A second version of the core file offers the same interface as the standard core file, but instead of reading input samples from the A/D converters on the six-channel board and sending output samples to the D/A converters, it reads and writes from test vectors generated in MATLAB.

Test vectors provide a method for testing your code with known input. Given this known input and the specifications of the system, we can use simulations to determine the expected output of the system. We can then compare the expected output with the measured output of the system. If the system is functioning properly, the expected output and measured output should match[19].

Testing your system with test vectors may seem silly in some cases, because you can see if simple filters work by looking at the output on the oscilloscope as you change the input frequency. However, they become more useful as you write more complicated code. With more complicated DSP algorithms, testing becomes more difficult; when you correct an error that results in one case not working, you may introduce an error that causes another case to work improperly. This may not be immediately visible if you simply look at the oscilloscope and function generator; the oscilloscope does not display the signal continuously and transient errors may be hidden. In addition, it is easy to forget to check all possible input frequencies by sweeping the function generator after making a change.

More importantly, the test vectors also allow you to test signals that cannot be generated or displayed with the oscilloscope and function generator. One important signal that cannot be generated or tested with the function generator and oscilloscope is the impulse function; there is no way to view the impulse response of a filter directly without using test vectors. The unit impulse represents a particularly good test vector because it is easy to compare the actual impulse response of a digital filter against the expected impulse response. Testing using the impulse response also exposes the entire range of digital frequencies, unlike testing using periodic waveforms generated by the function generator.

Lastly, testing using test vectors allows us to isolate the DSP from the analog input and output section. This is useful because the analog sections have some limitations, including imperfect anti-aliasing and anti-imaging filters. Testing using test vectors allows us to ensure that what we see is due only to the digital signal processing system, and not imperfections in the analog signal or electronics.

After generating a test vector in MATLAB, save it to a file that can be brought into your code using the MATLAB command `save_test_vector` (available as save_test_vector.m[20]):

```
≫ save_test_vector('testvect.asm',ch1_in,ch2_in);  % Save test vector
```

(where `ch1_in` and `ch2_in` are the input test vectors for input channel 1 and input channel 2; `ch2_in` can be omitted, in which case both channels of the test-vector input will have the same data.)

[19] Will the expected output and the actual output from the DSP system match perfectly? Why or why not?
[20] http://cnx.rice.edu/author/workgroups/90/m10017/save_test_vector.m

Next, modify your code to include the test-vector support code and the test-vector file you have created. This can be done by replacing the first line of the file (which is a linker directive to copy in `core.asm`) with two lines. Instead of:

```
.copy "core.asm"
```

use:

```
.copy "testvect.asm"
.copy "vectcore.asm"
```

Note that, as usual, the whitespace in front of the `.copy` directive is required. (Download vectcore.asm[21] into your work directory if you do not already have a copy.)

The test vectors occupy the `.etext` section of program memory between `08000h` and `0FEFFh`. If you do not use this section, it will not interfere with your program code or data. This memory block is large enough to hold a test vector of up to 4,000 elements. Both channels of input, and all six channels of output, are stored in each test vector element.

Now assemble and load the file, and reset and run as usual. After a few seconds, halt the DSP (using the Halt command under the Debug window) and verify that the DSP has halted at a branch statement that branches to itself: `spin b spin`.

Next, the test vector should be saved and loaded back into MATLAB. This is done by saving $6k$ memory elements (where k is the length of the test vector in samples, and the 6 corresponds to the six output channels) starting with location `08000h` in program memory. Do this by choosing `File->Data->Save...` in Code Composer, then entering the filename `output.dat` and pressing `Enter`. Next, enter `0x8000` in the Address field of the dialog box that appears, $6k$ in the Length field, and choosing "Program" from the drop-down menu next to "Page." (Always ensure that you use the correct length - six times the length of the test vector - when you save your results.)

Last, use the `read_vector` function (available as read_vector.m[22]) to read the saved test vector output into MATLAB. Do this using the following MATLAB command:

```
≫ [ch1, ch2, ch3, ch4, ch5, ch6] = read_vector('output.dat');
```

The MATLAB vectors `ch1` through `ch6` now contain the output of your program code in response to the input from the test vector.

3.2.2 Core File: Accessing External Memory on TI TMS320C54x[23]

3.2.2.1 Introduction

The TI DSP evaluation boards you use have a large amount of memory; in addition to the 32K words internal to the DSP, there are another 256K words of memory installed on the EVM board. For many exercises, the

[21] http://cnx.rice.edu/author/workgroups/90/m10017/vectcore.asm
[22] http://cnx.rice.edu/author/workgroups/90/m10017/read_vector.m
[23] This content is available online at <http://cnx.org/content/m10823/2.7/>.

data sets are small, and you worked with only the on-chip memory of the DSP and were not expected to consider how the use of memory impacted performance. However, the large delays often required in audio processing, for example, require that many thousands of samples be stored in memory. There is not enough memory on the DSP microprocessor itself to store a second or more of samples at a 44.1 kHz sample rate, so the off-chip memory must be used.

3.2.2.2 EVM Memory Maps

As you have seen, the TI TMS320C54x DSP has two separate memory spaces, called Program and Data. Usually, Program contains your assembled program, and Data contains data, but sometimes it may be convenient or more efficient to violate this convention. (For instance, the `firs` instruction requires filter coefficients in the Program address space.) The Data space is 64K long and is accessed using the 16-bit auxiliary registers (`ARx`). Although the Program space is normally accessed using 16-bit literals stored in your program code, the Program space is, in fact, significantly larger than 64K. Using special "extended addressing" instructions, the TI DSP can access up to 8192K-words of memory in the Program space. The extended addressing instructions include far calls and jumps that reset the full 23-bit program counter, as well as accumulator-addressed data-transfer instructions.

3.2.2.2.1 Internal and external memory

In many exercises, it is possible to store program instructions and data entirely in the DSP's on-chip ("internal") memory. This internal memory has several advantages over off-chip ("external") memory: it is much faster (data stored can be accessed without delay), and multiple reads and writes can access the DSP's on-chip memory simultaneously. However, many applications (including the audio delay effect of Using External Memory (Section 2.2.1)) require a data buffer too large to fit into the on-chip memory. For these large buffers, we must use the larger but slower external memory.

When writing programs that require large amounts of memory, use the internal memory to hold your code, filter coefficients, and any small buffers you need. External memory should be used for large buffers that you only access a few times per sample, like the delay buffer described in Audio Effects: Using External Memory (Section 2.2.1).

3.2.2.2.2 TMS320C549x DSP EVM memory maps

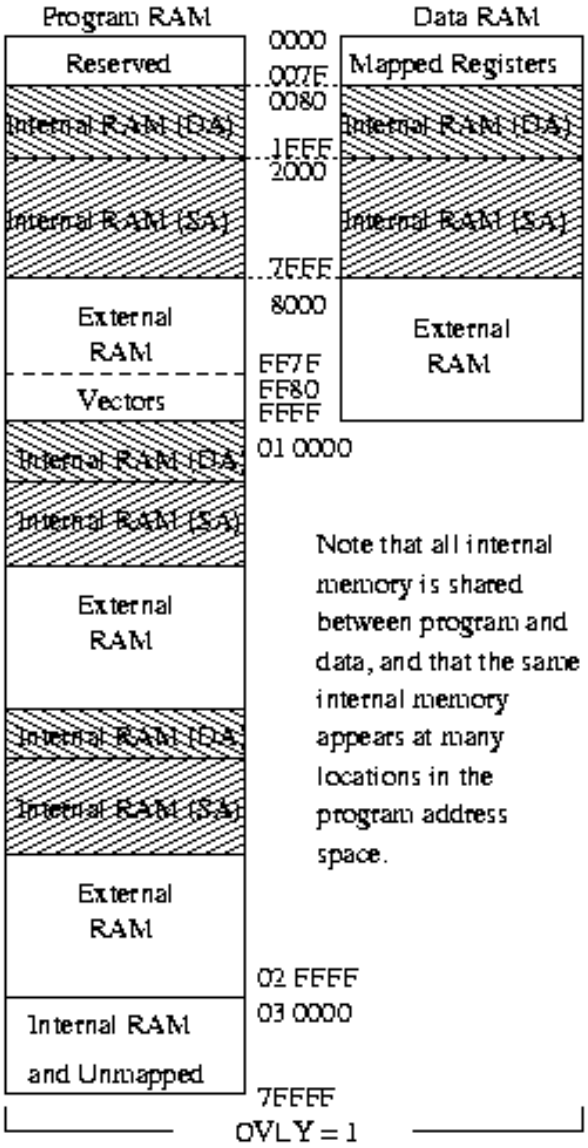

Figure 3.4: DSP EVM memory maps

As these memory maps show, the EVM's Data address space is addressed fully by the 16-bit auxiliary registers (`ARx`) and address-extension words and the mapping of Data memory is not affected by the `OVLY` bit. However, because the Program memory space is much larger than can be addressed by the 16-bit addressing register or the 16-bit literals stored in the program, it is split up into 64K (16-bit) pages by the hardware. Normal instructions, such as `call`, `firs`, and `mvpd` accept only 16-bit addresses, and can therefore only address the current "page" (usually address in the form `00xxxxh`, which corresponds to the addresses the linker uses for your program's code). To access the full 23-bit address space, the DSP offers special

accumulator-addressed load, store, and jump instructions.

Further complicating matters is the fact that the OVLY bit affects the mapping of the Program memory space. If you remember, before we load our DSP program, we have to change the PMST to FFE0h. We do this to set the OVLY bit in the PMST, which maps the internal memory into both the Program and Data spaces of the DSP. If OVLY is 1, the internal memory appears in both the Data and Program memory address space at locations 0080h to 07FFFh. Therefore, with OVLY set, anything written into Data memory below 07FFFh will overwrite a program stored in the same location. [24] In addition, copies of the internal memory also appear in the extended Program address space, occupying locations 0080h-7FFFh of each page. Therefore, with OVLY set, any addresses to Program memory locations in the form of xx0000h-xx7FFFh reference internal memory.

When OVLY is zero, internal memory is not mapped into the Program space at all; in this case, the Program space includes only external memory. In this mode, all 192K words of external Program RAM are accessible, although several wait states will be required for accessing each item of memory. In the overlay mode, only addresses in the ranges of 08000h-0FF00h, 1800h-1FFFFh, and 28000h-2FFFFh are available to store your data buffers; the remaining addresses are unmapped or map to the on-chip RAM.

To escape this confusion and allow the full 192K-words of external Program RAM to be used for your data buffers, the core file provides mechanisms for manipulating the PMST indirectly. Instead of accessing the external Program RAM directly, we can use the special macros to access the RAM that is normally "hidden" by the internal memory. This allows us to use the full range of external memory available: addresses 000000h-00FF00h and 010000h-02FFFF. However, since addresses 00FF00h-00FFFFh are reserved by the core file, you must be careful not to write to addresses in this range.

3.2.2.3 Accessing Extended Program RAM

The core file provides two macros for accessing data stored in the external Program RAM: READPROG and WRITPROG. These macros allow the processor to copy data between data memory and external Program memory. Both macros address external Program memory using the value in accumulator A. READPROG reads data from the external Program memory location pointed to by A and writes it to the data memory location pointed to by AR1. WRITPROG reads data from the memory location pointed to by AR1 and writes it to the location in external Program RAM specified by accumulator A. Both macros take one parameter, a count; specifying 1 reads or writes one word from external memory, and specifying some other number n transfers n words starting at the locations pointed to by A and AR1. AR1 is left pointing at the word after the last word read or written; no other registers are modified.

For instance, the following code fragment loads the value contained in memory location 023456h into the location 0064h in data memory using the READPROG macro:

```
1    stm #64h,AR1 ; load 64h into AR1
2    ld #02h,16,A ; load 02h in high part of A
3    adds  #3456h,A ; fill in low part of A
4 ; A contains 023456h
5   READPROG 1  ; read from 023456h in external Program RAM
6 ; into *AR1 in Data RAM
```

The WRITPROG macro can be used similarly to write into extended Program RAM:

[24] This is why the memory allocated for your program - 6000h-7FFFh - does not overlap with any of the space allocated for the data segments.

```
1   stm #64h,AR1 ; load 64h into AR1
2 ld #02h,16,A ; load 02h in high part of A
3 adds #3456h,A ; fill in low part of A
4 ; A contains 023456h
5 WRITPROG 1 ; write from *AR1 in Data RAM to
6 ; 023456h in external Program RAM
```

Note that Code Composer will not display or allow you to change the contents of the external Program RAM on the memory-dump or disassembly screen, though you can view or change it indirectly by watching the effects of the `READPROG` and `WRITPROG` macros on data memory.

3.2.3 Core File: Serial Port Communication Between MATLAB and TI TMS320C54x[25]

3.2.3.1 Using the Serial Port

The core file supports the serial port installed on the TI TMS320C54x DSP. The serial port on the EVM is connected with a cable to COM2 on the PC. Before jumping to your code, the core file initializes the EVM's serial port to 38,400 bits per second (bps) with no parity, eight data bits, and one stop bit (but it may be necessary to restart the DSP completely if the serial port does not work.) It then accepts characters received from the PC by the **UART** (**Universal Asynchronous Receiver/Transmitter**) and buffers them in memory until your code retrieves them. It also can accept a block of bytes to transmit and send them to the UART in sequence.

Two macros are used to control the serial port: `READSER` and `WRITSER`. Both accept one parameter. `READSER n` reads up to n characters from the serial input buffer (the data coming from the PC) and places them in memory starting at `*AR3`. (`AR3` is left pointing one past the last memory location written.) The actual number of characters read is left in `AR1`. If `AR1` is zero, then no characters were available in the input buffer.

`WRITSER n` adds n characters starting at `*AR3` to the serial output buffer; in other words, it queues them to be sent to the PC. `AR3` is left pointing one location after the last memory location read.

Note that `READSER` and `WRITSER` modify registers `AR0`, `AR1`, `AR2`, `AR3`, and `BK`, as well as the flag `TC`. Be sure you restore these registers after calling `READSER` and `WRITSER` if you need them later in your code.

Note also that the core file allows up to 126 characters to be stored in the input and output buffers. Neither the DSP hardware nor the core file protect against serial-buffer overflows, so you must be careful not to allow the input and output buffers to overflow. (The length of the buffers can be changed by editing `ser_rxlen` and `ser_txlen` values in core.asm[26] .) The buffers are 127 characters long; however, the code cannot distinguish between a completely-full and completely-empty buffer. Therefore, only 126 characters can be stored in the buffers.

It is easy to check if the input or output buffers in memory are empty. The input buffer can be checked by comparing the values stored in the memory locations `srx_head` and `srx_tail`; if both memory locations hold the same value, the input buffer is empty. Likewise, the output buffer can be checked by comparing the values stored in memory locations `stx_head` and `stx_tail`. The number of characters in the buffer can be computed by subtracting the head pointer from the tail pointer; add the length of the buffer (normally 127) if the resulting distance is negative.

[25] This content is available online at <http://cnx.org/content/m10821/2.7/>.
[26] http://cnx.rice.edu/author/workgroups/90/m10017/core.asm

The following example shows the minimal amount of code necessary to echo received data back through the serial port. It is available as ser_echo.asm[27] .

```
1 .copy "core.asm"
2
3 .sect ".data"
4  hold .word 0
5
6 .sect ".text"
7  main
8 stm #hold,AR3       ; Read to hold location
9
10 READSER 1 ; Read one byte from serial port
11
12 cmpm AR1,#1 ; Did we get a character?
13 bc main,NTC ; if not, branch back to start
14
15 stm #hold,AR3 ; Write from hold location
16 WRITSER 1 ; ... one byte
17
18 b main
```

Line 8 sets `AR3` to point to the location `hold` so that `READSER` will store serial data there. On *Line 9*, `READSER 1` reads one serial byte into `hold`; the byte is placed in the low-order bits of the word, and the high-order bits are zeroed. If a byte was read, `AR1` will be set to 1. `AR1` is checked in *Line 12*; *Line 13* branches back to the top if no byte was read. Otherwise, `AR3` is reset to `hold` (since `READSER` moved it), then on *Line 16*, `WRITSER` sends the word received. Finally, *Line 18* branches back to the start to receive another character.

3.2.3.2 Using MATLAB to Control the DSP

MATLAB allows you to create a visual interface with standard **graphical user-interface** (**GUI**) controls such as sliders, checkboxes, and radio buttons to call MATLAB scripts. The following scripts can be used to create a sample interface:

- ser_set.m[28] : Initializes the serial port and user interface
- wrt_slid.m[29] : Called when sliders are moved to send new data

3.2.3.2.1 Creating a MATLAB user interface

The following code (ser_set.m[30]) initializes the serial port COM2, then creates a minimal user interface consisting of three sliders.

[27] http://cnx.org/content/m10821/latest/ser_echo.asm
[28] http://cnx.rice.edu/author/workgroups/90/m10821/ser_set.m
[29] http://cnx.rice.edu/author/workgroups/90/m10821/wrt_slid.m
[30] http://cnx.rice.edu/author/workgroups/90/m10821/ser_set.m

```
1 % ser_set: Initialize serial port and create three sliders
2
3 % Set serial port mode
4 !mode com2:38400,n,8,1
5
6 % open a blank figure for the slider
7 Fig = figure(1);
8
9 % open sliders
10
11 %  first slider
12 sld1 = uicontrol(Fig,'units','normal','pos',[.2,.7,.5,.05],...
13  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
14
15 %  second slider
16 sld2 = uicontrol(Fig,'units','normal','pos',[.2,.5,.5,.05],...
17  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
18
19 %  third slider
20 sld3 = uicontrol(Fig,'units','normal','pos',[.2,.3,.5,.05],...
21  'style','slider','value',4,'max',254,'min',0,'callback','wrt_slid');
```

Line 4 of this code uses the Windows `mode` command to set up serial port COM2 (which is connected to the DSP) to match the serial port settings on the DSP evaluation board: 38,400 bps, no parity, eight data bits, and one stop bit. *Line 7* then creates a new MATLAB figure for the controls; this prevents the controls from being overlaid on any graph you may have already created.

Lines 12 through the end create the three sliders for the user interface. Several parameters are used to specify the behavior of each slider. The first parameter, `Fig`, tells the slider to create itself in the window we created in *Line 7*. The rest of the parameters are property/value pairs:

units - `normal` tells MATLAB to use positioning relative to the window boundaries.
pos - Tells MATLAB where to place the control.
style - Tells MATLAB what type of control to place. `slider` creates a slider control.
value - Tells MATLAB the default value for the control.
max - Tells MATLAB the maximum value for the control.
min - Tells MATLAB the minimum value for the control.
callback - Tells MATLAB what script to call when the control is manipulated. wrt_slid.m[31] is a MATLAB file that reads the values of the sliders and sends them to the DSP via the serial port.

3.2.3.2.1.1 User-interface callback function

Every time a slider is moved, the file wrt_slid.m[32] is called:

```
1 % wrt_slid: write values of sliders out to com port
2
3 % open com port for data transfer
```

[31] http://cnx.rice.edu/author/workgroups/90/m10821/wrt_slid.m
[32] http://cnx.rice.edu/author/workgroups/90/m10821/wrt_slid.m

```
4 fid = fopen('com2:','w');
5
6 % send data from each slider
7 v = round(get(sld1,'value'));
8 fwrite(fid,v,'uint8');
9
10 v = round(get(sld2,'value'));
11 fwrite(fid,v,'uint8');
12
13 v = round(get(sld3,'value'));
14 fwrite(fid,v,'uint8');
15
16 % send reset pulse
17 fwrite(fid,255,'uint8');
18
19 % close com port connection
20 fclose(fid);
```

Line 4 of wrt_slid.m[33] opens COM2 for writing. (It has already been initialized by ser_set.m[34] .) Then *Line 7* reads the value of the first slider using MATLAB's `get` function to retrieve the `value` property. The value is then rounded off to create an integer, and the integer is sent as an 8-bit quantity to the DSP in *Line 8*. (The number that is sent at this step will appear when the serial port is read with `READSER` in your code.) Then the other two sliders are sent in the same way.

Line 17 sends `0xFF` (255) to the DSP, which can be used to indicate that the three previously-transmitted values represent a complete set of data points. Your code can check for the value 255 to detect and correct synchronization errors.

Line 20 closes the serial port. Note that MATLAB buffers the data being transmitted, and data is often not sent until the serial port is closed. Make sure you close the port after writing a data block to the serial port.

3.3 Code Composer

3.3.1 Debugging and Troubleshooting in Code Composer[35]

3.3.1.1 Introduction

Code Composer provides a rich debugging environment that allows you to step through your code, set breakpoints, and examine registers as your code executes. This document provides a brief introduction to some of these debugging features.

3.3.1.2 Debugging Code

3.3.1.2.1 Controlling program flow

Breakpoints are points in the code where execution is stopped and control of the DSP is returned to the debugger, allowing you to view the contents of registers and memory. Breakpoints can be activated or

[33] http://cnx.rice.edu/author/workgroups/90/m10821/wrt_slid.m
[34] http://cnx.rice.edu/author/workgroups/90/m10821/ser_set.m
[35] This content is available online at <http://cnx.org/content/m10522/2.9/>.

deactivated by double-clicking on any line of code in the disassembly window.[36]

You may also want to step through your program code, executing one line at a time, to follow branches and watch memory change with the results of calculations. This can be done by choosing the "Step Into" or "Step Over" menu options from the "Debug" pull-down menu. (Unlike "Step Over," "Step Into" traces subroutine calls caused by "call" opcodes.)

Like most DSPs, the DSP we are using is a pipelined processor, which means that instructions execute in several stages over several clock cycles. Unfortunately, our debugger does not "flush" the pipeline of all current instructions when it halts your program; i.e., the DSP does not execute all remaining stages of instructions. As a consequence, when a program halts, the register values shown in the register and memory windows may not actually the last values written. Often, the values shown correspond to values written several cycles before the current instruction. If it is necessary to know the exact contents of the registers at any particular point in the program flow, simply insert three or more `nop` (no operation) instructions into your program after the instruction in question. Then, to debug, execute the instruction in question and the `nop` instructions that follow; this will flush the pipeline.

You can choose the "Run Free" option from the "Debug" pull-down menu to allow the your code to run freely, ignoring any breakpoints. The code will continue running until explicitly halted with the "Halt" command.

Note that stopping and restarting execution sometimes confuses the A/D and D/A converters on the six-channel surround-sound board. If this happens, the output will generally go to zero or become completely unrelated to the input signal. This can be fixed by simply resetting the DSP and starting your code from the beginning.

The bar on the left-hand side of the Code Composer Studio window contains shortcuts for many of the commands in the Debug menu.

> NOTE: Practice setting breakpoints in your program code and single-stepping by setting a breakpoint after the `WAITDATA` call and tracing through the program flow for several iterations of the FIR filter code. What code does the `WAITDATA` call correspond to in the disassembly window?

3.3.1.3 Troubleshooting

The DSP boards can behave unexpectedly. If there is no output, try the following (from less to more drastic):

- Use the Debug menu to halt and reset the DSP, verify that the `PMST` is set to `0xFFE0`, reload the code, reset the DSP, and restart the code.
- Press the "Reset" button on the DSP evaluation board, then use the Code Composer Studio menus to halt, reset the DSP, verify the `PMST`, reload, reset the DSP again, and restart your code.
- Close Code Composer Studio, then **power-cycle** the DSP by unplugging power to the DSP board, waiting five seconds, and plugging it back in. Then restart Code Composer Studio. You will need to reset the `PMST` to `0xFFE0`, then reload, reset the DSP, and execute your code.

If code does not load correctly, close Code Composer Studio and power-cycle the DSP.

If problems persist after power-cycling the DSP, ensure that the DSP board is functioning properly by executing previously verified code. Do not forget to set the `PMST` and to reset the DSP from the Code Composer Studio menu.

If you try all of these steps and still see problems, ask a teaching assistant for help.

[36] They can also be set by pressing F9 on a line in the source-file window. However, verify that the breakpoint appears at the corresponding location in the disassembly window if you do this; there have been problems with breakpoints being set inaccurately by this method in the past.

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] 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.

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

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

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

[11] 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: *DSP Laboratory with TI TMS320C54x*
Edited by: Douglas L. Jones
URL: http://cnx.org/content/col10078/1.2/
License: http://creativecommons.org/licenses/by/1.0

Module: "Preface for U of I DSP Laboratory"
By: Douglas L. Jones
URL: http://cnx.org/content/m10681/2.12/
Pages: 1-2
Copyright: Douglas L. Jones
License: http://creativecommons.org/licenses/by/1.0

Module: "DSP Development Environment: Introductory Exercise for TI TMS320C54x"
Used here as: "Lab 0: Hardware Introduction"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10017/2.24/
Pages: 5-11
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
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: 13-15
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"
Used here as: "Lab 1: Lab"
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/m10023/2.19/
Pages: 16-20
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: 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: 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: 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: 23
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: 24
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: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade
URL: http://cnx.org/content/m10621/2.9/
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: 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: 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-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: 28-29
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: 30
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: Exercise on TI TMS320C54x"
Used here as: "Lab 3: Lab"
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/m10624/2.8/
Page: 31
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: "Spectrum Analyzer: Introduction to Fast Fourier Transform"
Used here as: "Lab 4: 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/m10027/2.19/
Page: 33
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: "Spectrum Analyzer: MATLAB Exercise"
Used here as: "Lab 4: Prelab"
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/m10625/2.8/
Page: 34
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: "Spectrum Analyzer: FFT Exercise on TI TMS320C54x"
Used here as: "Lab 4: Lab"
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/m10626/2.8/
Pages: 35-37
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: "Digital Transmitter: Introduction to Quadrature Phase-Shift Keying"
Used here as: "Lab 5: 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/m10042/2.19/
Pages: 39-41
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: "Digital Transmitter: MATLAB Exercise for Quadrature Phase-Shift Keying"
Used here as: "Lab 5: Prelab"
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/m10627/2.9/
Pages: 42-43
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: "Digital Transmitter: Optimization Exercise with QPSK on TI TMS320C54x"
Used here as: "Lab 5: Lab"
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/m10628/2.8/
Page: 44
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: "Vector Signal Analyzer: Testing a QPSK Transmitter on Hewlett Packard 89440A"
Used here as: "Lab 5: Testing"
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/m10667/2.6/
Page: 45
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: "Digital Receivers: Symbol-Timing Recovery for QPSK"
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/m10485/2.14/
Pages: 47-58
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: "Digital Receiver: Carrier Recovery"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10478/2.16/
Pages: 58-62

Module: "Audio Effects: Using External Memory"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10480/2.17/
Pages: 62-64

Module: "Audio Effects: Real-Time Control with the Serial Port"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10483/2.24/
Pages: 64-65

Module: "Surround Sound: Passive Encoding and Decoding"
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/m10484/2.13/
Pages: 65-67

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: 67-70

Module: "Adaptive Filtering: LMS Algorithm"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10481/2.14/
Pages: 71-73

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: 73-76
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/
Pages: 76-77
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: 77-78
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: "Two's Complement and Fractional Arithmetic for 16-bit Processors"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs, Jason Laska
URL: http://cnx.org/content/m10808/2.9/
Pages: 79-81
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs, Jason Laska
License: http://creativecommons.org/licenses/by/1.0

Module: "Addressing Modes for TI TMS320C54x"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10806/2.7/
Pages: 81-86
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: "Core File: Introduction to Six-Channel Board for TI EVM320C54"
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10513/2.13/
Pages: 86-93
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

DSP Laboratory with TI TMS320C54x
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.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.