Digital Signal Processing Laboratory (ECE 420 55x)

Collection Editors:
  Thomas Shen
  David Jun
Digital Signal Processing Laboratory (ECE 420 55x)

Collection Editors:
Thomas Shen
David Jun

Authors:
Swaroop Appadwedula
Matthew Berry
Spencer Brady
Mark Butala
Richard Cantzler
Mark Haun
Jake Janovetz
Douglas L. Jones
David Jun

Matt Kleffner
Michael Kramer
Arjun Kulothungun
Jason Laska
Robert Morrison
Dima Moussa
Daniel Sachs
Thomas Shen
Brian Wade

Online:
< http://cnx.org/content/col10397/1.10/ >

CONNEXIONS
Rice University, Houston, Texas
# Table of Contents

## 1 Weekly Labs
1.1 Lab 0 ................................................................. 1
1.2 Lab 1 ................................................................. 8
1.3 Lab 2 ................................................................. 17
1.4 Lab 3 ................................................................. 20
1.5 Lab 4 ................................................................. 26
1.6 Lab 5 ................................................................. 55
Solutions ................................................................. ??

## 2 Project Labs
2.1 Digital Receiver ....................................................... 61
2.2 Audio Effects ......................................................... 76
2.3 Surround Sound ....................................................... 78
2.4 Adaptive Filtering .................................................... 84
2.5 Speech Processing ..................................................... 86
2.6 Video Processing ..................................................... 91
Solutions ................................................................. ??

## 3 General References
3.1 Processor ............................................................. 117
3.2 RTDX (Real Time Data Transfer) .................................. 124
3.3 Miscellaneous ......................................................... 130
Solutions ................................................................. ??

## 4 Other Labs
4.1 UIUC DSP Lab Fall 2002 Archive .................................. 135
4.2 UIUC DSP Lab Spring 2003 Archive .............................. 136
4.3 UIUC DSP Lab Spring 2004 Archive .............................. 137
4.4 UIUC DSP Lab Fall 2004 Archive .................................. 138
Solutions ................................................................. ??

## Bibliography
................................................................. 139

## Index
................................................................. 140

## Attributions
................................................................. 144
Chapter 1

Weekly Labs

1.1 Lab 0

1.1.1 Lab 0: Hardware Introduction

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 TMS320C5510A-200 digital signal processor chip mounted on a Spectrum Digital TMS320VC5510 evaluation board. The DUAL3006, a daughtercard produced by Educational DSP, is mounted on the external peripheral interface of the board to enable four-input/four-output capability. The evaluation module should be connected to a PC running Windows and will be controlled using the PC application Code Composer Studio v4.0, a debugger and development environment. We will be using a 48kHz sample rate. The DSP board can also communicate with user code or a terminal emulator running on the PC via a USB interface.

NOTE: If you are not using Code Composer Studio v4.0, the instructions on this page do not apply. Please see the revision history of this module for instructions if using CCS v3.x

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). The figure may not be up to date, so ask a TA if you need help.

\footnote{This content is available online at \url{<http://cnx.org/content/m13811/1.16/>}.}
Note that with this configuration, you will have only one signal going into the DSP board and two signals coming out. The output on channel 1 is the filtered input signal, and the output on channel 2 is the unfiltered input signal. This allows you to view the raw input and filtered output simultaneously on the oscilloscope. Turn on the function generator and the oscilloscope.

1.1.1.2.2 Step 2: Log in

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

When you log in, two shared networked drives should be mapped to the computer: the W: drive, which contains your own private network work directory, and the V: drive, where the necessary files for ECE 420 are stored. Be sure to save any files that you use for the course to the W: drive. Temporary files may be stored in the C:\Users\netID\workspace directory; however, since files stored on the C: drive are local to each computer, and may be erased at any time, do not store course files on the C: drive. On the V: drive, the directory V:\ece420\55x\ccs4 contains the files necessary to assemble and test code on the TI DSP evaluation boards.

Although you may want to work exclusively in one or the other of lab-partners’ network account, you should be sure that both partners have copies of the lab assignment assembly code.
WARNING: Not having the assembly code during a quiz because "it's on my partner's account" is NOT a valid excuse!

For copying between partners’ directory on W: or for working outside the lab, access to your files is available. See http://www.ece.illinois.edu/cts/storage/² for instructions on how to set that up.

1.1.1.3 The Development Environment

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

1.1.1.3.1 Step 3: Assemble filter code

Setup Code Composer

By default, a shortcut to CCS is available by going to Start > All Programs > Texas Instruments > Code Composer Studio v4. When CCS starts for the first time, Workspace Launcher will start because it will need to set up your workspace.

Create or make sure you have the following directory: W:\workspace\ECE420. In Workspace Launcher, hit Browse..., navigate to this folder, and make sure to check "Use this as the default and do not ask again".

NOTE: In the future, verify that you are in the correct workspace by going to File > Switch Workspace...

Import Project

In CCS, go to View > C/C++ Projects. A panel will pop up on the left side of the window. Right-click somewhere in this panel and choose Import...

1. Expand "CCS" and choose "Existing CCS/CCE Eclipse Project"
2. Hit Next and browse to W:\ece420\55x\ccs4\filter
3. Check "Copy projects into workspace"

Build Project

Once the project is copied into your workspace, we can proceed to build it by selecting Project > Build Active Project. In a successful build, there will be zero errors and maybe a few warnings and remarks. The output file will be placed in a Debug folder within the project’s directory. In this example, the executable binary code will be located at .\Debug\filter.out.

1.1.1.3.2 Step 4: Verify filter execution

Connect to the DSP

1. Select View > Target Configurations
2. In the panel that comes up, expand Projects > filter
3. Right-click on dsk5510.cxxml and select "Launch Selected Configuration"

Once CCS connects to the DSP, select Target > Connect Target

Load and Run Program

Now, load your assembled filter file (filter.out) onto the DSP by selecting Target > Load Program. Finally, execute the code by selecting Target > Run.

²See the file at <http://cnx.org/content/m13811/latest/http://www.ece.illinois.edu/cts/storage/>
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 48 kHz and have an anti-aliasing filter and an anti-imaging filter, respectively, that in the ideal case would eliminate frequency content above 24 kHz.

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? The converters on the board are also AC coupled and cannot pass DC signals.

Configure Function Generator and Oscilloscope

Set the amplitude on the function generator to 1.0 V peak-to-peak and the pulse shape to sinusoidal. Adjust the function generator so that it expects a high impedance load. The sequence of button presses to accomplish this on the function generator in the lab is Shift -> Enter -> Right -> Right -> Right -> Down -> Down -> Right -> Enter.

Make sure the oscilloscope is set to 1M impedance. This can be accomplished by pressing channel 1 or 2 and then selecting 1M Ohm from the Imped menu.

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 48 kHz?

Characterize Filter Response

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. There is a second set of filter coefficients already in your project folder. In Windows Explorer, navigate to W:\workspace\ece420\filter and do the following:

- Rename coef.asm to coef1.asm
- Rename coef2.asm to coef.asm

Repeat the assembly and testing process with the new filter by repeating steps required to build (Step 3 (Section 1.1.1.3.1: Step 3: Assemble filter code)) and execute (Step 4 (Section 1.1.1.3.2: Step 4: Verify filter execution)) the code.

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.

View Coefficients in DSP Memory

The FIR filter coefficients included in the file coef.asm are stored in memory on the DSP. To view the contents of the DSP memory, first suspend any running program by going to Target > Halt and then select View > Memory.

In the panel that comes up, there is a text box for you to type in the name of the variable that you are interested in viewing. This variable name is actually a mnemonic for a memory address. In the case of our coefficients, the mnemonic coef1 is used to point to the starting address of our coefficients. The memory content can be displayed in many different formats. In the drop-down box, choose 16-Bit Signed Int.
NOTE: Make sure you understand where the coef1 label comes from. [Hint:] Select View > C/C++ Projects and double click on filtercode.asm to view the source code.

In this example, the filter coefficients are placed in memory in decreasing order; that is, the last coefficient, \( h[7] \), is at location coef1 and the first coefficient, \( h[0] \), is stored at coef1+7.

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

\[
\begin{align*}
\gg & \ h = [-10, 20, -10]
\gg & \ \text{freqz}(h)
\end{align*}
\]

Note that you will have to enter eight values, the contents of memory locations coef1 through coef1+7, into the coefficient vector, \( h \).

TIP: You must divide the coefficients by 32768. Where does this scaling factor come from?

How 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 \( h = \text{gen_filt}; \) at a MATLAB prompt. Then save this vector of filter coefficients by typing \( \text{save coef('coef.asm',fliplr(h));} \) Make sure you save the file in your own directory. (The scripts that perform these functions are available as \text{gen_filt.m}\(^3\) and \text{save_coef.m}\(^4\). They are also available at V:/ece420/55x/m_files)

The \text{save_coef} MATLAB script will save the coefficients of the vector \( h \) into the named file, which in this case is \text{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 coef1 through coef1+7 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 coef1 through coef1+7 update accordingly.

1.1.1.3.6 Step 8: Modify filter coefficients in memory

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

NOTE: The DSP must be in a halted state in order to overwrite the memory.

Change the contents of memory locations coef1 through coef1+7 such that the coefficients implement a scale and delay filter with impulse response:

\[
h[n] = 81926[n - 4]
\]  

\(^3\)See the file at <http://cnx.org/content/m38811/latest/gen_filt.m>
\(^4\)See the file at <http://cnx.org/content/m38811/latest/save_coef.m>
Note that the DSP interprets the integer value of 8192 as a fractional number by dividing the integer by 32,768 (the largest integer possible in a 16-bit two's complement register). The result is an output that is delayed by four samples and scaled by a factor of $\frac{1}{4}$. More information on the DSP's interpretation of numbers appears in Two's Complement and Fractional Arithmetic for 16-bit Processors (Section 3.1.1).

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

**TIP:** Take advantage of the "Quick Measure" feature on the oscilloscope!

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

### 1.1.1.3.7 Step 9: Test-vector simulation

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

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

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

```matlab
≫ 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.

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

The `main.c` file needs to be told to take input from memory on the DSP. Fortunately, the changes have already been made in the files. The test vector is stored in a block of memory on the DSP just like other variables. 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 all four channels of input and from four channels of output.

To run your program with test vectors, you will need to modify `main.c` as well as `filtercode.asm`. Both are simply text files and can be edited using the editor of your preference, including WordPad, Emacs, and VI. (The changes have already been made, but please visually verify the changes are there.) Within `main.c`, uncomment the `#define FILE_INPUT` line so that your program will rewrite input from the A/D with the test vector you specified and then save the output into a block of memory.

In `filtercode.asm`, uncomment the `.copy "testvect.asm"` line. Make sure this Matlab generated file is in the same directory as `filtercode.asm`.

---

5 See the file at <http://cnx.org/content/m13811/latest/save_test_vector.m>

6 See the file at <http://cnx.org/content/m13811/latest/sweep.m>
NOTE: In TI assembly, the semi-colon ; signifies a comment.

These changes will copy in the test vector. 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 Target menu). How many seconds do you think it should take?

Saving DSP Memory to File
Next, we will save the test output file and load it back into MATLAB. We are interested in the first 500 output samples, starting at address tv_outbuf in Data memory. There are four output channels and the memory is interleaved in time. Therefore, we will have to collect 2000 (4 channels time 500 samples) memory elements.

• Select View > Memory
• Click on the "Save" icon, a green square with an angled arrow (top left in the Memory panel)
• Name the file output.dat and save filetype as TI data format
• On the next screen, use the following options:
  - format: hex
  - start address: tv_outbuf
  - memory page: data
  - length: 2000

Last, use the read_vector (available as read_vector.m7) function to read the saved result into MATLAB. Do this using the following MATLAB command:

> [ch1,ch2,ch3,ch4] = read_vector('output.dat');

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

NOTE: Because of quantization error introduced in saving the test vector for the 16-bit memory of the DSP, the vector ch2 will not be identical to the MATLAB generated test vector.

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

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

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

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

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

7See the file at <http://cnx.org/content/m13811/latest/read_vector.m>
1.1.1.3.8 Step 10: Closing Down

Before exiting Code Composer, make sure to disconnect properly from the DSP:

- Halt any program running on the DSP (Target > Halt)
- Disconnect from the DSP (Target > Connect will toggle between connecting and disconnecting)

Finally, make sure to return all of the cables to the wall rack.

1.2 Lab 1

1.2.1 Lab 1: Prelab

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 TMS320C55x (Section 3.1.2), and the Mnemonic Instruction Set manual for help.

```
1 FIR_len .set 3
2
3 ; Assume:
4 ; BK03 = FIR_len
5 ; firStateIndex is stored at memory location 1008h
6 ; AR2 = 1000h
7 ; AR3 = 1004h
8 ; FRCT = 1
9
10 BSET AR3LC ; sets circular addressing for AR3
11 mov mmap(AR3), BSA23
12 mov #firStateIndex, AR4
13 mov *AR4, AR3
14 mov LO(AC0), *AR3+
15 mov #0, AC0
16 rpt #(FIR_len-1)
17 macm *AR2+, *AR3+, AC0
```

Anything following a ";" is considered a comment. In this case, the comments indicate the contents of the auxiliary registers, the BK03 register, and the address registers before the execution of the first instruction, `mov`. The line `FIR_len .set 3` defines the name `FIR_len` as equal to 3. The BK03 register contains the length of the circular buffer we want to use for auxiliary register 0 through 3. The `BSET AR3LC` modifies the increment operator + so that it behaves as a circular buffer. This means circular addressing will be used for AR3. Refer to Section 6.11 of the CPU Reference Guide for help on circular addressing.

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.

---

*This content is available online at <http://cnx.org/content/m13810/1.5/>.*
### Data Memory Assignment (before execution)

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000h</td>
<td>1000h</td>
</tr>
<tr>
<td>1001h</td>
<td>0000h</td>
</tr>
<tr>
<td>1002h</td>
<td>4000h</td>
</tr>
<tr>
<td>1004h</td>
<td>1000h</td>
</tr>
<tr>
<td>1005h</td>
<td>1000h</td>
</tr>
<tr>
<td>1006h</td>
<td>4000h</td>
</tr>
<tr>
<td>1007h</td>
<td>1000h</td>
</tr>
<tr>
<td>1008h</td>
<td>0000h</td>
</tr>
</tbody>
</table>

**Figure 1.2:** Data Memory Assignment (before execution)

After familiarizing yourself with the *mov, rpt, and macm* instructions, step through each line of code and record the values of the accumulator AC0 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 in.
When working through the exercise, take into account that the accumulator AC0 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.) Multiples performed by the ALU (via the `macm` 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).

<table>
<thead>
<tr>
<th>AC0</th>
<th>AR2</th>
<th>AR3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000 8000h</td>
<td>1000h</td>
<td>1004h</td>
<td>at start of code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>after <code>mov</code> instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>line 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>after <code>mov</code> instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>line 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>after <code>mov</code> instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>line 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>after <code>mov</code> instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>line 14</td>
</tr>
<tr>
<td><strong>continued on next page</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|              |       |       | after `mov` instruction |
|              |       |       | line 15                |
|              |       |       | after `rpt` instruction |
|              |       |       | line 16                |
|              |       |       | after first `macm` instru |
|              |       |       | ction                   |
|              |       |       | after second `macm` instr |
|              |       |       | uction                  |
|              |       |       | after third `macm` instr |
|              |       |       | uction                  |

**Figure 1.3:** Execution Results
1.2.2 Lab 1: Lab\textsuperscript{9}

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 filtercode.asm\textsuperscript{10}. For help with circular addressing, view Addressing Modes for TI TMS320C55x (Section 3.1.2).

\textsuperscript{9}This content is available online at <http://cnx.org/content/m13791/1.8/>.
\textsuperscript{10}See the file at <http://cnx.org/content/m13791/latest/filtercode.asm>.

---

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000h</td>
<td></td>
</tr>
<tr>
<td>1001h</td>
<td></td>
</tr>
<tr>
<td>1002h</td>
<td></td>
</tr>
<tr>
<td>1004h</td>
<td></td>
</tr>
<tr>
<td>1005h</td>
<td></td>
</tr>
<tr>
<td>1006h</td>
<td></td>
</tr>
<tr>
<td>1007h</td>
<td></td>
</tr>
<tr>
<td>1008h</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Figure 1.4:} Data Memory Assignment (after execution)
CHAPTER 1. WEEKLY LABS

filtercode.asm

.ARMS_off ; enable assembler for ARMS=0
.CPL_on ; enable assembler for CPL=1
.mmemregs ; enable mem mapped register names

.global _filter
.global _inPtr
.global _outPtr

.copy "macro.asm" ; Copy in macro declaration

.sect ".data"

FIR_len1 .set 8 ; This is a 8-tap filter

.align 32 ; Align to a multiple of 16
coef1 ; assign label "coef1"
.copy "coef.asm" ; Copy in coefficients

.align 32
inputBuffer .space 16*FIR_len1 ; Allocate 8 words of storage for filter state
new_sample_index ; Allocate storage to save index in inputBuffer

.copy "testvect.asm"

.sect ".text2"

_filter

ENTER_ASM ; Call macro. Prepares registers for assembly

MOV #0, AC0 ; Clears AC0 and XAR3
MOV AC0, XAR3 ; XAR3 needs to be cleared due to a bug

MOV dbl (*(#_inPtr)), XAR6 ; XAR6 contains address to input
MOV dbl (*(#_outPtr)), XAR7 ; AR7 contains address to output

BSET AR2LC ; sets circular addressing for AR2

MOV #inputBuffer, AR2 ; State pointer is in AR2
MOV mmap(AR2), BSA23 ; BSA23 contains address of inputBuffer
MOV #new_sample_index, AR4 ; State index pointer is in AR4
MOV *AR4, AR2 ; AR2 contains the index of oldest state

MOV #coef1, AR1 ; initialize coefficient pointer
MOV #FIR_len1, BK03 ; initialize circular buffer length for register 0-3

MOV *AR6+ ≪ #16, AC0 ; Receive ch1 into AC0 accumulator
MOV AC0, AC1 ; Transfer AC0 into AC1 for safekeeping

MOV HI(AC0), *AR2+ ; store current input into state buffer

RPT #FIR_len1-1 ; Repeat next instruction FIR_len1 times

MACM *AR1+,*AR2+,AC0,AC0 ; multiply coef. by state & accumulate
round AC0 ; Round off value in 'AC0' to 16 bits

MOV HI(AC0), *AR7+ ; Store filter output (from AC0) into ch1
MOV HI(AC1), *AR7+ ; Store saved input (from AC1) into ch2

MOV AR2, *AR4 ; Save the index of the oldest state back into new_sample_index

LEAVE_ASM ; Call macro to restore registers

RET

Figure 1.1
filtercode.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 the filter folder from v:\ece420\55x\ccs4\filter to your work directory. 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 48 kHz sample rate of the system, then use the MATLAB command firpm to generate this filter; you can type help firpm 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 filtercode.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_len1 to 20. FIR_len1 is set using the .set directive, which assigns a number to a symbolic name. You will need to change this to FIR_len1.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 TMS320C55x 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 coef1 .copy "coef1.asm" % Copy FIR filter coefficients
3
4 .align 32        % Align to a multiple of 32
5 firState1 .space 16*20  % Allocate 20 words of data space
```

Assemble your code, load the output file, and run. Ensure that it 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 filtercode.asm file from Part 1 (Section 1.2.2.2: Part 1: Single-Channel FIR Filter). Work from this copy; do not modify your working filter from the previous part. You will use that code again later.

Next, modify your code so that in addition to sending the output of your first filter (with a 4 kHz to 8 kHz passband) to output channel 1 and the unfiltered input to output channel 2, it sends the output of your second filter (with a 8 kHz to 12 kHz passband) to output channel 3. To do this, you will need to use the .align and .copy directives to load the second set of coefficients into data memory. You will also need to
add instructions to initialize a pointer to the second set of coefficients and to perform the calculations for the second filter.

**Exercise 1.1**

**Extra Credit Problem**

One extra credit point will be awarded to you and your partner if you can implement the dual-channel system without using the auxiliary registers AR0, AR3 and AR5. Why is this more difficult? Renaming the registers using the .asg directive does not count!

Using the techniques introduced in DSP Development Environment: Introductory Exercise for TI TMS320C55x (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 TMS320C55x (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 `firsadd` 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 `firsadd`.

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 `firsadd` requires the states to be broken up into two separate circular buffers. Refer to the `firsadd` instruction on page 5-152 in the Mnemonic Instruction Set manual.

```assembly
1    mov   *AR1, *AR2-        ; write x(-N/2) over x(-N)
2    mov   HI(AC0), *AR1      ; write x(0) over x(-N/2)
3    add   *AR1-, *AR2-, AC0  ; add x(0) and x(-(N-1))
4    ; (prepare for first multiply)
5    rpt   #(FIR_len1/2-1)
6    firsadd *AR1-, *AR2-, *CDP+, AC0, AC1
7    round  AC1
8    amar  ????????????????? ; Fill in these two instructions
9    amar  ????? ; They modify AR1 and AR2
10   ; note that the result is now in the
11   12   ; AC1 accumulator
```

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

```assembly
1    bset  AR1LC               ; sets circular addressing for AR1
2    bset  AR2LC               ; sets circular addressing for AR2
3
4
5    mov   #firState1, AR1
6    mov   #firState1Index, AR4
7    mov   mmap(AR1), BSA01
8    mov   *ARM4, AR1          ; get pointer to oldest delayBuf in AR1
```

[^1]: http://focus.ti.com/lit/ug/spru374g/spru374g.pdf
There are also a couple other changes that need to be made before the code will compile successfully. Read the comments carefully and understand how the `firs add` instruction works to make the necessary changes.

Hint: Make sure accumulator usage (AC0, AC1, AC2) and what is sent to output is correct.

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

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

### 1.2.2.5 Quiz Information

The quiz for Lab 1 is broken down as follows:

- 1 point: Prelab (must be ready to show the TA the week before the quiz)
- 4 points: Working code: you must demonstrate that your code works using input from function generator and that it works using input from appropriate test vectors. Have an `.asm` file ready to demonstrate each. Of the 4 points, you get 0.5 points for a single 20-tap filter, 2 points for the two-filter system, and 1.5 points for the system using the `firs` opcode.
- 5 points: Oral quiz score.
- 1 extra credit point: As described above (p. 13).

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

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

Use the TI documentation, specifically the *Mnemonic Instruction Set*\(^\text{12}\) manual. Also, feel free to ask the TAs to help explain the code that you have been given.

\(^{12}\text{http://focus.ti.com/lit/ug/spru374g/spru374g.pdf}\)
1.2.3 Resources

1.2.3.1 Fixed-Point Number Representation

Fixed-point arithmetic is generally used when hardware cost, speed, or complexity is important. Finite-precision quantization issues usually arise in fixed-point systems, so we concentrate on fixed-point quantization and error analysis in the remainder of this course. For basic signal processing computations such as digital filters and FFTs, the magnitude of the data, the internal states, and the output can usually be scaled to obtain good performance with a fixed-point implementation.

1.2.3.1.1 Two’s-Complement Integer Representation

As far as the hardware is concerned, fixed-point number systems represent data as $B$-bit integers. The two’s-complement number system is usually used:

\[
 k = \begin{cases} 
 \text{binary integer representation if } 0 \leq k \leq 2^{B-1} - 1 \\
 \text{bit-by-bit inverse } (-k) + 1 \text{ if } -(2^{B-1}) \leq k \leq 0
\end{cases}
\]

Figure 1.5

The most significant bit is known as the sign bit; it is 0 when the number is non-negative; 1 when the number is negative.

1.2.3.1.2 Fractional Fixed-Point Number Representation

For the purposes of signal processing, we often regard the fixed-point numbers as binary fractions between $[-1, 1)$, by implicitly placing a decimal point after the sign bit.

\[
 x = -b_0 + \sum_{i=1}^{B-1} b_i 2^{-i}
\]

Figure 1.6

or

\[
 x = -b_0 + \sum_{i=1}^{B-1} b_i 2^{-i}
\]

This interpretation makes it clearer how to implement digital filters in fixed-point, at least when the coefficients have a magnitude less than 1.

\[\text{This content is available online at } \text{<http://cnx.org/content/m11930/1.2/>} .\]
1.2.3.1.3 Truncation Error

Consider the multiplication of two binary fractions

![Figure 1.7](image)

Note that full-precision multiplication almost doubles the number of bits; if we wish to return the product to a $B$-bit representation, we must truncate the $B - 1$ least significant bits. However, this introduces truncation error (also known as quantization error, or roundoff error if the number is rounded to the nearest $B$-bit fractional value rather than truncated). Note that this occurs after multiplication.

1.2.3.1.4 Overflow Error

Consider the addition of two binary fractions;

![Figure 1.8](image)

Note the occurrence of wraparound overflow; this only happens with addition. Obviously, it can be a bad problem.

There are thus two types of fixed-point error: roundoff error, associated with data quantization and multiplication, and overflow error, associated with data quantization and additions. In fixed-point systems, one must strike a balance between these two error sources; by scaling down the data, the occurrence of overflow errors is reduced, but the relative size of the roundoff error is increased.

**NOTE:** Since multiplies require a number of additions, they are especially expensive in terms of hardware (with a complexity proportional to $B_x B_h$, where $B_x$ is the number of bits in the data, and $B_h$ is the number of bits in the filter coefficients). Designers try to minimize both $B_x$ and $B_h$, and often choose $B_x \neq B_h$.

1.3 Lab 2

1.3.1 Lab 2: Theory

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 1.9. 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 $\frac{F_s}{D}$.

---

14This content is available online at <http://cnx.org/content/m10024/2.21/>. 
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?

1.3.2 Lab 2: Prelab (Part 1)

1.3.2.1 Multirate Theory Exercise

Consider a sampled signal with the DTFT $X(\omega)$ shown in Figure 1.10.

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 1.11 (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 $\omega$ to...
analog frequency depends on the sampling rate. Therefore, the conversion is different for \( X(\omega) \) and \( W(\omega) \).

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

(1.2)

\[
Y(\omega) = W(U\omega)
\]

(1.3)

\[X(\lambda) \xrightarrow[D]{} W(\lambda) \xrightarrow[U]{} Y(\lambda)\]

Figure 1.11: Multirate System

1.3.3 Lab 2: Prelab (Part 2)\(^{16}\)

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 1.9). 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.

\[
H(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + \cdots = (z_1 - z^{-1})(z_2 - z^{-1})(z_4 - z^{-1})\cdots
\]

(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 \( \pm \left( \frac{\pi}{8} \right), \pm \left( \frac{\pi}{4} \right), \pm \left( \frac{3\pi}{8} \right), \pm \left( \frac{5\pi}{8} \right), \pm \left( \frac{3\pi}{4} \right), \) 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 \texttt{poly} is very useful for this; type \texttt{help poly} in MATLAB for details.

Once you have determined the coefficients of the filters, use MATLAB function \texttt{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?

1.3.4 Lab 2: Lab\(^{17}\)

1.3.4.1 Implementation

Before implementing the entire system shown in Multirate Processing: Introduction (Figure 1.9), we recommend you design a system that consists of a cascade of filters FIR 1 and FIR 2 without the sample-rate

\(^{16}\)This content is available online at \(<http://cnx.org/content/m10815/2.6/>\).

\(^{17}\)This content is available online at \(<http://cnx.org/content/m3792/1.4/>\).
compressor or expander. After verifying that the response of your two-filter system is correct, proceed to implement the complete multirate system and verify its total response. At first, use fixed compression and expansion factors of \( D = U = 4 \). After you have verified that the multirate system works at a fixed rate, you should modify your code so that the rate can be changed easily. **You must be able to quickly change the compression and expansion factors when you demo your code.**

1.3.4.1.1 Compressed-rate processing

In order to perform the processing at the lower sample rate, implement a counter in your code. Your counter will determine when the compressed-rate processing is to occur, and it can also be used to determine when to insert zeros into FIR 3 to implement the sample-rate expander. Some instructions that may be useful for implementing your multirate structure are the `add` and `bcc` (branch conditional) instructions. You may also find the `b` (branch) instruction useful. The conditional fields that can be used with `bcc` can be found on page 1-7 of the 55x Mnemonic Instruction Set.

1.4 Lab 3

1.4.1 Lab 3: Theory

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 Quantization 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 1.12. Note that in Direct Form II, the states (delayed samples) are neither the input nor the output samples, but are instead the intermediate values \( w[n] \).

---

18This content is available online at <http://cnx.org/content/m10025/2.22/>.
1.4.2 Lab 3: Prelab (Part 1)

1.4.2.1

The transfer function for the second-order section shown in IIR Filtering: Introduction (Figure 1.12) is

\[ H(z) = G \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \]  \hspace{1cm} (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 \( H(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)W(z)}{X(z)W(z)} \) 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

\[ [B,A] = \text{ellip}(4,.25,10,.25) \]
\[ \text{freqz}(B,A) \]

**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.

\(^{19}\)This content is available online at <http://cnx.org/content/m13799/1.2/>.
NOTE: If you use the \texttt{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, \texttt{zeros}, and roots of the denominator, \texttt{poles}, so that you can group them to create two second-order sections. The MATLAB commands \texttt{roots} and \texttt{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 \texttt{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, memory and register use, and testing procedure. Write out pseudo-code for the processor implementation of a biquad. 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 TMS320C55x (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)\textsuperscript{20}

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 \texttt{notch filter} generated using the \texttt{butter} command: $[B,A] = \texttt{butter}(2, [0.07, 0.10], 'stop')$.

\textsuperscript{20}This content is available online at <http://cnx.org/content/m10813/2.5/>.
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 \texttt{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 \texttt{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.

1.4.4 Lab 3: Lab\textsuperscript{21}

1.4.4.1 Implementation

On the DSP, you will implement the elliptic low-pass filter designed using the \texttt{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 \(a_0\) and \(a_2\) coefficients (\(a_0\) and \(a_2\), for example) will always be less than or equal to 1.0. However, the magnitude of the \(a_1\) 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.

\begin{itemize}
\item This content is available online at \url{http://cnx.org/content/m13798/1.3/}.
\end{itemize}
Because the IIR code will have to run inside the block I/O loop and this loop uses the block repeat counter (BRC0), you must use another looping structure to avoid corrupting the BRC0.

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

```
mov    #num_stages-1, AR1

start_stage

; IIR code goes here

BCC start_stage, *AR1- != #0
```

### 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 sfts opcode) before saving the output to memory.

### 1.4.4.2 Grading

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

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

### 1.4.5 Resources

#### 1.4.5.1 Fixed-Point Quantization

The fractional $B$-bit two’s complement number representation evenly distributes $2^B$ quantization levels between $-1$ and $1 - 2^{-(B-1)}$. The spacing between quantization levels is then

$$\left(\frac{2}{2^B} = 2^{-((B-1))} \equiv \Delta_B\right)$$

Any signal value falling between two levels is assigned to one of the two levels.

$X_Q = Q[x]$ is our notation for quantization. $e = Q[x] - x$ is then the quantization error.

One method of quantization is rounding, which assigns the signal value to the nearest level. The maximum error is thus $\frac{\Delta_B}{2} = 2^{-B}$.

---

22This content is available online at <http://cnx.org/content/m11921/1.2/>. 
Another common scheme, which is often easier to implement in hardware, is \textit{truncation}. \( Q[x] \) assigns \( x \) to the next lowest level.

The worst-case error with truncation is \( \Delta = 2^{-(B-1)} \), which is twice as large as with rounding. Also, the error is always negative, so on average it may have a non-zero mean (i.e., a bias component).

Overflow is the other problem. There are two common types: two's complement (or \textit{wraparound}) overflow, or \textit{saturation} overflow.
Obviously, overflow errors are bad because they are typically large; two's complement (or wraparound) overflow introduces more error than saturation, but is easier to implement in hardware. It also has the advantage that if the sum of several numbers is between $[-1, 1)$, the final answer will be correct even if intermediate sums overflow! However, wraparound overflow leaves IIR systems susceptible to zero-input large-scale limit cycles, as discussed in another module. As usual, there are many tradeoffs to evaluate, and no one right answer for all applications.

1.5 Lab 4

1.5.1 Lab 4: Intro

1.5.1.1 Introduction

In this lab you are going to apply the Fast Fourier Transform (FFT) to analyze the spectral content of an input signal in real time. You will also explore algorithms that estimate a stationary random signal’s Power Spectral Density (PSD). Finally, you will be introduced to using the C environment and code optimization in a practical application. This knowledge will be applied in optimizing a reference implementation of a PSD estimator.

1.5.1.2 Fast Fourier Transform

First, samples of the power spectrum of a deterministic signal will be calculated via the magnitude squared of the FFT of the windowed signal. You will transform a 1024-sample block of input data and send the power spectrum to the output for display on the oscilloscope. After computing the FFT of a 1024-sample block of input data, you will then compute the squared magnitude of the sampled spectrum and send it to the output for display on the oscilloscope. In contrast to the systems you have implemented in the previous labs, the FFT is an algorithm that operates on blocks of samples at a time. In order to operate on blocks of samples, you will need to use interrupts to halt processing so that samples can be transferred.

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

DFT:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}nk}$$

(1.6)

where $n, k \in \{0, 1, \ldots, N-1\}$
Your implementation will include windowing of the input data prior to the FFT computation. This is simple a point-by-point multiplication of the input with an analysis window. As you will explore in the prelab exercises, the choice of window affects the shape of the resulting spectrum.

A block diagram of the spectrum analyzer you will implement in the lab, including the required input and output locations, is depicted in Figure 1.16.

![Block Diagram of Spectrum Analyzer](image)

**Figure 1.16:** FFT-based spectrum analyzer

#### 1.5.1.3 Pseudo-Noise Sequence Generator

Second, you will generate a colored, pseudo-noise (PN) sequence as input to the power spectrum algorithm. The noise sequence will be generated with a linear feedback shift register, whose operation is as shown in Figure 1.17 (Pseudo-Noise Generator). This PN generator is simply a shift-register and an XOR gate. Bits 0, 2, and 15 of the shift-register are XORed together and the result is shifted into the lowest bit of the register. This lowest bit is the output of the PN generator, and the highest bit is discarded in the shift process. The LSB is used to generate a value of $\pm M$ and this sequence will repeat with a period of $2^B - 1$, where $B$ is the width in bits of the shift register and $M$ is a constant. The power spectral density of this sequence is flat (white) and it will be "colored" via a fourth-order IIR filter. PN generators of this type are a useful source of random data bits for system testing. They are especially useful as a data model in simulating communication systems as these systems tend to randomize the bits seen by the transmission scheme so that bandwidth can be efficiently utilized. However, this method will not produce very "random" numbers. For more on this, see *Pseudorandom number generator[_number_generator?]*, *Linear feedback shift register[_feedback_shift_register?]*, and chapter 7, section 4 of *Numerical Recipes*[?].
1.5.1.4 Power Spectral Density Estimation

The direct-power-spectrum (DPS) algorithm outlined above is insufficient for estimating the PSD of a stationary noise signal because the variance of the estimated PSD is proportional to the value of the actual PSD. For the third part of this lab you will try to reduce the variance of the PSD estimate by windowing the autocorrelation of the noise signal and computing the FFT.

The autocorrelation of a sequence is the correlation of the sequence with itself:

\[ R[m] = \sum_{i=0}^{N-1-|m|} (x[i] x[i+|m|]) \]  

(1.7)

where \( m \in \{-((N-1)), -((N-2)), \ldots, N-1\} \)

For random signals, the autocorrelation here is an estimate of the actual autocorrelation. As \(|m|\) is increased, the number of samples used in the autocorrelation decreases. The windowed-DPS algorithm is equivalent to taking the FFT of the autocorrelation of the windowed data. Windowing of the data adds even more noise to the autocorrelation estimate, since the autocorrelation is performed on a distorted version of the original signal. An improvement can be made by constructing an accurate estimate of the autocorrelation (using a rectangular window), applying a window and computing the FFT. The motivation for applying the window at the latter stage is that emphasis should be given to accurate autocorrelation values while less accurate values should be de-emphasized or discarded.

A good empirical characterization of a random process requires sufficient data, and both of the PSD-estimation algorithms defined above can be extended to accommodate more data. There is one caveat, however: many real-world processes are modeled as short-time stationary processes (non-stationary models are hard to deal with), so there is a practical limit to how much data is available for a PSD estimate. Additional data is added to the direct-PSD estimation algorithm by adding multiple spectra together, thereby smoothing the PSD estimate. Additional data is added to the windowed-autocorrelation method by computing the autocorrelation of the total data set before windowing. You will explore the windowed-autocorrelation method on the DSP.

1.5.1.5 Compiling and Optimization

A second objective of this lab exercise is to introduce the TMS320C5510 C environment in a practical DSP application. The C environment provides a fast and convenient way to implement a DSP system using C and assembly modules. You will also learn how to optimize a program and when to use C or assembly. In future labs, the benefits of using the C environment will become clear as larger systems are developed.
In previous labs, processing was done on a sample-by-sample basis. In the next two labs, we will be working on 1024-blocks of input/data. The buffering has already been setup, leaving the user the task of adding code that processes the data.

1.5.2 Lab 4: Prelab

1.5.2.1 MATLAB Exercise, Part 1

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.

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

% Generate vector of frequencies to test
omega = pi/8 + [0:num_freqs-1]'/num_freqs*pi/4;

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

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

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

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

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

---

24This content is available online at <http://cnx.org/content/m12379/1.2/>. 
1.5.2.2 MATLAB Exercise, Part 2

Download and run the MATLAB file `lab4b.m` to observe direct and autocorrelation-based PSD estimates. A pseudo noise generator is filtered with a fourth-order IIR filter and various PSD estimates are computed and plotted.

The first plot contains PSD estimates, using a 1024-point FFT, from the first 512 samples of the 1024-sample sequence. The direct method is to take the squared-magnitude of the FFT of the sequence. The autocorrelation (AC) method is to take the magnitude of the FFT of the autocorrelation of the sequence. In this case rectangular windows were used in both FFTs. Why do the estimates look exactly the same? Will the estimates be alike if all 1024 samples are used with a 1024-sample FFT, with all other conditions being equal? Why or why not?
Figure 1.19: Second plot

The second plot contains PSD estimates, using a 1024-point FFT, from the first 32 samples of the 1024-sample sequence. The direct and AC estimates are computed in the same manner described above, except a hamming window has been applied to the sequence in the direct-PSD estimate. Why are these estimates different? What will make these estimates identical?
The third plot contains PSD estimates, using a 1024-point FFT, from all 32-sample blocks of the 1024-sample sequence. The direct-PSD estimate is computed by summing the hamming-windowed PSD estimates of each 32-sample block. The AC-PSD estimate is computed by taking the magnitude of the FFT of 63 samples of the autocorrelation of the entire 1024-point sequence. Why are 63 samples used in comparing the AC method to the direct method?
The fourth plot contains the unfiltered spectrum of the PN sequence and the impulse response of the coloring filter.

Try various the block lengths (Np) and direct-PSD window types. Observe the changes. Are there any tradeoffs? Does either the direct method or the autocorrelation method have an advantage over the other? Is there a direct-method window that results in identical block-based estimates? For simplicity, the autocorrelation window in this lab is rectangular. Can we, however, consider this an unbiased autocorrelation estimate that is windowed by a rectangular window? (Hint: see the MATLAB documentation for the xcorr function.) If not, how would you create an unbiased autocorrelation estimate before windowing, and what is the effective window that we have applied to the unbiased autocorrelation?

1.5.3 Lab 4: Lab\textsuperscript{25}

1.5.3.1 Implementation

As this is your first experience with the C environment, you will have the option to add most of the required code in C or assembly. A C skeleton will provide access to input samples, output samples, and interrupt handling code. You will add code to transfer the inputs and outputs (in blocks at a time), apply a hamming

\textsuperscript{25}This content is available online at <\text{http://cnx.org/content/m13809/1.12}/>.
window, compute the magnitude-squared spectrum, and include a trigger pulse. After the hamming window is created, either an assembly or C module that bit-reverses the input and performs an FFT calculation is called.

As your spectrum analyzer works on a block of samples at a time, you will need to use interrupts to pause your processing while samples are transferred from/to the CODEC (A/D and D/A) buffer. Fortunately, the interrupt handling routines have been written for you in a C shell program available at v:\ece420\55x\lab4\main.c. For this lab, you will be working with the code available at v:\ece420\55x\lab4.

1.5.3.1.1 Interrupt Basics

Interrupts are an essential part of the operation of any microprocessor. They are particularly important in embedded applications where DSPs are often used. Hardware interrupts provide a way for interacting with external devices while the processor executes code. For example, in a key entry system, a key press would generate a hardware interrupt. The system code would then jump to a specified location in program memory where a routine could process the key input. Interrupts provide an alternative to polling. Instead of checking for key presses at a predetermined rate (requires a clock), the system could be busy executing other code. On the TI-C55x DSP, interrupts provide a convenient way to transfer blocks of data to/from the CODEC in a timely fashion.

1.5.3.1.2 Interrupt Handling

The main.c, dma.c, and lab4.c code are intended to make your interaction with the hardware much simpler. These files handle the buffering of data using interrupts from the CODEC (A/D and D/A) and the USB port. Here, we will describe the important aspects of the code necessary to complete the assignment.

In the first few labs, data was processed on a sample-by-sample basis, so no buffering was necessary. However, the spectrum analyzer to be implemented in this lab works over a block of \( N = 1024 \) samples. If it were possible to compute a 1024-point FFT in the sample time of one sample, then no additional interrupt handling routines would be necessary. Samples could be collected in a 1024-length buffer and a 1024-point FFT could be computed uninterrupted while the buffer fills. Unfortunately, the DSP is not fast enough to accomplish this task.

We now provide an explanation of the shell C program main.c. The main.c file contains the main function that sets up the McBSP (Multi Channel Buffered Serial Port), DMA (Direct Memory Access), and interrupts. Then it returns and allows the DSP/BIOS scheduler to take over.

Many of the important interrupt routines are defined in the DSP/BIOS scheduler. The settings can be viewed by expanding the DSP/BIOS Config folder and double-clicking on the .cdb file. Changes do not have to be made, but it is good to know that the most important parts are the Scheduling and the Chip Support Library. Under Scheduling, there is the Hardware Interrupt Manager and Software Interrupt Manager. The Chip Support Library includes the DMA controller and the MCBSP. Various settings for these and many other modules can be set using a graphical interface instead of straight up coding.

Code for the DMA is defined in dma.c. The buffers are defined in this file, as well as the hardware interrupts. The hardware interrupts are initialized by calling the function init_DMA in the main function. When the hardware interrupts are triggered, they call the HWI_DMA0_Transmit() and HWI_DMA1_Receive() functions. At the end of these two functions, the software interrupt SWI_Process() is posted with different variables. Posting SWI_Process() will call the SWI_ProcessBuffer() function which will require modification.

The SWI_ProcessBuffer() function is defined in lab4.c. It is called every time the software interrupt SWI_Process is posted, which is set to happen every \( N = 1024 \) samples. As given, the function will simply copy the inputs to the outputs. (After commenting out some lines and uncommenting others.) Follow the example and comments to modify the code to perform the necessary operations.

Although each of these operations may be performed in C or assembly, we suggest you follow the guidelines suggested.
1. Transfer inputs (C)
2. Apply a Hamming Window (C/assembly)
3. Bit-reverse the input (C and assembly) (Already done)
4. Apply an N-point FFT (C and assembly) (Already done)
5. Compute the magnitude-squared spectrum and place in output (C/assembly)
6. Include a trigger pulse (C/assembly)

Note: Bit-reversing and application of the FFT may be done in reverse order depending on implementation.

Near the beginning of the SWI_ProcessBuffer function, the input samples need to be copied to specific buffers for processing. In C, pointers may be used as array names so that pdest[0] is the first word pointed to by pdest. The input samples are not in consecutive order and must be accessed with offsets. The four channels of input are offset from psrc respectively by 4i and 4i + 2, i = 0, ..., BlockLen - 1. The four output channels are accessed consecutively as offsets from pdest. On channel 1 of the output, the input is echoed out. You are to fill channel 2 with the windowed magnitude-squared FFT values by performing the operations listed above. For the first step, take a look at the way we make the DSPLIB cfft call to find out where to transfer the inputs to. (You may change the function call cfft to pass in different values if you like. Just remember that bit_rev() expects its input in a specific location.) Likewise, take a look at the C FFT code (declared in lab4fft.c to find out where to copy the inputs to.

1.5.3.1.3 Assembly FFT Routine

As the list of operations indicates, bit-reversal and FFT computation are to be done in both C and assembly. For the assembly version, make sure that the line defining C_FFT is commented out in lab4.c. We are providing you with a shell assembly file, available at v:\ece420\55x\lab4\c_fft_given.asm and shown in Appendix B (Section 1.5.3.4: Appendix B), containing many useful declarations and some code. The code for performing bit-reversal and other declarations needed for the FFT routine are also provided in this section. However, we would like you to enter this code manually, as you will be expected to understand its operation.

The assembly file c_fft_given.asm contains two main parts, the data section starting with .sect " .data" and the program section starting with .sect " .text". Every function and variable accessed in C must be preceded by a single underscore _ in assembly and a .global _name must be placed in the assembly file for linking. In this example, bit_rev_fft is an assembly function called from the C program with a label _bit_rev_fft in the text portion of the assembly file and a .global _bit_rev_fft declaration. In each assembly function, the macro ENTER_ASM is called upon entering and LEAVE_ASM is called upon exiting. These macros are defined in v:\ece420\55x\macro.asm. The ENTER_ASM macro saves the status registers and AR1, AR6, and AR7 when entering a function as required by the register use conventions. The ENTER_ASM macro also sets the status registers to the assembly conventions we have been using (i.e., FRCT = 1 for fractional arithmetic and CPL = 0 for DP referenced addressing). The LEAVE_ASM macro just restores the saved registers.

1.5.3.1.3.1 Parameter Passing

The parameter passing convention between assembly and C places the parameters into registers depending on the size of the parameters. Data pointers (16 or 23 bit) will be placed in (X)AR0 through (X)AR4 in that order. 16-bit data will be placed in T0, T1, and AR0 through AR4 in that order. 32-bit data will be placed in accumulators AC0 through AC2. If there are no available registers of the correct type, the parameters will be passed onto the stack. For more details, see page 6-16 of the Optimizing C/C++ Compiler User’s Guide (spru281e26).

26 http://focus.ti.com/lit/ug/spru281e/spru281e.pdf
1.5.3.1.3.2 Registers Modified

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

Now, we explain how to use the FFT routine provided by TI for the C55x. TI provides a library of commonly used functions. These functions include FFT, FIR, IIR, and some math operations. More information can be found in the DSP Library Programmer’s Reference\textsuperscript{27}. The CFFT function will be used for the DSPLIB implementation. Refer to the reference guide to figure out the correct syntax in calling the function.

The length of the FFT is defined as a label K_FFT_SIZE and the bit-reversing algorithm assumes that the input starts at data memory location _t_data. To have your code assemble for an N-point FFT, you will have to include the following label definitions in your assembly code.

N .set 1024
K_FFT_SIZE .set N ; size of FFT

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

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Binary Representation</th>
<th>Bit-Reversed Representation</th>
<th>Output Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1.1

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

\textsuperscript{27}http://focus.ti.com/lit/ug/spru422zs/prnu422z.pdf
As mentioned, in the above code _fft_data is a label indicating the start of the input data and _bit_rev_data is a label indicating the start of a circular buffer where the bit-reversed data will be written.

In general, to have a pointer index memory in bit-reversed order, the T0 register needs to be set to one-half the length of the circular buffer; a statement such as ARx+T0B is used to move the ARx pointer to the next location. For more information regarding the bit-reversed addressing mode, refer to Chapter 6 in the DSP Programmer’s Reference Guide [7]. Is it possible to bit-reverse a buffer in place?

1.5.3.1.4 C FFT Routine

A bit-reversing and FFT routine have also been provided in lab4fft.c, listed in Appendix C (Section 1.5.3.5: Appendix C:). Again, make sure you understand how the bit reversal is taking place. In main.c, the line defining C_FFT must not be commented for use of the C FFT routine. The sine tables (twiddle factors) are located in sinetables.h 28. This fft requires its inputs in two buffers, the real buffer real and the imaginary buffer imag, and the output is placed in the same buffers. The length of the FFT, N, and logN are defined in lab4.h, which is also listed in Appendix C (Section 1.5.3.5: Appendix C:). When experimenting with the C FFT make sure you modify these length values instead of the ones in the assembly code and main.c!

1.5.3.1.5 Creating the Window

As mentioned, you will be using the FFT to compute the spectrum of a windowed input. For your implementation you will need to create a 1024-point Hamming window. First, create a Hamming window in matlab using the function hamming. Use the matlab function write_intvector_headerfile 29 with name set to 'window' and elemperline set to 8 to create the header file that is included in lab4.c. The window has already been created for you.

As men tioned, in the above code _fft_data is a label indicating the start of the input data and _bit_rev_data is a label indicating the start of a circular buffer where the bit-reversed data will be written.

In general, to have a pointer index memory in bit-reversed order, the T0 register needs to be set to one-half the length of the circular buffer; a statement such as ARx+T0B is used to move the ARx pointer to the next location. For more information regarding the bit-reversed addressing mode, refer to Chapter 6 in the DSP Programmer’s Reference Guide [7]. Is it possible to bit-reverse a buffer in place?

1.5.3.1.4 C FFT Routine

A bit-reversing and FFT routine have also been provided in lab4fft.c, listed in Appendix C (Section 1.5.3.5: Appendix C:). Again, make sure you understand how the bit reversal is taking place. In main.c, the line defining C_FFT must not be commented for use of the C FFT routine. The sine tables (twiddle factors) are located in sinetables.h 28. This fft requires its inputs in two buffers, the real buffer real and the imaginary buffer imag, and the output is placed in the same buffers. The length of the FFT, N, and logN are defined in lab4.h, which is also listed in Appendix C (Section 1.5.3.5: Appendix C:). When experimenting with the C FFT make sure you modify these length values instead of the ones in the assembly code and main.c!

1.5.3.1.5 Creating the Window

As mentioned, you will be using the FFT to compute the spectrum of a windowed input. For your implementation you will need to create a 1024-point Hamming window. First, create a Hamming window in matlab using the function hamming. Use the matlab function write_intvector_headerfile 29 with name set to 'window' and elemperline set to 8 to create the header file that is included in lab4.c. The window has already been created for you.

28 http://cnx.org/content/m13809/latest/sinetables.h
29 http://cnx.org/content/m13809/latest/write_intvector_headerfile.m
1.5.3.1.6 Displaying the Spectrum

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

\[ |X(k)|^2 = (\Re(X(k)))^2 + (\Im(X(k)))^2 \]  

You may find the assembly instructions \texttt{sqrm} and \texttt{sqam} useful in implementing (1.8).

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

1.5.3.1.7 Intrinsics

If you are planning on writing some of the code in C, then you may be forced to use intrinsics. Intrinsic instructions provide a way to use assembly instructions directly in C. An example of an intrinsic instruction is \texttt{bit\_rev\_data[0]=_smpyr(bit\_rev\_data[0],window[0])} which performs the assembly signed multiply round instruction. You may also find the \texttt{_lsmpy} instruction useful. For more information on intrinsics, see page 3-29 of the \textit{TI-C55x DSP Programmer’s Guide}.

The following lines of code were borrowed from the C FFT to serve as an example of arithmetic operations in C. Save this code in a file called \texttt{mathex.c} and create a new project by going to Project->New... In the following window, enter mathex as the name for the project and save it in its own folder on the W:
drive. Verify that the Project Type is Executable (.out) and that the target is TMS320C55XX before clicking Finish.

```c
int s1, s2;
int t1, t2;
int i1, i2;
int n1 = 16383, n2 = 16382, n3 = 16381, n4 = 16380;

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

    /* Code for TI TMS320C55X series */
    t1 = ((long int)(n1*n2) + (long int)(n3*n4)) >> 15;
    t2 = ((long int)n1 + (long int)n2) >> 1;

    /* Intrinsic code for TMS320C55X series */
    i1 = _sadd(_smpy(n1,n2), _smpy(n3,n4));
    i2 = _sshl(_sadd(n1, n2),-1);

    while(1);
}
```

Add the \texttt{mathex.c} file to the project by left-clicking on the \texttt{mathex.pjt} file in the left-hand window and selecting Add Files to Project... By default, the generated assembly code is not saved. To save the
generated assembly for later comparison, go to Project->Build Options. Under the Compiler tab, click on the Assembly category and make sure Keep Generated .asm Files is selected.

Compile your project before looking at the resulting assembly file and investigating the differences between each block. Be sure to reference page 3-32 of the DSP Programmer’s Guide to find out what the state of the FRCT and OVM bits are. Run this program on the DSP, halt the program, and compare the output values in a memory window. Does each block work properly for all possible values?

1.5.3.1.8 Compiling and Linking

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

Close the mathex project and go back to the original Lab 4 project. In the future if there are additional source code files to include in the project, just follow the above instructions. Once you have completed lab4.c and c_fft_given.asm, select Project->Rebuild All. Load the output file onto the DSP as usual and confirm that valid FFTs are calculated. Once valid output is obtained, measure how many clock cycles it takes to compute both the assembly and C FFT.

1.5.3.2 Quiz Information

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

1.5.3.3 Appendix A:

lab4.c

```c
#include "dsk5510_dual3006cfg.h"
#include "dsk5510.h"
#include "swi_process.h"
#include "dsplib.h"

#define N 1024
#define logN 10
#include "window.h"

/* comment the next line to use DSPLIB fft */
#ifndef C_FFT /* Use C FFT */

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

/* FFT data buffers */
```

[30]http://cnx.org/content/m13809/latest/lab4.c
int real[N]; /* Real part of data */
int imag[N]; /* Imaginary part of data */
#include "lab4fft.c"
#else /* Use DSPLIB FFT */
/* Function defined by c_fft_given.asm */
void bit_rev(void);
/* FFT data buffers (declared in c_fft_given.asm) */
extern int bit_rev_data[N*2]; /* Data output for bit-reverse function */
extern int fft_data[N*2]; /* In-place FFT & Output array */
#endif /* C_FFT */

// all data processing should be done in SWI_ProcessBuffer

void SWI_ProcessBuffer()
{
static unsigned int mbox_value = 0;
short *psrc, *pdest;
unsigned int i;

mbox_value |= SWI_getmbox();

// buffers are only processed when both transmit and receive are ready
if((mbox_value & DMA_RECEIVE_DONE) && (mbox_value & DMA_TRANSMIT_DONE)) {
    mbox_value = 0;
    // get buffer pointers
    psrc = receive_buffer[receive_buffer_to_process_index];
    pdest = transmit_buffer[transmit_buffer_to_fill_index];

    // samples are interleaved in input buffer 3-4-1-2
    // output buffer is organized 3-4-1-2-3-4-1-2
    // The following code would copy input from each input channel to the
    // respective output channel:
    /*
    for (i = 0; i < 1024; i++)
    {
        pdest[4*i] = psrc[4*i]; //channel 3 output is channel 3 input
        pdest[4*i+1] = psrc[4*i+1]; //channel 4 output is channel 4 input
        pdest[4*i+2] = psrc[4*i+2]; //channel 1 output is channel 1 input
        pdest[4*i+3] = psrc[4*i+3]; //channel 2 output is channel 2 input
    }
    */
#endif C_FFT /* Use C FFT */
/* Insert code to fill */
/* C FFT buffers */
#else /* Use DSPLIB FFT */
   /* Insert code to fill */
   /* assembly FFT buffers*/
#endif /* C_FFT */

#else /* Use DSPLIB FFT */
   
   /* Insert code to fill */
   /* assembly FFT buffers*/
#endif /* C_FFT */

#elifdef C_FFT /* Use C FFT */
   
   /* Insert code to fill */
   /* assembly FFT buffers*/
#endif /* C_FFT */

receive_buffer_processed = 1; // flag receive buffer as processed
transmit_buffer_filled = 1; // flag output buffer as full
}
1.5.3.4 Appendix B:

c_fft_given.asm\(^{31}\)

```
.ARMS_off ; enable assembler for ARMS=0
.CPL_on ; enable assembler for CPL=1
.mmregs ; enable mem mapped register names

.global _bit_rev_data
.global _fft_data
.global _window
.global _bit_rev

.copy "macro.asm"

.sect ".data"

N .set 1024
K_FFT_SIZE .set 1024

.align 4*N
_bit_rev_data .space 16*2*N ; Output of bit reversing function

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

.sect ".text"

_bit_rev

ENTER_ASM

MOV #_fft_data, AR3
MOV #_bit_rev_data, AR7
MOV AR7, AR2
MOV #K_FFT_SIZE-1, BRC0
MOV #K_FFT_SIZE, T0
RPTB bit_rev_end-1
MOV dbl(*AR3), AC0
MOV AC0, dbl(*AR2+)
AMAR *(AR3+T0B)
bit_rev_end:

LEAVE_ASM

RET
```

\(^{31}\)https://cnx.org/content/m13809/latest/c_fft_given.asm
; If you need any more assembly subroutines, make sure you name them
; _name, and include a "._global _name" directive at the top. Also,
; don't forget to use ENTER_ASM at the beginning, and LEAVE_ASM
; and RET at the end!

1.5.3.5 Appendix C:

lab4fft.c

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

http://cnx.org/content/m13809/latest/lab4fft.c
#include "sinetables.h"

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

    j = 0;          /* bit-reverse */
    n2 = N >> 1;
    for (i=1; i < N - 1; i++)
    {
        n1 = n2;
        while ( j >= n1 )
        {
            j = j - n1;
            n1 = n1 >> 1;
        }
        j = j + n1;
        if (i < j)
        {
            t = real[i];
            real[i] = real[j];
            real[j] = t;
            t = imag[i];
            imag[i] = imag[j];
            imag[j] = t;
        }
    }

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

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

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

/* End standard 32-bit code */

/* Code for TI TMS320C54X series */

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

/* End code for TMS320C54X series */

/* Intrinsic code for TMS320C55X series */

Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
real[k] = _sshl(_sadd(real[k], Wr),-1);
imag[k] = _sshl(_sadd(imag[k], Wi),-1);

/* End intrinsic code for TMS320C55X series */

/* Intrinsic code for TMS320C54X series */

Wr = _ssub(_smpy(c, real[k+n1]), _smpy(s, imag[k+n1]));
Wi = _sadd(_smpy(s, real[k+n1]), _smpy(c, imag[k+n1]));
real[k+n1] = _sshl(_ssub(real[k], Wr),-1);
imag[k+n1] = _sshl(_ssub(imag[k], Wi),-1);
real[k] = _sshl(_sadd(real[k], Wr),-1);
imag[k] = _sshl(_sadd(imag[k], Wi),-1);

/* End intrinsic code for TMS320C54X series */

```c
} 
}
return;
```
1.5.4 Lab 4: Extension

1.5.4.1 Reference Implementation of a PSD estimator

We provide for you in Appendix D and E (Section 1.5.4.3: Appendix E: Additional routines for PSD estimator) a complete C implementation of a PSD estimator. The input is an IIR-filtered pseudo-noise (PN) sequence generator and the PSD estimate is based on windowing the autocorrelation with a rectangular window. The code consists of the files lab4b.c, lab4b.h, intrinsics.h, pn.c, iirfilt.c, autocorr.c, c_fft_given_iirc.asm, and the previously-given TI FFT routine. The assembly file c_fft_given_iirc.asm differs from c_fft_given.asm in that the window array has been removed and variables and arrays associated with IIR filtering have been added. Note that the multiply functions in the functions are actually compiler directives contained in intrinsics.h. Make sure you know which ones are used and why; note that VPO is not defined by the TI compiler, therefore the corresponding section of the #ifdef statement is not used. Open up the lab4b.pjt project and Rebuild All. Load lab4b.out onto the DSP and run the code. Make sure that an IIR-filtered PN sequence appears on channel 1 and its PSD estimate appears on channel 2.

Does the output match your expectations based on the theory? Does this application illustrate any limitations of the FFT implementation? (Hint: note that most of the values in the FFT input are zero.) The previously-given C implementation uses a similar algorithm as the TI FFT; take a look at the C code for help. What are the limitation(s) of the FFT that show up in this application?

In lab4b.h \#sets the number of autocorrelation points that are calculated. What is the maximum value of \( M \) that allows the reference routines to run in real time? In determining this value you may find it useful to connect a waveform generator to the input and copy input on that channel into channel 1 of output. You may limit \( M \) to powers of 2 minus 1.

1.5.4.2 Appendix A: Main routine, header files for PSD estimator

lab4b.h

```c
#define N 1024 /* Length of output buffers * /
#define L N /* Length of input data */
#define logL 10 /* log base 2 of L */
#define M 31 /* Compute 2*M+1 autocorrelation points */

/* #define M (L/2-1) */ /* Be sure to use ()'s in this case */
/* or algebraic substitution bugs */
/* can be introduced */
```

---

33 This content is available online at <http://cnx.org/content/m15847/1.1/>.
34 http://cnx.org/content/m15847/latest/lab4b.h
35 http://cnx.org/content/m15847/latest/intrinsics.h
36 http://cnx.org/content/m15847/latest/lab4b.c
37 http://cnx.org/content/m15847/latest/lab4bmain.c
/* Compiler intrinsics for the TI compiler */
/* and the Very Portable Optimizer (VPO) port */
/* to TMS320C54x series DSPs */
/* */
/* Use compile option -DVPO when using VPO */
/* */
/* Copyright September 2005 by Matt Kleffner */
/* under the Creative Commons Attribution License */
/* Works with TMS320C55X series */

#ifndef INTRINSICS_H
#define INTRINSICS_H

#ifdef VPO

long int vpo_l_mul_ii(int w0, int w1);

/* fractional multiply without fractional mode (long result) */
#define _l_mul_fract_fb0_ii(w0,w1) (vpo_l_mul_ii(w0,w1) ≪ 1)

/* fractional multiply with fractional mode already on (long result) */
#define _l_mul_fract_fb1_ii(w0,w1) (vpo_l_mul_ii(w0,w1))

/* fractional multiply without fractional mode (int result) */
#define _i_mul_fract_fb0_ii(w0,w1) (vpo_l_mul_ii(w0,w1) ≫ 15)

/* fractional multiply with fractional mode already on (int result) */
#define _i_mul_fract_fb1_ii(w0,w1) (vpo_l_mul_ii(w0,w1) ≫ 16)

#define _set_fract_bit() vpo_set_fract()
#define _reset_fract_bit() vpo_reset_fract()
#define _set_ovm_bit() vpo_set_ovm()
#define _reset_ovm_bit() vpo_reset_ovm()

#define _l_add_shiftl_li(w0,w1) (((int32)(w0)) + (((int32)(int16)(w1) ≪ 16))
#define _l_sub_shiftl_li(w0,w1) (((int32)(w0)) - (((int32)(int16)(w1) ≪ 16))

#else

/* fractional multiply without fractional mode (long result) */
#define _l_mul_fract_fb0_ii(w0,w1) (((long int)w0 * (long int)w1) ≪ 1)

/* fractional multiply with fractional mode already on (long result) */
#define _l_mul_fract_fb1_ii(w0,w1) (vpo_l_mul_ii(w0,w1))

#endif
#endif

#endif
(((long int)w0 * (long int)w1))

/* fractional multiply without fractional mode (int result) */
#define _i_mul_fract_fb0_ii(w0,w1) 
    (((long int)w0 * (long int)w1) >> 15)

/* fractional multiply with fractional mode already on (int result) */
#define _i_mul_fract_fb1_ii(w0,w1) 
    (((long int)w0 * (long int)w1) >> 16)

#define _set_fract_bit() asm(" ssbx frct")
#define _reset_fract_bit() asm(" rsbx frct")
#define _set_ovm_bit() asm(" ssbx ovm")
#define _reset_ovm_bit() asm(" rsbx ovm")

#endif /* VPO */
#endif /* INTRINSICS_H */

// lab4b.c
// Uses PN generation, IIR filtering, and autocorrelation
// code by Matt Kleffner -9/2004
// Based on swi_process.c by Educational DSP

#include "dsk5510_dual3006cfg.h"
#include "dsk5510.h"
#include "swi_process.h"
#include "dsplib.h"

#include "lab4b.h"  // Define N here in header file */

/* function defined in pn.c */
void rand_fillbuffer(void);

/* IIR values and buffers (declared in c_fft_given_iirc.asm) */
#define IIR_order 4
extern int scale;
extern int coef[IIR_order];
extern int state[IIR_order];

/* Pointer to state buffer location */
int iirptr;
extern unsigned int *iseed;  // seed for rand_fillbuffer() and randbit() */

/* function defined in iirfilt.c */
void iirfilt(void);

/* function defined in autocorr.c */
void autocorr(void);
/ Function defined by c_fft_given_iirc.asm */
//void bit_rev_fft(void);

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

/* Our input/output buffers */
int autocorr_in[N];
int autocorr_out[N];

// all data processing should be done in SWI_ProcessBuffer

void SWI_ProcessBuffer()
{
  static unsigned int mbox_value = 0;
  short *psrc, *pdest;
  unsigned int i;

  mbox_value |= SWI_getmbox();

  // buffers are only processed when both transmit and receive are ready
  if((mbox_value & DMA_RECEIVE_DONE) && (mbox_value & DMA_TRANSMIT_DONE)) {
    mbox_value = 0;

    // get buffer pointers
    psrc = receive_buffer[receive_buffer_to_process_index];
    pdest = transmit_buffer[transmit_buffer_to_fill_index];

    /* First, transfer inputs and outputs */
    for (i = 0; i < N; i++) {
      pdest[4*i] = autocorr_in[i];
      pdest[4*i+1] = bit_rev_data[i*2] ≪ 8;

      /* Some statements useful in debugging */
      /* pdest[4*i] = psrc[4*i+2]; */
      /* Be sure to comment out PN-sequence generation */
      /* when using the next two lines */
      //autocorr_in[i] = psrc[4*i+2];
    }

    /* Last, set the DC coefficient to -1 for a trigger pulse */
    pdest[0] = -32768;

    /* Generate PN input */
    rand_fillbuffer();
}
/* Filter input */
iirfilt();

/* Calculate autocorrelation */
autocorr();

/* Transfer autocorr output to FFT input buffer */
for (i = 0; i < N; i++) {
    fft_data[i*2] = autocorr_out[i];
    fft_data[i*2+1] = 0;
}

// Bit-reverse and compute FFT
cfft((DATA *)fft_data,N, SCALE);
cbrev((DATA *)fft_data,(DATA *)bit_rev_data,N);

/* Done, wait for next time around! */
receive_buffer_processed = 1; // flag receive buffer as processed
transmit_buffer_filled = 1; // flag output buffer as full

}
DSK5510_init(); // init BSL
DSK5510_rset(DSK5510_MISC, 0x03); // route McBSP0/1 to J3

// Initialize autocorr_out to zero since some values will remain zero
for (i = 0; i < N; ++i)
{
    autocorr_out[i] = 0;
}

for (i = 0; i < IIR_order; ++i)
    state[i] = 0;

// Start McBSP0 I2S slave
    MCBSP_start(hMcbsp0, MCBSP_XMIT_START | MCBSP_RCV_START |
                 MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);

// Start McBSP1 I2S master
    MCBSP_start(hMcbsp1, MCBSP_XMIT_START | MCBSP_RCV_START |
                 MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);

init_DMA(); // configure DMA and interrupts

*iseed = 1;
iirptr = 0;

return; // let DSP/BIOS scheduler take over
}

1.5.4.3 Appendix E: Additional routines for PSD estimator

pn.c\textsuperscript{38}
iirfilt.c\textsuperscript{39}
autocorr.c\textsuperscript{40}
c_fft_given_iirc.asm\textsuperscript{41}

/* ECE420, Lab 4, Reference PN Generator Implementation (Non-Optimized) */
/* Matt Kleffner 08/04 */
/* Original by Michael Frutiger 02/24/04 */
/* Use governed by the Creative Commons Attribution License */

#include "lab4b.h"

extern unsigned int *iseed;
extern int autocorr_in[N];

\textsuperscript{38}http://cnx.org/content/m15847/latest/pn.c
\textsuperscript{39}http://cnx.org/content/m15847/latest/iirfilt.c
\textsuperscript{40}http://cnx.org/content/m15847/latest/autocorr.c
\textsuperscript{41}http://cnx.org/content/m15847/latest/c_fft_given_iirc.asm
/ * Returns as an integer a random bit, based on the 15 lowest significant
  * bits in iseed (which is modified for the next call). */
int randbit()
{
    int newbit;
    /* XOR bits 15, 1 and 0 of iseed */
    newbit = (*iseed >> 15) & 1 ^ (*iseed >> 1) & 1 ^ *iseed & 1;
    /* Leftshift the seed and put the result of the XOR's in bit 1. */
    *iseed = (*iseed << 1) | newbit;
    return (newbit);
}

void rand_fillbuffer(void)
{
    int i;

    for (i = 0; i < N; ++i)
    {
        if (randbit()) autocorr_in[i] = 32767;
        else autocorr_in[i] = -32767;
    }
}

/* Simple, unoptimized IIR filter (feedback only) */
/* for TMS320C54X series DSPs */
/* Copyright September 2005 by Matt Kleffner */
/* under the Creative Commons Attribution License */
/* Works for TMS320C55X series as well */
#include "lab4b.h"
#include "intrinsics.h"

/* IIR values and buffers (declared in c_fft_given_iirc.asm) */
#define IIR_order 4
extern int scale;
extern int coef[IIR_order];
extern int state[IIR_order];

/* Arrays declared in main routine */
extern int autocorr_in[N];
extern int autocorr_out[N];

/* Pointer to state buffer location */
extern int iirptr;

void iirfilt()
{
    int i, j;
for (i = 0; i < N; ++i)
{
    int sum = 0;
    /* Calculate and sum all feedback terms except the "oldest" one */
    for (j = 0; j < (IIR_order-1); ++j)
    {
        sum += _i_mul_fract_fb1_ii(coef[j],state[iirptr]);
        /* Avoid usage of "modulo" routine */
        iirptr++;
        if (iirptr == IIR_order) iirptr = 0;
    }
    /* Calculate and sum oldest feedback term without incrementing iirptr */
    sum += _i_mul_fract_fb1_ii(coef[IIR_order-1],state[iirptr]);
    /* Calculate direct input contribution */
    sum += _i_mul_fract_fb1_ii(scale,autocorr_in[i]);
    autocorr_in[i] = sum;
    state[iirptr] = autocorr_in[i];
}
_reset_fract_bit();

/***********************************************************/
/* autocorr.c */
/* Copyright August 2004 by Matt Kleffner */
/* under the Creative Commons Attribution License */
/* */
/* Simple, unoptimized autocorrelation function */
/* for ECE 420 (TMS320C54X series) */
/* */
/* #defines expected in lab4b.h */
/* */
/* L: length of data in autocorr_in buffer */
/* N: length of data in autocorr_out buffer */
/* logL: log base 2 of L (used for scaling output) */
/* M: Largest positive lag of autocorrelation desired */
/* (must be < L and < N/2) */
/* */
/* 16-bit-limited input/output (must be defined elsewhere) */
/* autocorr_in: buffer for input data (L pts) */
/* autocorr_out: buffer for output data (N pts) */
/* N must be >= 2*M+1 */
/* assumed to be full of zeros at start */
/* output in zero-phase form */
/***********************************************************/
/* Works for TMS320C55X series */
#include "lab4b.h"
#include "intrinsics.h"

extern int autocorr_in[L];
extern int autocorr_out[N];

void autocorr(void)
{
    int i,j,temp;

    _set_f fract_bit();
    for(i=0;i<=M;++i)
    {
        long int sum=0;
        for(j=0;j<(L-i);++j)
        {
            temp = _i_mul_f fract_fb1_ii(autocorr_in[j],autocorr_in[j+i]);
            sum += temp;
        }
        autocorr_out[i]=(int)(sum >> logL);
    }
    _reset_f fract_bit();

    /* Copy values for negative indeces at end of buffer */
    for(i=1,j=N-1;i<=M;++i,--j)
    {
        autocorr_out[j]=autocorr_out[i];
    }
}

; c_fft_given_iirc.asm
; Designed for use in lab4b for ECE420

.ARM S_off
.CPL_on
.mmregs
.global _bit_rev_data
.global _fft_data
.global _state
.global _scale
.global _coef
.copy "macro.asm"
.sect ".data"

N .set 1024
.align 4*N
_bit_rev_data .space 16*2*N

.align 4*N
_fft_data .space 16*2*N

; IIR filter
.align 4
_coef
.word 0
.word 0
.word 0
.word -13421
_state
.space 16*4
_scale
.word 19345

.sect ".text"

1.6 Lab 5

1.6.1 Lab 5: Optimization Theory

1.6.1.1 Introduction to Code Optimization

Most practical DSP applications have a clock-cycle and/or memory budget. Initial implementations typically don’t meet these budgets; therefore, the code must be optimized. Code development usually follows six steps:

1. Develop algorithm on paper
2. Simulate in MATLAB
3. Develop and simulate more efficient implementations
4. Implement algorithm in C
5. Use library routines when available
6. Use optimizing compiler
7. Manually write assembly routines for key routines

1.6.1.2 Develop algorithm on paper

The algorithm to be implemented should first be designed on paper. In addition to equations describing the algorithm, its design should include inputs and outputs of each routine and a flow chart of the operation of the complete program. Never design an algorithm or a program at a computer. While it may be possible to implement some basic algorithms and programs this way, compilers cannot overcome any design flaws that are likely to be introduced.

42This content is available online at <http://cnx.org/content/m12380/1.2/>.
1.6.1.3 Simulate in Matlab

Before any C code is written, the algorithm should be developed and simulated in MATLAB since problems with the algorithm design can be found more easily. This is done by applying the algorithm to test vectors and inspecting and/or plotting the results. Once the algorithm is completely defined, C implementation can begin. Recall that in each of the previous labs a MATLAB simulation step was given before assembly implementation.

1.6.1.4 Develop and simulate more efficient implementations

An efficient algorithm used as few multiplications and additions as possible. Applying DSP theory to "simplify" the algorithm is a common way of doing this. The autocorrelation function is a simple example: a simple-minded way of implementing this is to compute sums for each lag, but careful inspection of the autocorrelation function reveals that it is actually a function of the absolute value of the lag. Therefore each value at a negative lag is identical to the value at the corresponding positive lag. Approximately half the apparent multiplications and additions are therefore required. If the entire autocorrelation sequence is needed, the autocorrelation can be optimized even further if it is treated as a convolution of two sequences. An FFT and an inverse FFT can then be used to compute the autocorrelation.

1.6.1.5 C Implementation

After the algorithm has been designed and optimized, an initial implementation in C is done. Tips on writing code with efficiency in mind can be found here[7]. It is important, however, to get a working implementation of the algorithm in a reasonable amount of time as some optimizations cannot be anticipated by the programmer. Some of these coding techniques in the URL reference can be applied later on when it is clear which routines need the most optimization. This implementation can serve as a reference implementation to compare the correctness and speed of optimized versions.

1.6.1.6 Library routines

If the algorithm uses common mathematical operations, such as the cosine and FFT operations, it is usually wise to use existing library routines instead of "reinventing the wheel." As many library routines are readily available from DSP manufacturers and over the internet, the first factor to consider in using a library is its license: do you have permission to use it in your application? Libraries can often be used freely for educational and research purposes, but any other use requires inspection of the library license.

The second factor to consider is the design goal of the library: was it designed for speed or low memory usage? Typically speed can be bought with more memory and vice-versa, so when selecting a library it is important to decide which budget (speed or memory) is more important with respect to the routine.

1.6.1.7 Compiler Optimization

Recall that the basic operation of a C compiler is to translate C source code into assembly instructions and then into an executable.

"Compiler optimization is used to improve the efficiency (in terms of running time or resource usage) of the executables output by a compiler. These techniques allow programmers to write source code in a straightforward manner, expressing their intentions clearly, while allowing the computer to make choices about implementation details that lead to efficient execution. Contrary to what the term might imply, this rarely results in executables that are perfectly "optimal" by any measure, only executables that are much improved compared to direct translation of the programmer's original source." - [7]

An optimizing compiler traditionally groups optimizations into phases. Each phase contains a series of optimizations (or transformations) that are performed in a fixed order. These phases are usually turned on
with command-line flags such as -01, -02, etc. Each flag indicates an optimization "level" where the level includes all of the lower levels. At higher optimization levels bugs in the code are sometimes introduced, so it is important to check the behavior of a compiler-optimized program against the reference implementation. Keep the highest optimization level that produces accurate code.

At this point the compiled code should be checked against the budgetary constraints. Is it fast enough? Does it fit in available memory? Total memory usage is placed in a file produced by the compiler (sometimes a command-line flag is needed for this). Speed can be measured in a couple of ways. The most common method is the use of a profiler. A profiler tracks the performance of the program, providing data on how many times each function is called, as well as how much time each function takes in terms of cycles and percentages of total program cycles. A simulator also allows clock cycles to be measured, typically by allowing the user to place breakpoints around sections of code to be measured. If the speed and memory properties of the compiled code fit the budget, optimization is finished. If not, some of the routines must be hand-written in assembly.

1.6.1.8 Write key assembly routines manually

Finally, if the budget cannot be met with the other optimization techniques some routines must be written in assembly. Manually-written assembly code is usually the most efficient, but it is labor-intensive and it is not portable to other architectures. Therefore it is done as a last resort and only on routines that absolutely require it for budget constraints to be met. This is done by rewriting the routine that consumes the largest portion of the budget, followed by the next largest budget-consuming routine and so on until the budget is met. It should be noted that this step is almost always required in embedded applications as current state-of-the-art C compilers and optimizers do not produce sufficiently fast and/or small code.

If meeting the budget is unexpectedly difficult, remember that no compiler optimization or assembler can effectively overcome a poor algorithm design or implementation. If you are confident that your implementation is fast and accurate, then the budget may be too tight for the application. Either some parts of the application must be removed (extra "features", for example) or an architecture with more resources must be used.

1.6.2 Lab 5: Spectrum Analyzer

In this lab you are to implement and optimize the a pseudo-noise (PN) sequence generator, IIR filter, and autocorrelation routines that are part of the previous lab’s PSD estimator. For the lab grade, you will be judged on the execution time of your system (memory usage need not be minimized).

1.6.2.1 Reference Implementation

After taking a look at the source code of the PSD estimator reference implementation, you will likely discover inefficiencies. This implementation is provided as the "reference implementation" of the optimization process and to define the expected input and output of the application. The computational efficiency of your code will be judged against this implementation. While the given code might serve as a starting point, you should do whatever you need to do to make your code as efficient as possible, while operating in an equivalent manner as the given code.

The exact portion of the code to be optimized is defined below. You may write in C, assembly, or any combination of the two; choose whatever will allow you to write the fastest code. The optimization process will be smoother if you plan for optimization before you begin any programming.

1.6.2.2 Optimization

Since a primary purpose of this lab is to learn optimization and efficient code techniques, your lab grade will be based primarily on the total execution time of your system. You are not required to

---

43 This content is available online at <http://cnx.org/content/m13805/1.3/>. 
optimize memory use. Note that by execution time we mean cycle count, not the number of instructions in your program. Remember that some of the TMS320C55xx instructions take more than one cycle. However, unlike the TMS320C54xx instructions, most operations take only one cycle and can be placed in parallel with other operations. Branch and repeat statements are the most common instructions that require several cycles to execute. Most C instructions take more than one cycle. The debugger can be used to determine the exact number of cycles used by your code. The instructions on how to do this can be found in Cycle Counts.\(^{44}\)

We will grade you based on the number of cycles used between the `rand_fillbuffer();` and `cfft((DATA *)fftdata,N, SCALE);` statements. Thus, you can optimize `rand_fillbuffer` function but optimizing the `fft` function will not help. Note that some instructions, like RPT, are non-repeatable instructions; their use may cause unnecessary glitches in I/O. For grading simplicity, your final code should not have modifications except between these two instructions, and \(M\) should be set to 31. If the number of cycles between the two points is variable, the maximum possible number of cycles will be counted. You must use the `dma.c` and `swi_process.h` files in `v:\ece420\55x\lab4` as provided by the TAs; these files may not be modified. We reserve the right to test your code by modifying the inputs.

1.6.2.3 Routine-Specific Optimization Tips

If you are programming the PN generator in assembly, you may wish to refer to the description of assembly instructions for logical operations in the \(C55x\) Mnemonic Instruction Set reference. Initialize the shift register to one. You can debug the PN output by comparing it to the output of the MATLAB code. Be prepared to prove to a TA that your PN generator works properly as part of your quiz.

Your IIR filtering routine can be debugged by writing an impulse followed by zeros in `autocorr_in` instead of `randsample`.

Your autocorrelation routine can be debugged by commenting out the IIR-filtering routine and writing the maximum DC value into `autocorr_in` in a similar manner as described the IIR-debugging step. Note that each of these tips is the most helpful if the output is inspected in memory.

1.6.2.4 Grading

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

- 2 points: Working code, implemented from scratch in assembly language or C.
- 5 points: Optimization. These points will be assigned based on your cycle counts and the optimizations you have made.
- 3 points: Oral quiz.

1.6.3 Lab 5: BPSK Transmitter\(^{45}\)

In this lab you are to implement and optimize a pseudo-noise (PN) sequence generator and a Binary Phase Shift Keying (BPSK) transmitter to encode and transmit the PN sequence. For the lab grade, you will be judged on the execution time of your system (memory usage need not be minimized).

1 Reference Implementation

You will find the source code for this lab in the following folder `v:\ece420\55x\bpsk_tx`. After taking a look at the source code for the BPSK transmitter reference implementation, you will likely discover inefficiencies. This implementation is provided as the ‘reference implementation’ of the optimization process and to define the expected input and output of the application. The computational efficiency of your code will be judged against this implementation. While the given source code might serve as a starting point,

\(^{44}\)http://cnx.org/content/m13805/latest/m14415

\(^{45}\)This content is available online at <http://cnx.org/content/m15202/1.1/>. 
you should do whatever you need to do to make your code as efficient as possible, while operating in an equivalent manner as the given code.

The exact portion of the code to be optimized is defined below. You may write in C, assembly, or any combination of the two; choose whatever will allow you to write the fastest code. The optimization process will be smoother if you plan for optimization before you begin any programming.

2 Optimization

Since a primary purpose of this lab is to learn optimization and efficient code techniques, your lab grade will be based primarily on the total execution time of your system. You are not required to optimize memory use. Note that by execution time we mean cycle count, not the number of instructions in your program. Remember that some of the TMS320C55xx instructions take more than one cycle. However, most operations take only one cycle and can be placed in parallel with other operations. Branch and repeat statements are the most common instructions that require several cycles to execute. Most C instructions take more than one cycle. The debugger can be used to determine the exact number of cycles used by your code. The instructions on how to do this can be found in Cycle Counts (m14415).

We will grade you based on the number of cycles used between the profile_me = 1; and profile_me = 0; statements. Thus, you can optimize code within this interval, but optimizing code outside of this interval will not help. For grading simplicity, your code should not have modifications except between these two instructions. If the number of cycles between these two points is variable, the maximum possible number of cycles will be counted. We reserve the right to test your code by modifying the inputs.

3 Routine Specific Optimization Tips

If you are programming the PN generator in assembly, you may wish to refer to the description of assembly instructions for logical operations in the C55x Mnemonic Instruction Set reference. Initialize the shift register to one. You can debug the PN output by comparing it to the output of the MATLAB code. Be prepared to prove to a TA that your PN generator works properly as part of your quiz. For more information about the PN generator, see Lab 4 Prelab.

Your BPSK transmitter can be debugged by looking at the output of the transmitter along with the corresponding PN values on channels one and two of the oscilloscope. The transmitter output corresponding to PN=1 should be 180 degrees out of phase with the transmitter output corresponding to PN=0. To verify that you are starting with a decent sine wave, you might try overwriting the PN sequence to a constant zero or one, which should produce a smooth sinusoid with no phase shifts at the transmitter’s output.

4 Grading

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

- 2 points: Working code, implemented from scratch in assembly language and/or C.
- 5 points: Optimization. These points will be assigned based on your cycle counts and the optimization you have made.
- 3 points: Oral quiz.
2.1 Digital Receiver

2.1.1 Digital Receiver: Carrier Recovery

2.1.1.1 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, if the digital carrier frequency is $\pi/2$ and the D/A is operating at 44.1 kHz, then the expected analog carrier frequency is $f_c = \frac{\pi}{2 \times 44.1} = 11.25$ kHz. If there is a slight change to the D/A sample rate (say $f_c = 44.05$ kHz), then there will be a corresponding change in the actual analog carrier frequency ($f_c = 11.0125$ 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.

\[1\] This content is available online at <http://cnx.org/content/m10478/2.16/>.

\[2\] "Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" <http://cnx.org/content/m10042/latest/>
2.1.1.1 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 2.1). A simple scheme for implementing an NCO is based on the following re-expression of the carrier sinusoid:

\[ \sin(\omega_c n + \theta_c) = \sin(\theta[n]) \]  \hspace{1cm} (2.1)

where \( \theta[n] = \omega_c n + \theta_c \) (\( \omega_c \) and \( \theta_c \) 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

\[ \theta[n] = \theta[n-1] + \omega_c \]  \hspace{1cm} (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:

\[ \theta[n] = \theta[n-1] + \omega_c + d_{pd}[n] \]  \hspace{1cm} (2.3)

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

2.1.1.2 Phase detector

The goal of the PLL is to maintain a demodulating sine and cosine that match the incoming carrier. Suppose \( \omega_c \) is the believed digital carrier frequency. We can then represent the actual received carrier frequency as
the expected carrier frequency with some offset, \( \omega_c = \omega_c + \hat{\theta} [n] \). The NCO generates the demodulating sine and cosine with the expected digital frequency \( \omega_c \) and offsets this frequency with the output of the loop filter. The NCO frequency can then be modeled as \( \omega_c = \omega_c + \hat{\theta} [n] \). Using the appropriate trigonometric identities\(^3\), the in-phase and quadrature signals can be expressed as

\[
\begin{align*}
z_0 [n] &= \frac{1}{2} \left( \cos \left( \hat{\theta} [n] - \hat{\theta} [n] \right) + \cos \left( 2\omega_c + \hat{\theta} [n] + \hat{\theta} [n] \right) \right) \\
z_Q [n] &= \frac{1}{2} \left( \sin \left( \hat{\theta} [n] - \hat{\theta} [n] \right) + \sin \left( 2\omega_c + \hat{\theta} [n] + \hat{\theta} [n] \right) \right)
\end{align*}
\]

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

\[
\begin{align*}
y_1 [n] &= \frac{1}{2} \cos \left( \hat{\theta} [n] - \hat{\theta} [n] \right) \\
y_Q [n] &= \frac{1}{2} \sin \left( \hat{\theta} [n] - \hat{\theta} [n] \right)
\end{align*}
\]

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

\[
\forall \theta, \text{small : } (\sin (\theta) \approx \theta)
\]

and let the current value of the quadrature channel approximate the phase difference: \( z_Q [n] \approx \hat{\theta} [n] - \hat{\theta} [n] \). With the exception of the sign error, this difference is essentially how much we need to offset our NCO frequency\(^4\). 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\(^5\).

### 2.1.1.3 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,

\[
y [n] = \beta x [n] + \alpha y [n - 1]
\]

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.

### 2.1.1.2 MATLAB Simulation

Simulate the PLL system shown in Figure 2.1 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 \( 2\pi \) in the phase update relation. This can be done by setting \( \theta [n] = \theta [n] - 2\pi \) whenever \( \theta [n] > 2\pi \).

---

\(^3\) \( \cos (A) \cos (B) = \frac{1}{2} (\cos (A - B) + \cos (A + B)) \) and \( \cos (A) \sin (B) = \frac{1}{2} (\sin (B - A) + \sin (A + B)) \).

\(^4\) If \( \hat{\theta} [n] - \hat{\theta} [n] > 0 \) then \( \hat{\theta} [n] \) is too large and we want to decrease our NCO phase.

\(^5\) What should the relationship between the I and Q branches be for a digital QPSK signal?
Figure 2.2 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.

![Graph showing PLL sub-system output for BPSK modulated carrier.](image)

Figure 2.2: 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?

### 2.1.1.3 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.

#### 2.1.1.3.1 Sine-table interpolation

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

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

You may simplify the interpolation problem by using "lower-neighbor" interpolation, i.e., by using the integer part of your pointer. Note that the full-precision, non-integer pointer must be maintained in memory so that the fractional part is allowed to accumulate and carry over into the integer part; otherwise, your phase will not be accurate over long periods. For a long enough sine table, this approximation will adjust the NCO frequency with sufficient precision.\(^6\)

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 \( 256 \).\(^7\) 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 \( p = \frac{\pi}{8} \frac{N}{2\pi} = 16 \). 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 \( 16 \times 2^7 = 2048 \).\(^8\)

2.1.1.3.2 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 \( \alpha \) 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?

2.1.2 Digital Receivers: Symbol-Timing Recovery for QPSK\(^9\)

2.1.2.1 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 2.1.1)) 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

---

\(^6\)Of course, nearest-neighbor interpolation could be implemented with a small amount of extra code.

\(^7\)How is this similar to the addition modulo \( 2\pi \) discussed in the MATLAB Simulation (Section 2.1.1.2: MATLAB Simulation)?

\(^8\)If this value were 2049, what would be the output frequency of the NCO?

\(^9\)This content is available online at <http://cnx.org/content/m10485/2.14/>. 
CHAPTER 2. PROJECT LABS

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 2.3 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, $s_I[n]$ and $s_Q[n]$. These operations are explained in Digital Receivers:
Carrier Recovery for QPSK (Section 2.1.1). The remaining operations are explained in this module. Both
branches are fed through a matched filter and re-sampled at the symbol rate. The matched filter is simply
an FIR filter with an impulse response matched to the transmitted pulse. It aids in timing recovery and
helps suppress the effects of noise.

Figure 2.3: Digital receiver system

If we consider the square wave shown in Figure 2.4 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 2.4: Clean BPSK waveform.
However, in the presence of noise, the received waveform may look like that shown in Figure 2.5. 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 $u[n] - u[n - T_{sym}]$, where $u[n]$ is the unit step function, for the simple rectangular pulse shape used in Digital Transmitter: Introduction to Quadrature Phase-Shift Keying$^{10}$. $^{11}$Figure 2.6 and Figure 2.7 show the result of passing both the clean and noisy signal through the matched filter.

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

11For 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 2.6: Averaging filter output for clean input.
CHAPTER 2. PROJECT LABS

Figure 2.7: 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 2.5.

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.

2.1.2.1.1 Early/late sampling

One simple method for recovering symbol timing is performed using a delay-locked loop (DLL). Figure 2.8 is a block diagram of the necessary components.
Consider the sawtooth waveform shown in Figure 2.6, 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 2.9 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 2.10 and Figure 2.11 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 2.9, the on-time sample is already at the peak, and the receiver knows that peaks are spaced by $T_{\text{symb}}$ samples. If it then takes the next on-time sample $T_{\text{symb}}$ samples after this on-time sample, it will be at another peak.
2. In Figure 2.10, the on-time sample is too early. Taking an on-time sample $T_{\text{symb}}$ 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 2.11, the on-time sample is too late. Taking an on-time sample $T_{\text{symb}}$ 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 2.9: Sampling at a peak.

Figure 2.10: Sampling too early.
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 2.9, Figure 2.10, and Figure 2.11. 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.

### 2.1.2.1.2 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 $T_{symb} - 1$, $T_{symb}$, or $T_{symb} + 1$, according to the offset decision block.

### 2.1.2.2 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 2.8. 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 2.12 and Figure 2.13 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 2.12: Symbol sampling before DLL lock.
2.1.2.3 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.

2.1.2.4 Extensions

As your final project will require some modification to the discussed BPSK signaling, you will want to refer to the listed references, (see Proakis[11] and Blahut[3], 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, ...])?
- What would the matched filter look like for different symbol shapes?

Figure 2.13: Symbol sampling after DLL lock.
• What other methods of symbol-timing recovery are available for your application?
• How does adding decision statistics help suppress the effects of noise?

2.2 Audio Effects

2.2.1 Audio Effects: Real-Time Control using RTDX\textsuperscript{12}

2.2.1.1 Implementation

For this exercise, you will extend the system from Audio Effects: Using External Memory (Section 2.2.2) to generate a feedback-echo effect. You will then extend this echo effect to use the USB port on the DSP EVM. The USB 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.

2.2.1.1.1 Feedback system implementation

First, modify code from Audio Effects: Using External Memory (Section 2.2.2) to create the feedback-echo system shown in Figure 2.14. A one-tap feedback-echo is a simple audio effect that sounds remarkably good. You will use both channels of input by summing the two inputs so that either or both may be used as an input to the system. Also, send several test signals to the six-channel board’s D/A converters:

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

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

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

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

\textsuperscript{12}This content is available online at <http://cnx.org/content/m15465/1.1/>.
2.2.1.1.2 MATLAB interface implementation

After studying the MATLAB interface outlined at the end of Using RTDX with a MATLAB GUI (Section 3.2.1), write MATLAB code to send commands to the USB 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 these 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.

2.2.2 Audio Effects: Using External Memory

2.2.2.1 Introduction

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

In this exercise, you will use external memory to implement a long audio delay and an audio echo.

2.2.2.2 Delay and Echo Implementation

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

2.2.2.2.1 Fixed-length delay implementation

First, implement the 131,072-sample delay shown in Figure 2.15. Store the samples in a buffer in SDRAM with alignment 0x40000h. Do this by adding a line in your userlinker.cmd file to create a new section in memory: Allocate a buffer of the size you need in the assembly file. Be sure to place it in the section you just created. Since the memory address is now greater than 16-bits, you will need to do a MOV dbl() in order to get the full address into an auxiliary register.

```plaintext
.bigbuffer: align=0x40000 {} > SDRAM
```

![Figure 2.15: Fixed-Length Delay](image)

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?) If you want, you may assume the delay will be a power-of-2 for easier implementation.

2.2.2.2.2 Variable-delay implementation

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

---

13\[This content is available online at <http://cnx.org/content/m15240/1.1/>.\]
Verify that your code works as expected by timing the delay from input to output and ensuring that it is approximately the correct length.

2.2.2.3 Feedback-echo implementation

Last, copy and modify your code so that the value taken from the end of the variable delay from Variable-delay implementation (Section 2.2.2.2: 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 2.16 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 2.16: Feedback Echo](image)

2.3 Surround Sound

2.3.1 Surround Sound: Chamberlin Filters

2.3.1.1 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.

2.3.1.2 Filter Topology

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

\[
H(z) = \frac{F_c^2 z^{-1}}{1 - (2 - (F_c Q_c - F_c^2)) z^{-1} + (1 - F_c Q_c) z^{-2}}
\]  

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. The topology of the filter is shown in Figure 2.17. 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

14This 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 2.17: Chamberlin Filter Topology

Figure 2.18 and Figure 2.19 show how the response of the filter varies with $Q_c$ and $F_c$. 
Figure 2.18: Chamberlin filter responses for various $Q_c$ ($F_c = 0.3$)
2.3.1.3 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?

2.3.2 Surround Sound: Passive Encoding and Decoding\textsuperscript{15}

2.3.2.1 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

\textsuperscript{15}This content is available online at <http://cnx.org/content/m10484/2.13/>.
principles as the more sophisticated commercial systems. Significantly more technical information regarding Dolby Pro Logic can be found at Gundry [6].

2.3.2.2 Encoder

You will create a MATLAB implementation of the passive encoder given by the block diagram in Figure 2.20.

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

The basic components of the encoder are multipliers, adders, a Hilbert transform, a band-pass filter, and a Dolby Noise Reduction encoder. If you wish to implement Dolby Noise Reduction, refer to Dressler [5]. The other components are discussed below.

The transfer function of the Hilbert Transform is shown in Figure 2.21. The Hilbert Transform is an ideal (unrealizable) all-pass filter with a phase shift of $-90^\circ$. 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 $-90^\circ$ phase-shifted version of the original signal. Plot each signal to confirm your expectations.
For the band-pass filter, design a second-order Butterworth filter using the `butter` function in MATLAB.

2.3.2.2.1 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.

2.3.2.3 Decoder

Implement the passive decoder shown in Figure 2.22 on the DSP. Use an appropriate time delay based on the distance between the front and back speakers and the speed of sound.

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?
2.3.2.4 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 \( (f \leq 80\text{Hz}) \). 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 2.3.1).

2.4 Adaptive Filtering

2.4.1 Adaptive Filtering: LMS Algorithm\(^{16}\)

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 \( y[n] \).

\[ x[n] \rightarrow H \rightarrow d[n] \rightarrow e[n] \]

\[ W \rightarrow y[n] \rightarrow + \]

\[ e[n] \rightarrow \]

**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

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

(2.11)

\(^{16}\)This content is available online at <http://cnx.org/content/m10481/2.14/>. 
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 $\mu$ is a step-size, which controls the amount of gradient information used to update each coefficient. After repeatedly adjusting each coefficient in the direction opposite to the gradient of the error, the adaptive filter should converge; that is, the difference between the unknown and adaptive systems should get smaller and smaller.

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

$$\frac{\partial}{\partial h[i]} \left( (|e|)^2 \right) = \frac{2}{\partial h[i]} (e) e$$

$$= \frac{2}{\partial h[i]} (d - y) e$$

$$= \left( \frac{2}{\partial h[i]} \left( d - \sum_{i=0}^{N-1} (h[i] x [n-i]) \right) \right) e$$

(2.12)

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

(2.13)

which in turn gives us the final LMS coefficient update,

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

(2.14)

The step-size $\mu$ directly affects how quickly the adaptive filter will converge toward the unknown system. If $\mu$ 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 $\mu$ 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\(^ {17}\).

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.

\(^{17}\)"IIR Filtering: Filter-Design Exercise in MATLAB" <http://cnx.org/content/m10623/latest/>
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\(^1\), 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 \(\mu\) 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 [7] 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\(^2\)

**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

\[
r_{xy}[l] = \sum_{n=-\infty}^{\infty} (x[n]y[n-l])
\]

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

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

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

\(^1\)"Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" [http://cnx.org/content/m10042/latest/]

\(^2\)This content is available online at [http://cnx.org/content/m10482/2.19/].
where \( s[n], n = \{-P, (-P) + 1, \ldots, N - 1\} \) are the known samples (see Figure 2.24) and the \( \frac{1}{N} \) is a normalizing factor.

\[
\begin{align*}
0 & \quad N - 1 \\
- P & \quad - l & \quad 0 & \quad N - l - 1 & \quad N - 1 \\
\text{multiply and accumulate to get } r_{ss}[l]
\end{align*}
\]

**Figure 2.24:** Computing the autocorrelation coefficients

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

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

(2.17)

where the summation is over \( N - l \) products (the samples \( \{s[-P], \ldots, s[-1]\} \) 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 [12]

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

(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

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

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

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

(2.19)

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

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

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

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

$$a = R^{-1}r$$

(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 [12] (p.115).

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

for $i = 1$ to $P$.

Steps

1. $k_i = \frac{1}{E_{i-1}} \left( r_{ss}[i] - \sum_{j=1}^{i-1} (\alpha_{j,i-1} r_{ss}[i-j]) \right)$
2. $\alpha_{j,i} = \alpha_{j,i-1} - k_i \alpha_{i-j,i-1} \quad j = \{1, \ldots, i-1\}$
   - $\alpha_{i,i} = k_i$  
3. $E_i = (1 - k_i^2) E_{i-1}$
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 $a$ coefficients must be factored into second-order stages with very small gains to yield a more stable implementation.

\[
\begin{align*}
\text{Figure 2.26:} & \quad \text{IIR lattice filter implementation.} \\
\end{align*}
\]

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 (mop, ace, easy, go, but) 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, $\frac{r_{xx}[T]}{r_{xx}[0]} \approx 0.25$, where $T$ is the pitch period.
2.5.2 Speech Processing: LPC Exercise in MATLAB\textsuperscript{20}

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 \texttt{xcorr} MATLAB function to compare with your own version of this function. At what time shift \( t \) 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 \( k \), \( E \), \( \alpha \), 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 \texttt{wavread}, \texttt{wavwrite}, \texttt{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 \texttt{levinson} or \texttt{1pc} 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\textsuperscript{21}

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 \texttt{thr6.asm}\textsuperscript{22} 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\textsuperscript{23} 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

\textsuperscript{20}This content is available online at <http://cnx.org/content/m10824/2.5/>.
\textsuperscript{21}This content is available online at <http://cnx.org/content/m10824/2.5/>.
\textsuperscript{22}http://cnx.org/content/m10825/latest/thr6.asm
\textsuperscript{23}Core File: Accessing External Memory on TI TMS320C54x" <http://cnx.org/content/m10823/latest/>
MATLAB.

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. 88) of the algorithm. Refer to the Applications Guide[?] and the subc instruction for a routine that performs integer division.

2.6 Video Processing

2.6.1 Real-time video processing in Linux with an IIDC camera: Lab

2.6.1.1 Download, compile, and run the example application

- From your home directory, download the tar file to the (backed-up) w_drive subdirectory.
- Open a terminal and type: cd w_drive
- Extract the files with tar -xvzf projectlab_linuxvideo.tar.gz. Typing a few letters of the filename followed by the Tab key will automatically fill in the file name.
- cd projectlab_linuxvideo
- Compile the application by running ./makescript. Open this file in a text editor to see what commands are executed.
- Plug in a Unibrain Fire-i RAW Color Digital OEM Board Camera and run the application: ./videoexample

Video from the camera should now be duplicated in the four quadrants of the screen, as well as some text, a line drawing of a moving sine wave, and a colored box. Press the Esc key to exit the application, or type Ctrl-C at the terminal.

The file videoexample.c contains comments and code illustrating how the video is transferred and displayed. Some additional comments on how waveforms are plotted are contained in screengraphics.h.

2.6.1.2 Copy and modify the example

Make a copy of the example application: cp videoexample.c myprojectlab.c ; cp makescript mymakescript. Modify mymakescript to compile myprojectlab instead of videoexample. Modify the Bayer downsampling code to use two green input pixels, one red input pixel, and one blue input pixel to compute each pixel in the color output image. Currently, only one green pixel is used. In other words, each non-overlapping, 2x2 square of single-color pixels in the input image will map to one color pixel in the output image. Can you think of and implement a method that more efficiently indexes (no multiplications, few additions) the raw-input and downsampled-output images?

Now modify myprojectlab.c so that it displays three different sets of processed video, accessible via the 1-3 keys, with 1 being the default. Make the contents of each screen as follows:

Screen 1: simple processing

- Top left: original downsampled video
- Top right: horizontally-flipped video
- Bottom left: vertically-flipped video
- Bottom right: inverted video - black becomes white and white becomes black, etc. Note that each pixel has a max value of 255.

24This content is available online at <http://cnx.org/content/m15246/1.2/>.
25http://www.ifp.uiuc.edu/~klenner/projectlab_linuxvideo.tar.gz
26http://www.libsdl.org
27http://en.wikipedia.org/wiki/Bayer_pattern
Screen 2: RGB display

- Top left: original downsampled video
- Top right: red channel
- Bottom left: green channel
- Bottom right: blue channel
- When the user presses the g key, toggle between displaying the channels as shades of gray or the channel colors. Which channel best approximates a grayscale version of the image?

Screen 3: YUV conversion and display

- Top left: original downsampled video
- Top right: Y channel
- Bottom left: U channel
- Bottom right: V channel
- When the user presses the a key, toggle between displaying the channels in their default ranges and auto-contrasting the channels (with a single offset and single scale factor per quadrant image) so that the maximum RGB-color component of each quadrant image is 255 and the minimum 0. Which channel contains the grayscale image?
- Note that displaying one of the YUV channels in RGB form requires conversion to YUV, selecting a channel, and converting back to RGB. Think carefully about how to do the conversions efficiently before programming them; eliminate redundant and/or unnecessary operations. In other words, six matrix transformations per video frame are not required to display the YUV channels. In fact, three matrix transformations are not required. An internet search should provide many references to RGB and YUV conversion.

Make your code as modular as possible so that it is easy to follow and debug; create functions, inline functions, and/or macros for operations that will be repeatedly performed, and group similar functions/macros into separate files.

2.6.2 Introduction to the IDK

2.6.2.1 Introduction

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

You will complete an introductory exercise to gain familiarity with the IDK programming environment. In the exercise, you will modify a C skeleton to horizontally flip and invert video input from the camera. The output of your video processing algorithm will appear in the top right quadrant of the monitor. In addition, you will analyze existing C code that implements filtering and edge detection algorithms to gain insight into IDK programming methods. The output of these "canned" algorithms, along with the unprocessed input, appears in the other quadrants of the monitor.

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

2.6.2.2 Video Processing Setup

The camera on the video processing lab bench generates an analog video signal in NTSC format. NTSC is a standard for transmitting and displaying video that is used in television. The signal from the camera is connected to the "composite input" on the IDK board (the yellow plug). This is illustrated in Figure 2-1 on

---

28This content is available online at <http://cnx.org/content/m10926/2.7/>. 
Notice that the IDK board is actually two boards stacked on top of each other. The bottom board contains the C6711 DSP, where your image processing algorithms will run. The top board is the daughterboard, which contains hardware for interfacing with the camera input and monitor output. For future video processing projects, you may connect a video input other than the camera, such as the output from a DVD player. The output signal from the IDK is in RGB format, so that it may be displayed on a computer monitor.

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

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

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

The TI C6711 DSP uses a different instruction set than the 5400 DSP’s you are familiar with in lab. The IDK environment was designed with high-level programming in mind, so that programmers would be isolated from the intricacies of assembly programming. Therefore, we strongly suggest that you do all your programming in C. Programs on the IDK typically consist of a main program that calls an image processing routine. The image processing routine may make several calls to specialized functions. These specialized functions consist of an outer wrapper and an inner component. The component performs processing on one line of an image. The wrapper oversees the processing of the entire image, using the IDM to move data back and forth between internal memory and external memory. In this lab, you will modify a component to implement the flipping and inverting algorithm.

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

2.6.2.3 Code Overview

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

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

The call to img_proc() takes us to the file img_proc.c. First, several variables are declared and defined. The variable quadrant will denote on which quadrant of the screen we currently want output; out_ptr will point to the current output spot in the output image; and pitch refers to the byte offset between two lines. This function is the high level control for our image-processing algorithm. See algorithm flow (Figure 2.27).
The first function called is the `pre_scale_image` function in the file `pre_scale_image.c`. The purpose of this function is to take the 640x480 image and scale it down to a quarter of its size by first downsampling the input rows by two and then averaging every two pixels horizontally. The internal and external memory spaces in the scratch pad are used for this task. The vertical downsampling will occur when only every other line is read into the internal memory from the input image. Within internal memory, we will operate on two lines of data (640 columns/line) at a time, averaging every two pixels (horizontal neighbors) and producing two lines of output (320 columns/line) that are stored in the external memory.

To accomplish this, we will need to take advantage of the IDM by initializing the input and output streams. At the start of the function, two instantiations of a new structure `dstr_t` are declared. You can view the structure contents of `dstr_t` on p. 2-11 of the IDK Programmer's Guide[1]. The structure contents are defined with calls to `dstr_open()`. This data flow for the pre-scale is shown in data flow (Figure 2.28).
Figure 2.28: Data flow of input and output streams.

To give you a better understanding of how these streams are created, let’s analyze the parameters passed in the first call to `dstr_open()`:

2.6.2.3.1 **External address: in_image->data**
This is a pointer to the place in memory serving as the source of our input data (it’s the source because the last function parameter is set to DSTR_INPUT).

2.6.2.3.2 **External size: (rows + num_lines) * cols = (240 + 2) * 640**
This is the total size of our input data. We will only be taking every other line from `in_image->data`, so only 240 rows. The extra two rows are for buffer.

2.6.2.3.3 **Internal address: int_mem**
This is a pointer to an 8x640 lexographic array, specifically `scratchpad->int_data`. This is where we will be putting the data on each call to `dstr_get()`.

2.6.2.3.4 **Internal size: 2 * num_lines * cols = 2 * 2 * 640**
The size of space available for data to be input into `int_mem` from `in_image->data`. Because double buffering is used, `num_lines` is set to 2.

2.6.2.3.5 **Number of bytes/line: cols = 640, Number of lines: num_lines = 2**
Each time `dstr_get()` is called, it will return a pointer to 2 lines of data, 640 bytes in length.
2.6.2.3.6 External memory increment/line: \( \text{stride} \times \text{cols} = 1 \times 640 \)

Left as an exercise.

2.6.2.3.7 Window size: 1 for double buffered

The need for the window size is not really apparent here. It will become apparent when we do the 3x3 block convolution. Then, the window size will be set to 3. This tells the IDM to send a pointer to 3 lines of data when \( \text{dstr_get()} \) is called, but only increment the stream's internal pointer by 1 (instead of 3) the next time \( \text{dstr_get()} \) is called. This is not a parameter when setting up an output stream.

2.6.2.3.8 Direction of input: DSTR_INPUT

Sets the direction of data flow. If it had been set to DSTR_OUTPUT (as done in the next call to \( \text{dstr_open()} \)), we would be setting the data to flow from the Internal Address to the External Address.

Once our data streams are setup, we can begin processing by calling the component function \( \text{pre_scale()} \) (in \( \text{pre_scale.c} \)) to operate on one block of data at a time. This function will perform the horizontal scaling by averaging every two pixels. This algorithm operates on four pixels at a time. The entire function is iterated within \( \text{pre_scale_image()} \) 120 times, which is the number of rows in each quadrant. Before \( \text{pre_scale_image()} \) exits, the data streams are closed, and one line is added to the top and bottom of the image to provide context necessary for the next processing steps. Now that the input image has been scaled to a quarter of its initial size, we will proceed with the four image processing algorithms. In \( \text{img_proc.c} \), the \( \text{set_ptr()} \) function is called to set the variable \( \text{out_ptr} \) to point to the correct quadrant on the 640x480 output image. Then \( \text{copy_image()} \), \( \text{copy_image.c} \), is called, performing a direct copy of the scaled input image into the lower right quadrant of the output.

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

In \( \text{conv3x3()} \), you will see that we perform a 3x3 block convolution, computing one line of data with the low pass filter mask. Note here that the variables \( \text{IN1[i]} \), \( \text{IN2[j]} \), and \( \text{IN3[k]} \) all grab only one pixel at a time. This is in contrast to the operation of \( \text{pre_scale()} \) where the variable \( \text{in_ptr[i]} \) grabbed 4 pixels at a time. This is because \( \text{in_ptr} \) was of type unsigned int, which implies that it points to four bytes of data at a time. \( \text{IN1} \), \( \text{IN2} \), and \( \text{IN3} \) are all of type unsigned char, which implies they point to a single byte of data. In block convolution, we are computing the value of one pixel by placing weights on a 3x3 block of pixels in the input image and computing the sum. What happens when we are trying to compute the rightmost pixel in a row? The computation is now bogus. That is why the wrapper function copies the last good column of data into the two rightmost columns. You should also note that the component function ensures output pixels will lie between 0 and 255.

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

The Sobel algorithm convolves two masks with a 3x3 block of data and sums the results to produce a single pixel of output. This algorithm approximates a 3x3 nonlinear edge enhancement operator. The brightest edges in the result represent a rapid transition (well-defined features), and darker edges represent smoother transitions (blurred or blended features).
2.6.2.4 Using the IDK Environment

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

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

```bash
copy V:\ece320\idk\c6000\IDK\Examples\NTSC\img_proc
copy V:\ece320\idk\c6000\IDK\Drivers\include
```

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

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

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

To compile all of the files when the ".out" file has not yet been generated, you need to use the "Rebuild All" command. The rebuild all command is accomplished by clicking the button displaying three little red arrows pointing down on a rectangular box. This will compile every file the `main.c` program uses. If you've only changed one file, you only need to do a "Incremental Build," which is accomplished by clicking on the button with two little blue arrows pointing into a box (immediately to the left of the "Rebuild All" button). Click the "Rebuild All" button to compile all of the code. A window at the bottom of Code Composer will tell you the status of the compiling (i.e., whether there were any errors or warnings). You might notice some warnings after compilation - don't worry about these.

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

2.6.2.5 Implementation

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

```
flip_invert(in_data, out_data, cols);
```

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

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

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

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

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

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

### 2.6.3 Video Processing Manuals

#### 2.6.3.1 Essential documentation for the 6000 series TI DSP

The following documentation will certainly prove useful:

- The IDK Programmer's Guide
- The IDK User's Guide
- The IDK Video Device Drivers User's Guide

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

### 2.6.4 Video Processing Part 1: Introductory Exercise

#### 2.6.4.1 Introduction

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

---

29 This content is available online at [http://cnx.org/content/m10889/2.5/].
30 [http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf](http://www-s.ti.com/sc/psheets/spru495a/spru495a.pdf)
33 [http://www.ti.com](http://www.ti.com)
34 This content is available online at [http://cnx.org/content/m11987/1.2/].
the IDK, the video processing lab bench is equipped with an NTSC camera and a standard color computer monitor.

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

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

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

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

2.6.4.1.1 Important Documentation

The following documentation will certainly prove useful:

- The IDK User's Guide\(^{35}\) Section 2 is the most important.
- The IDK Video Device Drivers User's Guide\(^{36}\). The sections on timing are not too important, but pay attention to the Display and Capture systems and have a good idea of how they work.
- The IDK Programmer's Guide\(^{37}\). Sections 2 and 5 are the ones needed. Section 2 is very, very important in Project Lab 2. It is also useful in understanding "streams" in project lab 1.

**Note:** Other manuals may be found on TI's website\(^ {38}\) by searching for TMS320C6000 IDK.

2.6.4.2 Video Processing - The Basics

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

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

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

The Image Data Manager (IDM) is a set of routines responsible for moving data between on-chip internal memory, and external memory on the board, during processing. The IDM helps the programmer by taking care of the pointer updates and buffer management involved in transferring data. Your DSP algorithms

---

[^38]: [http://www.ti.com](http://www.ti.com)
will read and write to internal memory, and the IDM will transfer this data to and from external memory. Examples of external memory include temporary "scratch pad" buffers, the input buffer containing data from the camera, and the output buffer with data destined for the RGB output.

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

The T1 C6711 DSP uses a different instruction set than the 5400 DSP's you are familiar with in lab. The IDK environment was designed with high level programming in mind, so that programmers would be isolated from the intricacies of assembly programming. Therefore, we strongly suggest that you do all your programming in C. Programs on the IDK typically consist of a main program that calls an image processing routine.

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

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

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

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

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

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

2.6.4.3 Code Description

2.6.4.3.1 Overview and I/O

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

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

The tskMainFunc(), in main.c, opens the handles to the board for image capture (VCAP_open()) and to the display (VCAP_open()) and calls the grayscale function. Here, several data structures are instantiated that are defined in the file img_proc.h. The IMAGE structures will point to the data that is captured by the FPGA and the data that will be output to the display. The SCRATCH_PAD structure points to our internal and external memory buffers used for temporary storage during processing. LPF_PARAMS is used to store filter coefficients for the low pass filter.
The call to `img_proc()` takes us to the file `img_proc.c`. First, several variables are declared and defined. The variable quadrant will denote on which quadrant of the screen we currently want output; out_ptr will point to the current output spot in the output image; and pitch refers to the byte offset (distance) between two lines. This function is the high level control for our image-processing algorithm. See algorithm flow.

![Algorithm Flow Diagram](image)

**Figure 2.29:** Algorithm Flow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In conv3x3(), you will see that we perform a 3x3 block convolution, computing one line of data with the low pass filter mask. Note here that the variables IN1[i], IN2[i], and IN3[i] all grab only one pixel at a time. This is in contrast to the operation of pre_scale() where the variable in_ptr[i] grabbed 4 pixels at a time. This is because in_ptr was of type unsigned int, which implies that it points to four bytes (the size of an unsigned int is 4 bytes) of data at a time. IN1, IN2, and IN3 are all of type unsigned char, which implies they point to a single byte of data. In block convolution, we are computing the value of one pixel by placing weights on a 3x3 block of pixels in the input image and computing the sum. What happens when we are trying to compute the rightmost pixel in a row? The computation is now bogus. That is why the wrapper function copies the last good column of data into the two rightmost columns. You should also note that the component function ensures output pixels will lie between 0 and 255. For the same reason we provided the two extra "copied" lines when performing the prescale.

Back in img_proc.c, we can begin the edge detection algorithm, sobel_image(), for the lower left quadrant of the output image. This wrapper function, located in sobel_image.c, performs edge detection by utilizing the assembly written component function sobel() in sobel.asm. The wrapper function is very similar to the others you have seen and should be straightforward to understand. Understanding the assembly file is considerably more difficult since you are not familiar with the assembly language for the c6711 DSP. As you’ll see in the assembly file, the comments are very helpful since an "equivalent" C program is given there.
The Sobel algorithm convolves two masks with a 3x3 block of data and sums the results to produce a single pixel of output. One mask has a preference for vertical edges while the other mask for horizontal ones. This algorithm approximates a 3x3 nonlinear edge enhancement operator. The brightest edges in the result represent a rapid transition (well-defined features), and darker edges represent smoother transitions (blurred or blended features).

2.6.4.4 Part One

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

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

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

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

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

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

To compile all of the files when the "out" file has not yet been generated, you need to use the "Rebuild All" command. The rebuild all command is accomplished by clicking the button displaying three little red arrows pointing down on a rectangular box. This will compile every file the main.c program uses. If you’ve only changed one file, you only need to do a "Incremental Build," which is accomplished by clicking on the button with two little blue arrows pointing into a box (immediately to the left of the "Rebuild All" button). Click the "Rebuild All" button to compile all of the code. A window at the bottom of Code Composer will tell you the status of the compiling (i.e., whether there were any errors or warnings). You might notice some warnings after compilation - don’t worry about these.

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

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

```
flip_invert(in_data, out_data, cols);
```

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

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

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

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

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

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

2.6.5 Video Processing Part 2: Grayscale and Color

2.6.5.1 Introduction

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

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

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

You may want to refer to the following TI manuals:

- IDK User's Guide
- IDK Video Device Drivers User's Guide
- IDK Programmer’s Guide

Section 2 is very, very important in this lab.

---

39 This content is available online at <http://cnx.org/content/m11988/1.2/>.
40 http://www-s.ti.com/sc/peheets/spru494a/spru494a.pdf
42 http://www-s.ti.com/sc/peheets/spru495a/spru495a.pdf
CHAPTER 2. PROJECT LABS

2.6.5.2 Prelab

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

2.6.5.2.1 Grayscale

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

\[
\text{auto_contrast_find_extrema(in_data, min, max, col)}
\]

2.6.5.2.2 Color

The NTSC camera acquires images in the color format YCbCr, where Y represents luminosity, Cb the blue component, and Cr the red component. Each image must be converted to 16-bit RGB for output on a standard color computer monitor. The function \text{"ycbc422pl_to_rgb565"} performs this conversion. Knowing how this function converts each pixel to RGB is relatively unimportant, however, knowing the packed (5:6:5) RGB format is essential.

Before we ignore the \text{ycbc422pl_to_rgb565} function completely, it is useful to look at how it operates. Find the run time of the function by examining the file \text{"ycbc422pl_to_rgb565.c"} and note that it must convert an even number of pixels at a time. If it were possible to have this function process the whole color image at in one function call, how many clock cycles would the function take? Since we are limited in the number of rows we can modify at a time, how many clock cycles should it take to process the whole image one row at a time? To demonstrate the overhead needed for this function, note how many clock cycles the function would take if it converted the whole image two pixels at a time.

![Figure 2.31](image.png)

\textbf{Figure 2.31:} RGB (5:6:5). A packed RGB pixel holds 5 bits for red, 6 bits for green, and 5 bits for blue.

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

\textbf{Example:}

- \text{int black = 0x00;} // black = 0
- \text{int foo_h = 0xF0;} // foo_h = 240
Another thing to note is that each pixel requires two bytes of memory, requiring two memory access operations to alter each pixel. Also NOTE that in a row of input color data, the indexing starts at 1. Thus RGB[1] contains red/green data and then RGB[2] contains the green/blue data — both for the first pixel.

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

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

```c
int pixel;
```

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

### 2.6.5.3 Implementation

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

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

#### 2.6.5.3.1 Grayscale

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

**Figure 2.32:** (left) Frequency of a grayscale image with pixel intensities ranging in value from 32 to 128, and (right) Frequency of the same grayscale image after performing an auto-contrast.

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

Example:

- `int opposite, adjacent;`
- `float tan;`
• \( \tan = \frac{(\text{float}\ \text{opposite})}{(\text{float}\ \text{adjacent})}; \)

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

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

2.6.5.3.2 Color

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

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

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

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

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

2.6.5.3.2.1 Code Briefing

The code provided merely performs a color conversion required to go from the input NTSC image to the output RGB image. The relevant streams of data are brought in using the \text{in\_luma}, \text{in\_cr}, \text{in\_cb} odd and even streams.

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

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

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

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

• \text{Right Shift}: \gg
• \text{Left Shift}: \ll
• \text{Masking}: Use a single ampersand, so to extract the first red component: \text{RGB[1] \& 0xF8}
### 2.6.5.3.2.2 Tips and Tricks

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

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

Here are a few recommendations:

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

### 2.6.6 Video Processing Part 3: Memory Management

#### 2.6.6.1 Introduction

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

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

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

- IDK Video Device Drivers User’s Guide
- IDK Programmer’s Guide

#### 2.6.6.2 Memory - The Basics

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

As seen earlier, this necessitates a system of transferring memory contents between the two memory systems. For example, an input color screen is in YCbCr format. This consists of 640 X 480 pixels with 8

---

43This content is available online at [http://cnx.org/content/m11989/1.2/](http://cnx.org/content/m11989/1.2/).
bits per pixel. This results in 300 Kbytes, which cannot be stored in internal memory. This same problem applies for the output buffer.

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

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

### 2.6.6.3 Memory - Setup

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

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

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

#### 2.6.6.3.1 Allocating Memory Space

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

- 1. Declare the type of memory space and it’s name. Use the `#pragma DATA_SECTION` command. There are two parameters:
  - a) the name of the memory spaces
  - b) and the type – internal or external

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

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

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

- We then specify the size of the memory space. We use a variable for the basic unit size (e.g. unsigned char for 1 byte) and a length for the number of basic units needed. Please note, the memory space is not delineated by ‘image’ rows or columns. The system thinks it is one long array of data, it is up to us to process this as separate lines of ‘image’ data.
// specify size as width 640
// height 480
// and 8 bytes per pixel
// which could represent an RGB screen of 640 X 480 with
// 2 bytes per pixel. Unsigned char = 8 bytes
unsigned char ext_mem[640 * 480 * 2];

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

Now have a look at the main.c file and take note of the memory spaces used. The internal memory
is of size 12 * 640. This single memory space is going to be used to store both the input lines from the
camera image and also the results of the color conversion, thus explaining its large size. Basically the internal
memory is partitioned by us for different buffers. The output data buffer needs only 4*640 bytes thus it’s
space starts at
int_mem + (8 * cols); //cols = 640
and ends at 12*cols – which gives us 4*cols of space. Though it is useful to partition internal memory
in such a way, it is recommended not to. It is very easy to mess up the other data too, so simple, so our
solution would have been to create a separate memory space of size 4*cols.

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

2.6.6.3.2 The INPUT and OUTPUT buffers and Main.c Details

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

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

  VDIS_open();
  VCAP_open();

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

  color(VCAP_NTSC, VDIS_640X480X16, numFrames);

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

  We then move on to the color(...) function within main.c

- 3. First we declare some useful pointers which we will use for the various images and their com-
   ponents and so forth. The IMAGE structure holds a pointer to the image array (img_data). In
   addition, it holds integers for the number of image rows (img_rows) and number of image columns
   (img_cols). (Implementation Details in img_proc.h) Declare more of these structures as needed for any
   memory spaces you create yourself. Furthermore, “scratch_pad” structures hold information about the
   location and size of internal and external memories. This is another use of pointers being used to hold
   the locations of the different memory spaces. (Implementation Details in img_proc.h) We also config-
   ury the display and capture formats using
CHAPTER 2. PROJECT LABS

VDIS_config(displayMode);
VCAP_config(captureMode);

- Following this we enter the loop:

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

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

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

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

    out_image.img_data = (unsigned char *) output;

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

2.6.6.4 Memory Streams

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

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

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

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

2.6.6.4.1 Creating and Destroying Streams

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

    dstr_t o_dstr;
Then use the

\texttt{dstr_open(...);} \\

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

Arguments (note: \texttt{out_rows = 480, out_cols = 640}):

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

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

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

\[(2 * \texttt{out_rows} * \texttt{out_cols})\]

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

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

- \texttt{unsigned short i_size}
  
  takes in the size of the internal data buffer to which we are reading from. This specifies the actual number of bytes of internal memory we will be traversing. So this is NOT necessarily the full size of the output buffer (i.e. NOT always \texttt{640 X 480 X 2}) For our example we are reading from the full screen hence we use

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

• unsigned short quantum

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

• unsigned short multiple

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

• unsigned short stride

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

• unsigned short w_size

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

• dstr_t dir

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

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

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

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

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

### 2.6.6.4.2 Memory Tricks and Tips

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

- Know how data flows in your system, this will help you increase efficiency and possibly eliminate complex stream use as well.
- The dsdr_get_2D and dsdr_put_2D are used for multiple line transfers. Use these to your advantage.
- You can use a simple memory ping-pong system to lessen memory use. If you need to use, say 200 X 300 rectangular region and filter it repeatedly. Then keep two memory 200 X 300 memory spaces. Write to the first, filter out to the second. Then filter the second out to the first, and so on until you’re done.

### 2.6.6.4.3 Limitations

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

### 2.6.6.5 IDK Libraries

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

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

```c
#include "sobel_h.h"
```
directive in the wrapper function at the top. Don’t forget to create a header function for your wrapper as well and add it to your project.

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

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

2.6.6.6 The Assignment

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

\[ V:\text{\textbackslash ece320\textbackslash projects\textbackslash colorcool} \]

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

2.6.6.6.1 Tips, Tricks and Treats

- Plan the system beforehand to make efficient use of modular functions and memory.
- For example, you only need just one “output area if size M” function to screen.
- Keep handy pointers to the different memory spaces.
- Use wrapper functions for the filter and copy_to_screen operations.
- Write the modules so that they can be tested independently.
- Be careful with color conversion. For example when copying the red-component of M, you need only 8 bits per pixel.
- Keep the previous lab in mind when deciding when/where to extract the area M.
Chapter 3

General References

3.1 Processor

3.1.1 Two’s Complement and Fractional Arithmetic for 16-bit Processors

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):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001₂</td>
</tr>
<tr>
<td>2</td>
<td>0010₂</td>
</tr>
<tr>
<td>6</td>
<td>0110₂</td>
</tr>
<tr>
<td>8</td>
<td>1000₂</td>
</tr>
</tbody>
</table>

Table 3.1

**NOTE:** 1000₂ 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.

¹This 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

<table>
<thead>
<tr>
<th>fractional</th>
<th>$0.5 \times 0.5 = 0.25$</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>$16384 \times 16384 = 4096 \times 2^{16}: 4096/32768 = 1/8$</td>
</tr>
<tr>
<td>hex</td>
<td>$4000_{16} \times 4000_{16} = 1000_{16} \times 2^{16}$</td>
</tr>
</tbody>
</table>

Table 3.2

<table>
<thead>
<tr>
<th>fractional</th>
<th>$0.125 \times 0.75 = 0.093750$</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>$4096 \times 24576 = 1536 \times 2^{16}: 1536/32768 = 0.046875$</td>
</tr>
<tr>
<td>hex</td>
<td>$1000_{16} \times 6000_{16} = 0600_{16} \times 2^{16}$</td>
</tr>
</tbody>
</table>

Table 3.3

You may wish to use 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.

<table>
<thead>
<tr>
<th>fractional</th>
<th>$-0.5 \times 0.5 = -0.25$</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>$49152 \times 16384 = 12288 \times 2^{16}$; $12288/32678 = 0.375$</td>
</tr>
<tr>
<td>hex</td>
<td>$C000_{16} \times 4000_{16} = 30000000_{16} = 3000_{16} \times 2^{16}$</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>fractional</th>
<th>$-0.5 \times 0.5 = -0.25$</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex</td>
<td>$1C000_{16} \times 4000_{16} = 70000000_{16} = 7000_{16} \times 2^{16}$</td>
</tr>
</tbody>
</table>

**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 TMS320C55x

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 six, **Addressing modes**, in the *CPU Reference Guide* contains extended descriptions of the addressing modes described below.

---

²This content is available online at [http://cnx.org/content/m14262/1.4/].
3.1.2.1 Accumulators: ACw, ACx, ACy, ACz, xsrc, xdst, src, dst

Whenever the abbreviations ACw, ACx, ACy, ACz, xsrc or xdst are used in the assembly language syntax description for an instruction, it means that only the accumulators AC0, AC1, AC2 and AC3 may be used for that particular operand. At times, src and dst may also be referring to the accumulators.

Examples:

\[
\begin{align*}
\text{MOV } &\begin{array}{c} *\text{AR3}, *\text{AR4}, \text{AC0} \\
\end{array} ; \text{AC0}(15-0) = \text{contents of memory location pointed to by AR3} \\
&\begin{array}{c} \text{AC0}(39-16) = \text{contents of memory location pointed to by AR4} \\
\end{array} \\
&\text{sign extended to 24 bits}
\end{align*}
\]

\[
\begin{align*}
\text{MOV } &\text{AC0, AC1} \\
&; \text{AC1} = \text{AC0}
\end{align*}
\]

\[
\begin{align*}
\text{MOV } &\text{AC0, *AR3} \\
&; \text{sets content of memory location pointed to by AR3} = \text{AC0}(15-0)
\end{align*}
\]

\[
\begin{align*}
\text{MOV } &\text{HI(AC0), *AR7+} \\
&; \text{sets (contents of memory location to by AR7)} = \text{AC0}(31-16), \\
&; \text{and then increments AR7 by one}
\end{align*}
\]

3.1.2.2 Memory-mapped Registers: MMR, MMRx, MMRy

Many of the TMS320C55x 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 accumulators AC0 through AC3, auxiliary registers AR0 through AR7. The temporary registers T0 - T3, BK, and various BSA (buffer start address) are also memory mapped. Memory mapped registers are stored from 00 0000h to 00 005Fh.

An mmr prefix can be used for indirect memory operands to assert that the memory access is to a memory-mapped register. This prefix can be used on Xmem, Ymem, indirect Smem, indirect Lmem, and Cmem operands. Refer to 1-19 of the SPRU374 for more information if necessary.

The mmap qualifier can be used with numerous instructions to force an access to a memory-mapped register. It can be used with Smem or Lmem direct memory access to prevent the dma access from being relative to the SP or DP. Instead, it will be relative to the memory-mapped register data page start address, which is 00 0000h.

Examples:

\[
\begin{align*}
\text{MOV } &\begin{array}{c} \text{mmap(AR6), BSA67} \\
\end{array} ; \text{sets BSA67} = \text{value in AR6} \\
&\begin{array}{c} \text{mmap must be used} \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{MOV } &\text{AR1, AR5} \\
&; \text{sets AR5} = \text{AR1}
\end{align*}
\]

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 TMS320C55x instructions allow immediate data of 3, 4, 5, 7, 8, 9, 12, 16, 23 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 #k16.
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:

```assembly
ADD #FFFFH, *AR3 ; The content addressed by AR3 is added to a signed 16-bit value and the result is store back in into the location addressed by AR3
AMOV #7FFFFFh, XAR0 ; The 23-bit value is loaded into XAR0
```

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

```assembly
MOV #coef, AR1 ; sets AR1 = memory address of label coef
```

mean to store the contents of memory location `coef` in `AR1`, or does it mean to store the memory address `coef` itself in `AR1`? The second interpretation is correct.

Many instructions have several versions allowing the use of different addressing modes (see `mov` 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.

### 3.1.2.4 Direct Addressing: Smem and others

In the modes called direct addressing by TI, the memory offset is combined with values in the DPH (the high part of the extended data page register) and DP (data page register) or the SPH (high part of the extended stack pointer) and SP (data stack pointer) to obtain a complete 23-bit data-memory address. The DSP uses SPH/SP or DPH/DP depending on the value of the CPL bit in status register ST1_55.

For our purposes, the CPL bit is set and so we will use SPH/SP for direct addressing. SP is initialized for you in the core file and should not need to be modified. SP-referenced direct addressing is used by the `psh` and `pop` instructions for stack manipulation, as well as by all subroutine calls and returns, which save program addresses on the stack.

Examples:

```assembly
MOV *SP(5), T2 ; T2 = value at location 00FF05h

; Assuming DPH = 3, DP = 0
MOV 00005h, T2 ; T2 = value at location 030005h

ADD *SP(6), AC0 ; AC0 = AC0 + (contents of memory location SPH:SP+6)
```

### 3.1.2.5 Absolute Addressing: *abs16(#k16), *(#k23)

This seems to be TI's term for all the forms of direct addressing which it does not call direct addressing! There are three types of absolute addressing: `k16`, `k23`, and I/O. We will only be using the first two. It is
represented in assembly-instruction syntax-definitions using one of the above abbreviations (*1k) addressing is available when the syntax definition says Smem or Lmem).

### 3.1.2.5.1 k16

k16 absolute addressing uses the operand *abs16(#k16) along with the 7-bit DPH to form a 23-bit address.

**Example:**

```
MOV *abs16(#2002h), T2 ; T2 = value at address 032002h
```

### 3.1.2.5.2 k23

k23 absolute addressing uses the operand *(#k23) as a 23-bit address.

**Example:**

```
MOV *(#032002h), T2 ; T2 = value at location 032002h
MOV AR1, *(#hold) ; sets (storage location at hold) = AR1
```

### 3.1.2.6 Indirect Addressing: Smem, Lmem, Xmem, Ymem, Cmem

**Indirect addressing** on the TMS320C55x uses the auxiliary registers AR0 through AR7 and the CDP. They can be used in place of Smem/Lmem or Xmem/Ymem.

#### 3.1.2.6.1 AR Indirect: Smem/Lmem

In Smem/Lmem indirect addressing, only one indirect address is used in the instruction and a number of variations is possible (see the table on page 6-39 of the CPU Reference guide). An asterisk is always used, which usually 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 T0, T1, or 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 Dual AR Indirect: 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, T0, or T1 with circular addressing.

```
ADD Xmem, Ymem, ACx
```

```
ADD *AR1+, *AR2+, AC0 ; Add values stored in memory locations referenced by AR1 and AR2 and store result in AC0.
```
3.1.2.6.3 CDP Indirect: Cmem

CDP indirect addressing uses the coefficient data pointer (CDP) to point to data. For accessing data memory/registers, the 16-bit CDP is combined with the 7-bit CDPH to generate a 23-bit address. When concatenated, they are called the XCDP. CDP indirect addressing can also be used to address a specific bit in the accumulators, auxiliary registers, and the temporary registers. Pre- and post-increment and decrement as well as an offset can be used with CDP. CDP can be used as an operand in place of Smem, Lmem, and Cmem.

```plaintext
MOV dbl(Lmem), Cmem
MOV dbl(*AR7), *CDP+ ; Values at XAR7 and XAR7 + 1 are read and stored at
          ; XCDP and XCDP +/- 1 depending on if XCDP was even or odd.
          ; CDP is incremented by 2 at the end.
```

3.1.2.6.4 Coefficient Indirect

Coefficient indirect addressing uses the same address generation process as CDP indirect addressing for data-space accesses. It is useful for instructions that need three memory operands per cycle. It can be used for finite impulse response filters, multiply [accumulate/subtract], and dual multiply [accumulate/subtract].

```plaintext
MPY Xmem, Cmem, ACx
  :: MPY Ymem, Cmem, ACy ; Cmem must be in a different memory bank from Xmem/Ymem
  ; for this to work in a single cycle

MPY *AR1+, *CDP+, AC0
  :: MPY *AR2+, *CDP+, AC1 ; The value at address XAR1 is multiplied by value at
  ; address XCDP and stored in AC0. At the same time,
  ; value at XAR2 is multiplied by value at address XCDP
  ; and stored in AC1. Then CDP is incremented.
```

3.1.2.6.5 Other Examples

```plaintext
AMAR  *AR3+ ; increments AR3 by 1
```

**NOTE:** The `amar` (modify auxiliary 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 Circular Addressing

Circular addressing is useful when implementing circular buffers. Circular addressing needs to be enabled for the specific register that is being used to point to memory. This is done by setting the corresponding ARnLC register using BSET ARnLC. When circular addressing is not needed, BCLR ARnLC on the corresponding ARn will disable circular addressing. The circular addressing length will depend on the BK03, BK47, and BKC register values. If you are using AR0 through AR3 for the addressing, then BK03 will be used. BKC is the buffer length for CDP. One thing to watch out for is the buffer start address registers (BSA01, BSA23, BSA45, BSA67, BSAC) are added to the auxiliary register or CDP register value whenever circular addressing is used. Be sure to re-initialize the BSA register when implementing multiple filters.

```
BSET AR1LC ; sets circular addressing for AR1
BCLR AR2LC ; normal addressing for AR2
MOV #13, BK03 ; set buffer length for AR0 through AR3

MACM *AR1+, *AR2+, AC0 ; AC0 = AC0 + (value at memory location AR1 + BSA01) x (value at memory location AR2);
; AR1 is incremented with circular addressing, length 13.
; AR2 is simply incremented by 1
```

3.1.2.8 Bit-Reversed Addressing

Bit-reversed addressing is often needed with the fast-fourier transform (FFT). This helps to set up a pointer for the next iteration. Enable bit-reversing on an operand by adding a B after the increment value. When a bit-reverse operand is used, the auxiliary register can not be used for circular addressing. If circular address is enabled, the corresponding buffer start address value (BSAxx) will be added to ARn, but ARn is not modified to stay within the circular buffer.

```
MOV Smem, dst

MOV *(AR4 + T0B), T2 ; T2 = value at memory location XAR4
; AR4 is incremented with T0, using reverse carry propagation.
```

3.2 RTDX (Real Time Data Transfer)

3.2.1 Using RTDX with a MATLAB GUI

3.2.1.1 Introduction

The USB port on the DSP box can also be used to transmit data between the DSP and the PC during real-time operation. Texas Instruments came up with Real Time Data Exchange (RTDX) to allow users real-time bidirectional exchange of data between the target and host. This allows the simulation of data input and output which can be used as feedback from the DSP for a variety of applications. Both input and output data are buffered until read, allowing transmission of larger amounts of data.

---

This content is available online at <http://cnx.org/content/m14388/1.5/>. 
3.2.1.2 Setting Up Input/Output Channels

The RTDX works by setting up input and output channels to the DSP. RTDX functionality is supplied through the include file `rtdx.h` so be sure it is included in your `main.c` and any other files that use rtdx functions.

Depending on whether there will be input and/or output from the computer in your project, add input and output channels in the `main.c` file using the commands. These are declared like global variables near the top of the code after the include files. They do NOT go in the `main()` function or any other function.

```c
RTDX_CreateInputChannel(ichan);
RTDX_CreateOutputChannel(ochan);
```

By default, these channels are disabled on the DSP. You may enable them in the main loop so that they will be enabled for the duration of the program. This can be accomplished with the following instructions:

```c
RTDX_enableInput(&ichan);
RTDX_enableOutput(&ochan);
```

In other C files that utilize the declared input and output channels, you will need to declare the input and output channels again with an extern so that your files know what the variables are.

```c
extern RTDX_input_channel ichan;
extern RTDX_output_channel ochan;
```

Lastly, RTDX MUST be manually enabled on the DSP boards. Go to Tools->RTDX->Configuration Control and select 'Enable RTDX' in the window that opens. Another helpful window for debugging is the 'Channel Viewer Control' accessed through Tools->RTDX->Configuration Control. This window displays the number of completed and outstanding transfers between computer and board.

3.2.1.3 Using the DSP to Access the USB Port

The data buffer can be written to in C, but requires the block method of input/output used in Lab 4 and Lab 5. The sample-by-sample method used in Labs 1 through 3 will not work with RTDX. RTDX functionality is supplied through the include file `rtdx.h` so be sure it is included in your `main.c` file and any other files that use rtdx functions.

3.2.1.3.1 Using C to Send/Receive

There are several functions for transmitting and receiving data within the C environment:

- **RTDX_readNB()** takes three arguments: the first is a pointer to the input channel, the second is a pointer to the variable in which to store read data, and the third is the number of bytes to read. This is a non-blocking read, meaning if no data is available to be read, it will merely return. However, there will then be an outstanding read and the variable will be updated when some data is finally read. It returns 'RTDX_OK' on success, '0' on failure (the target buffer is full), and 'RTDX_READ_ERROR' when the channel is currently busy reading.

- **RTDX_read()** also takes three inputs like `RTDX_readNB()` and but on successful read, it returns the number of bytes of data actually in the buffer. The difference from `RTDX_readNB()` is it’s a blocking read, meaning `RTDX_read()` won’t return until something is read. If the channel is busy or not enabled, 'RTDX_READ_ERROR’ will be returned.

- **RTDX_write()** takes three arguments: the first is the pointer to the output channel, the second is a pointer to the buffer containing the data to write, and the third is the size of the buffer in bytes. It returns an integer, non-zero on success and '0' on failure.
• **RTDX\_sizeofInput()** takes a pointer to an input channel and returns the number of bytes of data actually read from the buffer. It is used in conjunction with RTDX\_readNB() after a read operation is completed.

• **RTDX\_channelBusy()** takes a pointer to an input channel and returns an int indicating the status of the channel. A return of '0' means the channel is not busy while non-zero means the channel is busy. This is usually used in conjunction with RTDX\_readNB() to check if another read request needs to be issued.

More information about the RTDX module and the commands that can be used with it are in the TMS320 DSP/BIOS User’s Guide (spru423)\(^6\) and the TMS320C5000 DSP/BIOS API Reference Guide (spru404)\(^7\).

**Example 3.3**

The following example shows a simple C program that will echo received data back through the output channel. This assumes that the main.c file has declared and enabled the input and output channels. The project file and all the necessary files are available from v:/ece420/55x/block_rtdx MATLAB GUI files that are made to interface with this project are rtdx_text.m\(^8\) and rtdx_echotext.m\(^9\).

```c
#include "dsk5510_dual3006cfg.h"
#include "dsk5510.h"
#include "swi_process.h"
#include "dsplib.h"
#include "rtdx.h" // Include file for rtdx functionality

extern RTDX_input_channel ichan; // ichan has been declared in main.c
extern RTDX_output_channel ochan; // ochan has been declared in main.c

int recvd;
int sentNew = 0;

// all data processing should be done in SWI\_ProcessBuffer

void SWI\_ProcessBuffer()
{
    static unsigned int mbox_value = 0;
    short *psrc, *pdest;

    mbox_value |= SWI\_getmbox();

    // buffers are only processed when both transmit and receive are ready
    if((mbox_value & DMA\_RECEIVE\_DONE) && (mbox_value & DMA\_TRANSMIT\_DONE))
    {
        mbox_value = 0;

        // get buffer pointers
        psrc = receive_buffer[receive_buffer_to_process_index];
        pdest = transmit_buffer[transmit_buffer_to_fill_index];

        if (!RTDX\_channelBusy(&ichan)) // read only when not busy
            RTDX\_readNB(&ichan, &recvd, sizeof(recvd));
```

\(^6\)http://focus.ti.com/lit/ug/spru423f/spru423f.pdf

\(^7\)http://focus.ti.com/lit/ug/spru404g/spru404g.pdf

\(^8\)http://cnx.org/content/m14388/latest/rtdx_text.m

\(^9\)http://cnx.org/content/m14388/latest/rtdx_echotext.m


```c
    sentNew = 1;
}

if (sentNew == 1) { // echo back when data has been received
    RTDX_write(&ochan, &recvd, sizeof(recvd));
    sentNew = 0;
}

receive_buffer_processed = 1; // flag receive buffer as processed
transmit_buffer_filled = 1; // flag output buffer as full
```

### 3.2.1.4 Using MATLAB to Access the DSP Board (PC)

MATLAB can be used to access the data coming from the DSP board. A simple typing/echo GUI for the block_rtdx project has been provided. An interface can also be programmed in Visual Basic. The setup and transfer/receive commands for Matlab will be described below.

#### 3.2.1.4.1 Sending Data

Before accessing the DSP board, it must be initialized through MATLAB. This is done with this code:

```matlab
    h = actxserver('RTDX');
```

which sets the port with all necessary parameters. The port is still not open for writing. To open the port, specify the name of the channel you would like to open in the command:

```matlab
    invoke(h,'Open','ichan','W');
```

To write to the buffer, you invoke h with a 'Write' parameter and the data to send.

```matlab
    invoke(h,'Write',int16(v));
```

In this case, v was a char and we wanted to send the ASCII value. (There is a limitation to the ASCII converter in Matlab: it does not take all possible key presses.) Multiple channels can be opened in this manner. When there are multiple channels open, the write will be to the most recently opened channel.

Before finishing a function, or before executing a read, the port should be closed. The port is closed with the command:

```matlab
    invoke(h,'Close');
```

#### 3.2.1.4.2 Receiving Data

To read data from the DSP, a read channel must be set up. This is done using the `invoke open command` with an 'R' parameter:

```matlab
    invoke(h,'Open','ochan','R');
```

Reading data from the DSP board is a little more complicated. It seems that the DSP board buffers all the data. To get the latest piece of data, you must first 'Seek' to the current message.
[status, nummsgs] = invoke(h, 'GetNumMsgs');
status = invoke(h, 'Seek', nummsgs);

Once at the correct message, the actual reading can be done.

[status, values] = invoke(h, 'ReadI2');

As with writing, the port should be closed after reading.

### 3.2.1.5 Using MATLAB GUI Features

MATLAB has some nice Graphical User Interface (GUI) features which can be used to control the flow of data to and from the RTDX port. The basic implementation consists of a blank window (figure) which can have different interface elements placed on it. These elements can be sliders, buttons, text boxes, etc...

When an element is accessed (for instance, a slider is moved, or a button is pushed), MATLAB will execute a "callback routine" which is a MATLAB function defined by the user. Designing these interfaces is simple.

#### 3.2.1.5.1 Creating a User Interface with Sliders

Download These Files

- rtdx_sliders.m\(^\text{10}\) - User Interface
- rtdx_wrt_sliders.m\(^\text{11}\) - Callback File

**Example 3.4**

```matlab
1  % rtdx_sliders - initializes RTDX port and sets up three sliders
2 3 h = actxserver('RTDX');
4 5 % open a blank figure for the slider
6 Fig = figure(1);
7 % open sliders
8 % first slider
9 sld1 = uicontrol(Fig,'units','normal','pos', [.2, .7, .5, .05], ...
10     'style','slider','value',4,'max',254,'min',0,'callback','rtdx_wrt_sliders');
11 12 % second slider
13 sld2 = uicontrol(Fig,'units','normal','pos', [.2, .5, .5, .05], ...
14     'style','slider','value',4,'max',254,'min',0,'callback','rtdx_wrt_sliders');
15 16 % third slider
17 sld3 = uicontrol(Fig,'units','normal','pos', [.2, .3, .5, .05], ...
18     'style','slider','value',4,'max',254,'min',0,'callback','rtdx_wrt_sliders');
```

Lines 9 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 6. 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.

\(^\text{10}\)http://cnx.org/content/m14388/latest/rtdx_sliders.m
\(^\text{11}\)http://cnx.org/content/m14388/latest/rtdx_wrt_sliders.m
- **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. rtdx_wrt_sliders is a Matlab file that writes the values of the controls to the RTDX port.

Every time a slider is moved, the rtdx_wrt_sliders.m file is called:

**Example 3.5**

```matlab
1  % rtdx_wrt_sliders : writes values of sliders out to rtdx
2 3  % open rtdx port for data transfer
4  status = invoke(h,'Open','ichan','W');
5
6  % send value from sld1
7  v1 = round(get(sld1,'value'));
8  status = invoke(h,'Write',int16(v1));
9
10  % send value from sld2
11  v2 = round(get(sld2,'value'));
12  status = invoke(h,'Write',int16(v2));
13
14  % send value from sld3
15  v3 = round(get(sld3,'value'));
16  status = invoke(h,'Write',int16(v3));
17
18  % send reset pulse
19  status = invoke(h,'Write',int16(2989));
20
21  % close rtdx port
22  status = invoke(h,'Close');
```

Line 7 retrieves the value from the slider using the get function to retrieve the value property. The value is then rounded off to create an integer, and the integer is sent as an 16-bit quantity to the DSP in Line 8. The other two sliders are sent in the same way. Line 19 sends 2989 to the DSP, which can be used to indicate that the three previously-transmitted values represent a complete set of data points. (You can use whatever value you want.) This can be used to prevent the DSP and Matlab from losing synchronization if a transmitted character is not received by the DSP and provides some error detection.

**Note**: Line 22 closes the RTDX port. Make sure you close the port after sending a data block to the DSP.

### 3.2.1.5.2 Advanced features

The slider example shows some basic features of the gui tools. The handle for the RTDX server is generated into the workspace so that it can be used for writing. But other elements, such as text boxes cannot be dealt with as easily. The Parameters from these can be accessed through their returned handles. Some examples:

**Example 3.6**
%GUI.m

%****Sample GUI, Text and a Button***

%open a blank figure
Fig = figure(1);
set(Fig,'Name','Test GUI');

%Space to enter text
ed2 = uicontrol(Fig,'backgroundcolor','white','units','Normalized','pos',[.1,.6,.4,.05],
                'string','Default Text','style','edit');

%Button
but1 = uicontrol(Fig,'foregroundcolor','blue','units','Normalized','pos',[.1,.4,.5,.1],
                'string','Press Me!','style','pushbutton','callback','SampleGUI');

A Text box is created with default text in it that says: "Default Text". A button is also created,
which when pressed, will execute the callback function SampleGUI.m

%SampleGUI.m

%Get Text
testText = get(ed2,'string')

Now testText holds whatever string was entered into the text box. The function get() is used to
retrieve the data from the 'string' parameter in the ed2 handle. MATLAB help uicontrol gives
the full list of options for interface elements.

3.3 Miscellaneous

3.3.1 Cycle Counts

3.3.1.1 Cycle Counts

The number of cycles a block of code takes to run may be important. For example, code that executes during
an interrupt should be kept as short as possible to allow other interrupts execution time. When a complex
system is implemented on the DSP, hardware resources are scarce and code that takes too long to run will
prevent the system from running in real-time.

Two methods of obtaining the cycle count for a block of code will be outlined below. Breakpoints are a
fast method of getting a ballpark cycle count for a particular run of the code. However, different iterations
of the code may take a different number of cycles to execute. This generally happens when there are branch
and conditional instructions. Profiling allows the user to execute the code multiple times and obtain the
average, maximum, and minimum number of cycles.

3.3.1.2 Breakpoints

Setting two breakpoints will allow you to measure the clock cycles between the first and second breakpoint
for a particular run. Code Composer will also return a cycle count for running between the second breakpoint

\footnote{This content is available online at \url{http://cnx.org/content/m14415/1.2/}.}
and the first, but this count will include all the cycles from I/O handling. Multiple breakpoints can also be set to obtain counts for multiple sections of code, but there is a limit on the number of breakpoints that can be set.

The instructions to set breakpoints and obtain a cycle count for a piece of code:

1. Compile the code.
2. Select Clock->Enable and Clock->View under the Profile menu.
3. Set a software breakpoint by right-clicking on the line you would like to set it at and choosing "Toggle Software Breakpoint".
4. Load the program.
5. Run the program. It should halt at the first breakpoint. There will be a cycle count at the bottom right corner next to the clock symbol. Double click on the click to clear the cycle count.
6. Run the program again. It should halt at the second breakpoint. The cycle count from the code between the two breakpoints will be displayed.

Sometimes there may be errors when loading the program like "Can’t Set Breakpoint" or "Can’t Remove Breakpoint". If this happens, try disabling the breakpoints, reloading, and enabling them again. If the error persists, close Code Composer, disconnect and reconnect the power cable, and start the whole process again.

3.3.1.3 Profiling

Profiling allows you to obtain an average, maximum, and minimum cycle count for blocks of code. Multiple functions, loops, and ranges can all be profiled at the same time. Code Composer will provide you with many readings. The figure used in our DSP lab is the 'CPU Cycles: Incl. Max'.

The instructions to set profile points and obtain a cycle count for a piece of code:

1. View the profiling windows by selecting Profile->Setup and Profile->Viewer.
2. Load the program onto the DSP.
3. Click on the Stopwatch symbol in the Profiling Setup window on the right to enable profiling.
4. Select the 'Ranges' tab in the Profiling Setup window.
5. Highlight the range of code you would like to obtain the cycle count for and drag it to the 'Ranges' menu. (Another way to do the same is to highlight those lines, right click, and select Profile -> Range.)
6. Run the code for a while for Code Composer to collect data.
7. The default stats shown do not include what we are looking for. In the profiler viewer window, right click on the address range you would like stats for and select 'Columns and Rows Setting'. Different counts can be shown, but 'CPU Cycles: Incl. Max.' is probably the most important.

3.3.2 Using the I/O Ports of the DSK 5510

3.3.2.1 Introduction

The input/output ports of the DSK 5510 can be used when only two inputs and two outputs are needed. Unfortunately, it is currently impossible to utilize the four I/O ports provided by the daughtercard and the two I/O ports on the DSK at the same time. The advantages of the on-board I/O ports are higher signal-to-noise ratio and the ability to change the sampling rate.

The files necessary to utilize the original I/O ports of the DSP board can be found in the V drive under V:\ece420\55x\ dsp_board. Much of the functionality is provided in the core.c file. The code is similar to that used in Lab 4 (Section 1.5.3) of the Digital Signal Processing Laboratory (ECE 420 55x). Most changes will probably be made in the dspboard_app.c file. The processBuffer() function is called once

---

13 H"esshaft is available online at <http://cnx.org/content/m14478/1.2/>.
14 Digital Signal Processing Laboratory (ECE 420 55x) <http://cnx.org/content/col10397/latest/>
every 1024 samples so add changes there. Input will be taken from the 'Line In' port. Both output ports are on by default and are designed to drive different impedances.

As given, the code calls function echoInput() which echoes back input channels 1 and 2 on their respective output channels. The echoInput() function is defined in asmfunctions.asm. The input parameters of echoInput() are passed through registers T0, XAR0, and XAR1. To learn more about the passing convention when calling assembly functions from C, please view Section 6.4.1 of the TMS320C55x Optimizing C/C++ Compiler User’s Guide. The code also stores the input into a block of memory located at memory address 0x28000h. This is done to show the DSP can access addresses greater than 16-bit in length.

3.3.2.2 Changing I/O Parameters

The codec is configured with the following lines of code in core.c:

```c
/* Codec configuration settings */
DSK5510_AIC23_Config config = {
    0x0017, // 0 DSK5510_AIC23_LEFTINVOL Left line input channel volume
    0x0017, // 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume
    0x00d8, // 2 DSK5510_AIC23_LEFTHPVOL Left channel headphone volume
    0x00d8, // 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume
    0x0010, // 4 DSK5510_AIC23_ANAPATH Analog audio path control
    0x0000, // 5 DSK5510_AIC23_DIGPATH Digital audio path control
    0x0000, // 6 DSK5510_AIC23_POWERDOWN Power down control
    0x0043, // 7 DSK5510_AIC23_DIGIF Digital audio interface format
    0x0081, // 8 DSK5510_AIC23_SAMPLERATE Sample rate control
    0x0001 // 9 DSK5510_AIC23_DIGACT Digital interface activation
};
```

The default sampling rate is 48kHz but this can be changed quickly by changing the value of one of the registers in core.c. For example, a 44.1 kHz sampling rate can be obtained by changing register 8 from 0x0081 to 0x00a3. The rate is actually 44.117 kHz as detailed in the TLV320AIC23 Data Manual. A couple different sampling rates are available (e.g. 8kHz, 32kHz) and if necessary, the input and output rates can be different.

The default input is the 'Line In' port. Microphones that have a small output signal need to be connected to a port that will boost its signal. The 'Mic In' port will be perfect for this task. To change input to the 'Mic In' port, change register 4 from 0x0010 to 0x0014. The gain can also be adjusted, but keep in mind that there will be clipping if the input signal is too high. Unfortunately, the microphones we have in lab need to be amplified before being connected to the DSP. Please refer to the TLV320AIC23 Data Manual for further instructions.

3.3.3 Files for TI DSK 5510

Introduction

The files in this module are designed for use with the DSK 5510. Some of the files are only useful with the add-on dual3006 daughterboard built by Educational DSP. The code may also be useful for other DSKs. Please make sure to use the files that match the hardware used. One of the big differences between the files is sample-by-sample processing and block-processing. In sample-by-sample processing, the DSP receives one input sample per channel and expects one output sample per channel. For block-processing, the DSP will have a buffer (length N) of inputs samples per channel, and will expect N output samples per channel.
Block-processing is capable of RTDX, but has a longer delay. FFT applications will require the use of block-processing. Keep this in mind when choosing the files to use.

**DSK5510 with DUAL3006 daughterboard**

These files assume input and output are through the four input and four output ports provided by the DUAL3006 daughterboard. This category contains the files used in lab 0 through lab 4 of ECE420 at the University of Illinois as of Spring 2008.

- Lab 0\(^{19}\) - an FIR filter. Input and output processed on a sample-by-sample basis. RTDX not supported
- Lab 4\(^{20}\) - framework code for block processing. RTDX not supported
- BPSK Transmitter\(^{21}\) - poor implementation of a BPSK transmitter that utilizes block processing
- Block-processing\(^{22}\) - code for block processing. RTDX supported

**DSK5510**

These files assume input and output are through the two input and two outputs ports provided by the DSK5510. These files will probably be most useful to those not at the course at UIUC, or for those working on a final project that does not require four input/output channels.

- Sample-by-sample\(^{23}\) - an FIR filter with sample-by-sample processing. RTDX not supported
- Block-processing\(^{24}\) - code for block processing. RTDX supported

19 \[http://cnx.org/content/m16926/latest/filter.zip\]
20 \[http://cnx.org/content/m16926/latest/lab4.zip\]
21 \[http://cnx.org/content/m16926/latest/bpsk_tx.zip\]
22 \[http://cnx.org/content/m16926/latest/block_rtdx.zip\]
23 \[http://cnx.org/content/m16926/latest/dspboard_samplefilter.zip\]
24 \[http://cnx.org/content/m16926/latest/dspboard.zip\]
Chapter 4

Other Labs

4.1 UIUC DSP Lab Fall 2002 Archive

4.1.1 About this Module
These are the modules which where contained in the Fall 2002 semester of University of Illinois Urbana-Champaign’s DSP LAB course.

4.1.2 The Course

Lab 0 - • Hardware Introduction

Lab 1 - • PreLab

Lab 2 - • Theory (Section 1.3.1)
       • PreLab (Section 1.3.2)
       • PreLab (Section 1.3.3)
       • Lab

Lab 3 - • Theory (Section 1.4.1)
       • PreLab
       • PreLab (Section 1.4.3)
       • Lab

Lab 4 - • Theory
       • PreLab
       • Lab

Lab 5 - • Theory
       • PreLab
       • Lab

1This content is available online at <http://cnx.org/content/m12455/1.1/>.
2"DSP Development Environment: Introductory Exercise for TI TMS320C54x" <http://cnx.org/content/m10017/latest/>
3"FIR Filtering: Basic Assembly Exercise for TI TMS320C54x" <http://cnx.org/content/m10022/latest/>
4"Halfband vs Spectral Factors" <http://cnx.org/content/m11023/latest/>
5"Representing Proteins in Silico and Protein Forward Kinematics" <http://cnx.org/content/m11621/latest/>
6"IIR Filtering: Filter-Design Exercise in MATLAB" <http://cnx.org/content/m10623/latest/>
7"IIR Filtering: Exercise on TI TMS320C54x" <http://cnx.org/content/m10624/latest/>
8"Spectrum Analyzer: Introduction to Fast Fourier Transform" <http://cnx.org/content/m10027/latest/>
9"Spectrum Analyzer: MATLAB Exercise" <http://cnx.org/content/m10625/latest/>
10"Spectrum Analyzer: FFT Exercise on TI TMS320C54x" <http://cnx.org/content/m10626/latest/>
11"Digital Transmitter: Introduction to Quadrature Phase-Shift Keying" <http://cnx.org/content/m10042/latest/>
12"Digital Transmitter: MATLAB Exercise for Quadrature Phase-Shift Keying" <http://cnx.org/content/m10627/latest/>
13"Digital Transmitter: Optimization Exercise with QPSK on TI TMS320C54x" <http://cnx.org/content/m10628/latest/>
CHAPTER 4. OTHER LABS

- Lab (2)
- Testing

Alternate Lab 5 -
- PreLab
- Lab
- Theory

4.2 UIUC DSP Lab Spring 2003 Archive

4.2.1 About this Module

These are the modules which were contained in the Spring 2004 semester of University of Illinois Urbana-Champaign's DSP LAB course.

4.2.2 The Course

Lab 0 -
- Hardware Introduction

Lab 1 -
- PreLab
- Lab

Lab 2 -
- Theory (Section 1.4.1)
- PreLab
- PreLab (Section 1.4.3)
- Lab

Lab 3 -
- Theory
- PreLab (Section 1.3.2)
- PreLab
- Lab

Lab 4 -
- Theory
- PreLab
- Lab

Lab 5 -
- Introduction
- Filter Specification

---

14.“Digital Transmitter: Optimization Exercise with QPSK on TI TMS320C54x (ECE 320 specific)”
15.”Vector Signal Analyzer: Testing a QPSK Transmitter on Hewlett Packard 8940A”
16.http://cnx.rice.edu/content/m11849/1.1
17.”Vector Signal Analyzer: Testing a QPSK Transmitter on Hewlett Packard 8940A”
18.http://cnx.rice.edu/content/m11848/1.1
19.This content is available online at <http://cnx.org/content/m12456/1.1/>.
20.”DSP Development Environment: Introductory Exercise for TI TMS320C54x (ECE 420 Specific)”
21.”FIR Filtering: Basic Assembly Exercise for TI TMS320C54x”
22.”FIR Filtering: Exercise for TI TMS320C54x (ECE 320 specific)”
23.”IIR Filtering: Filter-Design Exercise in MATLAB”
24.”IIR Filtering: Exercise on TI TMS320C54x (ECE 320 specific)”
25.”Multirate Filtering: Introduction (ECE 320 specific)”
26.”Multirate Filtering: Filter-Design Exercise in MATLAB (ECE 320 specific)”
27.”Multirate Filtering: Implementation on TI TMS320C54x (ECE 320 specific)”
28.”Spectrum Analyzer: Introduction to Fast Fourier Transform (ECE 320 specific)”
29.”Spectrum Analyzer: MATLAB Exercise”
30.”Spectrum Analyzer: Processor Exercise Using C Language”
31.”Low-Pass Filter Implementation: Introduction”
32.”Low-Pass Filter Implementation: Filter Specification”
Alternate Lab 5 -

- Prelab
- Lab

4.3 UIUC DSP Lab Spring 2004 Archive

4.3.1 About this Module

These are the modules which where contained in the Spring 2004 semester of University of Illinois Urbana-Champaign's DSP LAB course.

4.3.2 The Course

Lab 0 -
- Hardware Introduction

Lab 1 -
- Prelab
- Lab

Lab 2 -
- Theory (Section 1.3.1)
- Prelab (Section 1.3.2)
- Prelab (Section 1.3.3)
- Lab

Lab 3 -
- Theory (Section 1.4.1)
- Prelab
- Prelab (Section 1.4.3)
- Lab

Lab 4 -
- Theory
- Prelab
- Lab

Lab 5 -
- Theory
- Prelab
- Lab

33"Low-Pass Filter Implementation: Prelab" <http://cnx.org/content/m11057/latest/>
34"Low-Pass Filter Implementation: Grading" <http://cnx.org/content/m11058/latest/>
35"Digital Transmitter: Introduction to Frequency Shift Keying" <http://cnx.org/content/m11849/latest/>
36"Digital Transmitter: Frequency Shift Keying Prelab Exercise" <http://cnx.org/content/m10661/latest/>
37"Digital Transmitter: Processor Optimization Exercise for Frequency Shift Keying" <http://cnx.org/content/m10662/latest/>
38This content is available online at <http://cnx.org/content/m12457/1.1/>.
39"DSP Development Environment: Introductory Exercise for TI TMS320C54x (ECE 420 Specific)"
<http://cnx.org/content/m11019/latest/>
40"FIR Filtering: Basic Assembly Exercise for TI TMS320C54x" <http://cnx.org/content/m10022/latest/>
41"FIR Filtering: Exercise for TI TMS320C54x [ECE 320 specific]" <http://cnx.org/content/m11020/latest/>
42"Multirate Filtering: Implementation on TI TMS320C54x" <http://cnx.org/content/m11810/latest/>
43"IIR Filtering: Filter-Design Exercise in MATLAB" <http://cnx.org/content/m10023/latest/>
44"IIR Filtering: Exercise on TI TMS320C54x [ECE 320 specific]" <http://cnx.org/content/m11021/latest/>
45"Spectrum Analyzer: Introduction to Fast Fourier Transform (ECE 320 specific)"
<http://cnx.org/content/m11828/latest/>
46"Spectrum Analyzer: MATLAB Exercise" <http://cnx.org/content/m10625/latest/>
47"Spectrum Analyzer: Processor Exercise Using C Language with C Introduction"
<http://cnx.org/content/m11827/latest/>
48"Digital Transmitter: Frequency Shift Keying Prelab Exercise" <http://cnx.org/content/m10661/latest/>
49"Digital Transmitter: Introduction to Frequency Shift Keying" <http://cnx.org/content/m11849/latest/>
50"Digital Transmitter: Processor Optimization Exercise for Frequency Shift Keying" <http://cnx.org/content/m11848/latest/>
CHAPTER 4. OTHER LABS

4.4 UIUC DSP Lab Fall 2004 Archive

4.4.1 About this Module

These are the modules which were contained in the Fall 2004 semester of University of Illinois Urbana-Champaign’s DSP LAB course.

4.4.2 The Course

Lab 0 -  
  - Hardware Introduction

Lab 1 -  
  - PreLab

Lab 2 -  
  - Theory (Section 1.3.1)
  - PreLab (Section 1.3.2)
  - PreLab (Section 1.3.3)
  - Lab

Lab 3 -  
  - Theory (Section 1.4.1)
  - PreLab
  - PreLab (Section 1.4.3)
  - Lab

Lab 4 -  
  - Intro
  - PreLab (Section 1.5.2)
  - Lab

Lab 5 -  
  - Optimization Theory (Section 1.6.1)
  - Lab

51This content is available online at <http://cnx.org/content/m12458/1.1/>.
52"DSP Development Environment: Introductory Exercise for TI TMS320C54x (ECE 420 Specific)" <http://cnx.org/content/m11019/latest/>.
53"FIR Filtering: Basic Assembly Exercise for TI TMS320C54x" <http://cnx.org/content/m10022/latest/>.
54"FIR Filtering: Exercise for TI TMS320C54x (ECE 320 specific)" <http://cnx.org/content/m1030/latest/>.
55"Multirate Filtering: Implementation on TI TMS320C54x" <http://cnx.org/content/m11810/latest/>.
56"IIR Filtering: Filter-Design Exercise in MATLAB" <http://cnx.org/content/m10623/latest/>.
57"IIR Filtering: Exercise on TI TMS320C54x (ECE 320 specific)" <http://cnx.org/content/m11021/latest/>.
58"Spectrum Analyzer: Introduction to Fast Fourier Transform and Power Spectra (ECE 420 specific)" <http://cnx.org/content/m12369/latest/>.
60"Spectrum Analyzer: Optimization Exercise" <http://cnx.org/content/m12392/latest/>.
Bibliography


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

k3, § 3.1.2(119)
k5, § 3.1.2(119)
k9, § 3.1.2(119)
lk, § 3.1.2(119)

5 55x, § 3.2.1(124)

A absolute addressing, § 3.1.2(119)
AC coupled, 4
accumulators, § 3.1.2(119)
adaptive filtering, § 2.4.1(84)
addressing modes, § 3.1.2(119)
algorith optimization, § 1.6.1(55)
aliasing, § 1.3.2(18)
antialiasing, § 1.3.3(19)
antialiasing filter, § 1.1.1(1), 4
anti-imaging, § 1.3.3(19)
anti-imaging filter, § 1.1.1(1), 4
Archive, § 4.1(135), § 4.2(136)
Archive course, § 4.3(137)
assembly, § 1.2.1(8), § 1.2.2(11), § 1.6.2(57), § 4.4(138)
assembly language, § 1.6.1(55)
audio effects, § 2.2.2(77)
autocorrelation, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46), § 1.6.1(55), § 1.6.2(57), § 2.5.1(86), § 2.5.2(89), § 2.5.3(90)
autocovariance, § 2.5.1(86), § 2.5.3(90)

B banz, § 1.3.4(19)
Bayer, § 2.6.1(91)
bi-quad, § 1.4.1(20), 20
bit-reversed, § 1.5.2(29)
block processing, § 1.5.3(33), § 1.5.4(46)
block repeat counter, § 1.4.1(20), § 1.4.4(23), 24
boxcar, § 1.5.2(29)
BPSK, § 1.6.3(58), § 2.1.1(61)
butter, § 1.4.1(20), 22, § 1.4.4(23)

C C, § 1.6.2(57)
C language, § 1.5.3(33), § 1.5.4(46), § 1.6.1(55)
camera, § 2.6.1(91)
canwire, § 2.6.1(91)
carrier recovery, § 2.1.1(61)
Chamberlin, § 2.3.1(78)
Chip Support Library, 93
Clock, § 3.3.1(130)
Code Composer, § 3.3.2(131)
Code Composer Studio, § 1.1.1(1)
Code Composer Studio 3.1, § 3.3.1(130)
Code Composer, DSP, 55x, TI, DSK 5510, § 3.3.3(132)
code optimization, § 1.6.1(55)
coefficient quantization, § 1.4.3(22)
coherent demodulation, § 2.1.1(61)
compiler, § 1.6.1(55)
conv, § 1.4.1(20), § 1.4.2(21), 23, § 1.4.4(23)
correlation, § 2.5.1(86)
cross-correlation, § 2.5.1(86)
CSL, 93
Cycle counts, § 3.3.1(130)

D data pointer, § 3.1.2(119)
decimation, § 1.3.1(17), § 1.3.4(19)
decision statistic, 73
decoder, § 2.3.2(81)
delay, § 2.2.1(76), § 2.2.2(77)
delay-locked loop, § 2.1.2(65), 70
DFT, 26, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)
difference equation, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
difference equations, 21
digital communications, § 2.1.1(61)
digital signal processing, § 1.5.3(33), § 1.5.4(46), § 1.6.1(55), § 1.6.2(57)
direct addressing, § 3.1.2(119), 119
direct form II, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
Discrete Fourier Transform, 26, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)
Discrete Time Fourier Transform, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)
division, § 2.5.3(90)
DLL, 70

dmad, § 3.1.2(119)

dolby pro logic, § 2.3.2(81)

down-sample, § 1.3.4(19)

downsampling, § 1.3.2(18)

dp, § 3.1.2(119)

dsk 5510, § 3.3.2(131)

dsp, § 1.1.1(1), § 1.2.1(8), § 1.2.2(11), § 1.3.1(17), § 1.3.2(18), § 1.3.3(19), § 1.3.4(19), § 1.4.1(20), § 1.4.2(21), § 1.4.3(22), § 1.4.4(23), § 1.5.2(29), § 2.2.1(76), § 2.2.2(77), § 2.3.1(78), § 2.3.2(81), § 2.4.1(84), § 2.5.1(86), § 2.5.2(89), § 2.5.3(90), § 3.1.1(117), § 3.1.2(119), § 3.2.1(124), § 4.1(135), § 4.2(136), § 4.3(137), § 4.4(138), dst, § 3.1.2(119)

dtft, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)

early sample, 71

echo, § 2.2.1(76), § 2.2.2(77)

einfinite impulse response, § 1.4.4(23)

eilliptic impulse response, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
eelliptic low-pass filter, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)

eencoder, § 1.4.1(20), § 2.2.1(76)

eexternal memory, § 2.2.2(77)

e fast algorithms, § 1.5.3(33), § 1.5.4(46), § 1.6.1(55)
efast fourier transform, 26, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)
efeed back, § 1.4.1(20), § 2.2.1(76)
efft, 26, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46), § 1.6.1(55)

efilter, § 1.2.3.1(16)

efloating-point, § 1.2.3.1(16)

e fixed-point, § 1.4.3(22)

e flip, § 2.6.1(91)
fourier transform, § 1.5.2(29)
fractional arithmetic, § 1.2.1(8), § 3.1.1(117)
fractional arithmetic mode, 10

efractional arithmetic, § 1.18

efractional number representation, § 1.2.3.1(16)
frequency domain, § 1.5.3(33), § 1.5.4(46)
fractional arithmetic, § 1.18

freqz, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
efunction generator, § 1.1.1(1)

gain, 22

gain factor, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
gradient descent, § 2.4.1(84)
gegraphical user interface, § 2.2.1(76)
gui, § 3.2.1(124)

ehamming, § 1.5.2(29)
ehexadecimal, § 1.2.1(8), 8

ehilbert transform, § 2.3.2(81)

eidk, 92

eidm, 93

eidc, § 2.6.1(91)
ehr, § 1.4.1(20), 20, § 1.4.2(21), § 1.4.4(23), § 2.3.1(78), 87

ehilbert filter, § 1.4.3(22), § 1.5.3(33), § 1.5.4(46), § 1.6.2(57)
eimage, § 2.6.1(91)
eimage data manager, 93

eimage developers kit, 92

eimaging, § 1.3.2(18)
eimmediate addressing, § 3.1.2(119), 119, 120

eimpulse response, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
eindirect addressing, § 3.1.2(119), 119, 122

einfinite impulse response, § 1.4.1(20), § 1.4.2(21), § 1.4.4(23)
einfinite impulse-resp onse, 20

einteger, § 1.4.3(22)
einterpolation, § 1.3.1(17), § 1.3.4(19), § 2.1.1(61)
einvert, § 2.6.1(91)

ke, § 3.1.2(119)

lab, § 4.1(135), § 4.2(136), § 4.3(137), § 4.4(138)
elabels, § 3.1.2(119)

elate sample, 71

elevinson-durbin, § 2.5.1(86)
elevinson-durbin algorithm, 88, § 2.5.2(89)
elibdc1394, § 2.6.1(91)

elibrary routines, § 1.6.1(55)
e linear predictive coding, § 2.5.1(86)
e linear prediction, § 87

e linear predictive coding, 86, § 2.5.2(89)
e linear time-invariant, § 1.4.1(20), 20, § 1.4.2(21), § 1.4.4(23)
elinux, § 2.6.1(91)
elms, § 2.4.1(84)
low-pass, § 2.3.1(78)
LPC, 86
LTI, § 1.4.1(20), 20, § 1.4.2(21), § 1.4.4(23)

M
mac, § 1.2.2(11)
macm, § 1.2.1(8)
mainlobe, § 1.5.2(29)
matched filter, § 2.1.2(65), 66
MATLAB, § 1.1.1(1), § 1.3.3(19), § 1.6.1(55), § 3.2.1(124)
memory map, § 2.2.2(77)
memory-mapped registers, § 3.1.2(119)
MMR, § 3.1.2(119)
MMRx, § 3.1.2(119)
MMRY, § 3.1.2(119)
mov, § 1.2.1(8)
multirate, § 1.3.2(18), § 1.3.3(19)
multirate processing, 17
multirate sampling, § 1.3.1(17), § 1.3.4(19)
multirate system, § 1.3.1(17), § 1.3.4(19)
music, § 2.3.1(78)

N
narrow-band, § 2.3.1(78)
NCO, 62
negative, § 2.6.1(91)
noise, § 2.1.2(65)
nonlinear phase, § 1.4.1(20)
notch filter, § 1.4.1(20), § 1.4.2(21), 22, § 1.4.4(23)
umerically-controlled oscillator, § 2.1.1(61), 62

O
on-time sample, 71
opcode, 119
optimization, § 1.6.1(55), § 1.6.2(57), § 1.6.3(58)
optimizing compiler, § 1.6.1(55)
overflow, § 1.2.3.1(16), 17, § 1.4.2(21), 22, § 1.4.5.1(24), § 3.1.1(117)

P
phase-locked loop, § 2.1.1(61), 61
PLL, 61
pmad, § 3.1.2(119)
PN, § 1.6.3(58)
PN generator, § 1.5.3(33), § 1.5.4(46)
poles, § 1.4.1(20), § 1.4.2(21), 22, § 1.4.4(23)
Power Spectra, § 1.5.2(29)
Power Spectral Density, 26, § 1.5.3(33), § 1.5.4(46), § 1.6.2(57)
Power Spectral Density Estimate, § 1.5.2(29)
power spectral density estimation, § 1.5.3(33), § 1.5.4(46)

Q
QPSK, § 2.1.1(61)
quadrature phase-shift keying, § 2.1.2(65)
quantization, § 1.2.3.1(16)
quantization error, 17, § 1.4.5.1(24)
quantize, § 1.4.1(20), 23, § 1.4.4(23)

R
receiver, § 2.1.2(65)
RGB, § 2.6.1(91)
rounding, § 1.4.5.1(24), 24
roundoff error, § 1.2.3.1(16), 17, § 1.4.5.1(24)
rpt, § 1.2.1(8)
RTDX, § 2.2.1(76), § 3.2.1(124)

S
sample-rate compressor, § 1.3.1(17), 17, § 1.3.2(18), § 1.3.3(19), § 1.3.4(19)
sample-rate expander, § 1.3.1(17), 17, § 1.3.2(18), § 1.3.3(19), § 1.3.4(19)
sampling rate, § 3.3.2(131)
saturation, § 1.4.5.1(24), 25
SDL, § 2.6.1(91)
sidelobe, § 1.5.2(29)
sign bit, 16
sign extension, 119
sign-extended, 119
simulation, § 1.6.1(55)
Smem, § 3.1.2(119)
SP, § 3.1.2(119)
spectral analysis, § 1.5.3(33), § 1.5.4(46)
spectrum, § 1.5.3(33), § 1.5.4(46)
speech, § 2.5.1(86), § 2.5.2(89)
speech analysis, § 2.5.2(89)
speech coding, § 2.5.1(86), § 2.5.2(89)
speech compression, § 2.5.1(86), § 2.5.2(89)
speech processing, § 2.5.3(90)
speech synthesis, § 2.5.1(86), § 2.5.2(89)
src, § 3.1.2(119)
stability, § 1.4.3(22)
stack pointer, § 3.1.2(119)
streaming, § 2.6.1(91)
surround sound, § 2.3.2(81)
system identification, § 2.4.1(84)

T
test vector, § 1.1.1(1)
INDEX

TI, § 4.4(138)
TMS320C55x, § 1.1.1(1)
truncation, § 1.2.3.1(16), § 1.4.5.1(24), 25
truncation error, 17, § 1.4.5.1(24)
twiddle-factor, § 1.5.2(29)
two's complement, § 1.2.3.1(16), § 3.1.1(117)
Two's-complement, 117
two's-compliment, § 1.2.1(8)

U
up-sample, § 1.3.4(19)
upsample, § 1.3.1(17), § 1.3.3(19)
upsampling, § 1.3.2(18)

V
VCO, 62
video, § 2.6.1(91)
Video Processing, § 2.6.4(98)
Video Processing on the TI67x, Memory Management, Image Developers Kit,
§ 2.6.6(109)
voltage-controlled oscillator, § 2.1.1(61), 62

W
windowing, § 1.5.2(29), § 1.5.3(33), § 1.5.4(46)
wraparound, 25
wraparound error, § 1.4.5.1(24)

X
xcorr, § 2.5.2(89)
Xmem, § 3.1.2(119)

Y
Ymem, § 3.1.2(119)
YUV, § 2.6.1(91)

Z
zero-pad, § 1.5.2(29)
zero-placement, § 1.3.3(19)
zeros, § 1.4.1(20), § 1.4.2(21), 22, § 1.4.4(23)
Attributions

Collection: Digital Signal Processing Laboratory (ECE 420 55x)
Edited by: Thomas Shen, David Jun
URL: http://cnx.org/content/col10397/1.10/
License: http://creativecommons.org/licenses/by/2.0/

Module: "DSP Development Environment: Introductory Exercise for TI TMS320C55x"
Used here as: "Lab 0: Hardware Introduction"
By: Thomas Shen, David Jun
URL: http://cnx.org/content/m13811/1.16/
Pages: 1-8
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: DSP Development Environment: Introductory Exercise for TI TMS320C54x (ECE 420 Specific)
By: Mark Butala, Jason Laska
URL: http://cnx.org/content/m11019/2.6/

Module: "FIR Filtering: Basic Assembly Exercise for TI TMS320C55x"
Used here as: "Lab 1: Prelab"
By: Thomas Shen, Douglas L. Jones
URL: http://cnx.org/content/m13810/1.5/
Pages: 8-11
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: FIR Filtering: Basic Assembly Exercise for TI TMS320C54x
By: Douglas L. Jones, Swaroop Appadweda, Matthew Berry, Mark Haun, Jake Janovetz, Michael Kramer, Dima Moussa, Daniel Sachs, Brian Wade, Jason Laska
URL: http://cnx.org/content/m10022/2.22/

Module: "FIR Filtering: Exercise for TI TMS320C55x"
Used here as: "Lab 1: Lab"
By: Thomas Shen, David Jun
URL: http://cnx.org/content/m13791/1.8/
Pages: 11-15
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: FIR Filtering: Exercise for TI TMS320C54x (ECE 320 specific)
By: Mark Butala
URL: http://cnx.org/content/m11020/2.6/

Module: "Fixed-Point Number Representation"
By: Douglas L. Jones
URL: http://cnx.org/content/m1930/1.2/
Pages: 16-17
Copyright: Douglas L. Jones
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: Thomas Shen, Douglas L. Jones
URL: http://cnx.org/content/m13799/1.2/
Pages: 21-22
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: IIR Filtering: Filter-Design 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/m10623/2.11/

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/
Pages: 22-23
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: "IIR Filtering: Exercise on TI TMS320C55x"
Used here as: "Lab 3: Lab"
By: Thomas Shen
URL: http://cnx.org/content/m13798/1.3/
Pages: 23-24
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: IIR Filtering: Exercise on TI TMS320C54x (ECE 320 specific)
By: Mark Butala
URL: http://cnx.org/content/m11021/2.4/

Module: "Fixed-Point Quantization"
By: Douglas L. Jones
URL: http://cnx.org/content/m11921/1.2/
Pages: 24-26
Copyright: Douglas L. Jones
License: http://creativecommons.org/licenses/by/1.0

Module: "Spectrum Analyzer: Introduction to Fast Fourier Transform (55x)"
Used here as: "Lab 4: Intro"
By: Thomas Shen, Douglas L. Jones
URL: http://cnx.org/content/m13804/1.2/
Pages: 26-29
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: Spectrum Analyzer: Introduction to Fast Fourier Transform and Power Spectra (ECE 420 specific)
By: Matt Kleffner, Douglas L. Jones
URL: http://cnx.org/content/m12369/1.2/
Module: "Binary Phase Shift Keying Transmitter: Optimization Exercise (55x)"
Used here as: "Lab 5: BPSK Transmitter"
By: Spencer Brady
URL: http://cnx.org/content/m15202/1.1/
Pages: 58-59
Copyright: Spencer Brady
License: http://creativecommons.org/licenses/by/2.0/

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: 61-65
Copyright: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs, Jake Janovetz, Michael Kramer, 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: 65-76
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: "Audio Effects: Real-Time Control using RTDX"
By: Thomas Shen
URL: http://cnx.org/content/m15465/1.1/
Pages: 76-77
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: 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/m10483/2.22/

Module: "Audio Effects: Using External Memory (55x)"
Used here as: "Audio Effects: Using External Memory"
By: Thomas Shen
URL: http://cnx.org/content/m15240/1.1/
Pages: 77-78
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: Audio Effects: Using External Memory
By: Douglas L. Jones, Swaroop Appadwedula, Matthew Berry, Mark Haun, Dima Moussa, Daniel Sachs
URL: http://cnx.org/content/m10480/2.15/

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: 78-81
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: "Using RTDX with a MATLAB GUI"
By: Thomas Shen
URL: http://cnx.org/content/m14388/1.5/
Pages: 124-130
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/
Based on: Using the Serial Port with a MATLAB GUI
By: Jason Laska, Matthew Berry, Daniel Sachs
URL: http://cnx.org/content/m12062/1.1/

Module: "Obtaining Cycle Counts in Code Composer Studio 3.1"
Used here as: "Cycle Counts"
By: Thomas Shen
URL: http://cnx.org/content/m14415/1.2/
Pages: 130-131
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/

Module: "Using the I/O Ports of the DSK 5510"
By: Thomas Shen
URL: http://cnx.org/content/m14478/1.2/
Pages: 131-132
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/

Module: "Files for TI DSK 5510"
By: Thomas Shen
URL: http://cnx.org/content/m16926/1.3/
Pages: 132-133
Copyright: Thomas Shen
License: http://creativecommons.org/licenses/by/2.0/

Module: "UIUC DSP Lab Fall 2002 Archive"
By: Jason Laska
URL: http://cnx.org/content/m12455/1.1/
Pages: 135-136
Copyright: Jason Laska
License: http://creativecommons.org/licenses/by/1.0

Module: "UIUC DSP Lab Spring 2003 Archive"
By: Jason Laska
URL: http://cnx.org/content/m12456/1.1/
Pages: 136-137
Copyright: Jason Laska
License: http://creativecommons.org/licenses/by/1.0

Module: "UIUC DSP Lab Spring 2004 Archive"
By: Jason Laska
URL: http://cnx.org/content/m12457/1.1/
Page: 137
Copyright: Jason Laska
License: http://creativecommons.org/licenses/by/1.0
Digital Signal Processing Laboratory (ECE 420 55x)
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.