DSP Development Environment: Introductory Exercise for TI TMS320C54x[*]

Douglas L. Jones
Swaroop Appadwedula
Matthew Berry
Mark Haun
Dima Moussa
Daniel Sachs

Abstract

This exercise introduces the hardware and software used in the course. By the end of this module, you should be comfortable with the basics of testing a simple real-time DSP system with Code Composer Studio, the debugging environment we will be using throughout the semester. First you will connect the laboratory equipment and test a real-time DSP system with provided 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 your code. Finally, you will create a filter in MATLAB and use test vectors to verify the DSP's output.

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.

[*]Version 2.22: Feb 25, 2004 12:16 pm -0600
[†]http://creativecommons.org/licenses/by/1.0

2 Lab Equipment

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

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

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 (Example Hardware Setup).

Example Hardware Setup

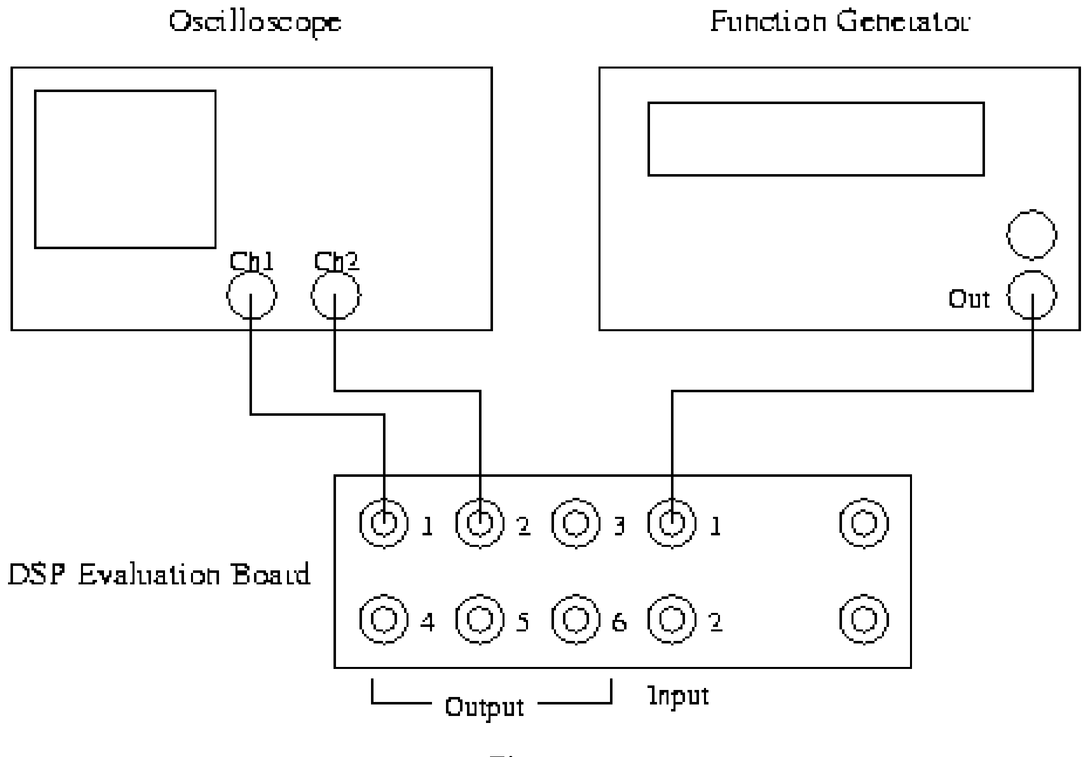

Figure 1

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

2.2 Step 2: Log in

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

3 The Development Environment

The evaluation board is controlled by the PC through the JTAG interface (XDS510PP) using the application Code Composer Studio. This development environment allows the user to download, run, and debug code assembled on the PC. Work through the steps below to familiarize yourself with the debugging environment and real-time system using the provided FIR filter code (Steps 3, 4 and 5 (Section 3.1: Step 3: Assemble filter code)), then verify the filter's frequency response with the subsequent MATLAB exercises (Steps 6 and 7 (Section 3.4: Step 6: Check filter response in MATLAB)).

3.1 Step 3: Assemble filter code

Before you can execute and test the provided FIR filter code, you must assemble the source file. First, bring up a `DOS` prompt window and create a new directory to hold the files, then copy filter.asm[1] , coef1.asm[2] , coef2.asm[3] , core.asm[4] , and vectcore.asm[5] into your directory.

Next, make a copy of `coef1.asm` called "coef.asm" and assemble the filter code by typing `asm filter` at the `DOS` prompt. The assembling process first includes the FIR filter coefficients (stored in `coef.asm`) into the assembly file `filter.asm`, then compiles the result to produce an output file containing the executable binary code, `filter.out`.

3.2 Step 4: Verify filter execution

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

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

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

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

[1]http://cnx.org/content/m10017/2.22/filter.asm
[2]http://cnx.org/content/m10017/2.22/coef1.asm
[3]http://cnx.org/content/m10017/2.22/coef2.asm
[4]http://cnx.org/content/m10017/2.22/core.asm
[5]http://cnx.org/content/m10017/2.22/vectcore.asm

this information, what differences do you expect to see between the signals at input channel 1 and at output channel 2?

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

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

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

Once you have determined the type of filter the DSP is implementing, you are ready to repeat the process with a different filter by including different coefficients during the assembly process. The different coefficients are in the file `coef2.asm`. Make a copy of `coef2.asm` and call it `coef.asm`.

You can now repeat the assembly and testing process with the new filter using the `asm` instruction at the `DOS` prompt and repeating the steps required to execute the code discussed in Step 4 (Section 3.2: Step 4: Verify filter execution).

Just as you did in Step 4 (Section 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.

3.4 Step 6: Check filter response in MATLAB

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

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

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

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

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

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

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

3.5 Step 7: Create new filter in MATLAB and verify

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

First, have MATLAB generate a "random" eight-tap filter by typing `h = gen_filt;` at a MATLAB prompt. Then save this vector of filter coefficients by typing `save_coef('coef.asm',flipud(h));`. Make sure you save the file in your own directory. (The scripts that perform these functions are available as gen_filt.m[6] and save_coef.m[7] .)

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

You may now re-assemble and re-run your new filter code as you did in Step 5 (Section 3.3: Step 5: Re-assemble and re-run with new filter).

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

3.6 Step 8: Modify filter coefficients in memory

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

Change the contents of memory locations `0x1000` through `0x1007` such that the coefficients implement a filter

$$h[n] = 8192\delta(n-4) \quad (1)$$

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

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

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

3.7 Step 9: Test-vector simulation

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

The first step in using test vectors is to generate an appropriate input signal. One way to do this is to use the MATLAB function `sweep` (available as sweep.m[8]) to generate a sinusoid that sweeps across a range of frequencies. The MATLAB function `save_test_vector` (available as save_test_vector.m[9] can then save the sinusoidal sweep to a file you will later include in the DSP code.

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

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

[6]http://cnx.org/content/m10017/2.22/gen_filt.m
[7]http://cnx.org/content/m10017/2.22/save_coef.m
[8]http://cnx.org/content/m10017/2.22/sweep.m
[9]http://cnx.org/content/m10017/2.22/save_test_vector.m

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

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

Now, modify the file `filter.asm` to use the alternative "test vector" core file, vectcore.asm[10] . Rather than accepting input from the A/D converters and sending output to the D/A, this core file takes its input from, and saves its output to, memory on the DSP. The test vector is stored in a block of memory on the DSP evaluation board that will not interfere with your program code or data.

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

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

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

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

use:

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

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

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

[10]http://cnx.org/content/m10017/2.22/vectcore.asm

Next, save the test output file and load it back into MATLAB. This can be done by first saving 3,000 memory elements (six channels times 500 samples) starting with location `0x8000` in program memory. Do this by choosing `File->Data->Save...` in Code Composer Studio, then entering the filename `output.dat` and pressing `Enter`. Next, enter `0x8000` in the Address field of the dialog box that pops up, `3000` in the Length field, and choose `Program` from the drop-down menu next to `Page`. Always make sure that you use the correct length (six times the length of the test vector) when you save your results.

Last, use the `read_vector` (available as read_vector.m[11]) function to read the saved result into MATLAB. Do this using the following MATLAB command:

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

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

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

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

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

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

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

You will observe that the two sequences are not exactly the same; this is due to the fact that the DSP computes its response to 16 bits precision, while MATLAB uses 64-bit floating point numbers for its arithmetic.

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

[11] http://cnx.org/content/m10017/2.22/read_vector.m