Lab 2: File I/O, processing signals and generation of simple waveforms

EE 352 DSP Laboratory (Spring 2019)

Lab Goals

- Basic file I/O ,i.e., reading/writing data from/to a file stored in your system into code composer studio
- Learn how to input a signal from an external source (signal generator or PC) into the processor, process it and then output the processed signal onto a digital storage oscilloscope (DSO), using AIC3204 codec present in the eZDSP board.
- You will generate sine, square and saw tooth waveforms and display them on both CCS graph window and DSO.

Work-flow you learned in the previous lab

- Connecting your device to CCS
- Creating a new project or copying an existing project into the workspace
- Configuring the linker options and file-search paths
- Building and running a project on the kit
- Making use of breakpoints for debugging the code and using watch window to track variable values.

The above work-flow will be frequently required in this and all the other lab-sessions. If you get stuck somewhere while performing them, go back to **Lab 1** handout.

1 File I/O

On some occasions you may be required to read data from a file stored on your system. In this section we will see, with the help of an example, how to read a file using probe point.

- Create a project (you can use the empty project that you have created earlier).
- Paste the `sine.c` and `sine_int16.dat` files (Uploaded on the course wiki page) into project directory.
- Open `sine.c` file. Define a global variable `x[100]` to store the data being read. Save the file.
- Rebuild the project and load the program (the one generated in the `Binaries` with extension `.out`).
- Now open the `sine.c` file again and put a breakpoint at `dataInput()`.
- Open the breakpoints window by clicking `View` → `Breakpoints` and select the `Actions` column (you need to scroll a bit to right) of the marked breakpoint.
- From the `Actions` column select `Read data from file`. A `Parameters` window will open. See Fig. 1
- Select the file `sine_int16.dat` from the same project.
- In the File I/O dialog, change the `Address` to `x` and the Length to 100. Also, put a check mark in the `Wrap Around` box.
- The `Address` field specifies where the data from the file is to be placed. The `Length` field specifies how many samples from the data file are read each time the `Probe Point` is reached. In this case we are taking 100 samples from the file each time.

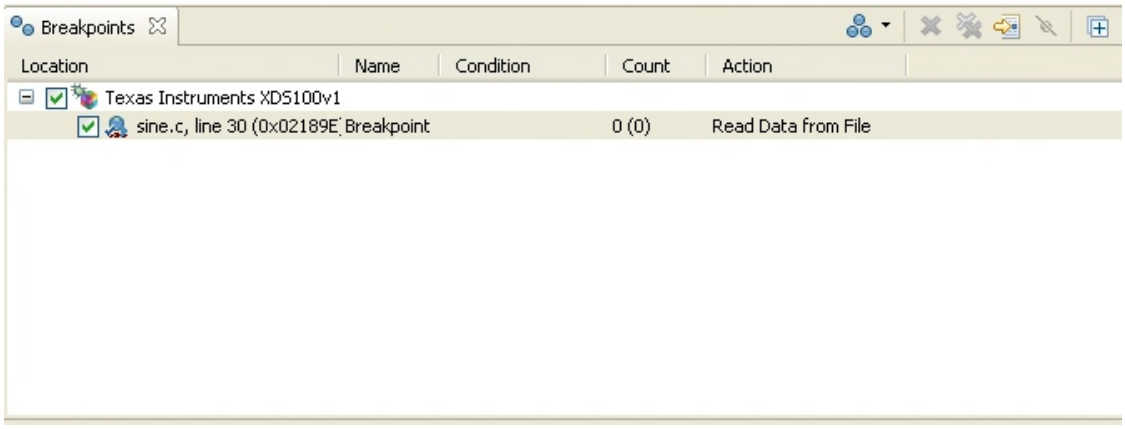

Figure 1: Debug window demonstrating use of File I/O to read data file.

- The `Wrap Around` option causes the IDE to start reading from the beginning of the file when it reaches the end of the file. This allows the data file to be treated as a continuous stream of data even though it contains only 1000 values and 100 values are read each time the `Probe Point` is reached.
- Select `Ok`. When you run the program, the data from `sine_int16.dat` will be read into the array `x`.

1.1 Graphing

The program should be running before starting Graphing.

- Goto `Tools -> Graph -> Single time`.
- You will get a `Graph Properties` dialog box.
- In the `Start address` field give the variable name (in this case `x`) that is to be viewed.
- `Acquisition Buffer Size` (It is the maximum data that can be read and stored at any instance) and `Display Data size` (It should always be lesser than the buffer size) should be set to 100 in this case.
- Specify correct `DSP Data Type` (16 bit signed).
- In the graph, observe the effect of changing the `Acquisition Buffer Size` and the `Display Data size`.

Note: If the sine wave is not observed properly in the graph then check the header of the .dat file. The header should be consistent with the type of data in the file. The file has 16 bit signed integer data. (For details of file format, refer the `Data File Formats.pdf` *file from the uploaded material.*

1.2 Learning Checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. From the material uploaded on the course-wiki page, find the `square_wave.m file` and run it in MATLAB (keep in mind that it is a function file, therefore you will have to open it and go through the information regarding its input and output arguments). It is well commented to help you to generate a `.dat` file of your own.

   Now, create a new project in CCS and add the `squarewv.c` file to it. Read the `.dat` file already generated in CCS using probe point into variable `x` and verify the signal (as the name of the file suggests) using graphing window.

2. You will be using `sine_checkpoint.c`, `sine_int16.dat` and `sine.h` for this checkpoint. You can reuse the project created in section 1.1 (You will have to delete the earlier `sine.c` file in it as 2 main functions cannot reside in a project) or you can create a completely new project. Use `sine_int16.dat` file as an input to receive the data into the variable `input`. Store the variable `output` in `sine_out.dat` using probe point.

Using the MATLAB function file `plot_dat.m` provided in the uploaded lab material, plot and compare the input, output signals and comment on the processing achieved.

2 Processing signals and waveform generation

2.1 Introduction to codec

The eZDSP board is provided with AIC3204 stereo codec for input and output of analog signals. The codec samples analog signals on the microphone inputs and converts them into digital data that can be processed by the DSP. When the DSP is finished with the data it uses the codec to convert the samples back into analog signals headphone output. Analog I/O is done through four 3.5mm audio jacks that correspond to microphone input and headphone output. The codec can select microphone input as the active input. The analog output is driven to headphone output (adjustable gain) connectors.

2.2 Accessing the codec AIC3204

The eZDSP is provided with software library called the Board Support Library (BSL). BSL provides a C-language interface for configuring and controlling the on board peripheral devices. It is divided into separate modules for different peripheral devices like codec, LEDs, DIP switches. In this case we will be using the codec module which contains functions pertaining to the AIC3204 codec. With the help of an example we will see how to use these functions. For all your future experiments, you will be provided with `main.c` file. It is a skeleton file which just takes in the external input using the codec, sends it to the DSP without modifying the data, and puts it back to the output. So the `main.c` allows the developer to work on the data.

The AIC3204 is connected to C5515 processor via two interfaces, i.e., I2C and I2S. I2C interface is used for writing control registers of a codec and I2S interface is used for digital audio data transmission and reception.

2.3 The `main.c` file

Open `main.c` file. It is a well commented file which contains descriptions for all the sections.

At the start we include 3 files, viz, `usbstk5515.h`, `usbstk5515_gpio.h` and `usbstk5515_i2c.h`.

- `usbstk5515.h` contains function for initialization of eZDSP board.
- `usbstk5515_i2c.h` contains functions for reading and writing into I2C bus interface.

After these header files, we see a commented integer array `sinetable`, which contains 48 sample values of 1 cycle of a sine wave. You will be using this look up table in your assignments to generate sine waves.

Provided skeleton program contains following functions and registers

- `AIC3204_rget(Uint16 regnum, Uint16* regval)`

  This function reads value of specified register of AIC3204 whenever this function is called. Anyways, we do not need to use this function in this experiment.

- `Int16 AIC3204_rset(Uint16 regnum, Uint16 regval)`

  This function sets value of specified register of AIC3204 whenever this function is called.

- `AIC3204_config(Uint8 sampling_freq)`

  This function is used for configuring the codec, e.g., changing the input volume, output volume, sampling frequency etc.

- `USBSTK5515_init()`

  This function resets `PCGCR1` and `PCGCR2` system registers so that processor will boot from default location (for more information go through TMS320C5515 datasheet)

- `I2S0_IR`

  This register is I2S bus interrupt flag register. So whenever processor completes transmission or reception of data from AIC3204, corresponding bit of this register is set by the processor.

- Finally, we write 0X00 to Serializer Control Register (i.e `I2S0_CR`) for closing I2S bus interface.

3 Signal generation and resampling

In this example, we will use a look-up table to generate a sine wave of 1 kHz with the codec rate fixed at 48 kHz. Each cycle of the sine wave should contain 48 samples.

- In order to generate the desired wave, each cycle of the sine wave should contain 48 samples. Uncomment the lookup table in `main.c` file. The table has 48 sample values of 1 cycle of a sine wave.
- Set the sampling Frequency of codec to 48 kHz using `AIC3204_config()`.
- Now initialize `sample` variable of data type `int` and replace the code inside `while(TRUE)` loop with

```
while(TRUE)
{
    for(sample=0;sample<48;sample++)
    {
         while((Xmit & I2S0_IR)==0);        // Wait for transmit interrupt to be pending
         I2S0_W0_MSW_W=(sinetable[sample]);
         I2S0_W1_MSW_W=(sinetable[sample]);
    }
}
```

- Generally, in serial communication scenarios like this, utmost care needs to be taken by the programmer to check that any data already present in any internal register is not overwritten. To simplify this, the processor designer gives the user access to certain flags which indicate the completion (or change of state) of a process.
- Since, we want to send the values in the look-up table one after the other to the DSO (in this case), we would like to make sure that the previous value successfully went to the DSO. To do this, the flag that we are going to check is the `Xmit` bit from the control register `I2S0_IR`. Having understood these concepts regarding the serial communication implemented on the DSP, we are now ready to understand the way to execute this in C.
- Let's see what the code inside the `for` loop does. Note that, the `while` loop contains just the condition and no body (the `while` condition is followed immediately by a semicolon). This is a standard construct used when the program is supposed to wait for something. Here, we want to wait for the `Xmit` bit from the control register `I2S0_IR` to set. Now, can you guess how the condition in the `while` loop achieves this? You may want to have a look at the definition of the macro `Xmit` towards the beginning of the `main.c` file. If you are not able to understand this, seek out your TA/RA. (The & is known as the `bitwise AND` operator in C.)
- Finally, once we are sure that the previous value successfully made its way to the DSO, next sample needs to be sent. Again, we are at the mercy of the DSP designer! She has given us access to two transmit data registers `I2S0_W0_MSW_W` and `I2S0_W1_MSW_W` (one for each channel) which when correctly configured and written to, directly send the data to the codec and hence to the receiver.

4 Learning Checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Understand the working of the code inside the infinite loop. What would happen if you change the number in the `for` loop condition? Explain this to your TA. Using the available look-up table, try generating sine waves of frequencies 500 Hz and 2000 Hz.

2. Fix the codec sampling rate at 24 kHz and generate square and saw-tooth waveforms of fundamental frequencies: 1 kHz, 2.4 kHz. Are you getting the desired waveform on the DSO ?

3. You will be using the `main_2.c` file for this and next checkpoints. Set the codec sampling frequency at 12 kHz. Input a sine wave from the signal generator with each of the following frequencies: 1 kHz, 5 kHz, and 8 kHz. Read the samples using provided code.

   - Observe the output sinusoids on the DSO and determine their frequencies. Listen to the tones using head-phones. Do you hear distinct and clear tones ?
   - Now add code to downsample the signal by a factor of 2, i.e., drop every other sample from the input signal. The signal to be given as input should be from the wave file provided in the updated lab material. Listen to the different wav files with and without downsampling and answer following question.
     - Which frequencies are getting aliased?
     - What are the aliased frequencies?
     - Are you actually downsampling? If yes, then do you hear the desired result? If no, why not?

4. Repeat the previous part with the sampling frequency set to 24 kHz. Relate to the Nyquist sampling rate for each of the sinusoids. Comment on what you observe and justify.

5. Now input a square wave (1kHz, 5kHz) to the DSP using the signal generator and output the same to the DSO. Observe the output on DSO. Comment on it.

6. In this task you will construct an electronic instrument by synthesizing the 12 notes of an octave given the frequencies in the Table 1. You have to play the scale: *Sa Re Ga Ma Pa Dha Ni*.

Table 1: Notes, their frequencies and the tone generated

Note	Frequencies(Hz)	Tone
C	131	Sa
C#	139	-
D	147	Re
D#	156	-
E	165	Ga
F	175	Ma
F#	185	-
G	196	Pa
G#	208	-
A	220	Dha
A#	233	-
B	247	Ni
C2	262	-

Each tone is to be generated for 1 second, one after the other in the form of trains of triangular waves of the corresponding frequencies. The peak value of the triangular wave can be the maximum value the datatype-range allows. Minimum value can be zero. You will be using the `main_saregama.c` file as the skeleton code for this exercise. Use 12 KHz as sampling frequency. The `transmit()` function can come handy here.

Notes

(a) The array `wavelength` holds 7 constants, `a0` to `a6`, where, every `ai` should be calculated to be the number of samples per cycle for the tone i.

(b) The array `no_of_cycles` again holds 7 constants, `b0` to `b6` where every `bi` should be calculated to be the number of cycles for the tone i.

Finally, use these 2 arrays to generate the 7 tones. The while(1) loop is waiting for you!

7. Next, here is a very recognizable song that you need to program your synthesizer to play. Each tuple specifies (Note, #Beats). Take 1 beat = 400 ms.

   (C,1) (C, 1) (D,2) (C,2) (F,2) (E,4);
   (C,1) (C,1) (D,2) (C,2) (G,2) (F,4);
   (C,1) (C,1) (C2,2) (A,2) (F,2) (E,2) (D,2);
   (A#,1) (A#,1) (A,2) (F,2) (G,2) (F,4);