EE 423: Class Demo 1

Introduction

In lab 1, two different methods for generating a sinusoid are used. In `sine_gen.c`, the built-in C function `sin()` is used to approximate the function sin(·) and interrupts are used to send the samples to the codec. For this implementation, the samples of the sine function are calculated at every interrupt. If the discrete-time sequence being generated is periodic, this implementation is a waste of resources[1]. When the discrete-time sinusoid sequence is periodic, a pre-computed lookup table may be stored in memory and samples may be read from the table and outputted to the codec. This is the way that `sine_lookup_poll.c` is coded. However, in this implementation, a polling (instead of interrupt driven) method is used to send the samples to the codec.

In this demo, a mixture of these two methods mentioned above will be used. The idea of having a lookup table from `sine_lookup_poll.c` will be used to store samples of a discrete-time periodic. However, instead of calculating and storing these values by hand, the built-in C function `sin()` will be used to generate the lookup table. The values in the lookup table will be accessed in the same manner as `sine_lookup_poll.c`; however, the codec will be accessed using interrupts as in `sine_gen.c`.

In this demo, you will study:

- code debugging,
- overdriving the codec, and
- code modification.

Demo 1 Preliminaries

Before continuing with this demo, make sure that the following tasks from the Lab 1 write-up have been completed (in the given order):

1. establish the appropriate file structure on your local u:\ drive, and
2. successfully implement `sine_gen.pjt` on the DSK board.

Once `sine_gen.pjt` is correctly producing a sine wave, move on to the rest of the demo.

[1]The method used in `sine_gen.c` is useful for generating sinusoids with an arbitrary frequency. Using a lookup table may be impossible when the period of the discrete-time sinusoid sequence is very large or if the sequence is aperiodic. In the communications lab (lab 7), the carrier frequency will be generated using the method in `sine_gen.c`.

Creating A New Project File From An Existing Project File

For the first part of the demo, you will want to create the folder 'Sine_500Hz' in the path u:\DSP\Lab_01\. This can be done using Windows Explorer. This project will be interrupt-based, so you will want to use the project `sine_gen.pjt` as your template for this new project. NB: If you were going to create a polling-based program, then you would want to use `sine_lookup_poll.pjt` as your template. To create this new project, do the following.

1. Copy the file `sine_gen.pjt`, from the path u:\DSP\Lab_01\Sine_gen, into your newly created path u:\DSP\Lab_01\Sine_500Hz\.
2. Change the name of `sine_gen.pjt` to `sine_500Hz.pjt`.
3. Go to the class webpage and follow the link Schedule and Assignments. Download the file `sine_500Hz.c` (located in the third column of week 1) into your newly created path u:\DSP\Lab_01\Sine_500Hz\.
4. Open the new project `sine_500Hz.pjt` in Code Composer Studio. When you try to open this project, a CCS window should open up saying that it cannot find the file `sine_gen.c`. Since a new C source code file, namely `sine_500Hz.c`, has been downloaded, select 'Remove'.
5. Add the downloaded C source code file `sine_500Hz.c` to the project by selecting 'Project', then 'Add Files to Project', then select `sine_500Hz.c`, and then click 'Open'.
6. In CCS, select the pull down menu 'Project' and go to 'Build Options'. Click on the Linker tab. In the field 'Output Filename (-o):', change the word `sine_gen` in `.\Debug\sine_gen.out` to `sine_500Hz.out`. Click 'OK'.
7. In the CCS window, double-click on `sine_500Hz.c` in the left-hand window. This will open up the source code file on the right-hand side of the CCS window. There are a few errors in this program, which you will try to debug.

Debugging A Program

Try to build the executable file `sine_500Hz.out` it in CCS. When you do this, you should get the following warnings and errors:

```
"sine\_500Hz.c", line 4: error: expected a ";"
"sine\_500Hz.c", line 9: error: identifier "sine\_table" is undefined
"sine\_500Hz.c", line 20: error: expected a ")"
"sine\_500Hz.c", line 27: warning: parsing restarts here after previous syntax error
"sine\_500Hz.c", line 27: error: expected a statement
```

Try to figure out how to correct these mistakes. Use the given programming examples in Lab 1 to figure out what is wrong. We will go over the corrections in a few minutes.

Setting Up A Watch Window

The C code `sine_500Hz.c` (with errors) is shown below.

```
1.)   #include <math.h>               // needed for sin() function
2.)   #define PI 3.14159265359       // define the constant PI
3.)   short ctr
4.)   short sine_table[16];           // array of values sent to codec
5.)   short amp=20;
6.)
7.)   interrupt void c_int11()        // interrupt service routine
8.)   {
9.)    output_sample(amp*sine_table[ctr]);  // output each sine value
10.)
11.)   if (ctr < 15) ++ctr;
12.)   else ctr = 0;
13.)
14.)   return;                        // return from interrupt
15.)  }
16.)
17.)  void main()
18.)  {
19.)    // Create the lookup table here
20.)    for(ctr=0; ctr<16; ctr++      // 16 samples of one period of a sinsoid
21.)    {                             // being outputted at rate 8kHz = 500Hz sine wave
22.)      sine_table[ctr]=(short)1000*sin(2*PI*ctr/16);
23.)    }
24.)
25.)    // Initialize codec for interrupts and begin program
26.)    ctr=0;
27.)    comm_intr();                  // init DSK, codec, SP0 for interrupts
28.)    while(1)                      // wait for an interrupt to occur
29.)  }
```

In the first five lines of the `main()` function (lines 19 through 23), a `for` loop is used to store the samples of one period of a discrete-time sinusoid. Here, each sample point is $\frac{2\pi}{16}$ radians away from the previous sample. This is to say that the offset value between sample points (as it was defined in `sine_gen.c`) is

$$2\pi\frac{f_0}{f_s} = \frac{2\pi}{16}. \tag{1}$$

Since our codec is running at rate $f_s = 8$kHz, our observed sinusoid will have frequency

$$f_0 = \frac{f_s}{16} = \frac{8000}{16} = 500\text{Hz}, \tag{2}$$

which is what we want.

In CCS, open a watch window by clicking on the pull-down menu 'view', and selecting 'Watch Window'. In the sub-window that opens up in the bottom right part of the CCS, select the tab 'Watch 1'. Click on the field below 'Name' and enter the global variable name `amp`. Press 'Enter'. The value `20` should appear in the field just to the right of where you entered `amp`. We will use this window to change the value of `amp`. To observe the affects of the variable `amp`, click on the value `20` and change it to a positive integer value less than `33`. What happens to the peak-to-peak voltage of the sinusoid? When you listen to the sinusoid on headphones, how does the volume intensity change with the value stored in `amp`? Try negating the value stored in `amp` (e.g. change `20` to `-20`). How does this affect the output?

Overdriving The Codec

In `sine_500Hz.c`, the values in the array `sine_table[]` go from -1000 to +1000. With `amp=20`, the values of the output go from -20000 to 20000. This may be seen in line 9 where the values in `sine_table[]` are scaled by the value stored in the variable `amp`. The codec is setup to receive 16-bit signed integers, which are numbers in the range $-2^{15} = -32768$ to $2^{15} - 1 = 32767$. Any values that are greater than 32767 or less than -32768 will result in a 2's compliment overflow. This means that values of `amp` less than or equal to `32` will not overflow the codec and values greater than or equal to `33` will overflow the codec. Keeping in mind that `amp` must be integer valued, use a watch window to experiment with various values of `amp`, both in the underflow (less than or equal to `32`) and overflow (greater than `32`) ranges. Also, try negative values for `amp`. Listen to the results. Can humans detect polarity or phase offsets[2] when listening to musical tones?

Modifying The Code

So far, a C generated lookup table has been created that will produce a 500Hz sinusoid, provided the on-board codec is being used. In order to generate a sinusoid with a lower observed frequency, a larger sinusoid table that contained more samples of a single period of a sinusoid would need to be created. However, to generate a sinusoid with a higher observed frequency, samples in the lookup table may be skipped over.

Code Analysis

Let's examine the code given in the previous section before modifying it. In particular, look at line 11. If the variable `ctr` is less than `15` (values `0` to `14`), then it is incremented by one with the C code `++ctr`. Consequently, if `ctr` is equal to `15`, then the last sample

[2]Flipping the polarity of sine wave is the equivalent of introducing a 180 degree phase shift into the original sine wave. In other words, changing the polarity is a specific type of phase offset (a more general concept).

of the lookup table has been sent to the codec and `ctr` must be reset to point to the first element of the table (i.e. `ctr = 0;`). This may be seen by examining lines 11 and 12. Let's assume that `ctr` is NOT less than `15` (i.e. `ctr = 15`), so the `if` statement in line 11 is FALSE. In this case, the instruction following the `if` statement will not be executed. Since the `if` statement is FALSE, the program will look for an `else` statement, which it will find on the next line[3]. In this case, the program will execute the code in line 12, which will set the value of `ctr` to zero (i.e. it will point to the first element in the lookup table).

Code Modification

In this section, the code will be modified to increase the frequency of the observed sinusoid by an integer multiple of fundamental frequency. This will be accomplished by allowing the value stored in `ctr` to be incremented by more than one at each iteration. This may be done by creating another global variable, which will be labelled `step`. This new variable, `step`, will then be added to the variable `ctr` at each iteration. The pitfall is that the variable `ctr` will not necessarily be reset to zero once it is out of range of the lookup table (in this case, when `ctr` > 15). For example, if `step = 3`, two samples will be skipped in the lookup table at each iteration. Assuming that `ctr = 0` at the first interrupt, during the sixth interrupt the value in `ctr` will increment from 15 to 18. This means that the sample is incremented from the last sample (sample 16 or array index 15) of one period and skip to sample 3 (or array index 2) of the next period. To convert 18 to 2, the modulus (or remainder) operator will be used. There are two possible ways that we can code this. One is to use the built-in *modulus operator*, namely (`%`), and the other would be to code the remainder algorithm directly. Both methods are stated below. Since the modulus operator is a built in arithmetic function in C, it will be the preferred way to calculate the remainder.

In order to modify this program to implement the algorithm given above, make the following changes:

- Replace lines 11 and 12 with the following two lines in the given order

  ```
  ctr += step;
  ctr %= 16;
  ```

- Add the global variable `short step=1;` after line 5.

Rebuild the project, load it onto the DSK, and run it. You should be able to observe a 500Hz sine wave on the oscilloscope. Now, use a watch window to change the value of `step`. How does changing the value of step affect the frequency of the sine wave observed

[3]If there is no `else` statement and the conditional in the `if` statement is FALSE, the program will ignore the code associated with the `if` statement and continue with next line of code not associated with the `if` statement. (NB: Here, "associated with" refers to the code that appears on the same line as the `if` (or `else`) statement. If there is more than one line of code that is to be associated with an `if` (or `else`) statement, it must be coded within { and } following the `if` (or `else`) statement). As a last note, if the `if` statement is TRUE, then the code associated with the `else` statement will not be executed.

on the oscilloscope? If you program is working correctly, the observed frequency should be `step*500`, provided `step` is less than `16`. Try values of step that are greater than or equal to `16`. Can you explain what you see?

Once you have this program working properly, use a watch window to set the value of `step` equal to `2` and overdrive the codec again (as we did in the previous section). You should see a 3kHz sinusoid. This is a special phenomenon that occurs when you overdrive this 16-bit signed integer codec while implementing a 1kHz sinusoid. The reason for this will be discussed in class.

As an alternative, the modulus operator may be coded directly in C by replacing the line `ctr %= 16;` with either

```
while(ctr > 15)
{
  ctr = ctr - 16;
}
```

or if you prefer the short-hand notation, you may code this as

```
while(ctr > 15) ctr -= 16;
```

In the second notation, the one line instruction is coded on the same line as the `while` loop. This may only be done if there is only one instruction to be executed in a given loop. NOte that the compound instruction `-=` was used to decrement the variable `ctr`. This is not necessary, but it is the most compact way to code this particular loop.