RTDSP Lab2 Write-up – Learning C and Sine Wave Generation

Questions

1. *Provide a trace table of Sinegen for several loops of the code. How many samples does it have to generate to complete a whole cycle?*

Here is the trace table for 12 cycles, where x(n) represents the input and y(n) represents the output.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | x(n) | y(n) | y(n-1) | y(n-2) |
| 0 | 1 | 0.7071 | 0.0000 | 0.0000 |
| 1 | 0 | 1.0000 | 0.7071 | 0.0000 |
| 2 | 0 | 0.7071 | 1.0000 | 0.7071 |
| 3 | 0 | 0.0000 | 0.7071 | 1.0000 |
| 4 | 0 | -0.7071 | 0.0000 | 0.7071 |
| 5 | 0 | -1.0000 | -0.7071 | 0.0000 |
| 6 | 0 | -0.7071 | -1.0000 | -0.7071 |
| 7 | 0 | 0.0000 | -0.7071 | -1.0000 |
| 8 | 0 | 0.7071 | 0.0000 | -0.7071 |
| 9 | 0 | 1.0000 | 0.7071 | 0.0000 |
| 10 | 0 | 0.7071 | 1.0000 | 0.7071 |
| 11 | 0 | 0.0000 | 0.7071 | 1.0000 |

From the table it is clear that it takes 8 samples to generate a complete sinewave.

1. *Can you see why the output of the sinewave is currently fixed at 1 kHz? Why does the program not output samples as fast as it can? What hardware throttles it to 1 kHz?*

Sampling\_freq = 8kHz and it takes 8 samples to generate a complete sinewave, therefore the output sinewave is fixed at 8kHz/8 = 1kHz.

// send to LEFT channel (poll until ready)

**while** **(!**DSK6713\_AIC23\_write**(**H\_Codec**,** **((**Int32**)(**sample **\*** L\_Gain**))))**

**{};**

// send same sample to RIGHT channel (poll until ready)

**while** **(!**DSK6713\_AIC23\_write**(**H\_Codec**,** **((**Int32**)(**sample **\*** R\_Gain**))))**

**{};**

These lines in the while loop of main function wait until the left and right output channel is ready. The function DSK6713\_AIC23\_writereturnstrueafter a wait period that’s the reciprocal of the sampling frequency. If we can only write to the output 8000 times a second, and it takes 8 samples to complete a sine wave, this would throttle the output sinewave to 1kHz.

1. *By reading through the code can you work out the number of bits used to encode each sample that is sent to the audio port?*

As shown in the code snippet above, sample returned from the sinegen function is multiplied by the gain and type casted to a 32-bit integer. Therefore 32 bits are used to encode each sample that is sent to the audio port.

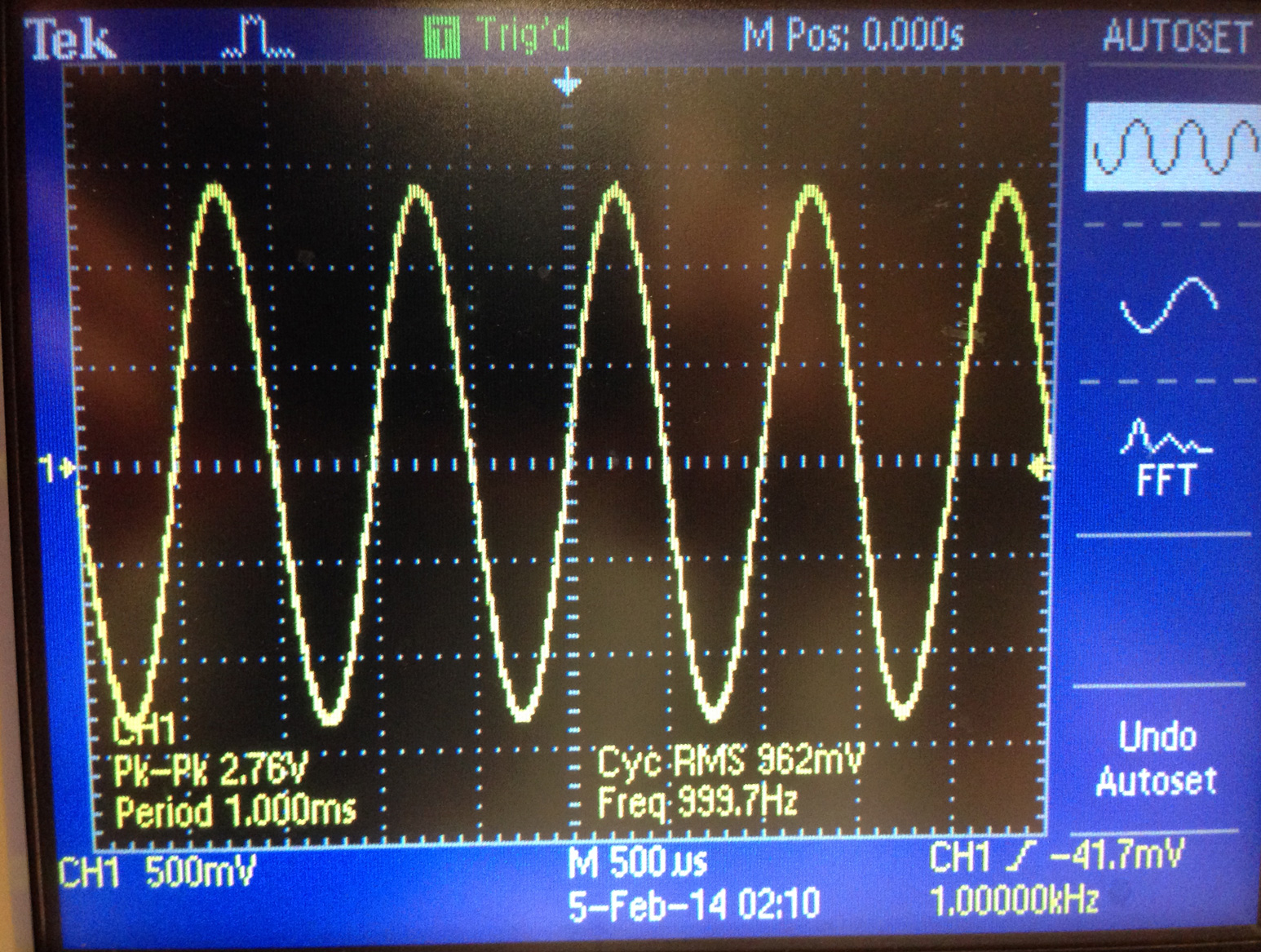
**Code Operation**

At the start of the main function we initialize the board, the audio ports and the values in the sine lookup table. A value of from the lookup table is then returned as sample from the sinegen function, which is then sent to the left and right channel of the DSK board at the sampling frequency.

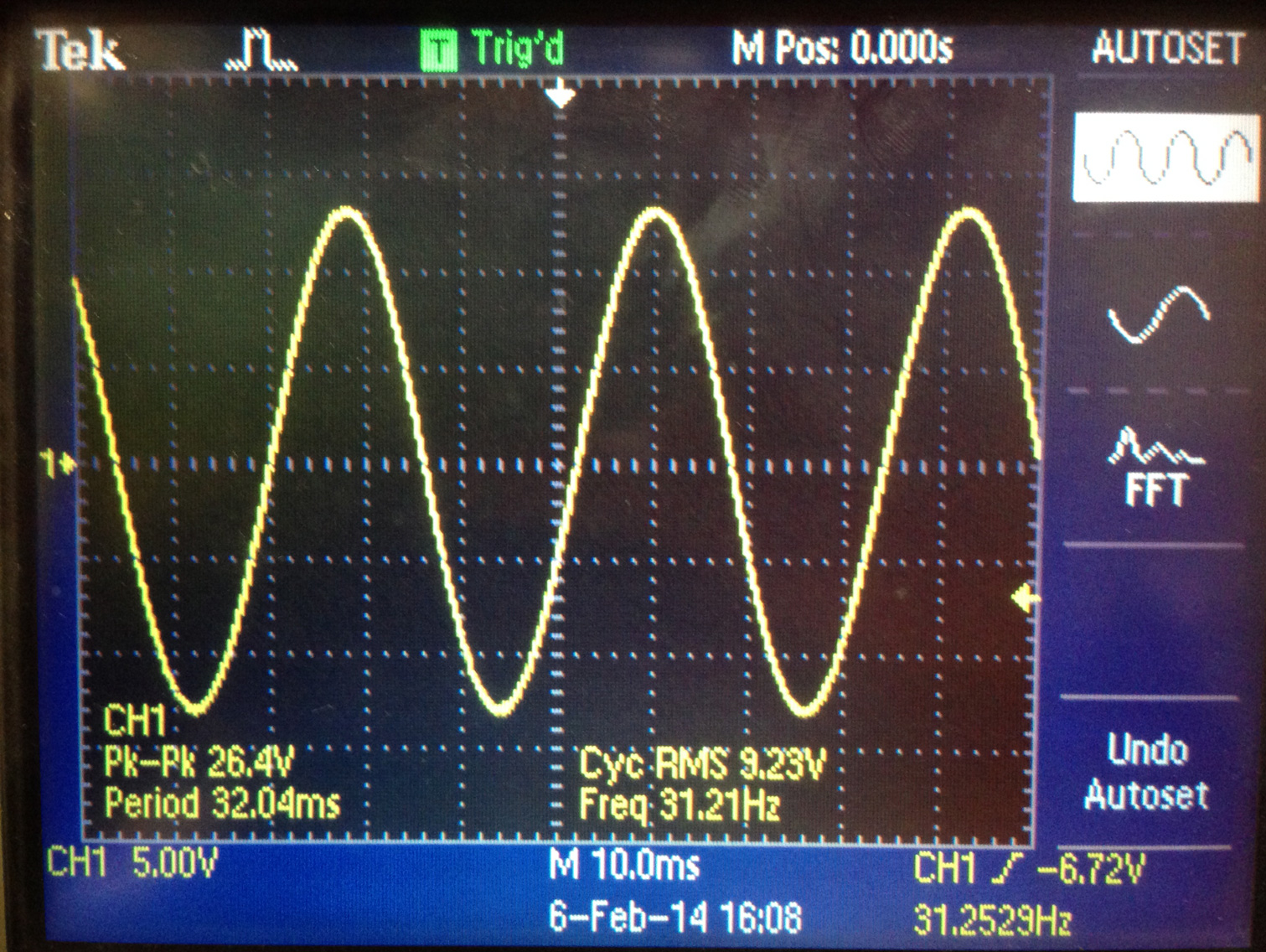
Inside the sinegen function, a variable jump is used to calculate the gap to the next entry of the lookup table required according to sine\_freq relative to sampling\_freq. A global variable x is used as the index and its value is incremented by jump every time sinegen is executed.

x **=** **(**int**)**x**%**SINE\_TABLE\_SIZE**;** wraps x around the size of the sine lookup table (in this case 256) to avoid array index overflow. It is also type casted as an int since x is used as an index. Finally the function returns the entry corresponding to the index x in the sine lookup table.

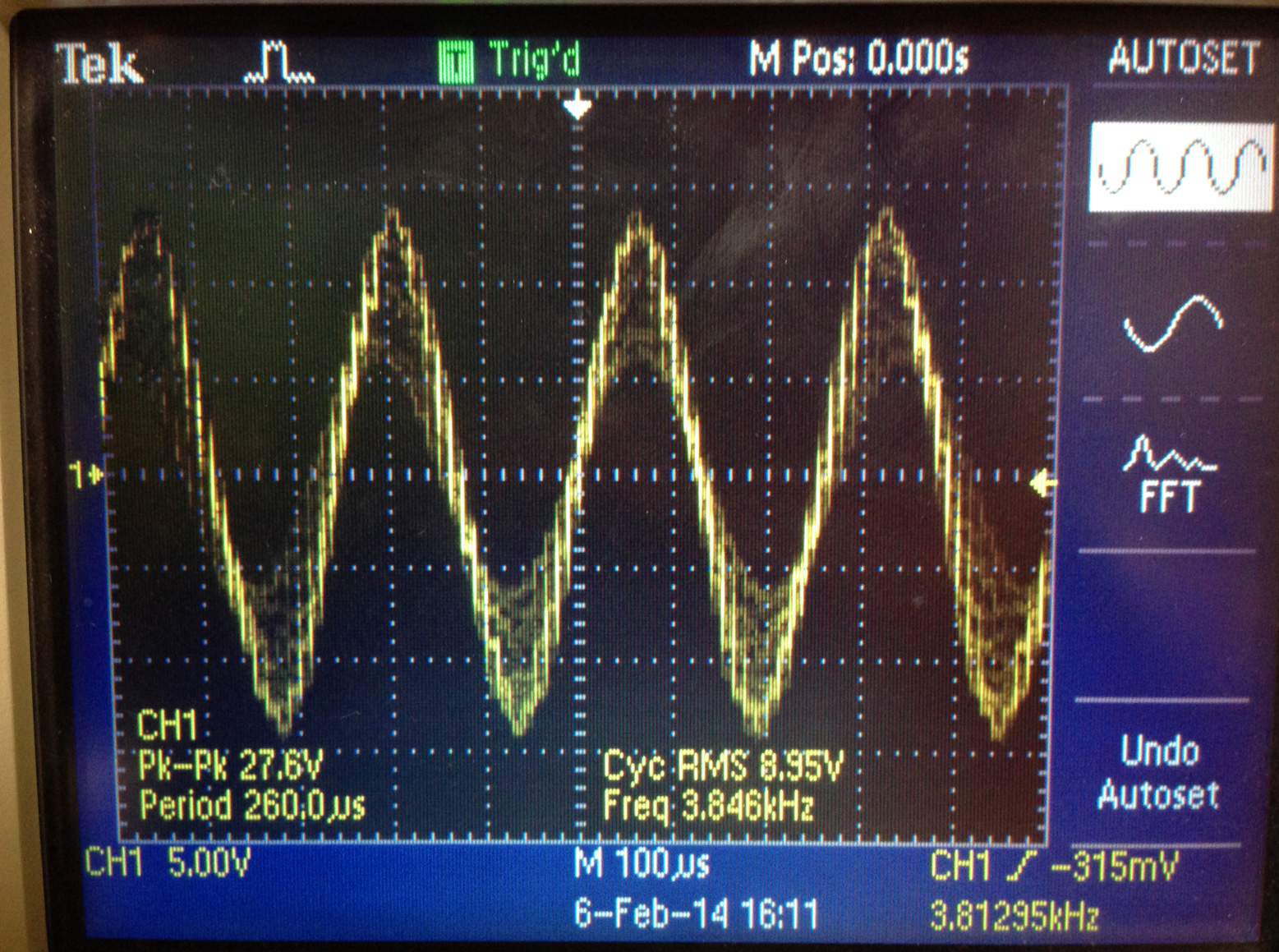
**Scope Traces**



*Scope trace showing 1kHz sine output*



*Scope trace showing lower limit of operation*



*Scope trace showing upper limit as we approach the nyquist freq.*

**Limitations of code operation**

Using this table lookup method means the resolution of the outputted sine wave is limited by the size of the table. To increase resolution without using a larger lookup table, we can store only a quarter of the sine wave and make use its symmetric nature, i.e. traverse the table in both directions and multiply the returned values by 1 or -1 according to the quadrant desired.

The theoretical maximum sine output frequency should be slightly under half the sampling frequency we set. This constraint arises due to the symmetric nature of sine waves; if we output at half the sampling frequency, we would be outputting 2 entries from the lookup table per period, one at the start, one halfway, both of which are zero. Reconstructing anything sensible out of this would be difficult. As we increase the desired output frequency closer to this limit, the sine wave became noisier due to any rounding error being much more pronounced with so few samples, as well as the actual timing of the output being not precisely as set by the sampling frequency.

Since we are simply adding the jump to find the next index in the array to read from, when this jump gets truncated to zero due to integer casting for the index, this would determine the lower bound of frequencies we can output. When using a sampling frequency of 8kHz this lower bound was around 30Hz.

Given jump = (256\*freq/sampling freq) + 0.5. The addition of 0.5 allows to us round off jump instead of taking the floor. This addition is unnecessary as we are only concerned about the actual outputted frequency; however, it provides better approximation for the lookup table index. We need a jump of at least 1 in order to increment through the sine lookup table. To achieve this, freq needs to be at least 31.25Hz. In other words, there aren’t enough entries in the lookup table when freq is less than 31.25Hz; our resolution is not high enough to keep outputting values at 8kHz to create a wave slower than 31.25 Hz. A side effect of this half addition is that the output frequency will still be smooth, stable 30Hz if we set the sine\_freq parameter to anything between that and 15 Hz, anything lower and we will get nothing at the output.

float sinegen**(**void**){**

// temporary variable used to output values from function

float x**;**

x **=** **(**SINE\_TABLE\_SIZE**\***sine\_freq**\***count**/**sampling\_freq**)+**0.5**;**

x **=** **(**int**)**x**%**SINE\_TABLE\_SIZE**;** //wrap round lookup table

count**++;** //count is a global variable of type int

**if** **(**x **==** 0**){**

count **=** 0**;** //reset counter

**}**

**return(**table**[**x**]);**

**}**

A workaround of this lower bound is to calculate the whole index of the array we want instead of incrementing the index as shown in the code snippet above.

In this sinegen function the index x is determined by the ratio between the desired sine\_freq and sampling\_freq, as well as a counter count. Count is incremented every time sinegen is executed and it is reset when x = 0 to avoid integer overflow.

Combining this operation with the previous algorithm with the condition where the index is calculated directly when sine\_freq is less than 30Hz, else otherwise, allows us to output sine waves from 10Hz up to the Nyquist frequency. A side effect of outputting at a frequency slower than 30Hz is that the sampling time is effectively reduced because of limited entries in the lookup table. To illustrate, for some samples, the calculated index will not have reached the next integer and we will end up outputting the same sample value. A much better solution would be to use the approach where only half or a quarter of the sine wave is stored to increase the resolution without increasing data usage as mentioned before.