ENCM 369 PIC Activity 6

ENCM 369 ILS Winter 2021 Version 1.0

Activity Objectives

- Apply your knowledge of timing in an embedded system
- Implement the DAC peripheral on the PIC to output various audible tones
- Practice following coding requirements and code review of a peer

Deliverables

- Light pattern design plan
- Check-in of you code to Github
- Completed rubric review of someone's code in your learning community

Background Information

Microcontrollers are digital systems where every computation and process happens as binary ones and zeros are processed by the system's logic gates. As you are learning in the MIPS portion of ENCM 369, the various data paths are set up to read an instruction, decode it, execute it, and writeback the result to a memory location or feed it forward for use in the next operation. The GPIO functionality that you have worked with so far is really just a set of memory locations, though they happen to be connected to logic gates that interface with the outside world. GPIO interaction still happens digitally though, with inputs and outputs limited to high and low signals.

Aside from all the human-made digital systems that exist, the world is primarily analog which means there must be a way to interface the two. Electrical and Computer engineers will inevitable become very familiar with Analog to Digital Converters (ADCs which we usually say "eh-dee-sees") and Digital to Analog Converters (DACs which we usually say as "dacks").

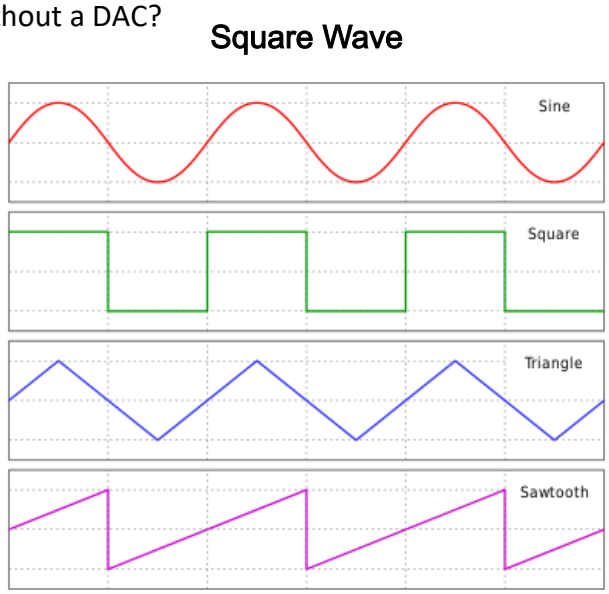
In most microcontrollers that have ADCs and/or DACs, both systems work by "tapping" a big series of fixed resistors that are built into the silicon of the microcontroller. Think of it like a potentiometer where the wiper moves in defined digital steps (yes, there are digital potentiometers than you can buy). The number of steps determines the resolution of the ADC or DAC. For example, an 8-bit DAC like in the PIC18F27Q43 has 256 discrete resistors. The resistor network is powered by a reference voltage of some sort on the high side, and either ground or another reference voltage on the low side. The simplest configuration is to use Vdd for the positive reference and Vss (ground) for the negative reference. The voltage between each discrete level (also called "steps" or "counts") is then (Vref+ - Vref-) / resolution.

Take a moment to calculate the DAC step size for the PIC18F27Q43 with Vdd = 5V and Vss = 0V. You should get 19.5mV per step. This means that the processor can output an analog voltage between 0 and 5V in 19.5mV steps. For many applications, this resolution is just fine. For other

$$(5v-0v)/256 = 19.53125mV$$

applications that require more accuracy, there are 10-bit, 12-bit, 16-bit and even 24-bit DACs (and ADCs) available, though usually these are external devices that you connect to the MCU.

The last piece of info you need to know are the basic functions (or waves) that you can get a microncontroller to output without a lot of effort. Which of these could you do right now with what you know even without a DAC?



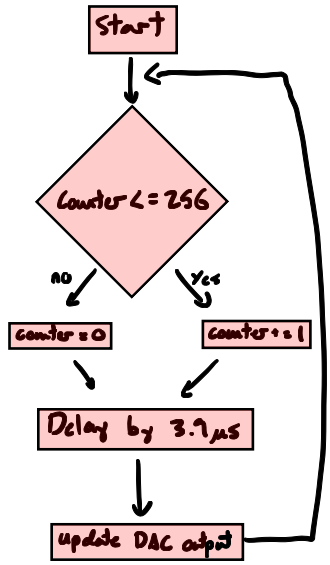
Source: https://en.wikipedia.org/wiki/Triangle_wave

The other three require a DAC to output voltage levels between Vdd and Vss. The Triangle and Sawtooth waves are straight forward as they are simply linear increases in the DAC output up to the peak value. What you should wrap your mind around right now is the timing. You have 256 DAC output levels available. **If you wanted a 1kHz Sawtooth wave, draw a simple flowchart for the program you would need to write.** Include this in your hand-in material. You can just write "Update DAC output" for the part where you adjust the output since you don't yet know the details on how to do that with the PIC. Make sure you use all 256 DAC levels and indicate the exact timing delay between adjustments of each output. Hint: What's the period of a 1kHz signal? Hopefully you can see that you need to update the DAC output faster than this.

Hardware Design

There is no new hardware on your PIC breadboard required for this activity. The only hardware challenge is finding which pins the DAC output is available on and then making sure we drive a suitable load. The DAC user guide suggests we are limited to RA2 or RB7 for the DAC output, but it's not totally clear if those are the only two pins available. A quick scan of the "Pin Allocation Table" on pages 15 and 16 of the user guide confirms that indeed these are the only two pins available for DAC output. **The table also confirms that the DAC in this processor is "DAC1" which is important to note.** Since we do not want to use the programming / debugging lines (RB6:7), that leaves us with RA2. You will have to commandeer one of your LEDs for the DAC output signal.

1kHz sawtooth wave:



$T = \frac{1}{f} = \frac{1}{1 \times 10^3} = 10^{-3}s = 1ms$

DAC update speed = $\frac{1 \times 10^{-3}}{256} = 3.9\mu s$

The other thing that you must know is that the current-driving capability of the DAC output is likely not going to be nearly as high as the digital output drivers. How do you know this aside from past experience with DACs? You need to look at the electrical characteristics in the user guide.

The DC characteristics on page 898 indicate the pins can sink 10mA and source 6 mA of current, but this is in DIGITAL mode (V_{OL} = Vout Low and V_{OH} = Vout High which are digital states). Great for driving LEDs but this has nothing to do with the DAC output capabilities.

Standard Operating Conditions (unless otherwise stated)							
Param. No.	Sym.	Device Characteristics	Min.	Typ.†	Max.	Units	Conditions
Output Low Voltage							
D360	V_{OL}	I/O PORTS	—	—	0.6	V	I_{OL} = 10.0 mA, V_{PIN} = 3.0V
Output High Voltage							
D370	V_{OH}	I/O PORTS	V_{DD} - 0.7	—	—	V	I_{OH} = 6.0 mA, V_{PIN} = 3.0V

When the DAC is operating, the figure on page 766 suggests that the output is unbuffered and is simply the series combination of the resistor ladder. If we hunt in the user guide for the DAC specs, we see table 47.4.10 on page 909 that tells us each resistor in the ladder is 20k. Although it does not give us the current-drive capability directly, we are left to simply infer it from the resistor information. Even without doing any math, we can assume the current drive capability is essentially nothing. 256 x 20k is over 5 MOhm, so any sort of load would significantly impact the DAC output. This INCLUDES a scope probe which is typically around 1 MOhm. So unless we're missing something in the spec, the only option is to buffer the output (i.e. into an op-amp). **Note:** it appears the scope probe does NOT load the output, which suggests there might be something different in the hardware than what can be gathered from the datasheet. The user guide is still in "Preliminary" release so this may be updated in the future.

47.4.10 8-Bit DAC Specifications

Table 47-16. 8-bit Digital-to-Analog Converter

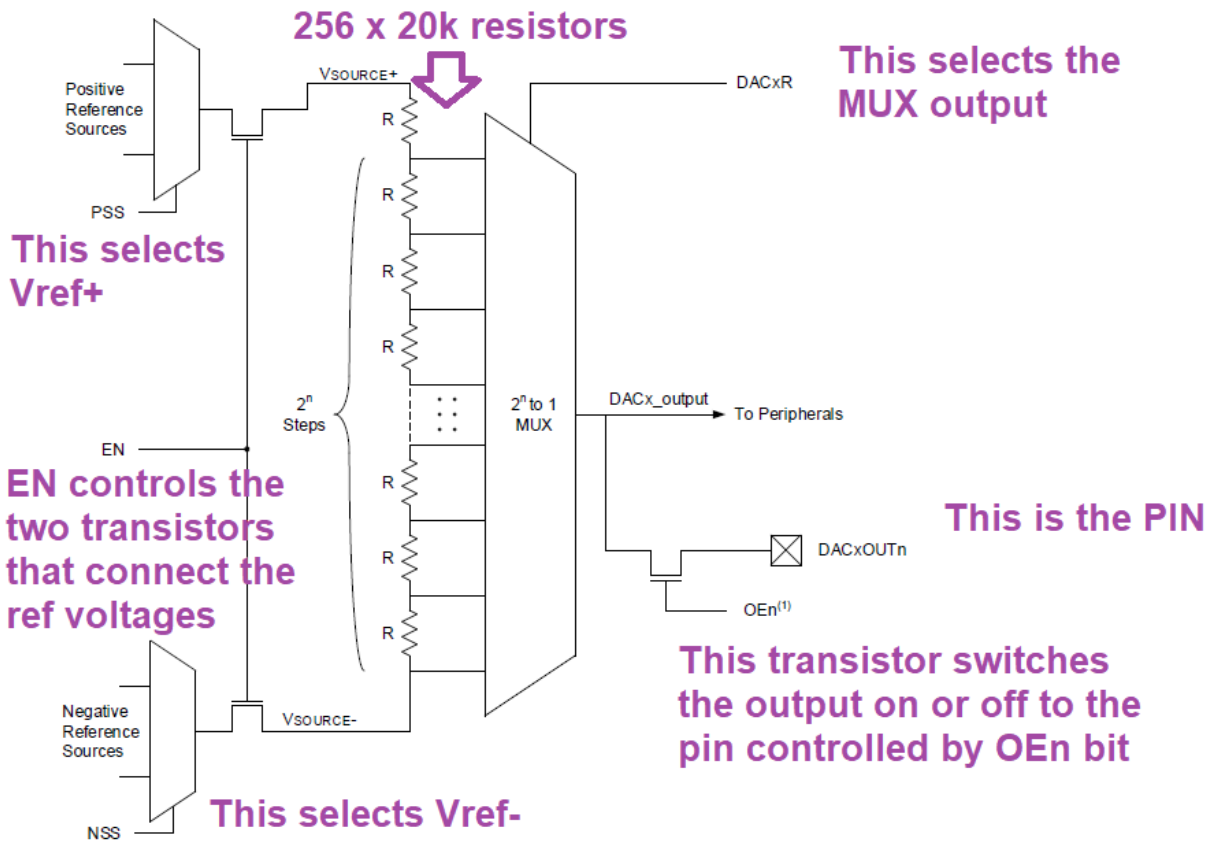
Standard Operating Conditions (unless otherwise stated) V_{DD} = 3.0V, T_A = 25°C						
Param No.	Sym.	Characteristic	Min.	Typ. †	Max.	Units
DSB01	V_{LSB}	Step Size	—	($V_{DACREF+}$ - $V_{DACREF-}$)/256	—	V
DSB02	V_{ACC}	Absolute Accuracy	—	—	±0.5	LSb
DSB03*	R_{UNIT}	Unit Resistor Value	—	20	—	kΩ
DSB04*	T_{ST}	Settling Time(1)	—	10	—	µs
DSB05*	V_{DBO}	DAC Buffer Offset	—	20	45	mV

All that said, simply pull out the LED that is connected to RA2 and plug a Scopy probe in to RA2 to monitor the output signal. **Do not try to drive the ENEL 343 audio amplifier circuit at this time** because we will be stopping the code and testing out some other things that could damage the speaker if it's connected.

DAC Module

Learning how the DAC on the PIC microcontroller works starts in exactly the same way as learning how Timer0 worked: find the section in the user guide and read it to learn what you need to do. In the current datasheet this is section 41 starting on page 766. You'll see there is hardly anything to learn. Some of the most important information comes from the figure 41-1 that shows the hardware. With your knowledge of basic circuits (aka resistor dividers) and basic logic from ENEL 353, you should be able to understand this. It's annotated below to help.

Figure 41-1. Digital-to-Analog Converter Block Diagram



The operation of the DAC is otherwise simple. We want the DAC enabled, RA2 output, Vdd positive reference, and Vss as negative reference. Write down the corresponding configuration value for DACxCON (and what is 'x'?).

DAC1CON: x=1

DAC1CON = 1010 0000

= 0xA0

Programming a Triangle Wave

Branch your PIC_Activity5 to PIC_Activity6. Delete all of the code in UserAppRun(). Make sure RA2 is floating (remove either the LED or resistor or both from that pin).

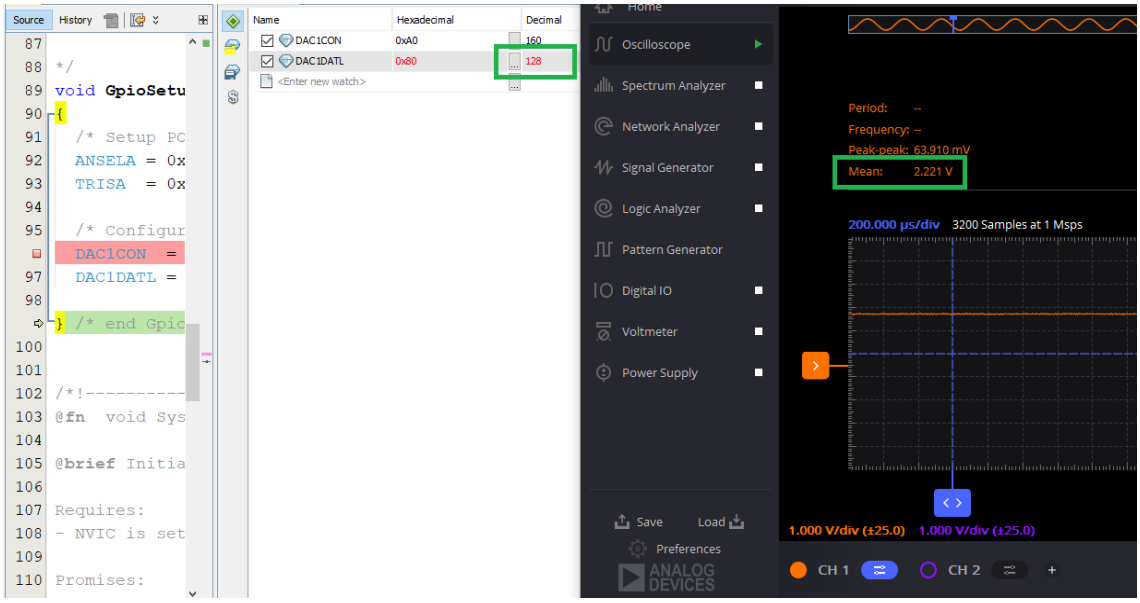
Since the hardware configuration is fixed, add the DAC1CON configuration to GpioSetup() along with a comment about what you're setting.

```
void GpioSetup(void)
{
  /* Setup PORTA for all digital output */
  ANSELA = 0x00;
  TRISA = 0x00;

  /* Configure DAC1 for Vdd and Vss references, on, and RA2 output. */
  DAC1CON = ;

} /* end GpioSetup() */
```

Adjust the system tick time to the required number of microseconds you need to update the DAC output to achieve close to 1kHz. Hint 1: round up and you'll be a little less than 1kHz but close enough. Hint 2: don't forget you need to go up and then back down in one period of 1kHz. What is the actual frequency of the signal you're going to get? You might also start to get an appreciation of timing in an embedded system when it comes to things like making audio…

In UserAppRun(), write code to update the DAC output register. For this part of the activity, you may directly increment or decrement the output register instead of trying to keep a separate variable to load.

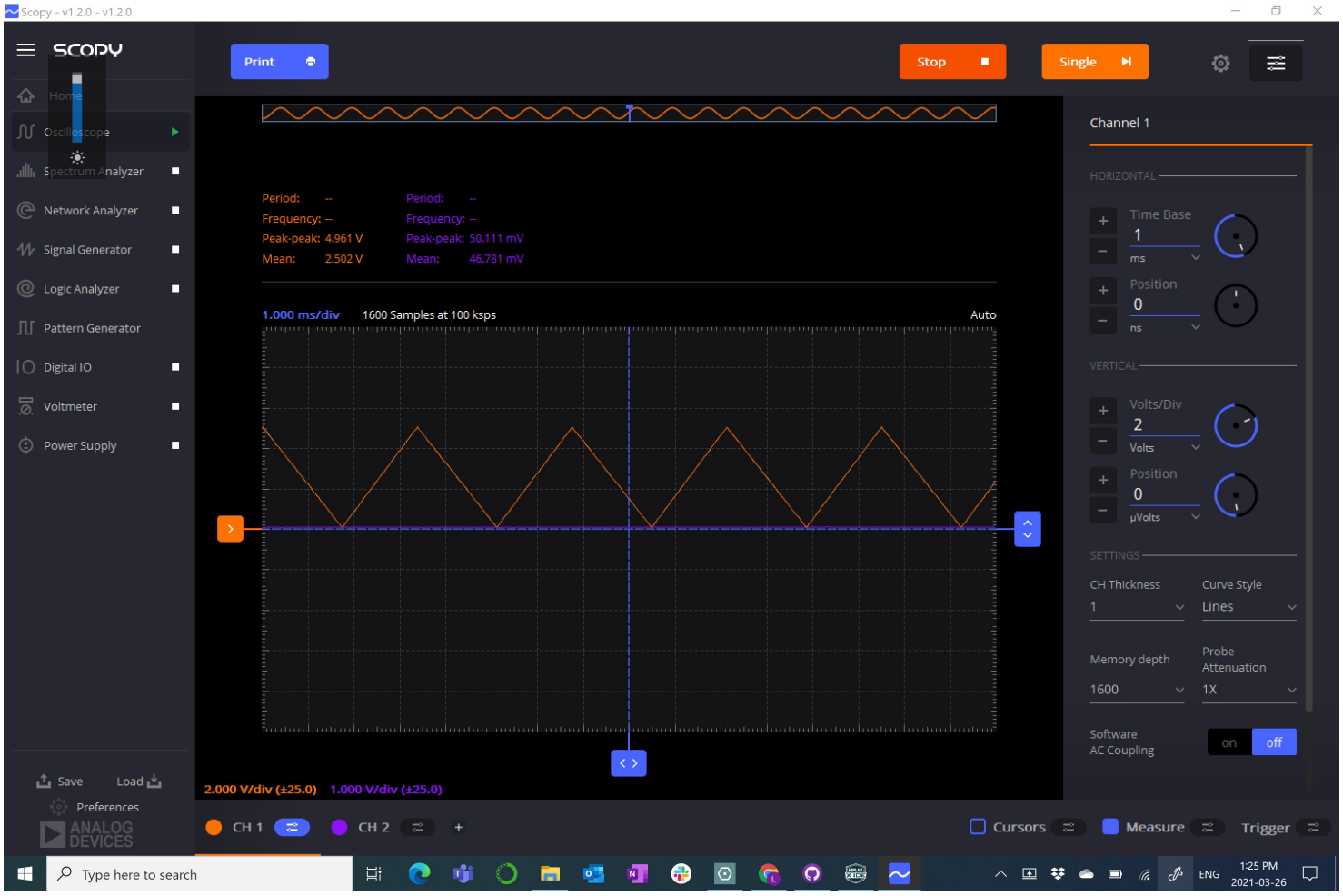
Run the code to test. It's suggested to put a breakpoint in GpioSetup() after DAC1CON is initialized and then manually change DAC1DATL to various values to see the response in Scopy. This way you know the DAC is on and Scopy is measuring correctly. In the example shown below, DAC1DATL is half of full scale, and we see the scope trace at 2.22 which is indeed half of Vdd (which is 4.4V running off the PICKIT).

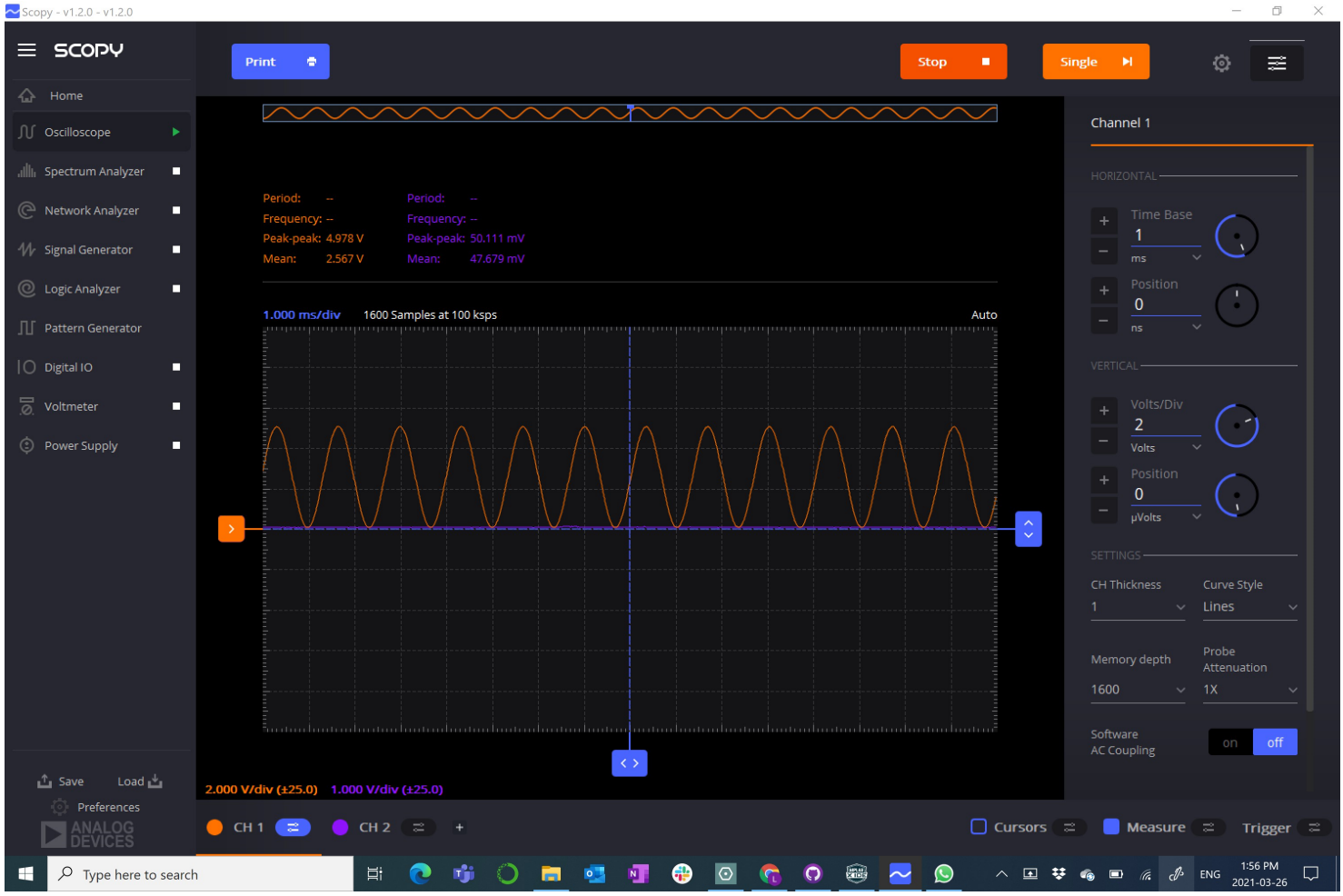


When you have verified that, let the program run and verify the output.

***** WARNING: THIS IS NOT GOING TO WORK AS EXPECTED!!! *****

You should notice that the frequency of the triangle wave is much less than the 1kHz you are aiming for. Even if you comment out the call to TimeXus(), the triangle wave frequency is low. Why?


Scopy - v1.2.0 - v1.2.0
SCOPY
Print
Stop
Single
Home
Oscilloscope
Spectrum Analyzer
Network Analyzer
Signal Generator
Logic Analyzer
Pattern Generator
Digital IO
Voltmeter
Power Supply
Period: --
Frequency: --
Peak-peak: 4.961 V
Mean: 2.502 V
Period: --
Frequency: --
Peak-peak: 50.111 mV
Mean: 46.781 mV
1.000 ms/div
1600 Samples at 100 ksps
Auto
2.000 V/div (±25.0)
1.000 V/div (±25.0)
Channel 1
HORIZONTAL
Time Base
1
ms
Position
0
ns
VERTICAL
Volts/Div
2
Volts
Position
0
µVolts
SETTINGS
CH Thickness
1
Curve Style
Lines
Memory depth
1600
Probe Attenuation
1X
Software AC Coupling
on
off
Save
Load
Preferences
CH 1
CH 2
Cursors
Measure
Trigger
Type here to search
ENG
1:25 PM
2021-03-26

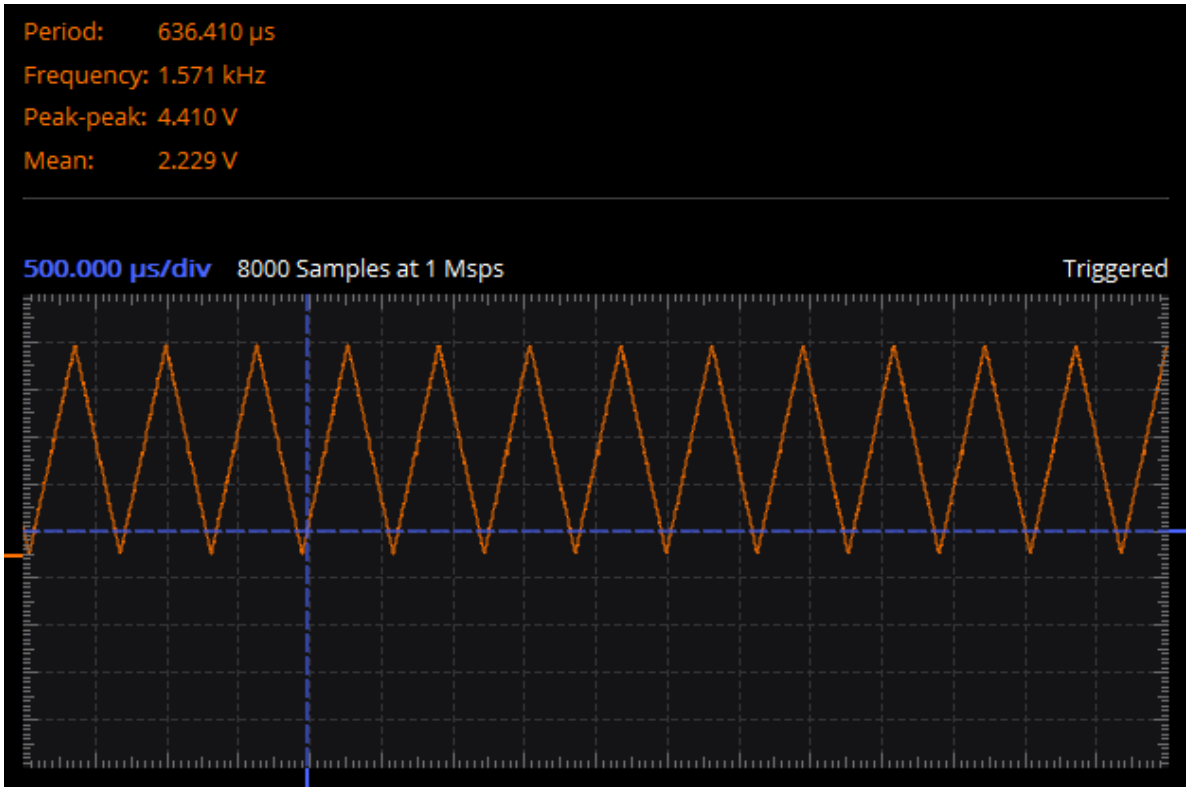

Scopy - v1.2.0 - v1.2.0
SCOPY
Print
Stop
Single
Home
Oscilloscope
Spectrum Analyzer
Network Analyzer
Signal Generator
Logic Analyzer
Pattern Generator
Digital IO
Voltmeter
Power Supply
Period: --
Frequency: --
Peak-peak: 4.978 V
Mean: 2.567 V
Period: --
Frequency: --
Peak-peak: 50.111 mV
Mean: 47.679 mV
1.000 ms/div
1600 Samples at 100 ksps
Auto
2.000 V/div (±25.0)
1.000 V/div (±25.0)
Channel 1
HORIZONTAL
Time Base
1
ms
Position
0
ns
VERTICAL
Volts/Div
2
Volts
Position
0
µVolts
SETTINGS
CH Thickness
1
Curve Style
Lines
Memory depth
1600
Probe Attenuation
1X
Software AC Coupling
on
off
Save
Load
Preferences
CH 1
CH 2
Cursors
Measure
Trigger
Type here to search
ENG
1:56 PM
2021-03-26

Capture your triangle wave and include the screenshot in your hand-in even though it's not the correct frequency.

Try commenting out (using an #if 0 and #endif structure) all of the code in the main while loop except the call to UserAppRun().

```
52    while(1)
53    {
54      /* Drivers */
55
56      /* Applications */
57      UserAppRun();
58
59  #if 0
60      /* System sleep */
61      HEARTBEAT_OFF();
62      SystemSleep();
63
64      /* Set the timer and wait out the period */
65      TimeXus(2);
66      while( PIR3bits.TMR0IF == 0 );
67
68      HEARTBEAT_ON();
69  #endif
70
71    } /* end while(1) main super loop */
```

Run the program again and see the frequency of the triangle wave.

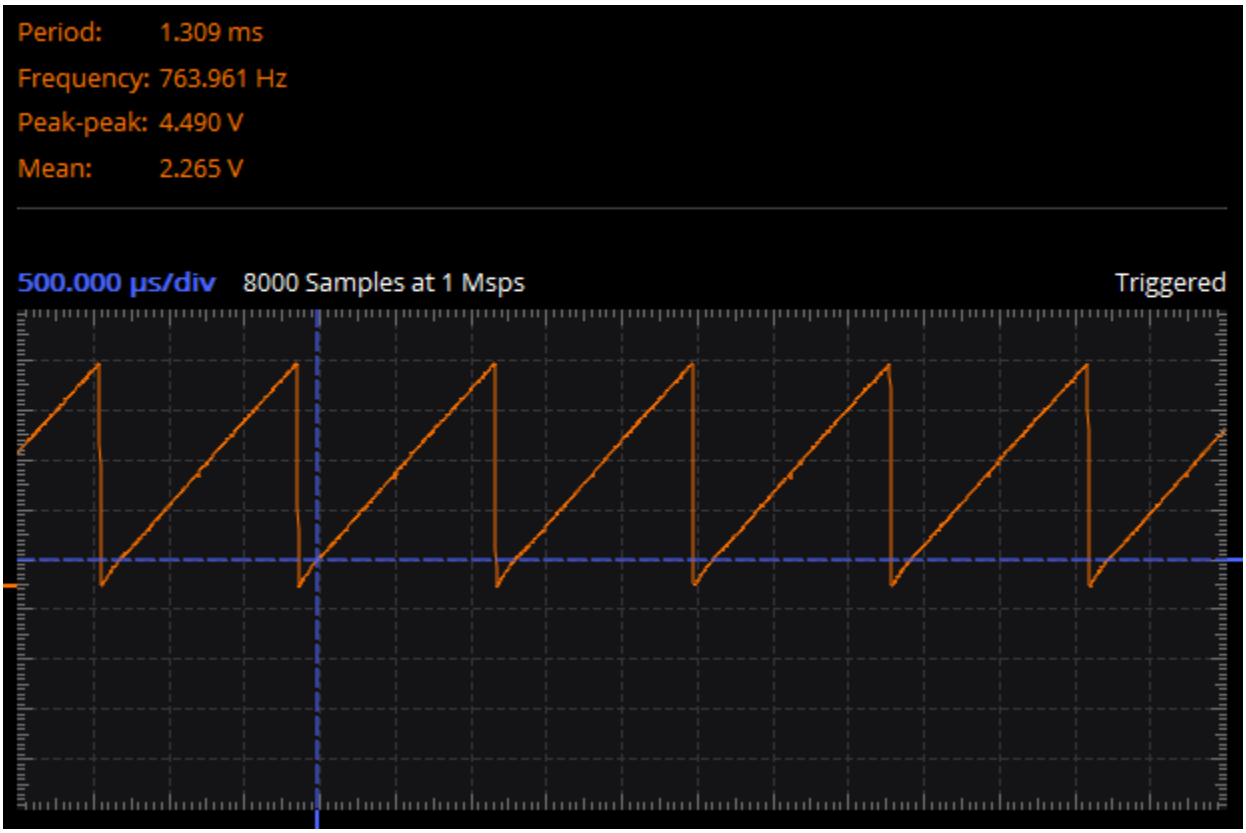


In our system, we're still only at 1.5kHz, which means the lines of code required to call UserAppRun(), check if it's incrementing or decrementing, and writing DAC1DATL still takes a significant amount of processing cycles.

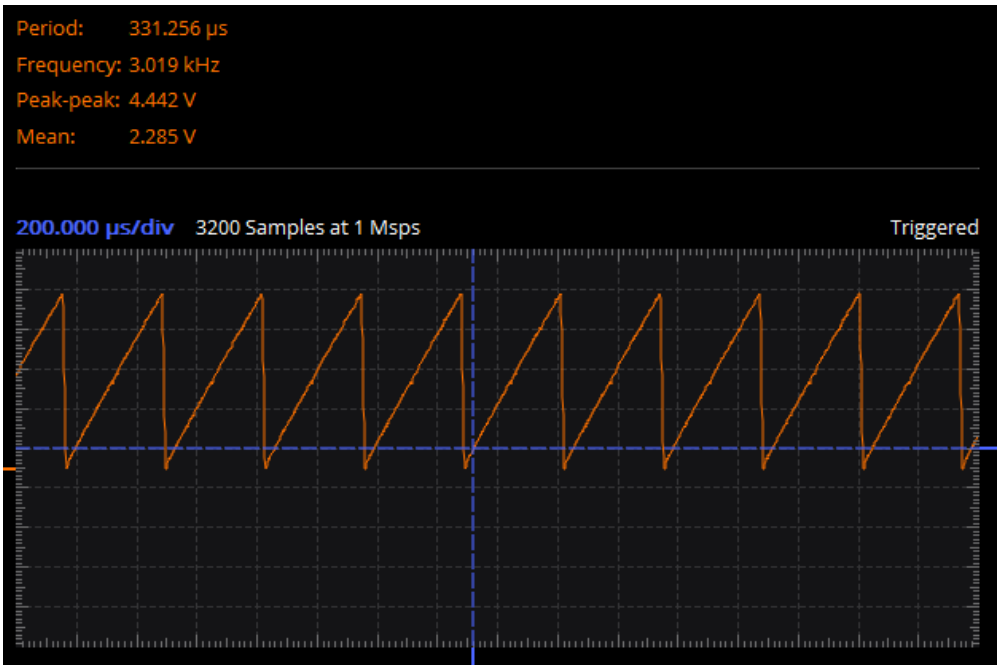
To achieve the best performance as possible, let's change the plan a little and produce a sawtooth wave instead. Do this directly in the main loop and remove all other code in the main loop (including the HEARTBEAT macros). The only thing you should do is increment DAC1DATL, call TimeXus(), and wait out the period by checking PIR3.

```
56 #if 1 /* Trying to run as fast as possible */
57     /* Set the timer and wait out the period */
58     TimeXus(2);
59     while( PIR3bits.TMR0IF == 0 );
60     DAC1DATL += 1;
61 #endif
```

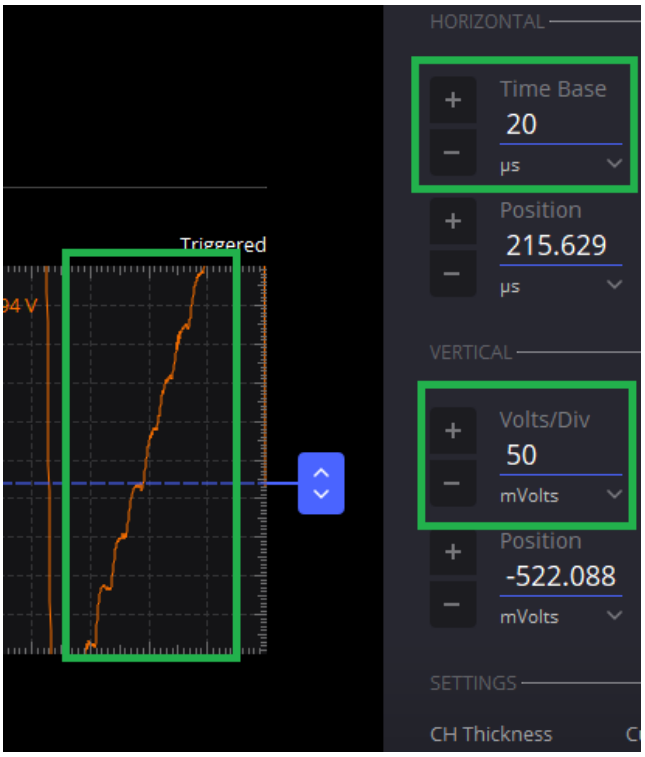
Your code should be pretty close to the code running to generate the examples here and thus your timing should be quite similar. With this minimalized approach, we can get as fast as 764 Hz.

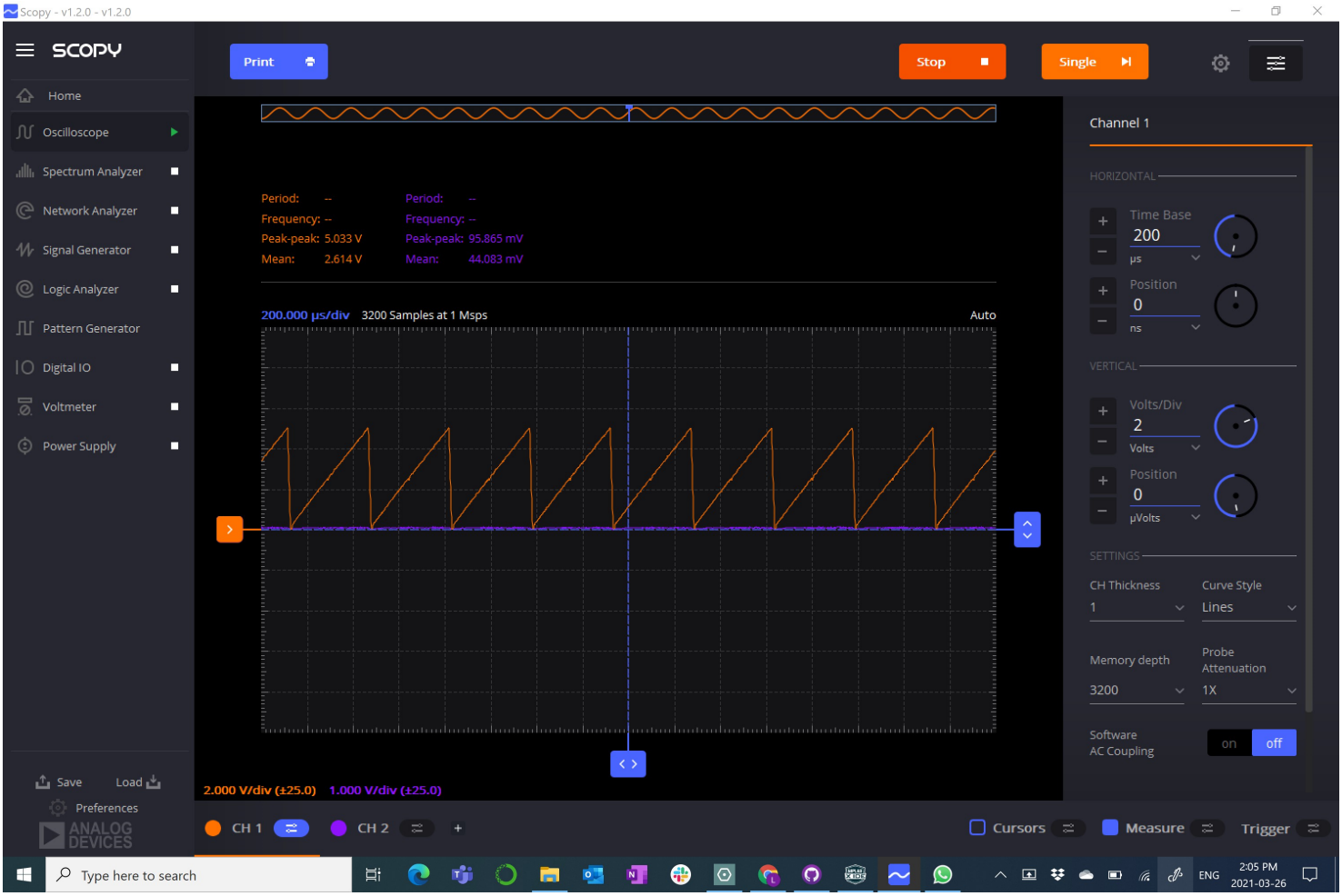


The last thing to try is to reduce the number of steps. Instead of 256 steps, use 64 steps. In other words, increment DAC1DATL by 4 instead of 1.



You should get slightly more than a 4x increase in speed because there is overall less overhead between each step. **Include a screenshot of your signal.** The signal still looks like a sawtooth wave, though if you zoom in you can see the discrete steps from the DAC output more clearly.




Scopy - v1.2.0 - v1.2.0
SCOPY
Print
Stop
Single
Home
Oscilloscope
Spectrum Analyzer
Network Analyzer
Signal Generator
Logic Analyzer
Pattern Generator
Digital IO
Voltmeter
Power Supply
Save
Load
Preferences
ANALOG DEVICES
Period: --
Frequency: --
Peak-peak: 5.033 V
Mean: 2.614 V
Period: --
Frequency: --
Peak-peak: 95.865 mV
Mean: 44.083 mV
200.000 µs/div
3200 Samples at 1 Msps
Auto
2.000 V/div (±25.0)
1.000 V/div (±25.0)
Channel 1
HORIZONTAL
Time Base
200
µs
Position
0
ns
VERTICAL
Volts/Div
2
Volts
Position
0
µVolts
SETTINGS
CH Thickness
1
Curve Style
Lines
Memory depth
3200
Probe Attenuation
1X
Software AC Coupling
on
off
CH 1
CH 2
Cursors
Measure
Trigger
Type here to search
ENG
2:05 PM
2021-03-26

The point of all of this is a cautionary tale about timing in an embedded system. When the operations you are trying to perform are on the same order of magnitude of speed as the processor clock itself, then unexpected behaviour can happen. There are ways around this, which again we will wait for the next activity to address.

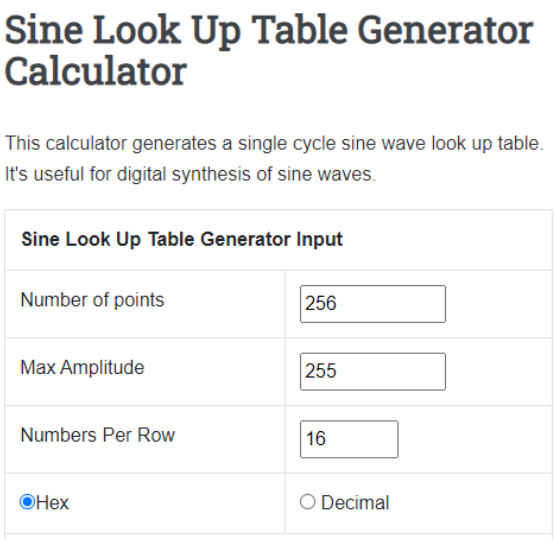
Outputting a Sinusoid

Aside from the timing issues, a triangle wave is fairly easy to make. However, if you drive a speaker with it, the sound won't be that great due to the harmonic frequencies produced by the sharp peaks. See the audio amplifier hands-on activity in ENEL 343 for more details! So instead, we want to create a sinusoidal output. Integer microcontrollers do have the capability to do trigonometric operations, but this is purely a matter of library firmware functions that are WAY too big and WAY too slow to use for real time audio.

Instead, we'll hard-code a look-up table for the DAC output values. A quick Google search of "sinusoid digital table" produces this handy website:

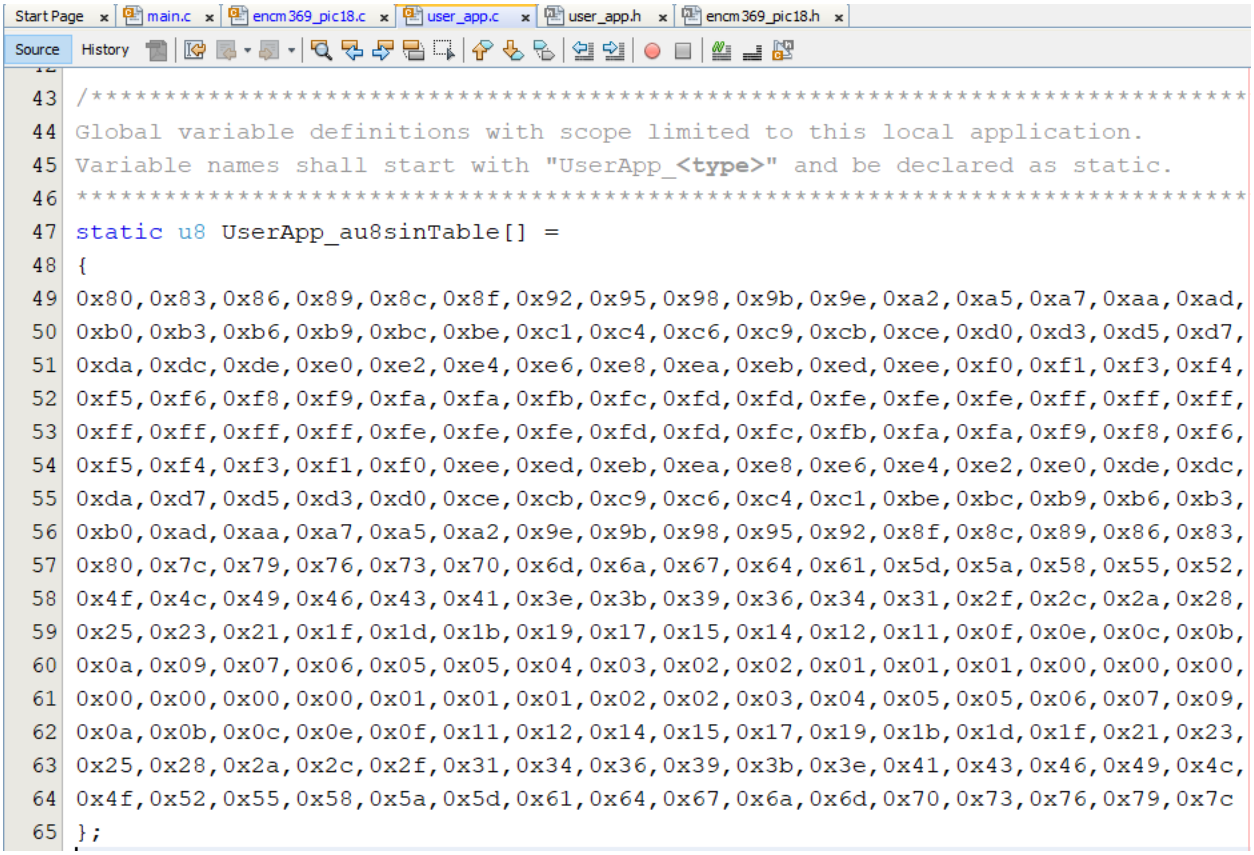
https://www.daycounter.com/Calculators/Sine-Generator-Calculator.phtml

Think about the settings that you need to enter. To save a lot of questions, here is what you should use – just make sure you understand why.



We'll stick with 256 points even though we have already shown that the method of updating DAC1DATL is too slow to produce frequencies above a few hundred hertz, but that's okay. Generate the table and paste it as the initialization values for a Global array variable accessible only to UserApp in user_app.c. Note the naming convention for this type of variable. The image below is what you should end up with. Note that leading zeros for the lowest numbers were added in manually so all the columns line up. This makes it much easier to read and makes it easy to visually confirm it is sized correctly (and it only took about a minute to do).

FYI, it's little things like this that differentiate good programmers from great programmers, so you can choose which you'd prefer to advertise yourself as.




```
/**********************************************************************************
Global variable definitions with scope limited to this local application.
Variable names shall start with "UserApp_<type>" and be declared as static.
**********************************************************************************/
static u8 UserApp_au8sinTable[] =
{
0x80,0x83,0x86,0x89,0x8c,0x8f,0x92,0x95,0x98,0x9b,0x9e,0xa2,0xa5,0xa7,0xaa,0xad,
0xb0,0xb3,0xb6,0xb9,0xbc,0xbe,0xc1,0xc4,0xc6,0xc9,0xcb,0xce,0xd0,0xd3,0xd5,0xd7,
0xda,0xdc,0xde,0xe0,0xe2,0xe4,0xe6,0xe8,0xea,0xeb,0xed,0xee,0xf0,0xf1,0xf3,0xf4,
0xf5,0xf6,0xf8,0xf9,0xfa,0xfa,0xfb,0xfc,0xfd,0xfd,0xfe,0xfe,0xfe,0xff,0xff,0xff,
0xff,0xff,0xff,0xff,0xfe,0xfe,0xfe,0xfd,0xfd,0xfc,0xfb,0xfa,0xfa,0xf9,0xf8,0xf6,
0xf5,0xf4,0xf3,0xf1,0xf0,0xee,0xed,0xeb,0xea,0xe8,0xe6,0xe4,0xe2,0xe0,0xde,0xdc,
0xda,0xd7,0xd5,0xd3,0xd0,0xce,0xcb,0xc9,0xc6,0xc4,0xc1,0xbe,0xbc,0xb9,0xb6,0xb3,
0xb0,0xad,0xaa,0xa7,0xa5,0xa2,0x9e,0x9b,0x98,0x95,0x92,0x8f,0x8c,0x89,0x86,0x83,
0x80,0x7c,0x79,0x76,0x73,0x70,0x6d,0x6a,0x67,0x64,0x61,0x5d,0x5a,0x58,0x55,0x52,
0x4f,0x4c,0x49,0x46,0x43,0x41,0x3e,0x3b,0x39,0x36,0x34,0x31,0x2f,0x2c,0x2a,0x28,
0x25,0x23,0x21,0x1f,0x1d,0x1b,0x19,0x17,0x15,0x14,0x12,0x11,0x0f,0x0e,0x0c,0x0b,
0x0a,0x09,0x07,0x06,0x05,0x05,0x04,0x03,0x02,0x02,0x01,0x01,0x01,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x01,0x01,0x01,0x02,0x02,0x03,0x04,0x05,0x05,0x06,0x07,0x09,
0x0a,0x0b,0x0c,0x0e,0x0f,0x11,0x12,0x14,0x15,0x17,0x19,0x1b,0x1d,0x1f,0x21,0x23,
0x25,0x28,0x2a,0x2c,0x2f,0x31,0x34,0x36,0x39,0x3b,0x3e,0x41,0x43,0x46,0x49,0x4c,
0x4f,0x52,0x55,0x58,0x5a,0x5d,0x61,0x64,0x67,0x6a,0x6d,0x70,0x73,0x76,0x79,0x7c
};
```

Now all you have to do is change your UserAppRun() code to index this huge array instead of just incrementing or decrementing the DAC output that you did for the triangle wave. Write the code and use the debugger to make sure you are correctly loading the DAC register. Notice that the first value directly follows the last value in the array, so make sure you seamlessly wrap around. It is fine to let your array index simply rollover back to 0, but be sure to put a comment in that says this is what you're intending.

Here's a trick question and a hint about accessing the table values: which is faster? Which is more clear? In the first line, does au8Array get indexed BEFORE or AFTER u8Index increments?

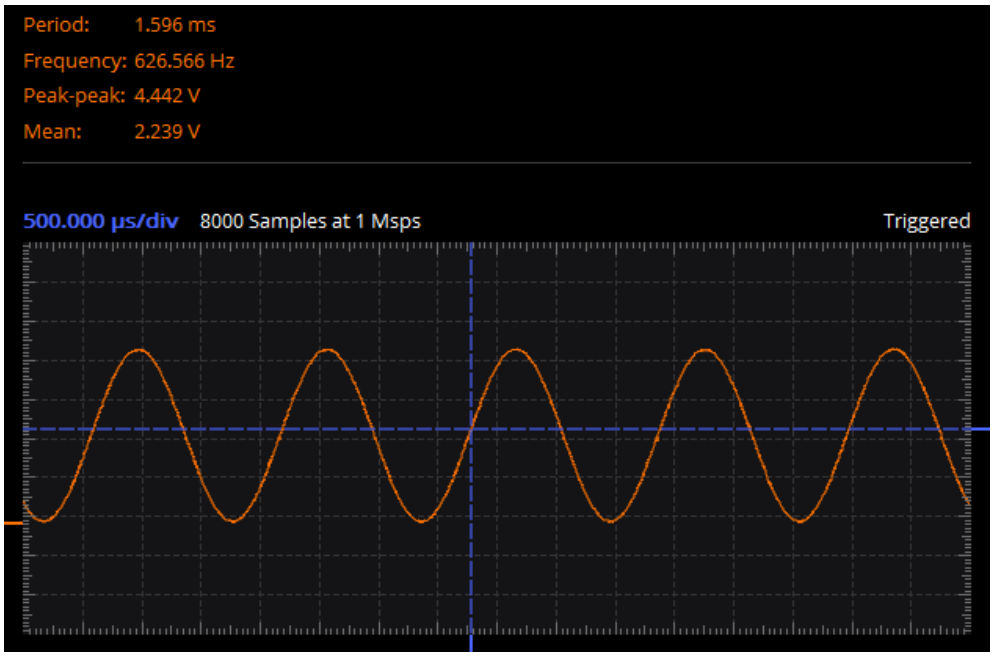
```
u8Foo = au8Array[u8Index++];
```

or

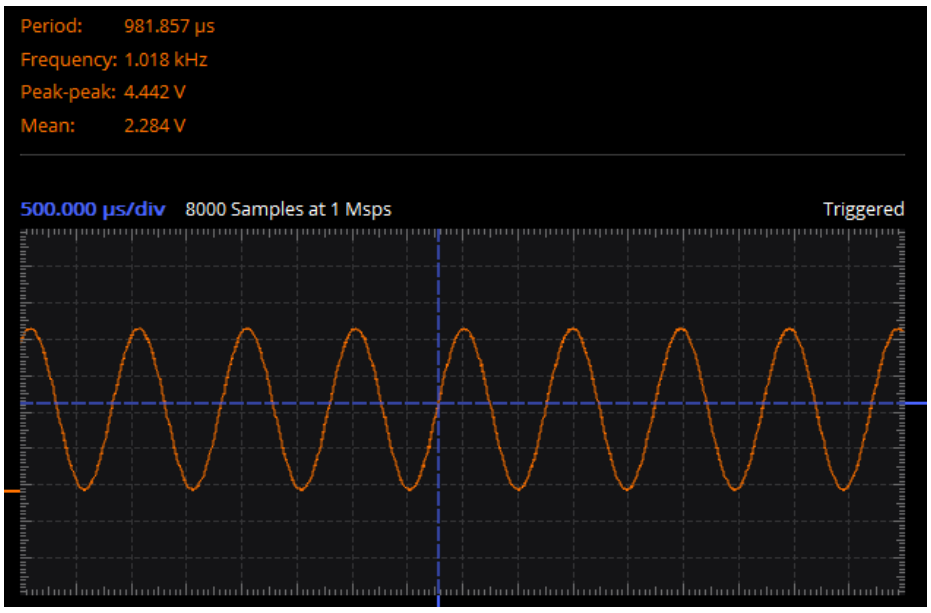
```
u8Foo = au8Array[u8Index];
u8Index++;
```

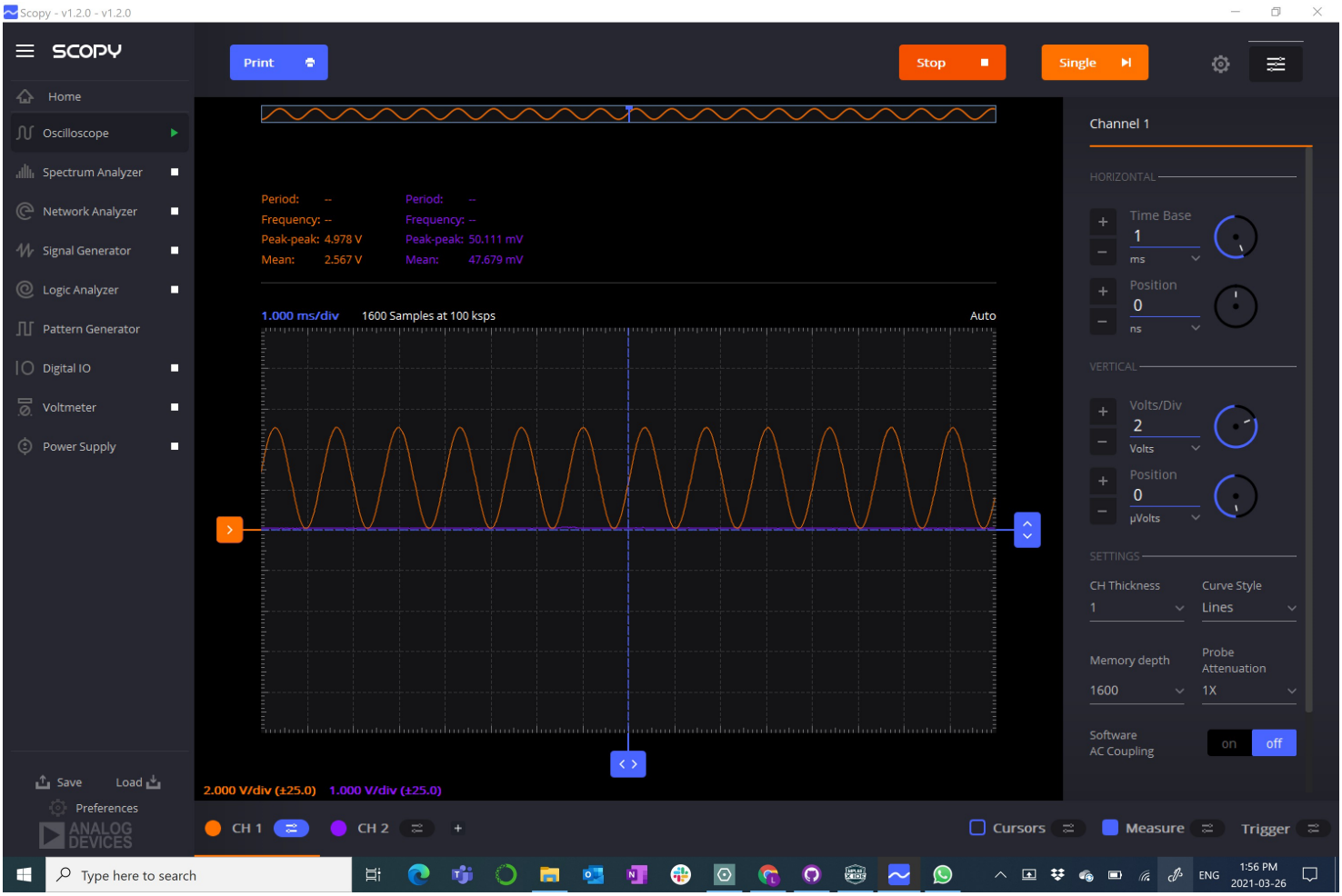
Keep your code short and concise as possible. In fact, it should be faster than the triangle wave. Don't forget to comment the code back in the main loop (just change #if 0 to #if 1).

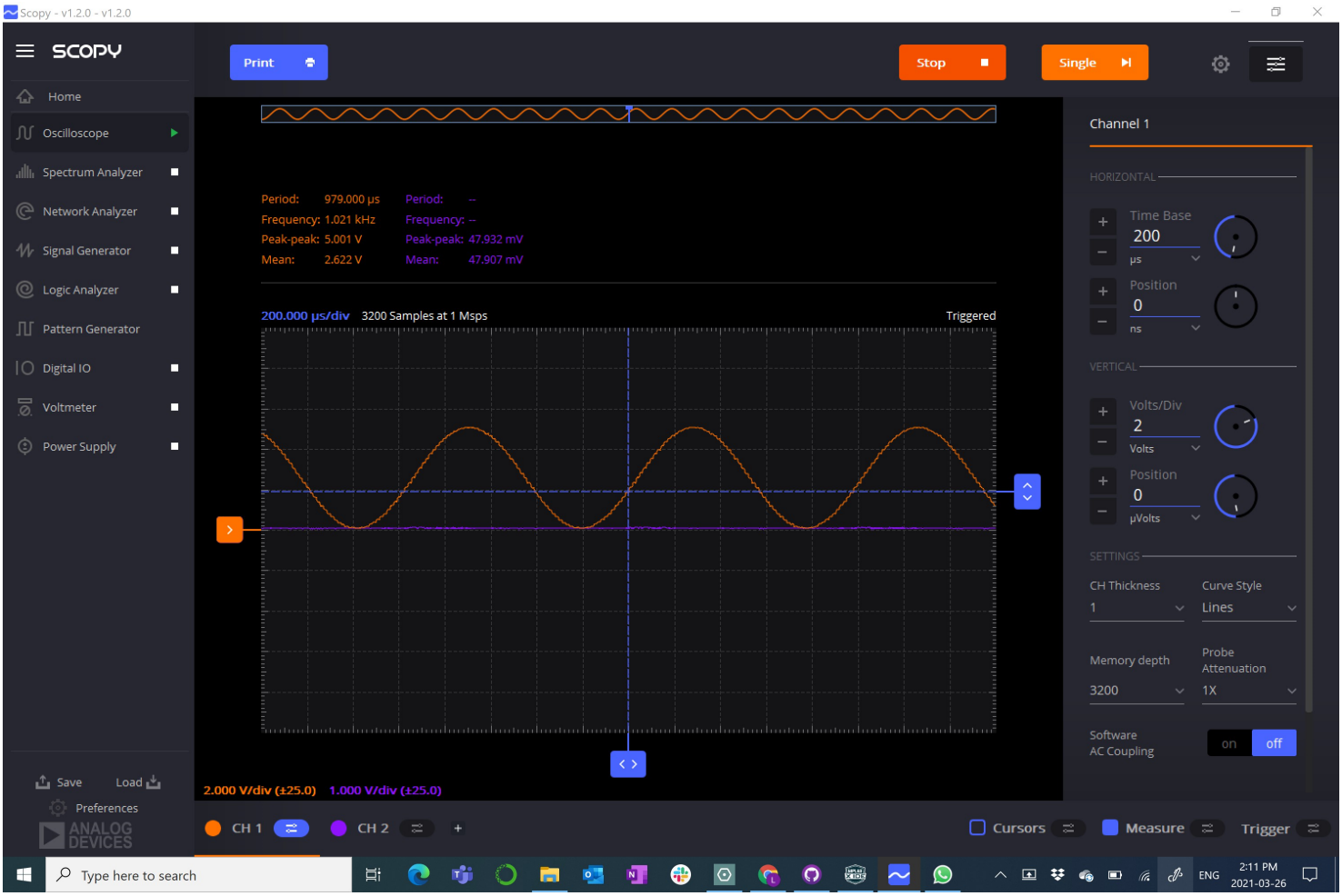
Depending on how tight your code is, you should get a sinusoidal output with a decently high frequency. **Include a screen shot of your sinusoid.**



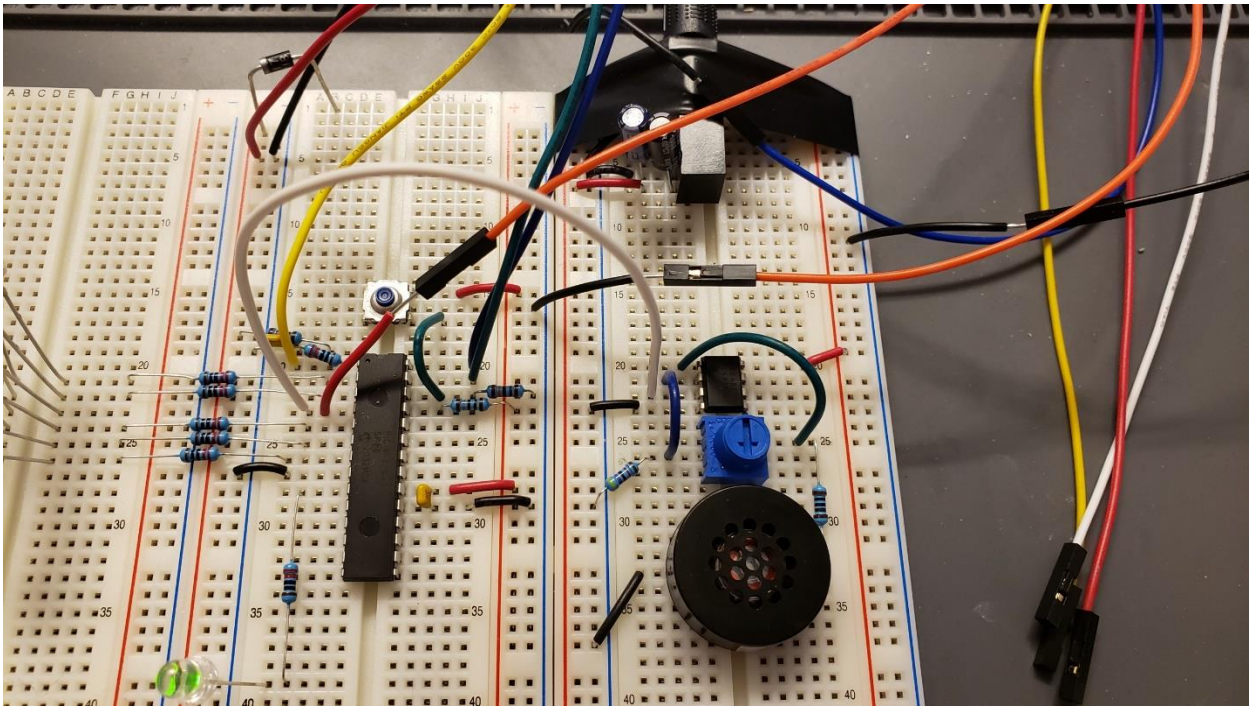
Try using only 64 values from the table (i.e. increment the index by 4). Now the difference in speed at which the code needs to runs and the number of overhead instructions is starting to be significant, so with a little tuning of the value used in TimeXus(), the output can be tweaked to yield very close to 1kHz. **Include a screen shot of your sinusoid close to 1kHz.** Everyone should calculate the required delay to be 15.6us but the actual value for TimeXus() will depend on the rest of the code. In the case of the example the actual value used was 11 to yield this result:




Scopy - v1.2.0 - v1.2.0
SCOPY
Print
Stop
Single
Home
Oscilloscope
Spectrum Analyzer
Network Analyzer
Signal Generator
Logic Analyzer
Pattern Generator
Digital IO
Voltmeter
Power Supply
Period: --
Frequency: --
Peak-peak: 4.978 V
Mean: 2.567 V
Period: --
Frequency: --
Peak-peak: 50.111 mV
Mean: 47.679 mV
1.000 ms/div
1600 Samples at 100 ksps
Auto
2.000 V/div (±25.0)
1.000 V/div (±25.0)
Channel 1
HORIZONTAL
Time Base
1
ms
Position
0
ns
VERTICAL
Volts/Div
2
Volts
Position
0
µVolts
SETTINGS
CH Thickness
1
Curve Style
Lines
Memory depth
1600
Probe Attenuation
1X
Software AC Coupling
on
off
Save
Load
Preferences
CH 1
CH 2
Cursors
Measure
Trigger
Type here to search
ENG
1:56 PM
2021-03-26


Scopy - v1.2.0 - v1.2.0
SCOPY
Print
Stop
Single
Home
Oscilloscope
Spectrum Analyzer
Network Analyzer
Signal Generator
Logic Analyzer
Pattern Generator
Digital IO
Voltmeter
Power Supply
Period: 979.000 µs
Frequency: 1.021 kHz
Peak-peak: 5.001 V
Mean: 2.622 V
Period: --
Frequency: --
Peak-peak: 47.932 mV
Mean: 47.907 mV
200.000 µs/div
3200 Samples at 1 Msps
Triggered
2.000 V/div (±25.0)
1.000 V/div (±25.0)
Channel 1
HORIZONTAL
Time Base
200
µs
Position
0
ns
VERTICAL
Volts/Div
2
Volts
Position
0
µVolts
SETTINGS
CH Thickness
1
Curve Style
Lines
Memory depth
3200
Probe Attenuation
1X
Software AC Coupling
on
off
Save
Load
Preferences
CH 1
CH 2
Cursors
Measure
Trigger
Type here to search
ENG
2:11 PM
2021-03-26

If you have an audio amplifier circuit built, you can safely jumper this over to the op-amp input and listen to the tone. Be careful about power supply voltage… You need a common ground between the PIC circuit and the op-amp circuit, but you don't need the same input voltage (you can run the op-amp from your 5V supply, and the PIC circuit from the PICKIT if you want (this is how the setup below is hooked up, though the common ground is out of frame at the bottom).



Make sure you set the amplifier gain to 1 since the DAC is outputting full scale and you don't want clipping. You can compare the DAC tone from the PIC to the function generator of the ADALM2000 and see if you hear a difference (we don't know the details of the ADALM2000 function generator, but it is likely a high-resolution DAC generating the signal). If it were an absolutely pure analog sinusoid, it would sound much cleaner. It actually sounds quite good, though!

What to Hand In / Review

Save your design documentation and screen shots in a single document in your source code folder and commit it and your code. Don't forget to push it to Github. Perform a code review on someone's code from your learning community who you have not already done a code review for. Upload the code review spreadsheet to the D2L Dropbox for the assignment and be sure to email the person whom you reviewed a copy.

More Things To try

OPTIONAL: this is not part of the assignment, but take a look and see if you can think about how you would do the following. Of course is you have extra time, take a stab at coding these things.

1. Code a sequence of several different tone frequencies that are cycled through every few seconds.
2. Change the tone cycling to work off an array of frequencies.
3. Figure out the frequencies to put in your array to play "Twinkle, Twinkle Little Star." You might not have enough resolution in TimeXus() to achieve this. You could modify TimeXus to reduce the prescaler and thus give you more options, or just wait until next activity where we will have much better control over small time increments.

In all these cases, make sure you don't add too much overhead code that will show up as distortion in the DAC output. Keep Scopy on RA2 to make sure there are no visible signs of disruption in the DAC output, though it will be difficult to see once you start changing the tones.