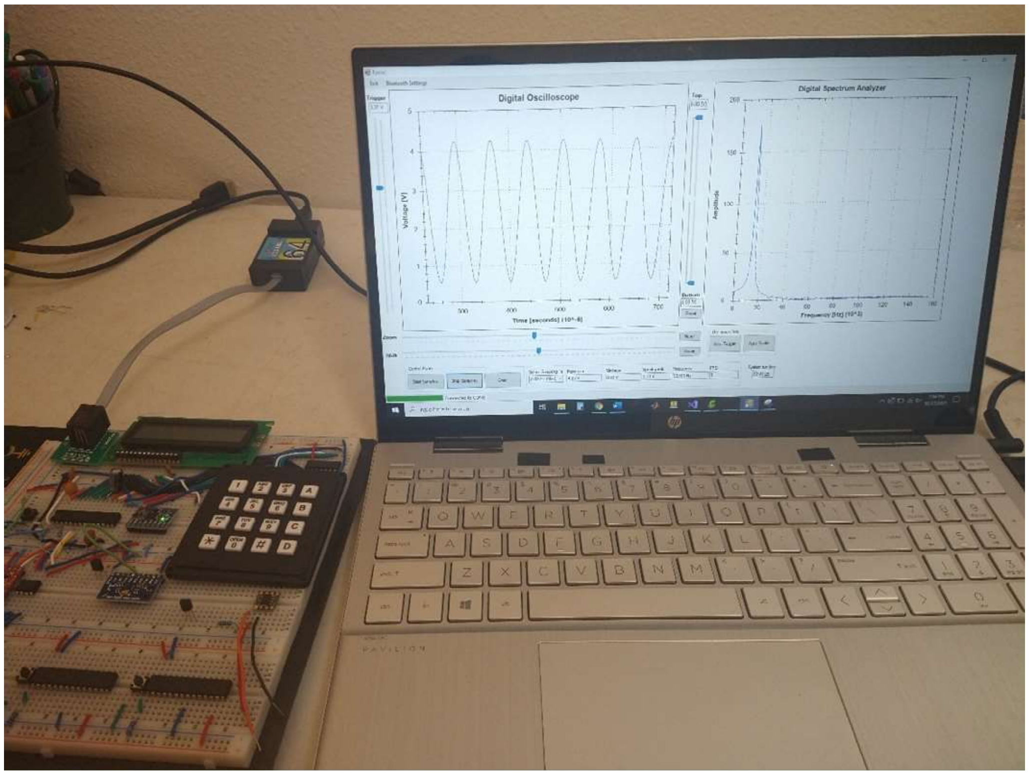
Wireless Digital Oscilloscope and Spectrum Analyzer

ECE 522 Midterm Project
October 27, 2021
Siong Moua



Saint Cloud State University

Electrical Engineering Department

Table of Contents

Objective

Design a digital oscilloscope using PIC24FV16KM202 as the sampling module and Visual Studio Window Form .Net Framework as the control and display module.

Design Specification

1. Use 8-bit ADC on the PIC24 to sample an input signal
2. Input signal will be a sinusoidal wave in the range of 100 Hz to 10 kHz
3. C# GUI should have at least these three buttons:
   ‘Start’ Sampling’
   ‘Stop Sampling’
   ‘Clear’
4. A combo box allowing user to select different sampling frequencies such as;
   - 100 kHz
   - 10 kHz
   - 1 kHz
   - 100 Hz
5. C# GUI needs to have a trigger method (track bar).
6. C# GUI should allow user to select different magnitude span of the oscilloscope
7. Horizontal and vertical axes of the oscilloscope must be properly labeled.
8. C# GUI should at least have four text boxes showing the following information:
   - Peak to peak voltage
   - Maximum voltage
   - Minimum voltage
   - Frequency of input signal
9. C# GUI must be able to display frequency of the sampled signal
10. C# GUI must have a reasonable display rate such that there is not much of delay (6 to 10 frame per second is good enough)
11. Can also add any other features to the designed oscilloscope to mimic a normal commercial oscilloscope.

Procedure

Note that not all code are provided, but only the key features of the code. These codes can be found in the Appendix section of this document.

In this document the term ‘C# GUI application’ and ‘C# oscilloscope’ are used interchangeably.

1. Wire the circuitry of the PIC24FV16KM202 as shown in Figure 1 on a breadboard.
2. Open CCS C and create a project from scratch
3. Note that we are using the ‘Project 24 Bit Wizard’ and our clock is ‘internal’ with frequency of 32 MHz
4. Open Visual Studio and create a new project with Window Form App (.NET Framework).

5. The link below will show how to create a general C# GUI application to establish a Bluetooth connection
https://www.youtube.com/watch?v=W4NyevEQnCI&list=WL&index=151&t=1s
https://www.youtube.com/watch?v=lTW7CDG-jkc&t=0s&ab_channel=RohitSuri
6. Back in the project in CCS C write a code to sample 256 data using a timer and input it in an array.
7. Modified the code in step 6 such that different sampling frequency can be achieved. These sampling frequencies are as specified in the 'Design Specification' number 4.
8. Once 256 data are collected, sent this to the C# GUI application.
9. Write a program in C# to process these 256 data and using ZedGraph plot it on the GUI. After C# processed the data, C# will send an acknowledgement 'P' back to PIC24 to notify that it had finished processing and want another set of 256 data to be send again from PIC24.
10. This process will repeat again and again, and in effect it looks like real time to the us
11. Test the system by letting it run continuously and interact with it. Observe the behavior of the C# oscilloscope and if there is any non-desired behavior. Fix the bug and test again
12. Repeat step 11 as many times as possible to make the system robust.

Design Techniques

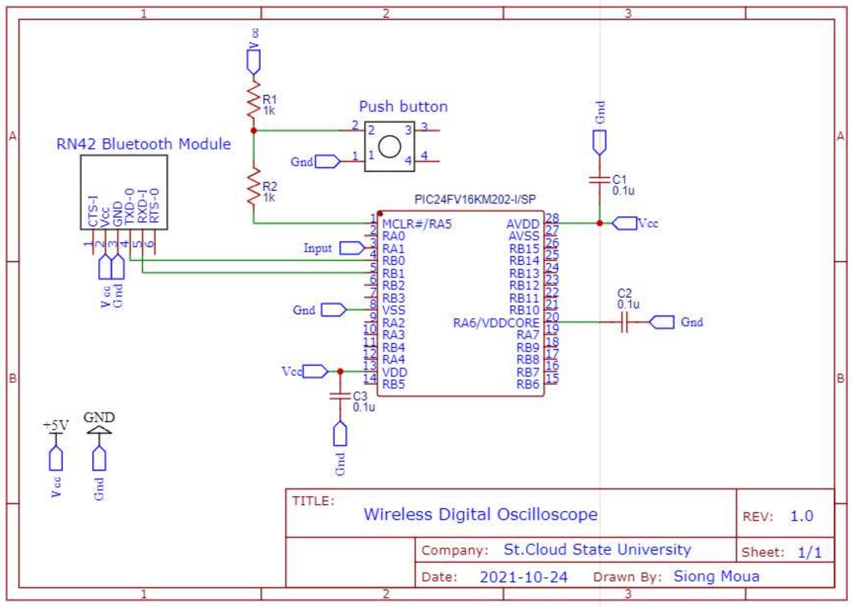


Figure 1: Circuit schematic of the component layout.

The circuit layout show in Figure 1 is of the PIC24FV16KM202 with the Bluetooth module RN42. In this project pin 3 (RA1) will be use as the ADC pin that will digitize the input signal.

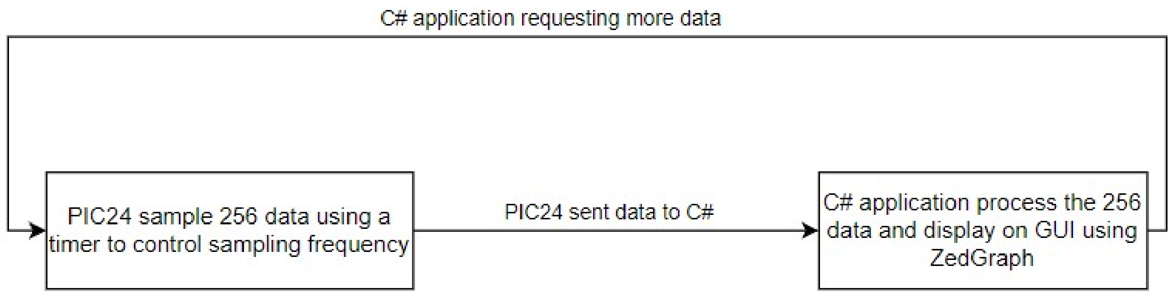

C# application requesting more data
PIC24 sample 256 data using a timer to control sampling frequency
PIC24 sent data to C#
C# application process the 256 data and display on GUI using ZedGraph

Figure 2: Handshake top view of program block diagram between PIC24 and C# GUI application.

The first most important part of this project is to establish a handshake communication protocol between PIC24 and the C# GUI application. This handshake protocol will ensure that there is no data collision between PIC24 and C# GUI which will allow the system to run neatly and uniformly without getting stuck.

The diagram show in Figure 2 shows the general idea of the handshake protocol that was developed and used for this project. At the start of the PIC24 program, it will wait for the C# application to request data. This mean that the user will initiate a start command using C# GUI button as shown below in Figure 7. Once PIC24 received the start command from C#, PIC24 will start a timer and samples whenever timer interrupt happens. When PIC24 finished collecting the data set (in this case it will be 256 data) it will disable timer and start sending these data to the C# application via the Bluetooth module using UART protocol. When finished sending the data, PIC24 will idle and wait for another request command from C# application.

Once C# received all 256 data, it will start to process these data and display accordingly. After C# finished processing the data set, it will automatically request more from PIC24. This process will happen again and again, and in effect the waveforms being displayed on the GUI will looks like real time.

In Table 1 below shows the different command C# will sent to PIC24. It describes what PIC24 should do upon receiving the command from C# application.

Table 1: The different commands that PIC24 will receive from C#.

C# requesting command	Description of command
"P" + "xxx" + "T"	Requesting data at normal sampling frequency ***with*** trigger control
"M" + "xxx" + "T"	Requesting data with **maximum** sampling frequency ***with*** trigger control
"Z"	Requesting data with **maximum** sampling frequency ***without*** trigger control

“X”	Requesting data at **normal** sampling frequency ***without*** trigger
“A”	Change Fs = 100 Hz
“B”	Change Fs = 1 kHz
“C”	Change Fs = 10 kHz
“D”	Change Fs = 50 kHz
“E”	Change Fs = 100 kHz
“F”	Change Fs = 140 kHz
“I”	Change Fs = 270 Hz (maximum)

Looking at Table 1, PIC24 will sample with or without using trigger, depending on the command sent from C# application. This decision algorithm is implemented in C# application. To allow the designed oscilloscope to mimic standard oscilloscope behavior. For example, when the trigger voltage is out of the input signal range, on a standard commercialized oscilloscope the wave that is being displayed is not steady and will wiggle around.

In the first two command of Table 1, the ‘xxx’ is the trigger voltage value. This mean that every time C# application request data from PIC24, it will send in a new trigger value as this trigger value can be changed by the user in the C# GUI shown in Figure 7 via the track bar on the left. Notice that this is only true for when C# is requesting data from PIC24 and want to use trigger, otherwise C# will request data and just want PIC24 to continuously sample without trigger.

The user can also use the combo box show in the C# GUI in Figure 7 to select a sampling frequency. And so, these commands are as shown respectively above in Table 1.

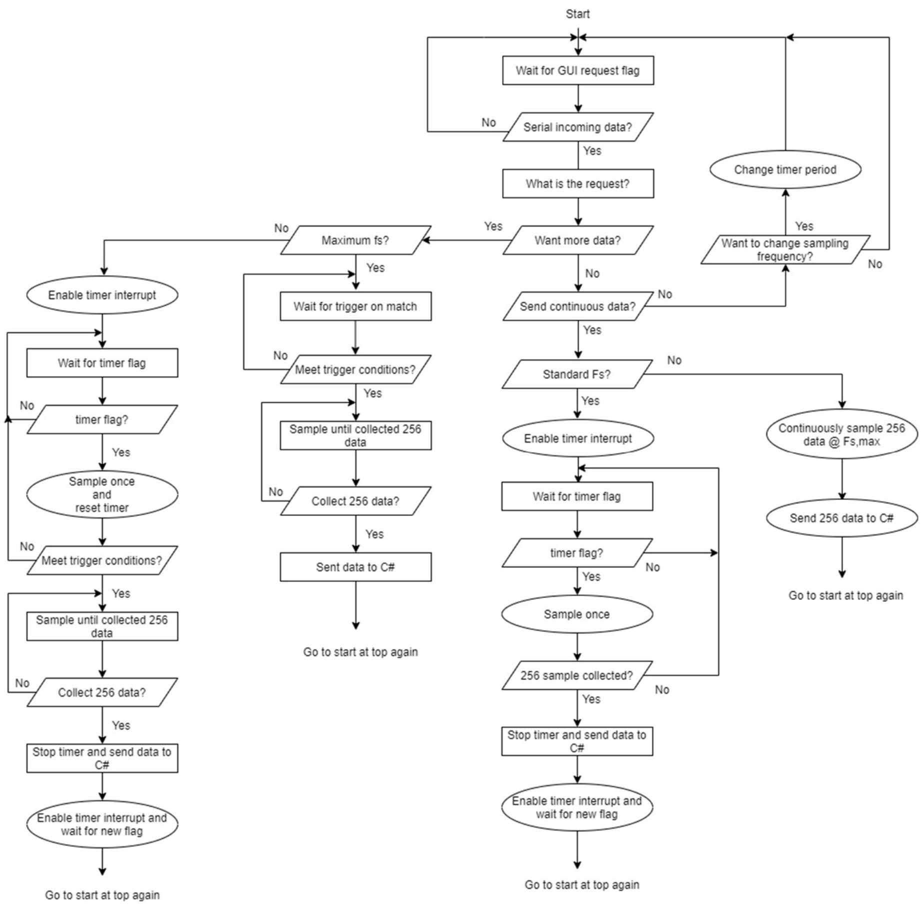


Figure 3: Logic flow of program implemented in PIC24FV16KM202.

Showed in Figure 3 is the ASM (arithmetic state machine) chart that implemented the logic in PIC24. Note that this logic flow diagram is not meant to display all details of the program. In this program diagram, what PIC24 will do is sample whenever C# application request data. PIC24 will follow the different commands that C# application sent, such as changing sampling frequency, or sample a data set with a controlled/non-controlled trigger value.

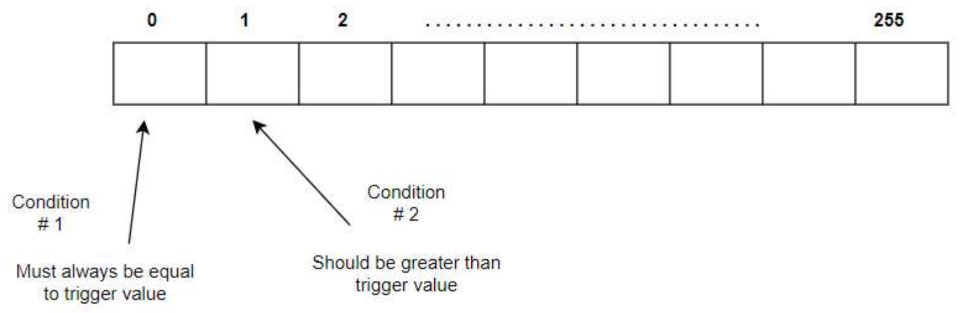

0
1
2
255
Condition # 1
Must always be equal to trigger value
Condition # 2
Should be greater than trigger value

Figure 4:The two conditions that trigger is based on.

To be able to display a nice steady waveform a triggering method (algorithm) must be in cooperated into this project. Figure 4 shows the two conditions that is required by the triggering algorithm before PIC24 will continuously sample and fulfill the full data set.

Assume that C# application requested data from PIC24. For the first sampled data that is put into this array of length 256, it must be the exact value as the trigger value. Otherwise, the timer will continue to run and PIC24 will continue to sample until it read a value that is the same as the trigger value.t If this first condition is met then PIC24 will put this sampled data (sampled input signal matching the triggered value) into the 0 index of the array.

Once the first condition is met, PIC24 will wait for timer interrupt to happen again before it sample the second time. This second sampled data (to be put in the 1st index of the array) MUST BE GREATER than the trigger value, otherwise PIC24 will discard the previously value stored in index 0 and start this triggering process all over again.

Since the algorithm make sure that the second sample data must be greater than the 0 index (trigger value), this implies that the system is triggering on a rising edge.

Again, this trigger algorithm will ensure that the waveform that is displayed on the C# GUI will be stable and steady. In other words, the different waveforms that is being displayed always starts at the same value (the triggered value).

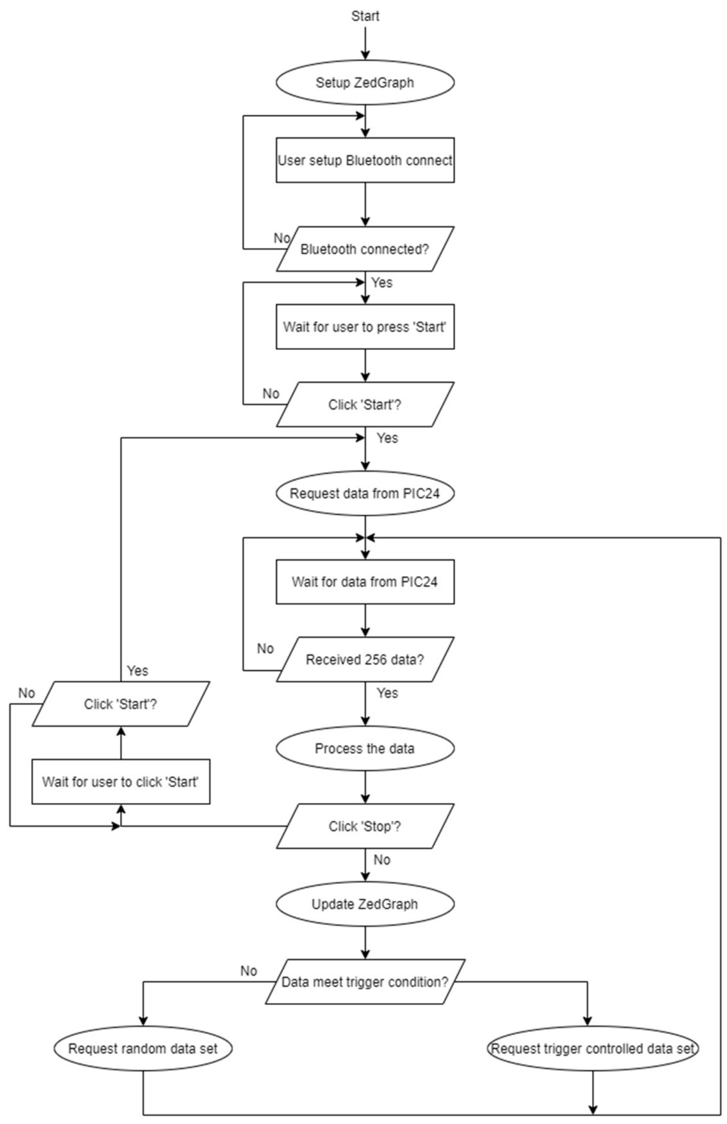


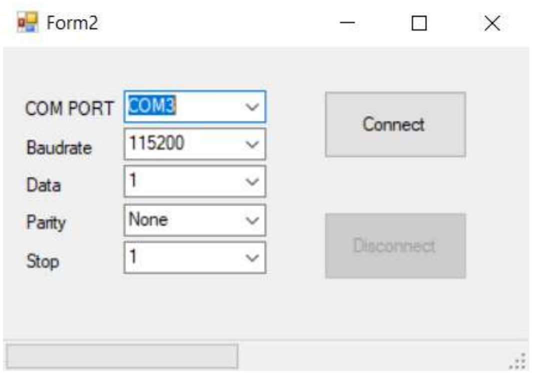
Figure 5: Logic flow in Visual Studio C# GUI application.

Showed in Figure 5 above is the logic flow implemented in the C# GUI application. Note that C# GUI is an event (interrupt) driven program and so the Figure 5 shown above do not show all the internal logic for the events being use. Though it displayed the general logic well enough.

At the start of the C# GUI application, it waits for the user to connect the system the Bluetooth module. Once the connect is established, the user will have to press the 'Start' button as shown in Figure 7 below. This will have C# initiate a data requesting command to PIC24 telling it to start sampling. While PIC24 is busy sampling data, the C# application will just hang and do nothing. And if the user decided to change anything in the C# GUI application (zoom, auto scale, shift, etc.), the update will not take effect until the 256 data have been received from PIC24. After C# received and processed the data, then the changes made by the user will change accordingly. If the 'Stop' button is clicked by the user, the waveform will no longer be update in the ZedGraph. Though the user can still make changes using the C# GUI as such (zoom, auto scale, shift, etc.) and the program will still update the ZedGraph right away. Again, it is only while the C# application is running and the user make changes to any of the features such as (zoom, auto scale, shift, etc.), C# will not update these changes until it received 256 data from PIC24.

This feature helps eliminate the complexity of the displaying and update of ZedGraph in the C# GUI application program.

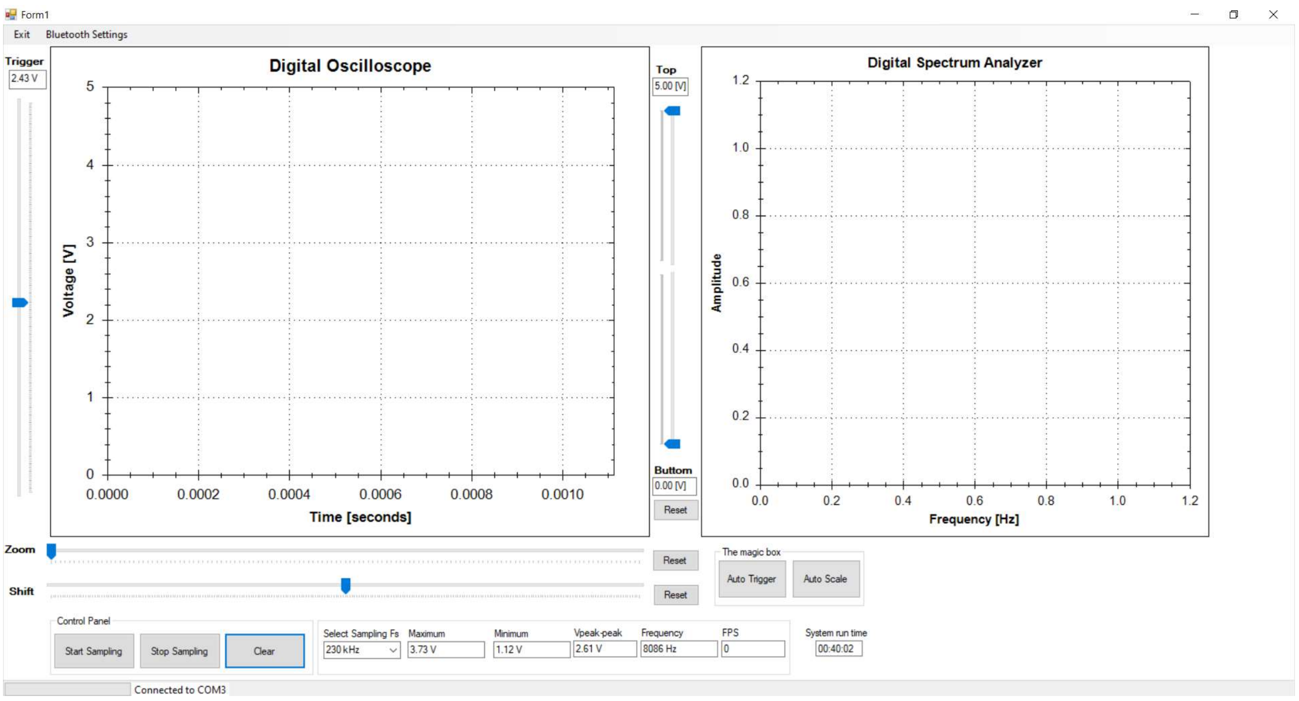
Result and Analysis



COM PORT	COM3
Baudrate	115200
Data	1
Parity	None
Stop	1

Figure 6: C# form for connecting to Bluetooth module.

In the Figure 6 above shows the form that was developed to allow the user to initiate a Bluetooth connection link between the Bluetooth module (RN42) and the C# GUI application. Note that there is a link to a YouTube video tutorial that was referenced for this Bluetooth connectivity design form. This link can be found in step 5 in the 'Procedure' section of this documentation.



Trigger	2.43 V
Top	5.00 [V]
Buttom	0.00 [V]
Select Sampling Fs	230 kHz
Maximum	3.73 V
Minimum	1.12 V
Vpeak-peak	2.61 V
Frequency	8086 Hz
FPS	0
System run time	00:40:02

Connected to COM3

Figure 7: The developed C# GUI application for the designed digital oscilloscope.

Above in Figure 7 shows the designed C# GUI window form that was developed. This C# digital oscilloscope GUI used the ZedGraph (open-source library) to plot the sampled data. Also notice that the GUI also have a digital spectrum analyzer on the right-hand side (using ZedGraph as well).

The sample data (time domain data) is plotted to the left ZedGraph. The C# GUI application also used the DFT to transform the sampled data (time domain data) to the frequency domain. This frequency domain data set is then plotted to the right ZedGraph, showing the frequency content of the input signal. How the DFT works and how it is implemented is presented in Appendix C of this document.

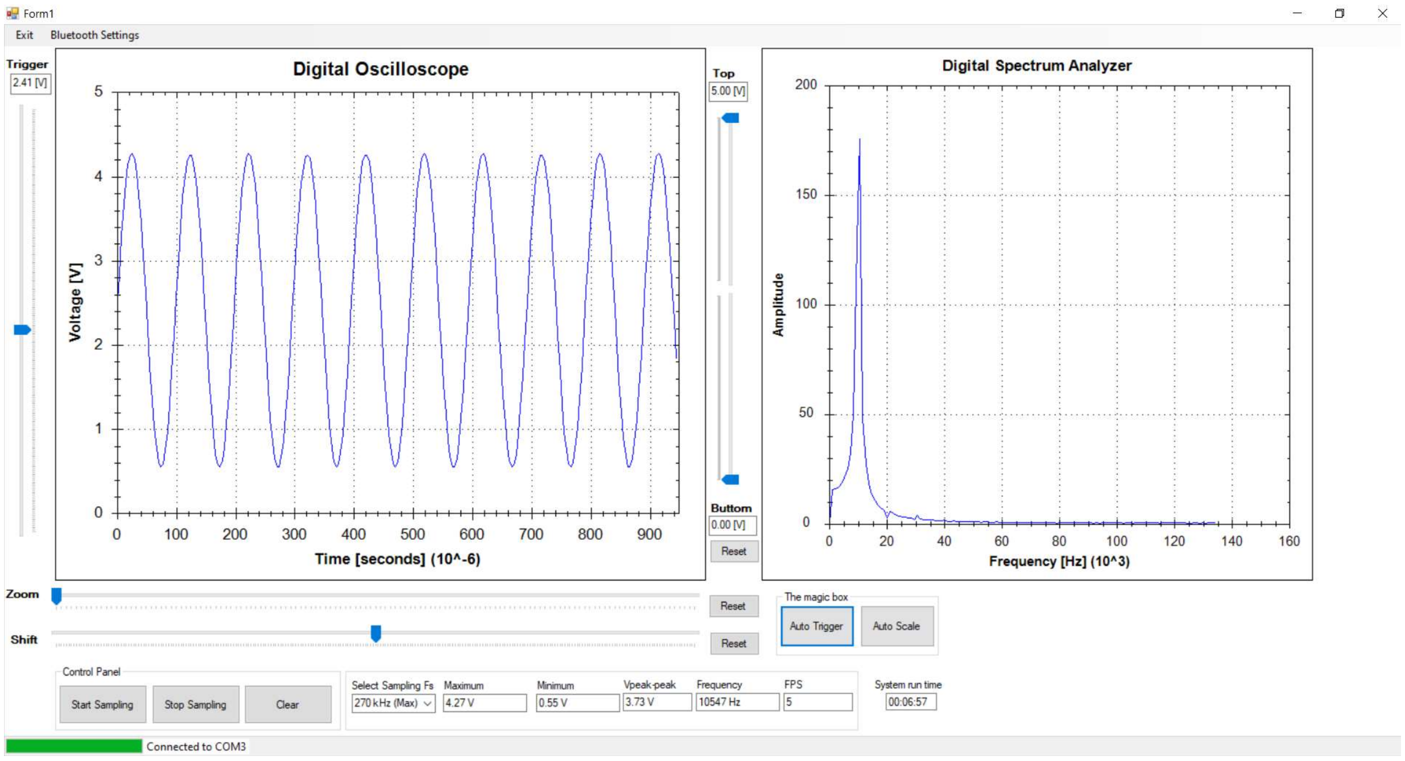
The C# GUI of Figure 7 have multiple trackbars that allow the user to perform these functionality

- Zoom in/out via the time axis (x-axis)
- Zoom in/out via the voltage axis (y-axis)
- Shift left/right (shifting in time axis (x-axis))
- Change trigger value

The C# GUI of Figure 7 also provided multiple text boxes to display information such as

- Maximum voltage
- Minimum voltage
- Peak to peak voltage
- Frequency of the input signal
- FPS (frame per second) of the re-fresh rate of the ZedGraph
- Elapsed time ever since Bluetooth connection has been established
- Trigger voltage
- Zoom in/out (x-axis)
- Zoom in/out (y-axis)

Another cool feature that is added to this C# GUI application is the two buttons inside the group box called ‘The magic box’. The ‘Auto Trigger’ button allow the program to auto trigger an incoming signal, so that the user don’t have to manually set the trigger voltage trackbar.. The ‘Auto Scale’ will zoom in onto the signal via the voltage axis (y-axis). This feature is very useful for zooming in onto signal that is very small.



Trigger: 2.41 [V]
Top: 5.00 [V]
Buttom: 0.00 [V]

Select Sampling Fs	Maximum	Minimum	Vpeak-peak	Frequency	FPS	System run time
270 kHz (Max)	4.27 V	0.55 V	3.73 V	10547 Hz	5	00:06:57

Connected to COM3

Figure 8: Maximum sampling frequency of 270 kHz with input signal of 10 kHz.

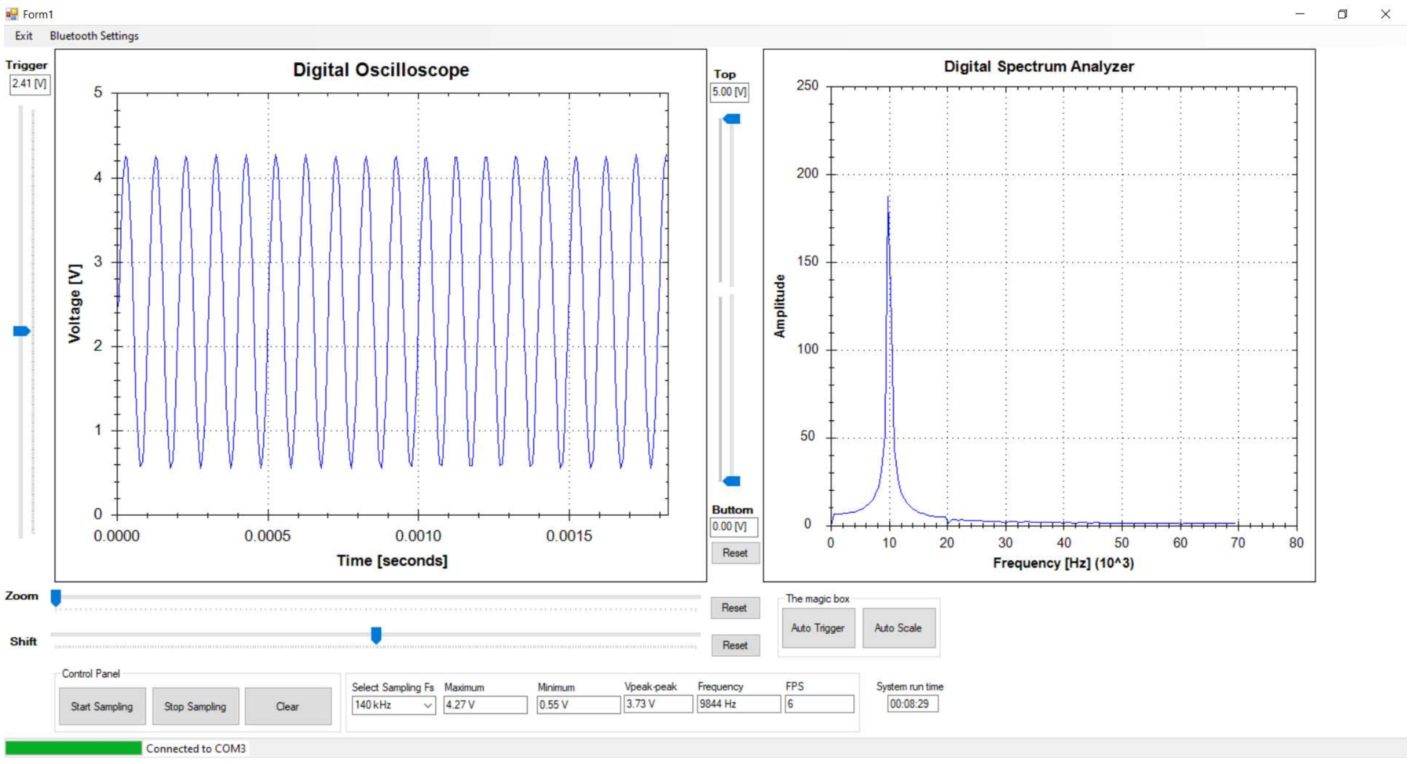
The maximum sampling frequency of the PIC24FV16KM202 without using a timer is 270 kHz. Shown in Figure 8 above is the PIC24 sampling at a sampling frequency of 270 kHz with an input signal of 10 kHz. One might ask how is the maximum sampling frequency determined? Simply setup a timer and before entering a WHILE loop to continuously sample the 256 data. And when done sampling 256 data, simply read the timer value. This timer value can be converted to time and divided by 256 and this will be the sampling frequency.

In the Figure 8 some information of the input signal is displayed such as maximum voltage, minimum voltage, peak to peak voltage, and as well as frequency.

Notice that the DFT determined the frequency to be 10547 Hz even though the input frequency is 10 kHz. This is due to the resolution of the sampling frequency.

Frequency resolution is Fs/N = 270k/256 = 1054.68 Hz. This implies that every mark on the frequency domain plot (right ZedGraph) will be an integer of 1054.68 Hz.

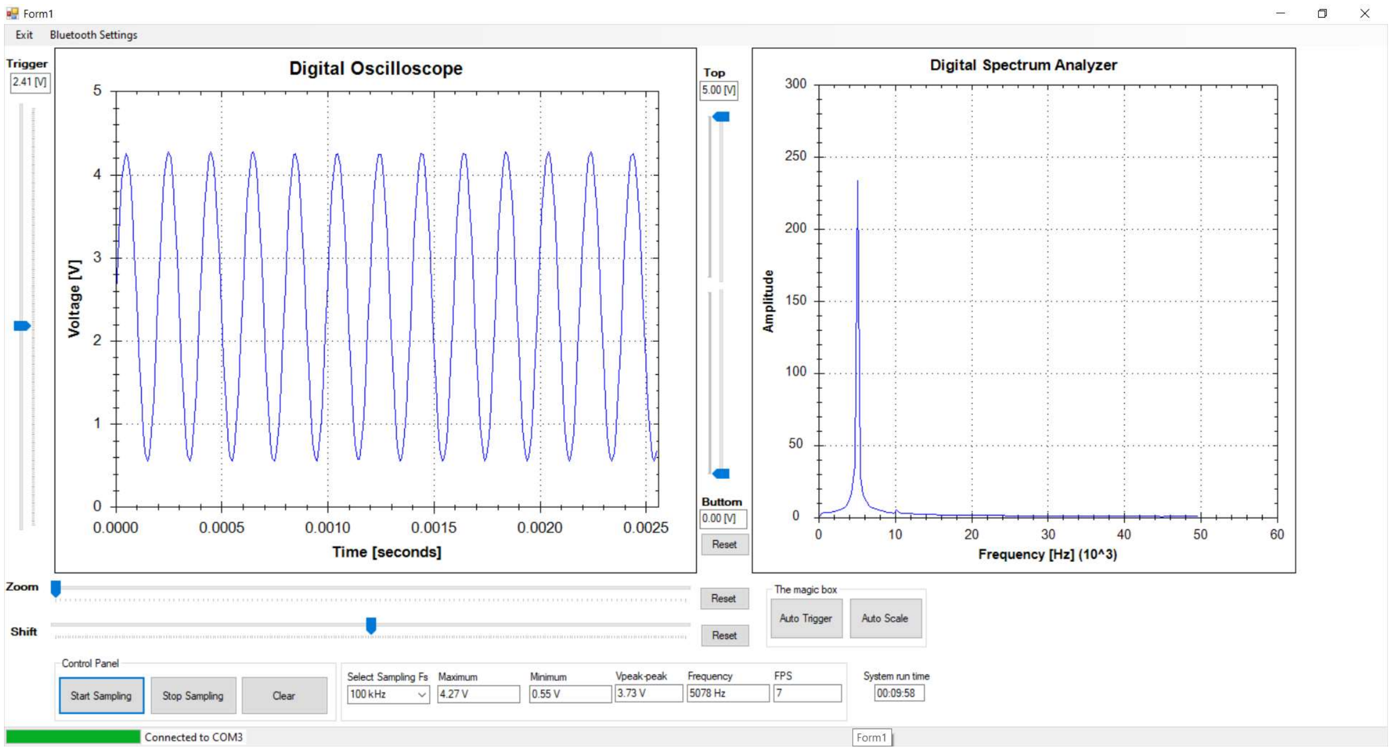
So, let us perform this math; 10547 Hz / 1054.68 Hz =~ 10. It is as expected, to get an integer from the division. So, the accuracy of the DFT is not accurate but good enough for our application.



Select Sampling Fs: 140 kHz
Maximum: 4.27 V
Minimum: 0.55 V
Vpeak-peak: 3.73 V
Frequency: 9844 Hz
FPS: 6
System run time: 00:08:29

Figure 9: The maximum sampling frequency when using a timer is 140 kHz.

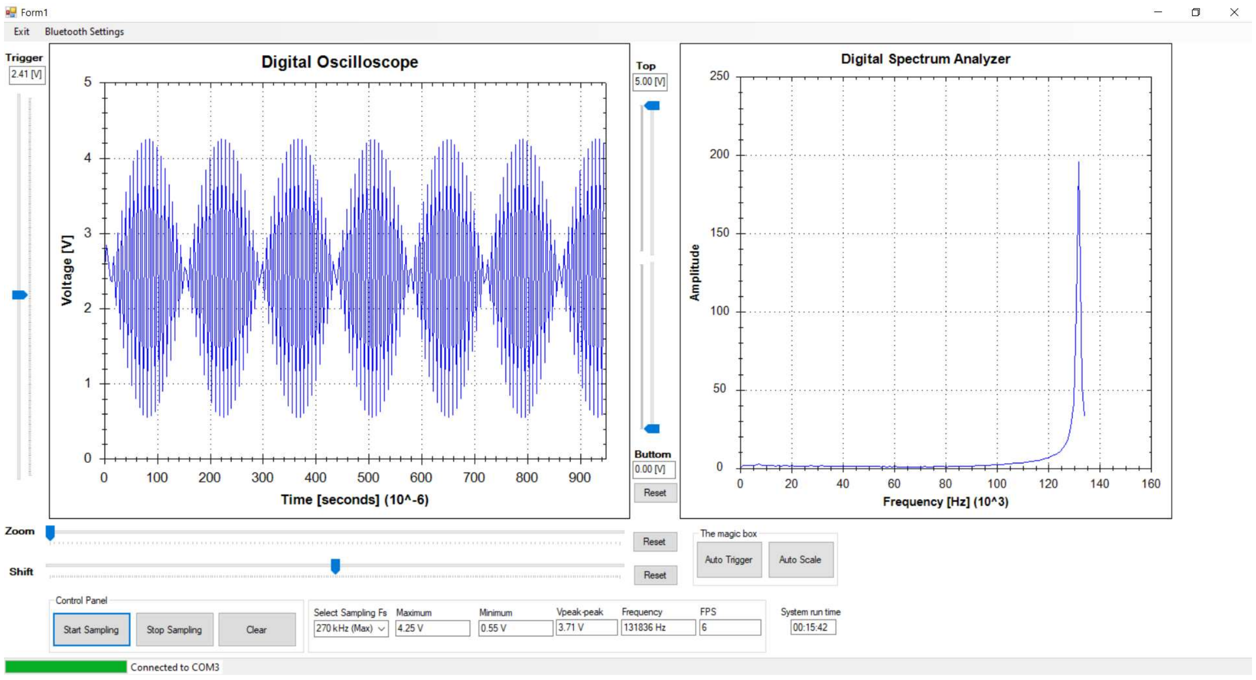
Figure 9 shows the PIC24 sampling at its maximum frequency is 140 kHz when using a timer interrupt. The input signal is still 10 kHz. Notice that the DFT now detect the input signal have a frequency of 9844 Hz. The resolution is getting better because of smaller sampling frequency as opposed to the one shown in Figure 8 above.



Select Sampling Fs: 100 kHz
Maximum: 4.27 V
Minimum: 0.55 V
Vpeak-peak: 3.73 V
Frequency: 5078 Hz
FPS: 7
System run time: 00:09:58

Figure 10: Sampling frequency of 100 kHz with input signal of 5 kHz.

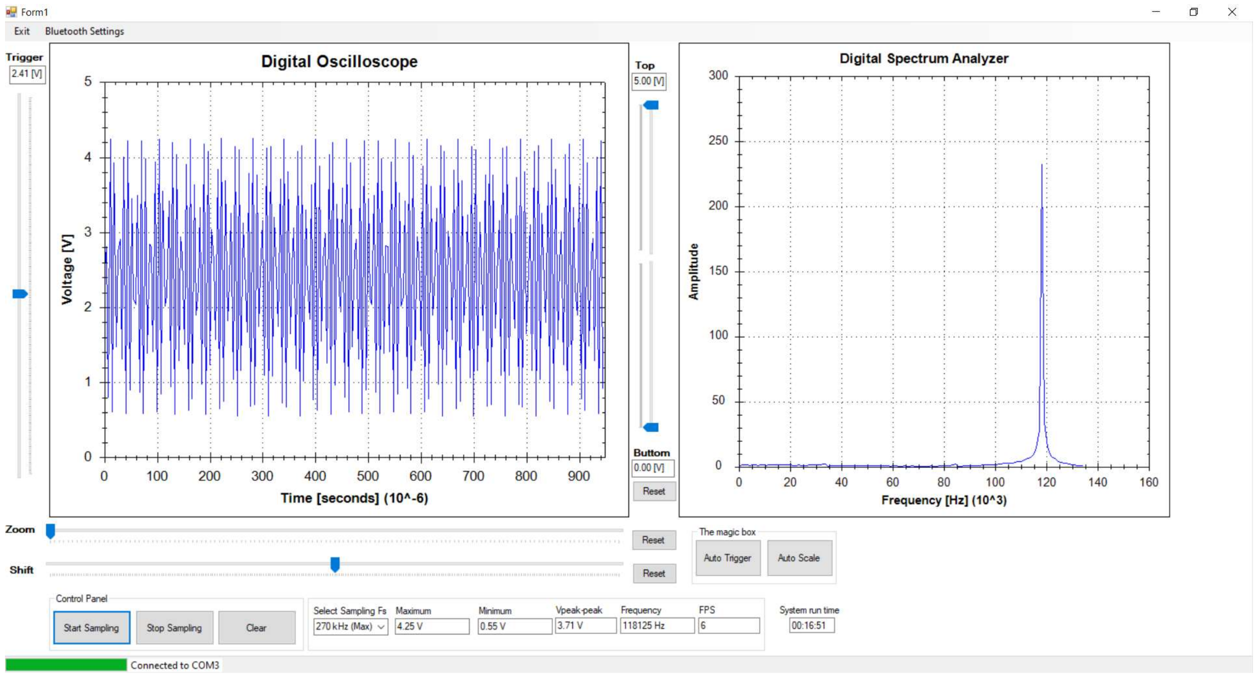
In the figure above PIC24 is digitizing an input signal of 5 kHz at a sampling frequency of 100 kHz. The DFT calculated the input signal to be 5078 Hz.



Select Sampling Fs: 270 kHz (Max)
Maximum: 4.25 V
Minimum: 0.55 V
Vpeak-peak: 3.71 V
Frequency: 131836 Hz
FPS: 6
System run time: 00:15:42

Figure 11: Exampling of ALMOST violating the sampling theorem and how it effect DFT performance.

In the Figure 11, the PIC24 is sampling at its maximum sampling frequency of 270 kHz without a timer. The input signal is 130 kHz. From the sampling theorem we know that we cannot sampling a signal that is higher than half the sampling frequency, in this case no signal over 135 kHz. This implies that with the input signal of 130 kHz and a sampling frequency of 270 kHz we are not violating the sampling theorem yet. Therefore, the DFT is still able to detect the input signal's frequency to be 131836 Hz which is fairly accurate (close enough).



Trigger 2.41 [V]
Top 5.00 [V]
Buttom 0.00 [V]

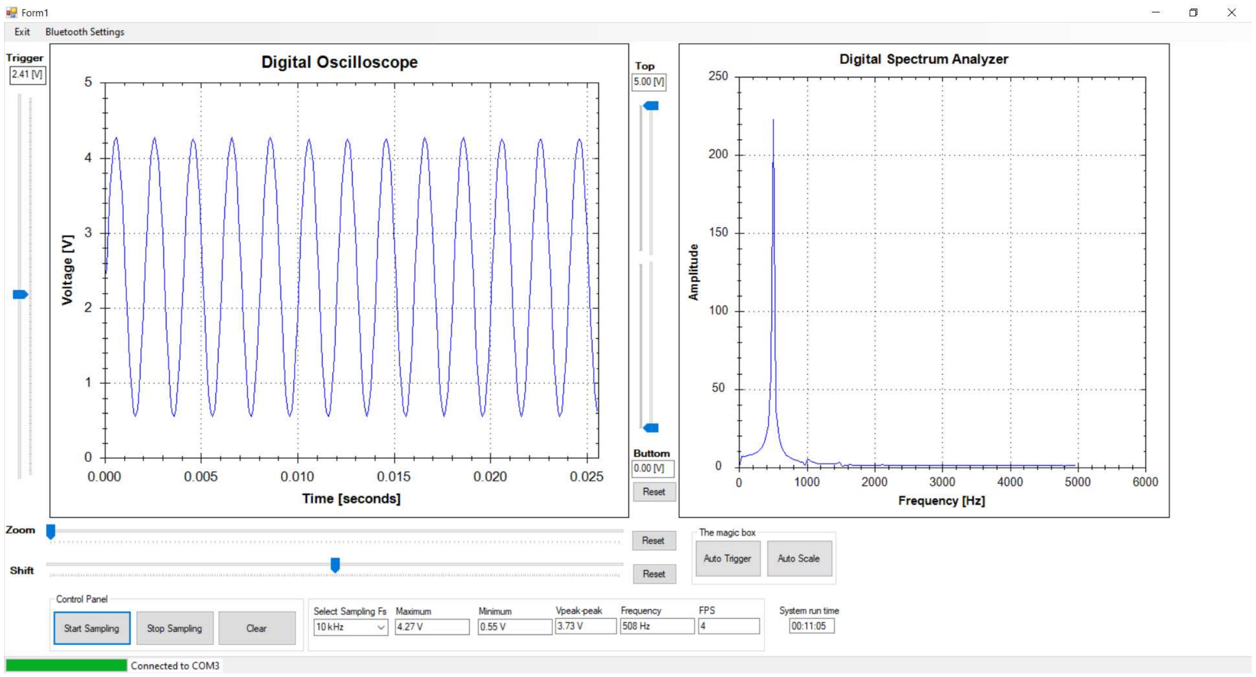
Select Sampling Fs	Maximum	Minimum	Vpeak-peak	Frequency	FPS	System run time
270 kHz (Max)	4.25 V	0.55 V	3.71 V	118125 Hz	6	00:16:51

Connected to COM3

Figure 12: Exampling of violating the sampling theorem and how it effect the DFT performance.

In Figure 12 the PIC24 is sampling at 270 kHz and the input signal is 150 kHz. This demonstrated the effect of aliasing. Aliasing is the term used when the sampling theorem is violated. This mean that we are sampling a signal that have a frequency that is larger than half of our sampling frequency. In this case 150 kHz is larger than 135 kHz.

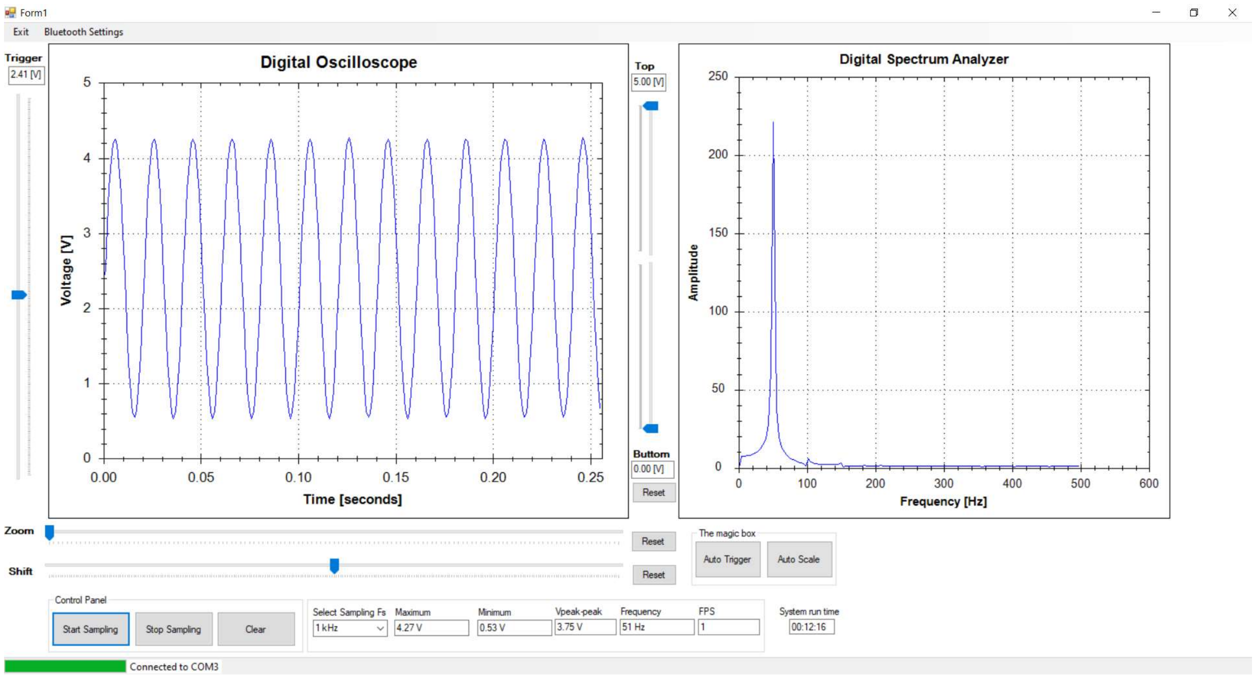
When the sampling theorem is violated, the DFT is no longer able to determine the input signal frequency accurately. With this begin said, in Figure 12 the DFT measured the input signal to have a of 118125 Hz. The input signal frequency is 150 kHz, but the DFT calculate it to be 118125 Hz. This here demonstrated how violating the sampling theorem can affect the performance of the DFT. This phenomenon is called frequency folding, where the magnitude of the peak moves back toward the DC (0 Hz (moving left)) if input frequency is increased more than half the sampling frequency.



Trigger 2.41 [V]
Top 5.00 [V]
Buttom 0.00 [V]
The magic box
Control Panel
Select Sampling Fs: 10 kHz
Maximum: 4.27 V
Minimum: 0.55 V
Vpeak-peak: 3.73 V
Frequency: 508 Hz
FPS: 4
System run time: 00:11:05
Connected to COM3

Figure 13: Sampling frequency of 10 kHz and input signal frequency is 500 Hz.

The sampling frequency show in Figure 13 is 10 kHz with an input signal of 500 Hz. This demonstrated that the designed system can sample at lower frequency too.

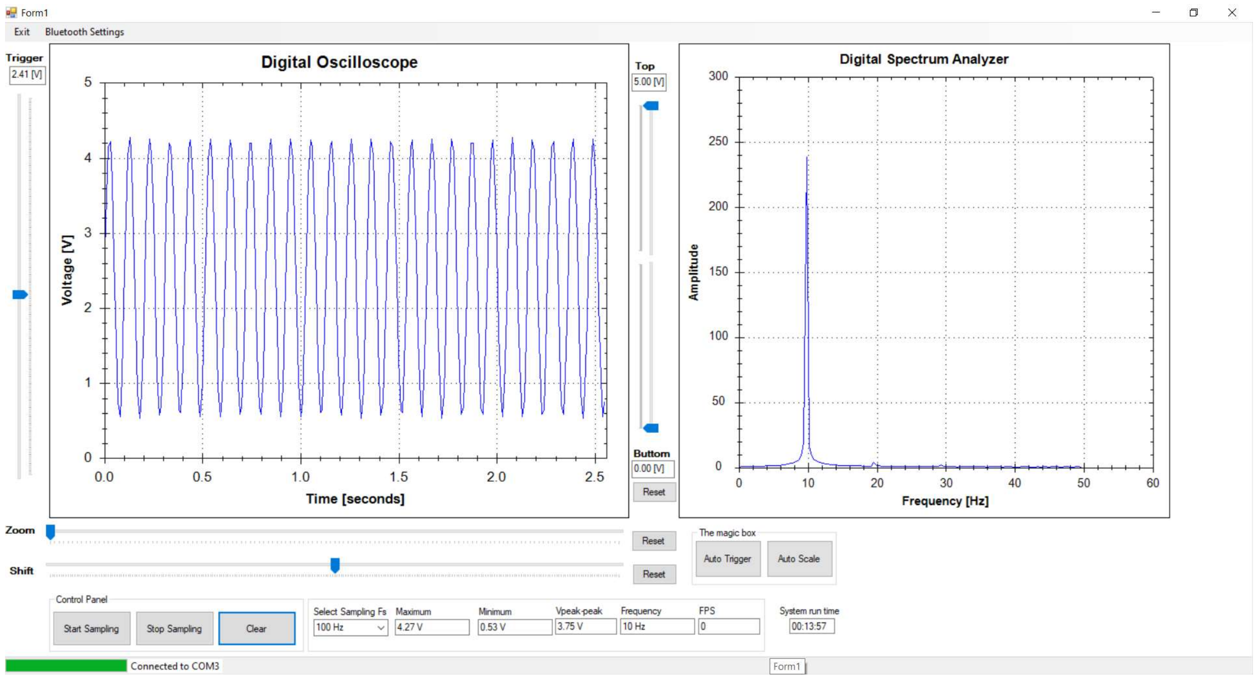


Trigger 2.41 [V]
Top 5.00 [V]
Buttom 0.00 [V]
The magic box
Control Panel
Select Sampling Fs: 1 kHz
Maximum: 4.27 V
Minimum: 0.53 V
Vpeak-peak: 3.75 V
Frequency: 51 Hz
FPS: 1
System run time: 00:12:16
Connected to COM3

Figure 14: Sampling frequency of 1 kH with input signal frequency of 50 Hz.

In Figure 14 the sampling frequency is 1 kHz and the input signal frequency is 50 Hz. The resolution is therefore Fs/N =1000/256 = 3.906 Hz.

Notice that in this system the sampling frequency changes, but the number of data is always the same 256. Therefore, at lower sampling frequency the frequency resolution gets better (finer).



Trigger 2.41 [V]

Top 5.00 [V]

Buttom 0.00 [V]

Select Sampling Fs	Maximum	Minimum	Vpeak-peak	Frequency	FPS	System run time
100 Hz	4.27 V	0.53 V	3.75 V	10 Hz	0	00:13:57

Connected to COM3

Figure 15: Sampling frequency of 100 Hz with input signal of 10 Hz.

In the Figure 15 the sampling frequency is 100 Hz while the input signal is 10 Hz. This have a frequency resolution of .39 Hz. Note that the DFT can accurately measure the input signal with not much of an error.

It is worth noting that the system can sample at low frequencies, but the refresh rate is very slow. This is due to the triggering algorithm and the number of data it PIC24 must collect. Since the sampling frequency is slow the triggering algorithm have a hard time satisfying its two conditions.

Discussion

This designed system of the digital oscilloscope can sample at a maximum frequency of 270 kHz without a timer. For sampling with a timer, the maximum accurate frequency the PIC24 can achieve is 140 kHz. The system can also sample at lower frequencies such as 10 kHz, 1 kHz, and 100 Hz.

Though at lower sampling frequencies the refresh rate of the C# GUI is slow. This is due to the triggering algorithm that is implemented in PIC24. The triggering algorithm have a hard time satisfying its two conditions as shown in Figure 4. Since the sampling frequency is slow (100 Hz and 10 kHz) PIC24 sample at a slower rate and will have a hard time sampling and matching the first sample to the trigger value.

Another reason that the refresh rate is slow is due to the 256 data set that needs to be collected. For example, PIC24 is sampling at 100 Hz and needs to collect 256 data. This implies that the amount of time that is needed to collect 256 data is as shown below

Time to sample 1 data: 1/100 = 10 ms;

Time to sample 256 data = (1/100) * 256 = 2.56 second

When PIC24 is commanded to sample at 100 Hz, it will take 2.56 seconds for it to collect 256 data! And this didn't even include the amount of time PIC24 discard its data set buffer due to the triggering algorithm.

The system also have a limitation to the allowable input frequencies. Since the maximum sampling frequency of the system is 270 kHz. Theoretically we can sample frequency as high as 135 kHz, but visually the resolution of the signal is very bad. This mean for the input signal wave with frequency of 135 kHz will have only 2 points representing a full period in the time domain.

Therefore, it is recommended to have at least 10 points representing a full period of any input signal. This mean that if PIC24 is sampling at 100 kHz, the input signal should only have maximum frequency of 10 kHz. Otherwise, if the input signal have higher frequency than 10 kHz, a full period will be presented with less than 10 points which the visualization resolution of the wave very bad. Meaning that the user will not see curves anymore, but rather sharp edges that supposed to be curves.

The system was well tested and ran for lengthy number of times with user interaction via the C# GUI. No strange behavior was observed. Though it is not guaranteed that the system is bug free, therefore the system should continue to be tested and improvement can always be made.

Conclusion

In this project a digital oscilloscope with a spectrum analyzer was designed and built. The system was used a PIC24FV16KM202 microcontroller as the digitization module, and a Visual Studio C# WinForms application as the display and controller module. The system can have a maximum sampling frequency of 270 kHz without using a timer, and maximum sampling frequency of 140 kHz with using a timer. The user can interact with the system via the C# GUI application such as changing the sampling frequency, triggering value, zooming, shifting, etc. The system can also display the frequency spectrum of the input signal by using the DFT. The designed system was well designed and tested and met all designed parameters, therefore this project was successfully implemented and met all designed specifications.

Acknowledgement

Thank you to Professor Zheng for this superior guidance and knowledge in making this great project available to all his ECE 422/522 students.

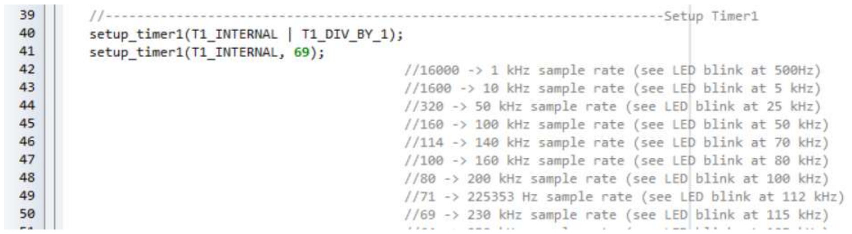
Appendix A (Code in PIC24)

This section of Appendix A will highlight the importance of the codes used in both PIC24 and C# program.

Note that ***NOT*** all code will be show. Only the important part of the code. If you desired to see all the code, please contact me directly via email: smoua@stcloudstate.edu

How to setup timer in PIC24

PIC24FV16KM202 have only one 16-bit timer. The use of this timer is very critical in this project. We must sample the incoming data at a well-controlled interval of time, and this controlled interval of time is called the sampling frequency (fs). This sampling of a continuous signal to the discrete form is also called digitization. Digitizing with a well-controlled sampling frequency is important in the field of digital signal processing such as digital filter design and labeling of frequency axis when using DFT.

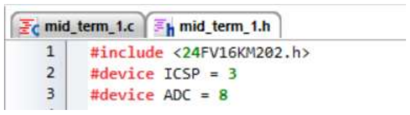


```
39  //-----------------------------------------------------------------------Setup Timer1
40  setup_timer1(T1_INTERNAL | T1_DIV_BY_1);
41  setup_timer1(T1_INTERNAL, 69);
42                                         //16000 -> 1 kHz sample rate (see LED blink at 500Hz)
43                                         //1600 -> 10 kHz sample rate (see LED blink at 5 kHz)
44                                         //320 -> 50 kHz sample rate (see LED blink at 25 kHz)
45                                         //160 -> 100 kHz sample rate (see LED blink at 50 kHz)
46                                         //114 -> 140 kHz sample rate (see LED blink at 70 kHz)
47                                         //100 -> 160 kHz sample rate (see LED blink at 80 kHz)
48                                         //80 -> 200 kHz sample rate (see LED blink at 100 kHz)
49                                         //71 -> 225353 Hz sample rate (see LED blink at 112 kHz)
50                                         //69 -> 230 kHz sample rate (see LED blink at 115 kHz)
```

Line 40 will tell PIC24 to use the internal clock frequency which is 16 MHz. Though PIC24 runs with a 32 MHz, but it takes two clock cycles to execute one instruction, and therefore the internal clock is 16 MHz.

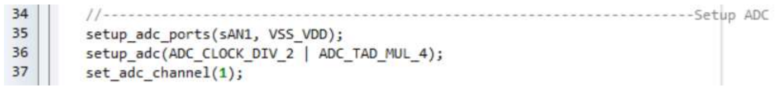
Line 41 load the period to the counter of the clock. This mean that when the counter counts to 69, it will overflow back to zero. Using this method of loading the period is better than using the 65535 counts as the overflow period, as we will have to always load a new starting value to the timer and will end up with timing errors due to overhead times.

How to setup ADC in PIC24 for digitization of input signal



```
1  #include <24FV16KM202.h>
2  #device ICSP = 3
3  #device ADC = 8
```

Even though PIC24 have 12-bit ADC, it can only sample up to 100 ksp/s maximum. For our application we are going to use 8-bit ADC so that we can achieve a higher sampling rate, though our resolution in digitization has decreased dramatically. In line 3 we are telling PIC24 to do exactly this, to digitize the input signal with 8-bit resolution. ***It is important to note that the place where this command is placed is also important, therefore it needs to be placed right at the beginning section of the program.***



```
//-----------------------------------------------------------------------Setup ADC
setup_adc_ports(sAN1, VSS_VDD);
setup_adc(ADC_CLOCK_DIV_2 | ADC_TAD_MUL_4);
set_adc_channel(1);
```

In line 35 PIC24 is commanded to use sAN1 (AN1 also mean pin 3) as the ADC pin and use the Vss and Vdd (0V and ~5 V) as the reference to the ADC module.

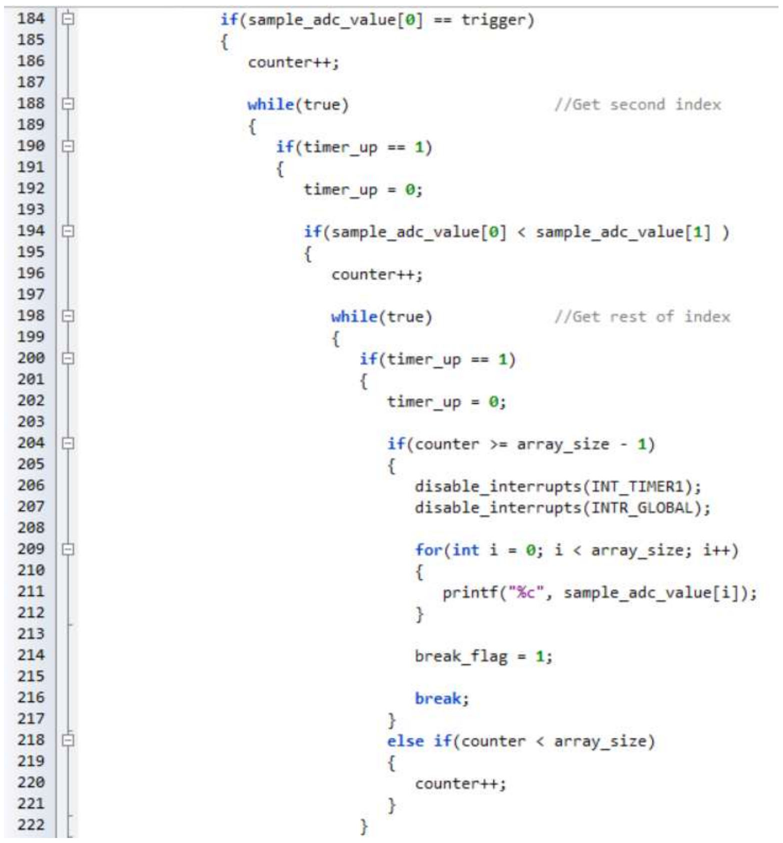
Line 36 tells PIC24 which clock to use to drive its internal ADC module. In this case we are dividing the 16 MHz clock by 2, and the aquations time of the ADC is set to 4 times the input clock. Meaning that we will give the ADC module some time (4*(1/8M) = .5 us) to tried to correctly determine the input voltage before it convert the reading to the digital domain. It is important to note that if we don't give the ADC module enough time to quantize the input signal, the reading of the ADC will not be accurate. In other words, we can tell the ADC module to digitize (1/16M = 62.5 ns) at its maximum speed but don't expect to get accurate reading!

Line 37 function like line 36 except for not having the referenced voltage specified. Therefore, without line 37, the code should still function properly.

How Triggering is implemented

When using a digital oscilloscope, the triggering feature plays an important role in stabilizing the display of the input signal. The same holds true in this project. A triggering method is needed to properly display the waveform to the C# GUI for the user to analyze, otherwise the C# GUI will display the input waveform with at different starting point in each re-fresh. And with this, it looks like the signal is moving and is not steady.

Below shows majority of the code that is used to implement the triggering feature in this design project.

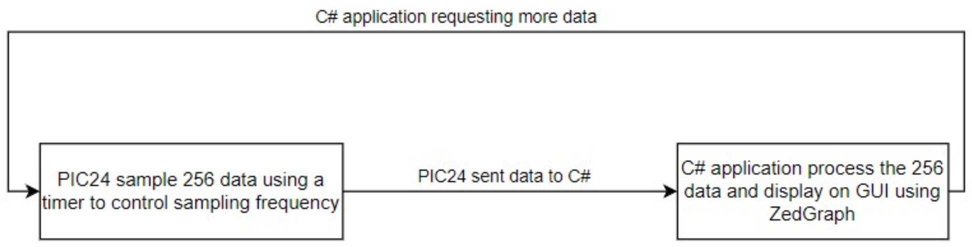


```
if(sample_adc_value[0] == trigger)
{
   counter++;

   while(true)                      //Get second index
   {
      if(timer_up == 1)
      {
         timer_up = 0;

         if(sample_adc_value[0] < sample_adc_value[1] )
         {
            counter++;

            while(true)             //Get rest of index
            {
               if(timer_up == 1)
               {
                  timer_up = 0;

                  if(counter >= array_size - 1)
                  {
                     disable_interrupts(INT_TIMER1);
                     disable_interrupts(INTR_GLOBAL);

                     for(int i = 0; i < array_size; i++)
                     {
                        printf("%c", sample_adc_value[i]);
                     }

                     break_flag = 1;

                     break;
                  }
                  else if(counter < array_size)
                  {
                     counter++;
                  }
               }
```

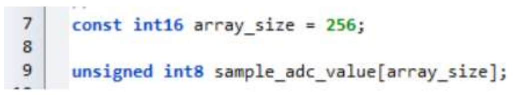
Appendix B (Handshake of PIC24 & C# GUI code)

Use of handshake between PIC24 and C# program

The figure show below demonstrates the handshake process between PIC24 and C# program.

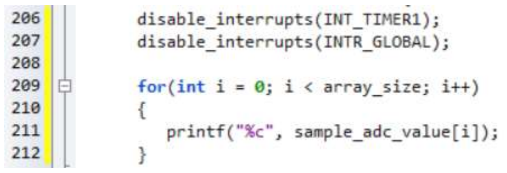

C# application requesting more data
PIC24 sample 256 data using a timer to control sampling frequency
PIC24 sent data to C#
C# application process the 256 data and display on GUI using ZedGraph

At the start of the system, user must use C# (pressing 'Start') to send a character 'P' followed by 'xxx' three characters representing the trigger voltage in integer form (0 to 255). Once PIC24 received this it will enable the timer and starts to sample 256 data. Once it finished sampling, it will disable the timer and send this array of 256 data to C#. Once C# received all 256 data it will start to process the data. After C# finished processing and displaying the data accordingly, it will automatically send the request flag 'P' followed by the 3 characters representing the trigger value to PIC24. This process will repeat again and again and in effect it looks like real time. Though the user can stop the process anytime and can always start it again.



```
const int16 array_size = 256;

unsigned int8 sample_adc_value[array_size];
```

It is important to note that the array used to collect the 256 data must be defined as 'unsigned int8' as we are using 8-bit ADC.



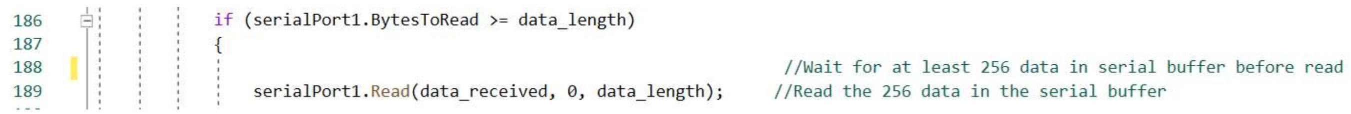
```
        disable_interrupts(INT_TIMER1);
        disable_interrupts(INTR_GLOBAL);

        for(int i = 0; i < array_size; i++)
        {
           printf("%c", sample_adc_value[i]);
        }
```

The above code shows the disabling of the timer and entering a FOR loop to print the 256 data to C# via the Bluetooth module using UART protocol. It is very critical to use the '%c' command inside the 'printf' statement as we will be sending each data as a character (0~255).

For example, if we want to send '100' and we are using '%i', we will be sending '1', '0', and '0'. Which will slow our system down! But since we are using '%c' instead we will be sending '100'.

Reading all 256 data at once in C# program

In the C# program, we must not read the data right away when we received it. Instead, we should wait until we finally received 256 data in our received buffer. This method will allow our system to run and display the waveform at a faster rate as opposed to reading one character on every received interrupt from the serial port in C#.

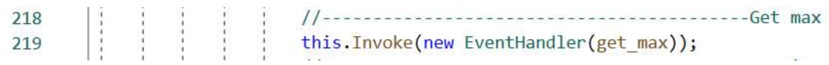


```
186    if (serialPort1.BytesToRead >= data_length)
187    {
188                                                        //Wait for at least 256 data in serial buffer before read
189        serialPort1.Read(data_received, 0, data_length);      //Read the 256 data in the serial buffer
```

In line 186 above, the program will not do anything (no reading the buffer) until there is 256 data in the received buffer. Now when the serial received interrupt fires, and it is the 256th data in the serial received buffer, the program will read all 256 data into the 'data_received' variable.

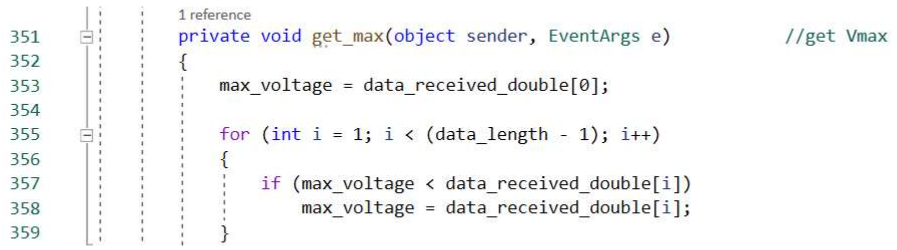
How to invoke a method when program is inside C# serial received method

When the 256 data are received, C# will start to process these data and it needs to display this information to the C# GUI for user to analyze. Therefore, we must be able to display this information after we finished processing the data. This mean that inside the C# serial received method after we have read all the 256 data into the 'data_received' variable and have finished processing, we must call other methods from within the C# serial received method. Though the C# serial received method runs at a different thread, and so we cannot just display information directly to a textbox from within inside this C# serial received method. We must invoke other methods, otherwise it will throw some 'cross threading' exception, and the program will stop and fail. Therefore, below will allow us to work around this 'cross threading' exception.



```
//----------------------------------------Get max
this.Invoke(new EventHandler(get_max));
```

Call the desired function as show above from within the C# serial received method.



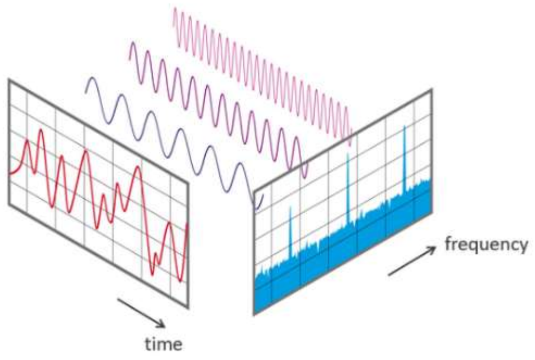
```
1 reference
private void get_max(object sender, EventArgs e)          //get Vmax
{
    max_voltage = data_received_double[0];

    for (int i = 1; i < (data_length - 1); i++)
    {
        if (max_voltage < data_received_double[i])
            max_voltage = data_received_double[i];
    }
```

Define the method as shown above. This will allow the program to halt from within the C# serial received method and jumps to another method. In this example we are searching for the maximum value within the 256 data set.

Appendix C (Understanding DFT and C# code implementation)

Understanding the DFT (discrete Fourier Transform)

The DFT (discrete Fourier Transform) is a mathematical tool that transform a signal from the time domain into the frequency domain. The figure shown below demonstrated this concept well. In essence the DFT is correlating the time domain signal with each of the basic functions (sinusoidal). Correlation is the detection of how likely a signal exist in another signal on comparison. In other words, it is a way (indeed the optimal way) of detecting a known signal inside another signal. And so, when we are performing the DFT on a time signal we are trying to detect which basic functions (sinusoidal functions of different frequency) exist inside the time domain signal.



Source: https://makeabilitylab.github.io/physcomp/signals/FrequencyAnalysis/index.html

The mathematical definition of the DFT is shown below;

- The DFT of N real data values x[n]: $x[n] \leftrightarrow X(k)$

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn}, \quad k = 0,1,\ldots,N-1 \\ &= \sum_{n=0}^{N-1} x(n)[\cos(\frac{2\pi}{N}kn) - j\sin(\frac{2\pi}{N}kn)] \\ &= \sum_{n=0}^{N-1} x(n)\cos(\frac{2\pi}{N}kn) - \sum_{n=0}^{N-1} x(n)j\sin(\frac{2\pi}{N}kn) \end{aligned}$$

Source: Borrowed from Dr. Zheng's slide show (ECE 522 lecture 18).

x(n) = discrete time signal (sample)

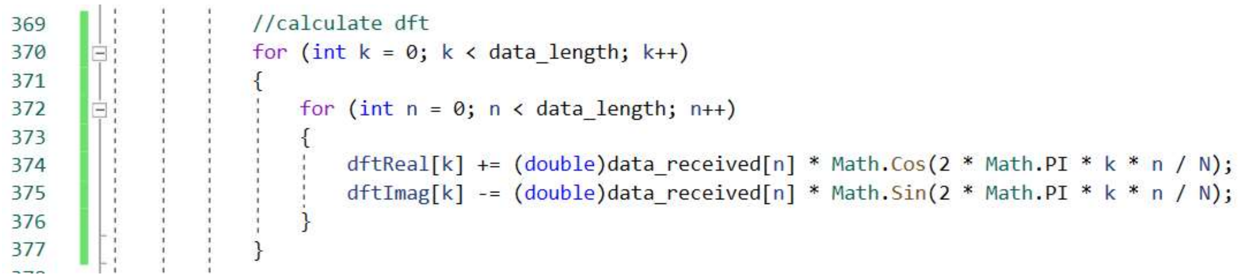
k = the number of the frequency domain point we are trying to calculate

N = number of total time domain samples (constant) n = index that loops through all the time domain samples to produce 1 single frequency domain point

From the equation above, we can write program that can implement this DFT equation. By analyzing the DFT equation, it can be concluded that 2 nested FOR loop will be needed to implement this.

Since the DFT equation required a lot of computations, it is implemented in C# as PIC24 will take a long time to implement this.

The below figure shows the use of the DFT formula in C#.



```
//calculate dft
for (int k = 0; k < data_length; k++)
{
    for (int n = 0; n < data_length; n++)
    {
        dftReal[k] += (double)data_received[n] * Math.Cos(2 * Math.PI * k * n / N);
        dftImag[k] -= (double)data_received[n] * Math.Sin(2 * Math.PI * k * n / N);
    }
}
```

Note that this program of the DFT was borrowed from Dr. Zheng's lecture 18 slide show.

How to find the frequency after performing DFT? Simply find the magnitude of the real and imaginary part, and loops through the 256 data points to determine which amplitude have the highest amplitude and keep tracks of the index of that highest amplitude. Then multiply the index by the frequency resolution to determine the frequency of the input signal.

Frequency resolution = fs/N; where fs is the sampling frequency

and N is the number of time samples (constant)

It is important to note that the DFT will produce a mirrored image of the frequency spectrum of the input signal. Therefore, we only need to use the first half of the magnitude array.

This also points out a very important concept of DSP regarding the sampling frequency. From the sampling theorem, we must have a sampling frequency twice that of the input signal that we are trying to digitize. So, if our sampling frequency is 10 kHz, and the input signal is 8 kHz the DFT will not be able to correctly identify this input signal's frequency. In this example the DFT can only successfully and correctly identify frequency that is below or equal to 5 kHz.