Programming a Waveform Generator On a PIC18 F87K22 Microprocessor

I. O. ADEWUMI

***Abstract*— The body of this report explores the use of a PIC18 microcontroller, mounted on an EasyPIC Pro 7 development board to construct a simple signal generator capable of creating 3 basic waveforms, a sine, a sawtooth and a square wave with variable frequency and amplitude. Each waveform could be given a frequency of 1, 0.25, 0.125, 0.0625 and 0.03125 Hz using inputs 4, 5, 6, 7, 8, and 9 from the keypad, and varied in amplitude from 0 V to 5 V using inputs from 2 of the EasyPIC Pro's inbuilt potentiometers. The output signal was read off a Digital to Analog Converter using a Picoscope 2000.**

I. Introduction

SIGNAL generation and processing has been the centre of much of the world's technological advancements over the past century. The Information age relied heavily on the introduction and development of computers which at the lowest level are often described as devices constructed mainly from transistors which process information via the transmission and reception of electrical signals through complex circuitry. Moore's Law [1] states that in microelectronics the number of transistor components which can be fitted into a dense area of integrated circuitry doubles every two years. Since his posit in 1975, Moore's law has held true, and in some years, even exceeded by real-world advancements in computing technology.

This unparalleled growth has meant that computers, from their advent until present day, have experienced a substantial reduction in size, accompanied by the phenomenal increase in computing power and capacity. The first computers [2] created only less than a century ago (mainframes) were the size of entire rooms. Today, we have programmable computers (smaller than a human palm), called microcontrollers [3]. Although nano-computers of size less than a few microns have also been developed, microcontrollers are much more common. Their ease of access, affordability, and high level of integrability into many other electrical circuit environments, makes them the heart and thrust of most micro-computing applications today.

This report details a simple 3-type waveform signal generator which was programmed from scratch onto a PIC18 microprocessor [4] using the low-level language Assembly.

As hardware, the product constitutes a few elementary electrical components which include 2 potentiometers, an analog-to-digital converter, digital-to-analog converter, and a keypad. The product's operating software was developed within MPLAB-X IDE [5], which compiles written Assembly code into machine language before writing it to the PIC18 microprocessor.

II. High Level Design

The wave-generator permitted a variable amplitude and vertical offset shifting, as well as adjustable wave frequencies. These frequency adjustments were implemented by varying the clock speed of the EasyPIC Pro 7's internal timers. A multiplexed (16-keyed) keypad was used to trigger these clock speed alterations, with different clock speed instructions assigned to each of 6 keys numbered 4 to 9. At the highest level, the main code of the product runs in a loop. This writes the values which have been read off the ADC potentiometers, and those keyed in from the keypad onto a 2x16 hexadecimal LCD mounted on the EasyPIC Pro 7 development board.

These values were read via an 'interrupt' of the main-code loop. The Interrupt branches off the main-code loop to perform a multi-step process that executes the generator's primary function of generating electrical wave signals. This is done using parameters read at each 'Interrupt' from the external input devices. The modular diagram (Fig. 1) provides a visual representation of the different segments (modules) of the code at their varying levels of complexity, reading down-up, and time of execution, reading left to right.

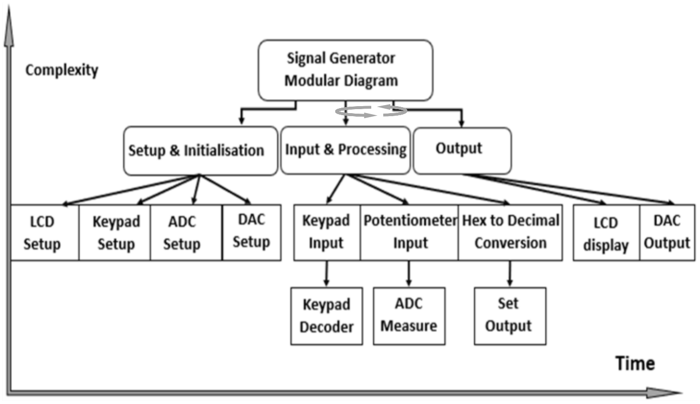

Fig. 1. Shows the signal generator modular diagram. Designed in 3 major segments, a setup stage, an input and processing stage and an output (signals) stage. The cyclic arrows around the processing and output stages show that these stages run in a continuous loop whilst the product is functioning.

III. Hardware & Software

A. *Design Overview*

As the PIC18 functions at a very low level, seamless transitions between software and hardware were required during the construction of our device. Indeed, both regimes are not always readily separable, however, handling these 2 regions as separate parts within this report provides leeway for more in-depth analyses of our product and its components.

B. Hardware

In terms of hardware, our product constitutes 5 key components: a digital-to-analog converter (DAC), an analog-to-digital converter ADC, a multiplexed 4x4 16-keyed keypad, a 2x16 LCD screen, and a PC oscilloscope.

1) DAC

The major component of the hardware set-up is the Digital-to-Analog converter. This receives binary-digit data values for the respective waveforms from the PIC18 via the Port J of the EasyPic Pro 7 board. It then converts these, within its internal structure (comprised of an 8-bit long R-2R ladder network) to electrical analog signals. The circuit diagram Fig. 2 [6] gives a good visual representation of this setup.

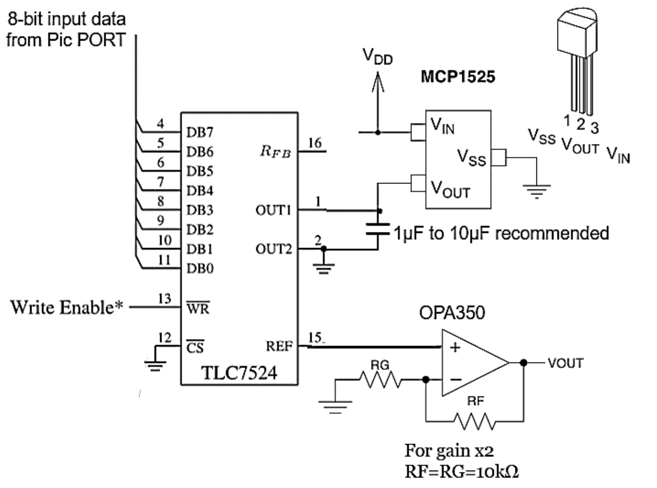

Fig. 2. Adopted from Mark Neil's Lectures on Microprocessors at Imperial College London. Shows the circuit diagram configuration for the (digital to analog converter) used in our product design-construction. The Write WR and CS pins were hardwired low, as just one DAC was utilized.

As shown in the circuit diagram, the TLC7524 Digital-to-analog converter has an 8-bit parallel data bus, connected to 8 of its input data pins. These transfer 8-bits of data to the DAC. As a voltage reference the DAC is connected in an inverted configuration with the driving voltage plugged into its V-out pin. The DAC could only receive a maximum voltage of 2.5V, so a voltage-dividing source MCPI525 was used. To smooth out the current flowing from this source a 10 µF capacitor is used, connected as shown in Fig. 2.

The requirement to reduce the source voltage from 5-2.5 V meant that~~,~~ to restore the voltage to its original level of 5 V, on output from the DAC, the voltage needed x2 amplification by a non-inverting operational amplifier with a gain of 2. These specific prerequisite conditions (and tools available to us) informed our decision to set-up the DAC in the inverted configuration. The opamp used was an OPA350 connected to two 10-ohm resistors as shown in the circuit diagram (2).

2) ADC

In our setup the Analog-to-Digital converter exists within the internal workings of the EasyPIC-Pro 7 development board and is enabled via the PIC18 program we developed. As a factory default, the EasyPIC Pro 7 board has 2 potentiometer settings; one ranges from 0 V at the negative input to +4.096 V at the positive input and another going from 0 V to +2.048 V at the positive end [7]. The first of these was chosen for our product, as this best suited our requirements for the signal generator. Choosing this voltage range also meant that a wider voltage-range from the 0 V to 5 V of the EasyPIC Pro 7's dev board could be used.

The variable voltage within this range is adjusted and measured from two of the board's inbuilt potentiometers with the 5 V maximum (the source voltage of the board) exceeding the preset potentiometer range of (0 - 4.096 V).

3) Keypad

The keypad's operational code was programmed from scratch using 2 major techniques; a multiplexing technique to store binary information of what keys have been pressed, and a decoder to map those binary values to the corresponding hexadecimal values which represent their letters or numbers. Finally, the keypad subroutine writes these values to a defined register termed 'keyvalue', to be called on by other subroutines for later use during the information processing stage.

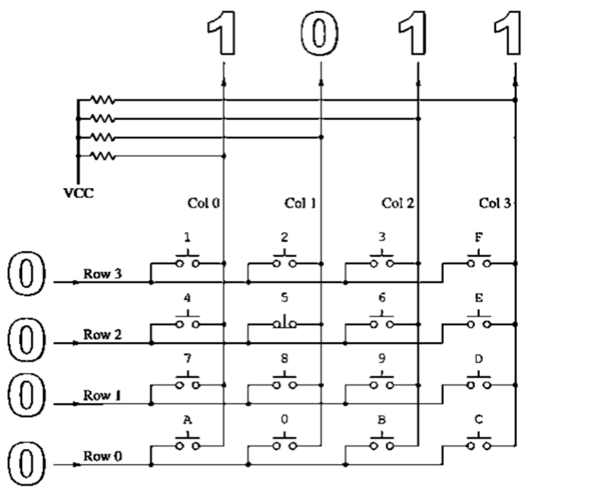

Fig. 3. Shows a diagram for the process of multiplexing a 4x4, 16-key keypad using an 8-bit parallel bus. The image corresponds to the hex value 0x0f being written to the keypad to raise all column inputs and lower the rows. The switch at button 5 is closed, corresponding to a button-press which pulls down the voltage of this column. The converse is done for determining the button row (0xf0 is sent to the keypad instead). This image was adopted from Mark Neil's Lectures on Microprocessors at Imperial College London.

Multiplexing was required because, although there were 16 keys available for use, the presence of only an 8-bit wide parallel data-bus (as well as the keypad having only 8 inputs) meant that only 8 bits of data could be used simultaneously for information transfer at any given point in time. The method of achieving 16-bit (16 keys) information transfer using multiplexing is as follows. The values 0x0f and 0xf0 are intermittently sent to the keypad via the 8bit bus to periodically raise all the rows high, keeping columns low and vice-versa (shown, for columns, in Fig. 3 [8]).

The general concept is that the transistors within each key pull its (row or column) value from high to low. Each pressed key would have a unique row and column combination. This way, 2 consecutive bytes of data; one containing information about a pushed key's row, and the other its column, are sent from the keypad back to the PIC18 for processing. Since each key has a unique row and column combination, each unique key pressed could be determined by reading off the 2-bytes of data within our code, combining both bytes with an inclusive-or function and storing the result in a register for decoding.

The decoder for the keypad was also written from scratch. Within this program, each numbered key was assigned a different function. The key numbered 1 called the saw-tooth wave, 2 the square wave, and 3 the sine wave. Keys 4 - 9 varied the frequency of the waves by altering the internal clock speed of the EasyPIC Pro 7.

4) *LCD*

To display the input parameters for the waveform a 2x16 (decimal-digit Liquid Crystal Display by Mikroelectronika was used. The LCD was mounted as shown in Fig. 4. The right-most values identified the waveform chosen (e.g., ‘squ’ for the square wave form), the middle digit the wave amplitude (set by the Port A potentiometer) and the last; the vertical offset (configured using the Port F potentiometer).

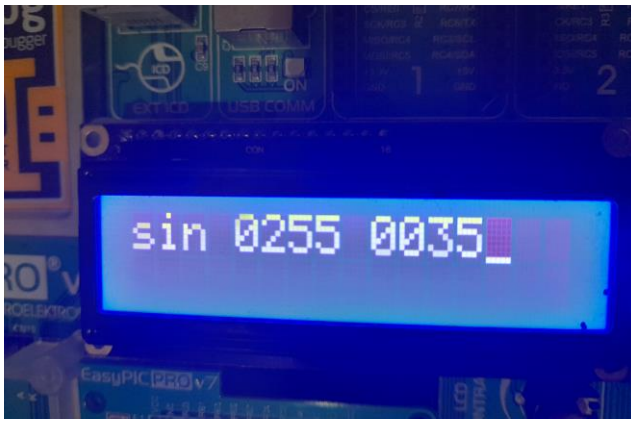

Fig. 4. Shows the LCD display during operation of the signal generating device. As can be seen by the image, a sinusoidal waveform, indicated by sin, has been selected, and the amplitude has been set to the maximum value 255 with the Port A potentiometer, whilst the vertical offset value

5) *PC Oscilloscope*

The output wave signal being transmitted by the DAC was measured using a PC oscilloscope by Pico technology, from the Picoscope 2000 series. The output cable from the V-out port of the OPA350 opamp was connected to the Analog input port of the Picoscope 2000 via an analogue cable. The Picoscope itself needed to be connected using a USB cable to a PC and signals were read off the PC using the Picoscope 2000's graphical user interface. A few images recorded from the Picoscope are shown in the Performance section of this report.

C. *Software*

1) *Main file*

As stated in the High-level Design section, the main part of our program, found within the ‘main.s' file contains two main parts. The first is a set-up subroutine, which calls the setup subroutines for the Keypad, LCD, ADC-potentiometers, and DAC Interrupt once at the start of the code. The second is a measure loop subroutine.

This subfunction first calls on another subroutine created, named 'Keypad_getKey' for determining what key on the keypad has been most recently pressed, and storing the information of this key within a register called 'keyvalue' in the PIC18 access memory. After this the 'measure_loop' subroutine moves the contents of the 'keyvalue' register to the W register where it is used by the 'wavedis' subroutine to write the first 3 letters of the correct waveform to the LCD, for example 'squ' is written for the square wave and 'sin' for sinusoidal. This LCD write is followed by a space ' ' written to the LCD. The subroutine then calls the first potentiometer subroutine named 'potentiometer1' to read off its manually set value and writes this to the LCD, also followed-up by a space, just as in the previous keypad case.

Next, it repeats the process with the second, 'potentiometer2'. After a fixed delay time, it calls the ‘LCD_ClearScreen’ instruction before then branching back to the start of the loop. This loop continues indefinitely until either the program is paused, or the device is switched off, and so the LCD screen appears to display a constant screen (Fig. 4), which only changes when keys are pressed. When the keys corresponding to frequency changes are pressed the screen flickers to a display with the selected clock frequency as shown below (Fig. 5) before rapidly reverting to the original (Fig. 4) with the waveform, amplitude, and offset.

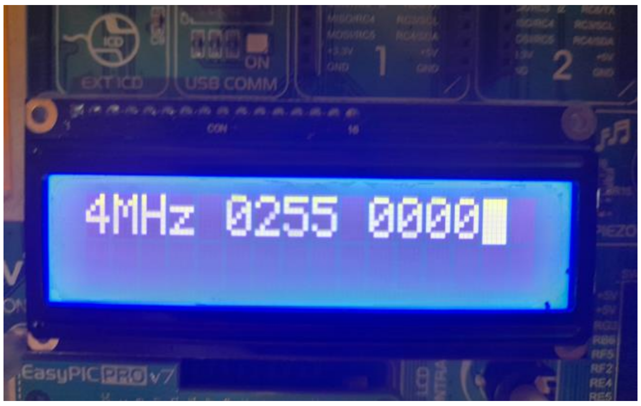

Fig. 5. Shows the display which the LCD momentarily switches to when a key instructing a frequency change is pressed (or held down). In this instant it shows 4MHz which corresponds to the key 4 on the keypad.

2) *DAC Interrupt file*

Following the set-up stage in the 'main.s' file, the interrupt is initiated using the EasyPIC-Pro's Timer0 clock counter with a timer0 pre-scaler of 1/16. This corresponds to a clock frequency of 1 Hz since the clock on the board’s natural oscillation frequency is 64MHz (translating to 16 million instruction-cycles per second at 4 clock-oscillations per instruction). Within the interrupt, the first subroutine called is the ‘check_keyvalue’ function. This determines what key has most recently been pressed, by comparing it sequentially against the list of possible values. Once a match is found, the corresponding signal-wave generating function is called.

3) *Waveforms*

Our code contains three 3 simple waveforms within its function generator library. A simple square wave function, a sawtooth wave generator, and a sinusoidal wave generator. The sawtooth and square waves operate using similar counting methods.

a. *Square wave*

The square wave works using 2 file registers as counters (named: sqcon1 and sqcon2). These are filled with the initial values 0x1d and 0x1F respectively. The function iteratively decrements the first file register within a subroutine called 'Highs' whilst simultaneously writing the value of the square-wave peak (its vertical offset added to its amplitude) to Port J. When the counter reaches zero, based on a set condition it branches to the subroutine for the square trough termed 'Lows' which then sends the value of the square-wave trough (the vertical offset alone) to the Port J whilst simultaneously decrementing the 2nd counter sqcon2. When this reaches zero it branches to reset both counters to their maximum values and restarts the process. The square wave was defined this way so that at default operation (before any frequency changes) it has a period of a second.

b. Sawtooth wave

The Sawtooth wave works using a similar method of incrementation. The maximum value of the Sawtooth is set as the value of the vertical offset added to that of the amplitude. The program then increments the Port J value until its value equals this maximum, before then resetting the Port J by sending it the value of the offset alone (the wave trough value). As a result of defining the sawtooth wave this way, its frequency has been inextricably linked to the amplitude and vertical offset which are manually determined by the 2 potentiometers. This challenge is discussed further in the improvements segment of this report.

To prevent wrapping around of both the square and sawtooth waveforms (that is; the maximum-sum value of vertical-offset with amplitude overshooting the allowed Port J range of 0xFF) a variable limit on the maximum vertical offset value was defined using the following condition.

If the difference between the maximum range 0xFF and the selected wave amplitude, exceeded the size of the offset, the offset was reset to equal this difference.

c. Sinusoidal wave

The sine waves were defined using a lookup table with predefined values for sine waves of different amplitudes. These values were gotten online [9].

To determine what sine wave is to be plotted, the subroutine we defined and named 'checksine' is used. In the 'checksine' subroutine, the two most significant bits of the measured voltage value from the amplitude potentiometer are compared against incrementing 0x05 intervals ranging from value 0x05 to 0xf5. For each of these, if the measured value is within a specific 0x05 range (for example 0xa0 - 0xa5), it calls and plots the sine-table for the sine wave which has amplitude equivalent to the higher value; in this case 0xa5. The range of values 0x05 - 0xf5 for defining sine amplitudes separated each by 0x05 required a total number of 32 sine-wave tables. These were stored as data in sine lookup-tables which we programmed onto the PIC18.

For the sine waves, wrapping around was avoided in a similar way as with the sawtooth and square waves, but with pre-defined limits for the vertical offset, of each specific amplitude.

IV. PERFORMANCE

Overall, the signal generator functions satisfactorily with the most significant error being the delay, approximately 5 s long, which occurs whenever the sine function is being called for the first time after the device has just been switched on and the program initialized. It occurs, regardless of if the sine wave function is the first to be called or not. However, after initialization, it ceases. We suspect this may be due to the sine function generator being much larger in size, relative to the other two, and thus having a longer initialization time.

The plot **Fig. 6** shows a switch from a square wave to a Sinusoidal waveform, after the keys; 2 (to initialize the square wave) and 3 to switch to a sine wave have been pressed.

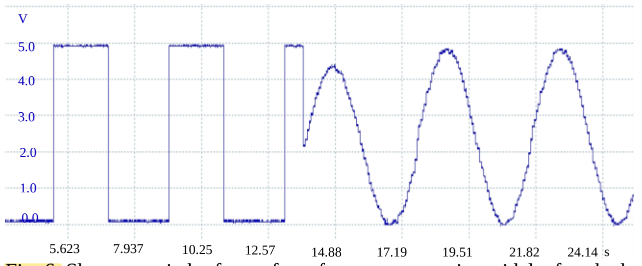

Fig. 6. Shows a switch of waveform from square to sinusoidal, after the key 3 on the keypad calling the sine function is pressed. Both waves are operating at default frequency 2.5 Hz.

Whilst programming and testing the 'checksine' lookup function for the sine waves, we observed that the oscilloscope plot for the 0x05 amplitude sine wave produced a waveform almost indistinguishable from the general electrical noise fluctuations. To a reasonable estimate we could therefore assume that the resolution of our signal generator was approximately equal to the magnitude of the noise fluctuations, and that these fluctuations had a magnitude of about 0x05, corresponding to 151 mV. Indeed, all other wave plots showed similar line thicknesses (resolutions) of about 151 mV.

The figures (7, 8 and 9), below show plots of the signal generator's response following various input commands. Alongside these plots are their respective descriptions

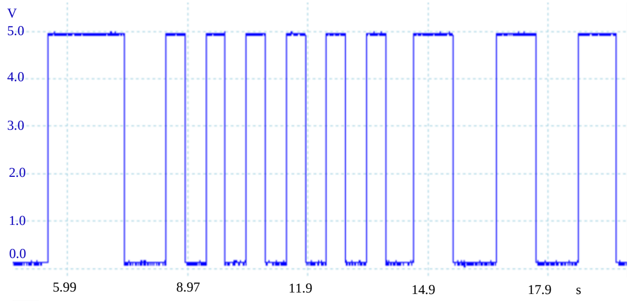

Fig. 7. Shows the frequency of a square wave generated by the signal generator being changed from a frequency of 0.25 Hz to 1 Hz and then to 0.5 Hz, following the keypad presses, 4 and 5.

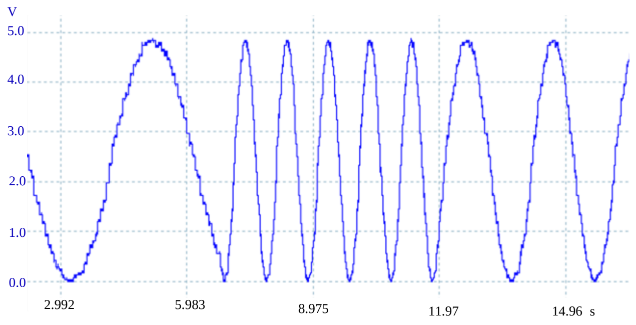

Fig. 8. Shows the frequency of a generated sine wave being altered from a frequency pf 0.25 Hz to 1 Hz and then to 0.5 Hz following the keypad presses, 4 and 5.

In the first 2 figures 7, 8 the frequency of the sine and square waves are shown switching from their default 0.25 Hz to a frequency of 1 Hz and then 0.5 Hz. The latter 2 frequencies match the pre-scaled clock speeds of 4 MHz and 2 MHz, respectively. These frequency responses followed the pressing of the keys numbered 4 and 5 on the keypad. Further toggles between the EasyPIC's clock frequencies 1, 0.5, 0.250 and 0.125 MHz (wave frequencies: 0.25, 0.125, 0.0625, 0.03125 Hz) could be executed by pressing the keys 6, 7, 8 and 9, respectively.

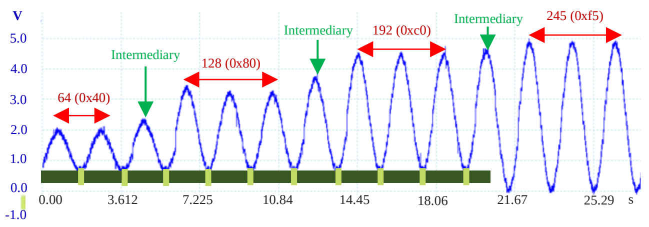

Fig. 9. Shows a sine wave with low offset shifting through 4 different amplitude regions of amplitude values 64(0x40), 128(0x80), 192(0xc0) and 245 (0xf5) indicated by the horizontal red double-headed arrows. The vertical dark-green arrows point to peaks with intermediary amplitude which occur whilst switching between the chosen desired amplitudes listed above. The horizontal striped dark-green line marks the fixed vertical offset which diminishes automatically when the sine amplitude is raised to a peak amplitude of 245 from 192.

This final plot shows the steady raising of the sinusoidal wave's amplitude through different amplitude regimes. The red horizontal arrows connect peaks of similar amplitude. The peaks in-between these horizontal arrows are amplitudes between the chosen discreet points of focus detailed below. The wave starts from amplitude 0x40 (which corresponds to a value of 64 on the LCD's predefined amplitude scale 0-255, and an electrical signal magnitude of about 1.25 V). It then switches through the amplitudes 0x80 (128 LCD), 0xc0 (192 LCD) and 0xf5 (245 LCD) gradually. These values correspond to 2.5 V, 3.75 V and 4.86 V, respectively.

During this process it was noted that both potentiometers of the EasyPIC pro 7 board were quite temperamental, so it was difficult to set the potentiometers precisely to the values above. In a number of instances, when the potentiometers are set to either extreme (maximum - 255 or minimum - 0) their faulty contacts cause a physically (electrically) induced wrapping around. This wrapping around from value 255 to 0 or the converse shows up on the wave plots of the PC oscilloscope. An improvement for overcoming this problem is discussed in the Improvements section.

As stated in the 'Software' section, the saw wave produced has a frequency coupled to the magnitude of its offset and amplitude. This, as previously hinted, is due to the way the waveform is defined. The wave is defined using incrementation of the file register Port J. The Port value is incremented until it reaches a magnitude equivalent to the sum of the wave amplitude and offset, before being reset to its trough position. Since the development board's clock speed is fixed at every predefined frequency, this means the time taken to complete these incrementation cycles is purely dependent on the magnitude of this sum (the vertical distance between wave-peak and trough).

For a wave signal generator, it is usually desirable that it have the capacity to definitively measure and set the frequency for all its periodic waveforms. Thus, the sawtooth problem had to be overcome. A few possible approaches to overcome this downside of our sawtooth waveform are looked at in the 'Improvements' section."

V. Improvements

As this signal generator was merely a proof-of-concept model, various improvements can be made on its design to rectify some of its pitfalls and give it added functionality.

First, as a rectification to the problem of the sawtooth frequency coupling to its wave amplitude and offset, the sawtooth function generator could be designed similarly to the sine wave generator; by using look-up tables for each amplitude; and thus, making the defined frequencies independent of amplitude and offset; since the number of data points within each period would always remain constant and wave-speed would be directly proportional to clock speed.

Another method for rectifying this problem was explored by my lab partner. Although more computationally intensive, it worked satisfactorily. In her approach she defined 4 different amplitude regimes. To each she assigned just a single triangle wave amplitude. Each of these 4 amplitudes, based on their vertical distance (peak to trough), were incremented by (amplitude-specific) values. These amplitude-specific values resulted in the number of steps taken to step through from trough to peak being equal across all 4 amplitudes. This is analogous to having the fixed number of 61 points within each wave-cycle of the predefined sawtooth wave lookup tables proposed earlier, and so achieves the same result. The subroutine for this is included within the 'dacc.s' (DAC Interrupt file) of our code, as an alternative to the original sawtooth wave, and is named 'sawtoothdis'.

As stated earlier, we had issues with the imprecision of the potentiometer knobs on the board. Since the board's ADC also accepts external input, better external potentiometers could be connected for use in place of the internal ones to increase our products functionality.

Other more general improvements which could be made include the addition of more complex waveforms within our signal generator since each waveform (like the sine waves) can be predefined and stored as data within lookup tables on the PIC18 microcontroller. More complex signal manipulation and processing such as wave amplitude modulations, linear multiplications additions and subtractions could also be programmed onto the PIC18 microprocessor giving it extra signal processing functionality. In theory the limits to what kind of wave processing can be achieved will rely on the total internal memory available within the microprocessor and its computational speed.

VI. Conclusion

We set out to develop a functioning wave signal generator which demonstrated the capacity to produce 3 of the most basic waveforms, sinusoidal, square, and sawtooth with controlled frequency and amplitude. This was achieved with adequate functionality. Each waveform could be given a frequency of *1, 0.25, 0.125, 0.0625 and 0.03125 Hz* using inputs from the keypad, as well as varied in amplitude from *0 V to 5 V* using potentiometer inputs. The amplitude of the square wave could be varied smoothly, (with amplitude incrementations of 0x01) and the sine wave's amplitude also appeared to vary quite smoothly when observed by eye. Within the program the sine wave was composed of 32 amplitudes ranging from hex value 0x05 to 0xf5 with spaced intervals of size 0x05. The improved sawtooth wave had an independent variable frequency but was only defined at 4 amplitudes. To overcome the issue of the waveform 'wrapping around' at large amplitudes and offsets the device was programmed with added functions which adjusted the vertical offset appropriately when the set amplitude and offset risked the wave overshooting the permitted range of values.

Looking forward, exploring future developments on the design model and its computational capacity for wave manipulation would provide an interesting research route for uncovering how powerful the PIC18 device can be at handling more modern and complex signal processing operations.

References

[1] Intel, “Over 50 Years of Moore’s Law,” Accessed: 4th April. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html

[2] Christopher Tozzi, “Mainframe History: How Mainframes Have Changed Over the Years,” Accessed: 24th April. 2021. [Online]. Available: https://www.precisely.com/blog/mainframe/mainframe-history

[3] Ben Lutkevich, “microcontrollers (MCU),” Accessed 4th April 2021. [Online] Available: https://internetofthingsagenda.techtarget.com/definition/microcontroller

[4] Microchip Technology, “PIC18F Product Family”, Accessed 24th April 2021. [Online]. Available: https://www.microchip.com/design-centers/motor-control-and-drive/motor-control-products/8-bit-mcus/pic18f

[5] Microchip Technology, “MPLAB X Integrated Development Environment”, Accessed 4th April 2021. [Online]. Available: https://www.microchip.com/mplab/mplab-x-ide

[6] Mark Neil, “Microprocessors – Lecture 10; Timers, Interrupts and DAC,” Dept. of Physics, Imperial College London, London, England, 2021.

[7] Nebojsa Matic, “EasyPIC PRO v7 - USER’S GUIDE”, pp. 32, MikroElektronika, 2013.

[8] Mark Neil, “Microprocessors – Lecture 8; Keypad,” Dept. of Physics, Imperial College London, London, England, 2021.

[9] Daycounter-Engineering Resources, “Sine Lookup Table Generator Calculator,” Accessed: 10th April 2021. [Online]. Available: https://www.daycounter.com/Calculators/Sine-Generator-Calculator.phtml