

Division of Engineering Programs

Senior Design Project Report

Spring 2021

May 3, 2021

Pure Sinusoidal Wave Inverter/Single to Three Phase Power Convertor

|  |  |
| --- | --- |
| **Student Name** | **Student Major** |
| Caroline Kucher | Electrical Engineering |
| Nikiforas Fokas | Electrical Engineering |
| Matthew Smith | Electrical Engineering |

|  |  |
| --- | --- |
| **Advisor name** | **Affiliation** |
| Mike Otis | SUNY New Paltz |

**Abstract**

There are two main types of ways to power electrical devices: by using a Direct Current (DC) source, or by using an Alternating Current (AC) source. In some cases, the source that the device calls for is not readily available. For example, in systems like vehicles, the main source of electricity is from the car’s battery which can only provide DC. The only way to provide current to devices that run on AC is to use a device such as an inverter which converts DC to AC. Typically, this is done using a modified square wave but for some very sensitive devices, this will not work. Another issue involved with powering devices is that not only do some devices run on AC, but they run on a special type of current known as three-phase AC. This project contains three parts: generation of a pure sinusoidal wave from a DC source, splitting this single-phase generated waveform into a three-phase balanced system, and filtering and amplifying the signal to drive a load. The first stage of the system is a variable frequency, variable voltage, single-phase AC power supply that utilizes a STM32 microprocessor with supporting circuitry to generate a sinusoidal waveform. The second stage of the system is a phase splitter that transforms the single-phase AC input with a phase angle of into a three-phase balanced output with phase angles , , and where the output phase angles can also be changed anywhere between andper phase. The third stage of the system incorporates a filtration design that includes a low pass filter at the output to filter out unwanted frequencies higher than 1k Hz as well as a power amplifier to allow higher voltage devices to be powered by the system. This system ensures that the design provides single-phase and three-phase sinusoidal waveforms for all types of AC devices and does so in a cost effective manner.

**Table of Contents**

1. About This Project…………………………………………………………………….5

2. Functional Description of The Design…………………………………………….......6

3. Preliminary Design and Results………………………………………………...……..7

3.1. Single-Phase AC Waveform Generation..………………………………….. 7

3.2. Single-Phase to Three-Phase………………………………………………...8

3.3. Three-Phase Data Acquisition……………………………………………….8

3.4. Power Amplification………………………………………….……………..9

3.5. Preliminary Results…………………………………………………....…….9

4. Final Design and Results………………………………………………………..……..10

4.1. Single-Phase AC Waveform Generation…...………………………………. 10

4.2. Emitter Follower Buffer……………………………………………………. 11

4.3. Single-Phase to Three-Phase……………………………………………….. 11

4.3.1. Motorized Potentiometer…………………………………………. 11

4.4. Three-Phase Data Acquisition and Control System……………………….. 12

4.4.1. Sine Wave to Digital Signal Conversion……………………….... 13

4.5. Power Amplification………………………………………………………...13

5. Recommendations for Future Work……………………………………………….......14

6. Conclusions……………………………………………………………………….…...14

7. Bibliography………………………………………………………………………..….15

8. Disciplines Used in This Project……………………………………………………….16

9. Design Constraints That Drove This Project………………………………………......16

10. Engineering Standards Used in This Project……………………………………....….17

11. New Skills Acquired During This Project…………………………………….……...17

12. Appendix………………………………………………………………………...…....18

A: Code for STM32 AC Waveform Generation………………………………...18

B: Code for Data Acquisition and Control System……………………………...31

1. **About This Project**

The motivation for the team members to undertake this project came from the project's complexity due to it’s the broad scope of engineering disciplines learned over the academic years. The team members wanted to challenge themselves and to incorporate a project that involved many disciplines from their electrical engineering course work. One of the team members, Niki, thought of the idea of this challenging project in the spring of 2020.

The problem is that AC power is not always readily available and can be difficult to generate from DC sources. The design objective is to build a pure sinusoidal wave inverter and phase splitter with a filter and power amplifier. This design answers this problem as this project provides single-phase as well as three-phase sinusoidal waveforms to power AC devices from a DC source.

This project costs less than $200 and provides multiple sinusoidal waveforms within an accuracy of 10%. This design is ideal for an individual that has a device that requires AC power and only has a DC power supply. This design is not financially burdensome and will provide extremely accurate solutions which will not damage the user’s device.

This report will cover the block diagram and functionality of each part of the design, all preliminary designs and results, the final design and results, recommendations for future additions to the project, disciplines used in the project, design constraints of the project, engineering standards used, new knowledge gained, as well as data sheets and codes used in the project.

**2. Functional Description of The Design**

The following diagram illustrates the block diagram of the overall project. In the left-most block, a DC source is turned into an AC supply that can handle a variety of loads. In the center block, the generated single-phase AC waveform is split into three phases. On the bottom of the center block, the sinusoid is filtered to ensure low noise and minimal harmonics and is amplified in order to power devices that require high power. On the right-most block, the design is used to drive a three-phase load which is then fed into a closed-loop feedback system, shown in the top block, which senses the voltage of the waveform and is adjusted to meet the device requirement.

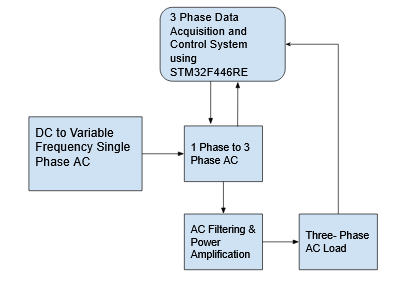


Figure 1. Block Diagram of Final Design.

**3. Preliminary Design and Results**

In this section, we will talk about the design we came up with in the very beginning and talk about what worked, what didn’t work, how we tested it, and how we came up with this initial idea in the first place.

Our goal was to:

* Generate a sinusoidally shaped AC waveform from a DC source.
* Design a way to split single-phase to three-phase.
* Design a way to automatically adjust phase angles in real time as the input frequency changes.
* Design a power amplifier capable of driving very low impedance loads. This amplifier must also have low output distortion.
* Combine all of the above into one unified system.

**3.1. Single Phase AC Waveform Generation**

* The onboard DAC buffer was difficult to calibrate so instead we opted to use an external op-amp buffer to ensure accuracy and limit DAC output current.
* The onboard DAC buffer was out of calibration so instead we opted to use an external op-amp buffer to ensure accuracy and limit DAC output current.
* Initially, we tried to set up the rotary encoder in timer encoder mode but we weren’t able to get that method working.
* Instead, we wrote a function to mimic encoder mode using an external interrupt and polling upon interrupt.
* This component was tested using an oscilloscope as well as digital multimeter.

**3.2. Single-Phase to Three-Phase**

* Two circuits, one that can shift a sine wave between 0° and 180°, and the other can shift between 0° and -180°.
* We used what we had on hand.
* Using prior experience in combination with some online and textbook research.
* The phase splitting worked, however, it created too much output noise.
* We tested this part by using a function generator to output a 1v 60 Hz sine wave and measuring the three output waveforms using a 4 Channel Oscilloscope. The scope can compute the phase angles.
* The three-phase output connects to the power amplifier.

**3.3. Three-Phase Data Acquisition**

* It measures and controls the phase angles of the three-phases. It will tune the potentiometers in single-phase to three-phase circuits based on frequency to maintain an angle of 120° between each phase.
* The Raspberry Pi is an easy to program platform with lots of compute power, the MCP3008 ADC has a fast sampling rate, has multiple channels, and can compare channels on the chip for full AC input.
* The design failed because the operating system on Raspberry Pi is Linux based which is not a real time operating system. Due to this, the measurements had poor precision and accuracy.
* We tested the data acquisition system by connecting a function generator outputting a 1v 60Hz sine wave to the MCP3008 adc which is connected to the Raspberry PI.
* The measured input would then be used to control the digital potentiometer to adjust the phase angle of the single-phase to three-phase circuit.

**3.4. Power Amplification**

* The power amplification stage uses the three-phase output as inputs so that its output can drive low impedance loads. This is necessary because the op-amp’s used can only output a maximum of 500mW. For example, if the phase 1 has an amplitude of 10 volts connected to a 10 ohm load and since P=V2/R the resulting power is 10W which is 20x what the op-amp can deliver on its own. The power amplifier stage takes the input power and multiplies it so it can maintain the same voltage regardless of the load impedance.
* We choose the components based on their power rating and cost.
* The LM833n op-amp was used in the design to drive the class B bjt based power amplifier. It worked but due to its internal construction it created a lot of output noise.
* The power amplification stage was tested using the three-phase output of the phase shifting circuit. The output and input were connected to an oscilloscope so we could compare them.

**3.5. Preliminary Results**

The advanced phase splitting circuit worked perfectly with a regular potentiometer but was incompatible with a digital potentiometer. The data acquisition and control system was not precise, accurate, or fast enough. The power amplification stage worked but the used op-amp needed to be replaced with one that has a different internal design.

**4. Final Design and Results**

In this section, we will talk about the final design we came up with and talk about what worked, what didn’t work, how we tested it, and our solution to what didn’t work.

* The Graphics LCD interface made changing settings very easy but needed to be properly debounced in order to get accurate results.
* The advanced phase splitting circuit worked very well.
* The LM358 op-amp in the phase splitting circuit was replaced with an LM833n.
* The digital potentiometer didn’t work so a custom electromechanical potentiometer had to be designed and fabricated.
* The control system had to be redesigned from the ground up.
* Converting a sine wave to a digital waveform was harder than expected; numerous revisions were made.
* Low speed TTL based chips were replaced with high speed CMOS based chips to reduce the propagation delay.

**4.1. Single Phase AC Waveform Generation**

The final design consisted of an ST7920 128x64 Graphics Liquid Crystal Display, interfaced to the STM32 board with a rotary encoder and pushbutton to vary the frequency and amplitude. The GLCD was run using SPI mode and the pushbutton and encoder utilized external interrupts. For the final design, OPA2134 op-amps were used due to their high signal to noise ratio and relatively cheap cost. The STM32 board was hooked up to external circuitry that smoothed the sine wave output as well as amplified the voltage slightly, and level shifted the circuit to be offset at ground. Our final design was wired up and encased in a 3D printed enclosure to make the component more compact and portable.

**4.2. Emitter Follower Buffer**

In the final design, emitter follower buffers were added between the input sine wave from DC to AC conversion and the inputs to the single-phase to three-phase circuit. These buffers were used to ensure that there is no interference between each phase. The design uses a 2n2222a BJT transistor, a 1n5819 Schottky diode, a capacitor and some resistors. The Schottky diode is connected from the base to the collector of the BJT, doing this ensures that the BJT will not go into saturation thus removing the need to bias the transistor. The parts were chosen based on what we had on hand. The design was simulated first using TINA and then constructed and tested with a function generator and oscilloscope.

**4.3. Single-Phase to Three-Phase**

The final design of the single-phase to three-phase system uses the advanced phase splitter design with a custom motorized potentiometer in place of the digital one in the preliminary design. The LM358n op-amp was replaced with a LM833n op-amp because it has a high slew rate, high operating supply voltage, and can output up to 500mW. This combination resulted in an output with little to no distortion. All the designs were simulated in TINA then tested using an oscilloscope.

**4.3.1. Motorized Potentiometer**

The motorized potentiometer design came about when the digital one didn’t work with an AC waveform and found out no such product exists. The design uses a readily available 12V DC motor with worm drive, a 10 turn 50k ohm rotary potentiometer, a 6mm shaft coupler, and a motor holder. The chosen worm drive is spec’d at 6 rpm and has a 6mm shaft, the potentiometer was chosen because it can be finely adjusted and has a 6.3mm shaft. The shaft coupler was chosen because it fit on the output shaft of the worm drive and with some slight modification it fit on the potentiometer shaft as well. The motor holder is a custom 3D printed part that was modelled in Autodesk Inventor 2020. An LM298n H-Bridge module was used to drive the motor, it was chosen because it’s cheap and easy to use.

**4.4. Three-Phase Data Acquisition and Control System**

The final design of the data acquisition and control system uses the STM32F446RE microcontroller programmed with MBED-OS. The STM32F446RE board was chosen because we used it in several other projects that require data acquisition and control. MBED was chosen because it is an open-source platform that supports a real time operating system (RTOS). This combination was chosen because a data acquisition system needs to be able to measure frequency and phase angles with high precision and accuracy. The control system needs to be capable of fast transient response with low steady state error, while also being able to provide feedback to the user. To measure the frequency as well as phase angles the sine wave outputs need to be converted to a digital signal. Once converted to a digital signal, we can measure the frequency by measuring the time between the rising and falling edges of phase 1 then doubling that value to obtain the period, and thus the frequency. To measure the phase angles, we needed to know the time delay between the rising edge of one phase with respect to another. The solution was to XOR the phases, doing this results in a signal where measuring the time between the rising and falling edges will give the time delay between two phases. By knowing the frequency and time delay the phase angle can be computed. This computation is done for phases 1-2, 1-3, and 2-3. The control system uses the measured phase angles and controls the two H-Bridge Drivers on the motorized potentiometers; the motors have three states rotate left, rotate right, and off. The controller takes the measured phase angle and checks if the measured is within .5% of the setpoint (in this case 120°). It will adjust up or down accordingly if not within the .5% otherwise it will turn the motors off. This system will update every 100uS and will send the frequency, current phase angles to a computer via USB.

**4.4.1. Sine Wave to Digital Signal Conversion**

Since the microcontroller and logic gate can only use digital signals, each phase needed to have a corresponding digital equivalent. This was done in two steps with the first being converting the sine wave to a square wave at the crossover points then level shifting said square wave resulting in a usable logic signal. The parts used for this are LM833n op-amps, 1n5819 Schottky diodes and some resistors. The op-amps are cheap and we had a lot on hand. The Schottky diodes were chosen for their low forward voltage and fast recovery time.

**4.5. Power Amplification**

The final design of the power amplifier is based on a Class B dual supply amplifier, but an op-amp is added to provide feedback to remove the crossover distortion that normally occurs with this design. The LM833n op-amp used in the preliminary design was replaced with a LM358. Other components used were the following TIP41C and TIP31C NPN power BJT in a Darlington configuration, as well as the complementary TIP42C and TIP31C PNP Power BJT’s also in a Darlington configuration. These were chosen for their low cost and high-power capabilities. When put together, the resulting power amplifier has excellent power gain, and very little output distortion

**Recommendations for Future Work**

Due to time constraints, the team was not able to put all of the circuitry onto one printed circuit board (PCB). This would be a great design addition as it would conceal most of the wires and it would make the overall design much smaller. Another recommendation would be to add on a variable DC power supply to the system so that the design could provide up to 250W DC power to make the design more versatile and able to power even more types of devices.

**Conclusion**

To summarize, this design provides a solution to providing single-phase as well as three-phase sinusoidal waveforms to power AC devices from a DC source in a reliable, cost-effective way. Overall, this project was a success, however, there is still potential to add onto this design and make it even better. Not only was this project a success, but it challenged the team and tied together many academic disciplines learned throughout their academic career and allowed them to applicably apply these disciplines.

**Bibliography**

[1] Globalspec.com. 2020. *Analog-To-Digital Converters Selection Guide | Engineering360*. [online] Available

at:<https://www.globalspec.com/learnmore/data\_acquisition\_signal\_conditioning/signal\_convert ing /analog\_to\_digital\_converters> [Accessed 6 October 2020].

[2]"Software generation of Three Phase Sinusoidal PWM", *Beingrealtimesystems.blogspot.com*, 2020. [Online]. Available:

http://beingrealtimesystems.blogspot.com/2014/08/software-generation-of-three-phase.html [Accessed: 06- Oct- 2020].

[3] T. Floyd, “RC Circuits.” [Online]. Available:

http://eon.sdsu.edu/~johnston/EE204\_PDF\_Slides/Chapters 8-10/CH10.pdf. [Accessed: 14-May-2020].

[4] “Non-inverting Operational Amplifier - The Non-inverting Op-amp,” *Basic Electronics Tutorials*, 24-Feb-2018. [Online]. Available:

https://www.electronics-tutorials.ws/opamp/opamp\_3.html. [Accessed: 15-May-2020]. [5] G Vijayaraghavan, Mark Brown, Malcolm Barnes. “Electrical noise and mitigation – Part 1: Noise definition, categories and measurement” *EE Times* [Online] Available: https://www.eetimes.com/electrical-noise-and-mitigation-part-1-noise-definition-categories-and measurement/# [Accessed: 1-October-2020]

[6] “Passive Low Pass Filters” *Basic Electronics Tutorials* [Online] Available: https://www.electronics-tutorials.ws/filter/filter\_2.html. [Accessed: 1-October-2020]

[7] “Introduction To The Amplifier” *Basic Electronics Tutorials* [Online] Available: https://www.electronics-tutorials.ws/amplifier/amp\_1.html. [Accessed:1-October-2020]

**Disciplines Used in This Project**

This project applied many of the ideas and theories learned from the engineering courses in the electrical and computer engineering disciplines at SUNY New Paltz. The design was heavily based on electronic circuit design and power electronics so the team used many concepts that were learned in classes like Circuits, Electronics 1, Electronics 2, and Energy Systems. Almost equally important was the computer programming aspect of the project which incorporated knowledge acquired from classes such as C Programming, Digital Logic, Microcontrollers, and Embedded Systems. Every component in the system required circuitry and every part except for the filtration system utilized an STM32 board. Another less prevalent area of coursework applied was signals and systems as well as control systems. There was a need for understanding how to use basic closed loop feedback and mathematically model our system as well. The course Technical Communications also assisted the team when it came time to submit documentation and give presentations during checkpoints throughout the project.

**Design Constraints That Drove This Project**

The design for this project had a few main driving constraints. The team needed the system to be able to produce a sinusoidal AC voltage from a DC source. The team needed the system to have an accuracy within 10% for the desired frequency, desired amplitudes, desired phase angles, and total harmonic distortion. The system is meant to be affordable so total design cost was not to exceed $200. Finally the design needed to be powered by an ATX 250W power supply or greater.

**Engineering Standards Used in This Project**

|  |  |
| --- | --- |
| **Engineering Standard** | **Where it was used** |
| **UL 817** | Power Supply Electrical Cord |
| **UL 1310** | Power Supply Wiring |
| **USB-IF ID# 46** | USB Cable type/rating |
| **ASTM D1351-14** | Polyethylene Wire Insulation |

**New Skills Acquired During This Project**

There were many new skills acquired during this project due to the circumstances surrounding this project as well as the actual difficulty of the design and implementation of this project. The team had to figure out how to efficiently share documents and videos on cloud platforms such as google drive so that every team member could contribute. The team became well versed in powerpoint as well as video editing since the final video had to be recorded in multiple different sections and compiled together. One of the most important skills that the team improved on constantly throughout the year was figuring out how to allocate the workload so that everyone was able to contribute and feel confident in the direction the design was heading. The gantt chart was a new tool used to help keep deadlines organized and the project flowing smoothly. As this was a project based heavily on coding and circuit design, team members became proficient in using circuit design software such as TINA TI to simulate various designs as well as troubleshooting tools for the programming portion of the project.

**Appendix**

Link to drive folder for parts datasheets:

<https://drive.google.com/drive/folders/1QjxbJt_J3zn9JfiAXnZpR-Cgw_PAFRhJ?usp=sharing>

**A: Code for STM32 Variable Sine Wave Generation:**

#include "main.h"

#include "delay.h"

#include "ST7920\_Serial.h"

#include <stdint.h>

#include "stm32f3xx\_hal.h"

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

DAC\_HandleTypeDef hdac1;

DMA\_HandleTypeDef hdma\_dac1\_ch1;

TIM\_HandleTypeDef htim1;

TIM\_HandleTypeDef htim4;

char MAIN[2];

int Button[2]={0,0};

int SubMenuCount=0;

int Clear=0;

int Scroll=0;

int ScrollOld=-1;

int ArrayElementOld=-1;

int ArrayElement;

int Loop=0;

int CountAmplitude[1]={0};

int CountFrequency[1]={0};

int Arrow=0;

int ArrowOld=1;

int Count=0;

int CountOld=0;

int PrevCount=0;

int PrevCountOld=0;

int MenuElementCount=0;

int MenuElementCountOld=0;

int State=0;

int StateOld;

int AnalogIn09=1;

int PA12=1;

int DACvalue[400]={2048,2080,2112,2144,2176,2208,2240,2272,2304,2336,2368,2400,2431,

2463,2494,2525,2557,2588,2619,2650,2680,2711,2741,2771,2801,2831,2861,2890,2919,2948,

2977,3006,3034,3062,3090,3117,3145,3172,3198,3225,3251,3277,3302,3328,3353,3377,3402,3425,3449,3472,3495,3518,3540,3562,3583,3604,3625,3645,3665,3685,3704,3723,3741,

3759,3776,3793,3810,3826,3842,3857,3872,3886,3900,3914,3927,3939,3951,3963,3974,3985,3995,4004,4014,4022,4031,4038,4046,4052,4059,4065,4070,4075,4079,4083,4086,4089,4091,4093,4094,4095,4095,4095,4094,4093,4091,4089,4086,4083,4079,4075,4070,4065,4059,4052,4046,4038,4031,4022,4014,4004,3995,3985,3974,3963,3951,3939,3927,3914,3900,3886,3872,3857,3842,3826,3810,3793,3776,3759,3741,3723,3704,3685,3665,3645,3625,3604,3583,3562,3540,3518,3495,3472,3449,3425,3402,3377,3353,3328,3302,3277,3251,3225,3198,3172,3145,3117,3090,3062,3034,3006,2977,2948,2919,2890,2861,2831,2801,2771,2741,2711,2680,2650,2619,2588,2557,2525,2494,2463,2431,2400,2368,2336,2304,2272,2240,2208,2176,2144,2112,2080,2048,2015,1983,1951,1919,1887,1855,1823,1791,1759,1727,1695,1664,1632,1601,1570,1538,1507,1476,1445,1415,1384,1354,1324,1294,1264,1234,1205,1176,1147,1118,1089,1061,1033,1005,978,950,923,897,870,844,818,793,767,742,718,693,670,646,623,600,577,555,533,

512,491,470,450,430,410,391,372,354,336,319,302,285,269,253,238,223,209,195,181,168,

156,144,132,121,110,100,91,81,73,64,57,49,43,36,30,25,20,16,12,9,6,4,2,1,0,0,0,1,2,4,6,9,12,

16,20,25,30,36,43,49,57,64,73,81,91,100,110,121,132,144,156,168,181,195,209,223,238,253,

269,285,302,319,336,354,372,391,410,430,450,470,491,512,533,555,577,600,623,646,670,

693,718,742,767,793,818,844,870,897,923,950,978,1005,1033,1061,1089,1118,1147,1176,

1205,1234,1264,1294,1324,1354,1384,1415,1445,1476,1507,1538,1570,1601,1632,1664,1695,1727,1759,1791,1823,1855,1887,1919,1951,1983,2015};

int DACvaluePlaceHolder[400]={2048,2080,2112,2144,2176,2208,2240,2272,2304,2336,2368,2400,2431,

2463,2494,2525,2557,2588,2619,2650,2680,2711,2741,2771,2801,2831,2861,2890,2919,2948,

2977,3006,3034,3062,3090,3117,3145,3172,3198,3225,3251,3277,3302,3328,3353,3377,3402,3425,3449,3472,3495,3518,3540,3562,3583,3604,3625,3645,3665,3685,3704,3723,3741,

3759,3776,3793,3810,3826,3842,3857,3872,3886,3900,3914,3927,3939,3951,3963,3974,3985,3995,4004,4014,4022,4031,4038,4046,4052,4059,4065,4070,4075,4079,4083,4086,4089,4091,4093,4094,4095,4095,4095,4094,4093,4091,4089,4086,4083,4079,4075,4070,4065,4059,4052,4046,4038,4031,4022,4014,4004,3995,3985,3974,3963,3951,3939,3927,3914,3900,3886,3872,3857,3842,3826,3810,3793,3776,3759,3741,3723,3704,3685,3665,3645,3625,3604,3583,3562,3540,3518,3495,3472,3449,3425,3402,3377,3353,3328,3302,3277,3251,3225,3198,3172,3145,3117,3090,3062,3034,3006,2977,2948,2919,2890,2861,2831,2801,2771,2741,2711,2680,2650,2619,2588,2557,2525,2494,2463,2431,2400,2368,2336,2304,2272,2240,2208,2176,2144,2112,2080,2048,2015,1983,1951,1919,1887,1855,1823,1791,1759,1727,1695,1664,1632,1601,1570,1538,1507,1476,1445,1415,1384,1354,1324,1294,1264,1234,1205,1176,1147,1118,1089,1061,1033,1005,978,950,923,897,870,844,818,793,767,742,718,693,670,646,623,600,577,555,533,

512,491,470,450,430,410,391,372,354,336,319,302,285,269,253,238,223,209,195,181,168,

156,144,132,121,110,100,91,81,73,64,57,49,43,36,30,25,20,16,12,9,6,4,2,1,0,0,0,1,2,4,6,9,12,

16,20,25,30,36,43,49,57,64,73,81,91,100,110,121,132,144,156,168,181,195,209,223,238,253,

269,285,302,319,336,354,372,391,410,430,450,470,491,512,533,555,577,600,623,646,670,

693,718,742,767,793,818,844,870,897,923,950,978,1005,1033,1061,1089,1118,1147,1176,

1205,1234,1264,1294,1324,1354,1384,1415,1445,1476,1507,1538,1570,1601,1632,1664,1695,1727,1759,1791,1823,1855,1887,1919,1951,1983,2015};

int PrevState=0;

int PrevStateOld;

int Six = 60;//Arrow ASCI

int Blank = 32;//Blank ASCI

int Element=2;

int Frequency=0;

int ARR;

float Amplitude=0;

int Loop\_if=0;

int Loopreturn=0;

void SystemClock\_Config(void);

static void MX\_GPIO\_Init(void);

static void MX\_DMA\_Init(void);

static void MX\_TIM4\_Init(void);

static void MX\_TIM1\_Init(void);

static void MX\_DAC1\_Init(void);

void Send\_Arrow(void);

void Calculate\_Arrow\_and\_Scroll(void);//int n =number of States, seven for State0-6

void Calculate\_Menu\_Element\_Count(void);//this c1 is used for menu Scrolling, input 2n-1

void Send\_Menu\_Item(void);

void Populate\_Menu\_Settings\_Names\_For\_LCD(void);

void Send\_Values(int Value1, int Value2);

void Send\_Units(void);

void Loop\_Return\_Check(void);

void Get\_State(void);

void Reset\_Variables(void);

void Save\_Variables(void);

void New\_Sin\_Amplitude(void);

void New\_Sin\_Frequency(void);

int main(void)

{

HAL\_Init();

SystemClock\_Config();

MX\_GPIO\_Init();

MX\_DMA\_Init();

MX\_TIM4\_Init();

MX\_TIM1\_Init();

MX\_DAC1\_Init();

delay\_init();

ST7920\_Init();

Populate\_Menu\_Settings\_Names\_For\_LCD();//sets up States to be displayed as names for //LCD

ST7920\_Clear();//Clears Old LCD

Send\_Menu\_Item();//converts number of Button presses to switch case State and sets State //upon interrupt or Scroll

Send\_Arrow();//sends Arrow position of menu

Get\_State();//helps with menu stuff

HAL\_DAC\_Start\_DMA(&hdac1,DAC\_CHANNEL\_1,&DACvalue,400,DAC\_ALIGN\_12B\_R);

HAL\_TIM\_Base\_Start(&htim4);

while (1)

{

if(Loop\_if==1)//if interrupt or Button press occurred

{

Send\_Menu\_Item(); //Controls which case to execute and sends necessary //information to display

Calculate\_Menu\_Element\_Count(); //calculates MenuElementCount value using Element

Get\_State(); //get State of menu

Calculate\_Arrow\_and\_Scroll(); // get Arrow\_and\_Scroll

Send\_Arrow(); // display Arrow

Loop\_Return\_Check(); //check if case requires another loop through

PrevState=State; //set previous state to state

Loop\_if=0; //disable loop starter

PrevCount=Count; //set previous state to state

}

if (AnalogIn09==0)//if Button pressed store whatever element cursor is on that specific //submenu

{

switch (SubMenuCount)

{

case 0:

Button[0]=State;

break;

case 1:

Button[1]=State;

break;

}

SubMenuCount++; // increase the variable that stores how many submenus in //you are

Save\_Variables(); //save variables

Reset\_Variables(); //reset variables

Loop\_if=1; //enable loop to send new information to display

AnalogIn09=1; //Signal button is not pressed again

}

}//end while

}//end main

void SystemClock\_Config(void)

{

RCC\_OscInitTypeDef RCC\_OscInitStruct = {0};

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

RCC\_PeriphCLKInitTypeDef PeriphClkInit = {0};

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSE;

RCC\_OscInitStruct.HSEState = RCC\_HSE\_ON;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_ON;

RCC\_OscInitStruct.PLL.PLLSource = RCC\_PLLSOURCE\_HSE;

RCC\_OscInitStruct.PLL.PLLMUL = RCC\_PLL\_MUL9;

RCC\_OscInitStruct.PLL.PREDIV = RCC\_PREDIV\_DIV1;

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK|RCC\_CLOCKTYPE\_SYSCLK

|RCC\_CLOCKTYPE\_PCLK1|RCC\_CLOCKTYPE\_PCLK2;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_PLLCLK;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV2;

RCC\_ClkInitStruct.APB2CLKDivider = RCC\_HCLK\_DIV1;

PeriphClkInit.PeriphClockSelection = RCC\_PERIPHCLK\_TIM1|RCC\_PERIPHCLK\_TIM34;

PeriphClkInit.Tim1ClockSelection = RCC\_TIM1CLK\_HCLK;

PeriphClkInit.Tim34ClockSelection = RCC\_TIM34CLK\_HCLK;

static void MX\_DAC1\_Init(void)

{

DAC\_ChannelConfTypeDef sConfig = {0};

hdac1.Instance = DAC1;

sConfig.DAC\_Trigger = DAC\_TRIGGER\_T4\_TRGO;

sConfig.DAC\_OutputBuffer = DAC\_OUTPUTBUFFER\_ENABLE;

}

static void MX\_TIM1\_Init(void)

{

TIM\_ClockConfigTypeDef sClockSourceConfig = {0};

TIM\_MasterConfigTypeDef sMasterConfig = {0};

htim1.Instance = TIM1;

htim1.Init.Prescaler = 72-1;

htim1.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim1.Init.Period = 65535;

htim1.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

htim1.Init.RepetitionCounter = 0;

htim1.Init.AutoReloadPreload = TIM\_AUTORELOAD\_PRELOAD\_DISABLE;

sClockSourceConfig.ClockSource = TIM\_CLOCKSOURCE\_INTERNAL;

sMasterConfig.MasterOutputTrigger = TIM\_TRGO\_RESET;

sMasterConfig.MasterOutputTrigger2 = TIM\_TRGO2\_RESET;

sMasterConfig.MasterSlaveMode = TIM\_MASTERSLAVEMODE\_DISABLE;

}

static void MX\_TIM4\_Init(void)

{

TIM\_ClockConfigTypeDef sClockSourceConfig = {0};

TIM\_MasterConfigTypeDef sMasterConfig = {0};

htim4.Instance = TIM4;

htim4.Init.Prescaler = 3-1;

htim4.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim4.Init.Period = 1000;

htim4.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

htim4.Init.AutoReloadPreload = TIM\_AUTORELOAD\_PRELOAD\_DISABLE;

sClockSourceConfig.ClockSource = TIM\_CLOCKSOURCE\_INTERNAL;

sMasterConfig.MasterOutputTrigger = TIM\_TRGO\_UPDATE;

sMasterConfig.MasterSlaveMode = TIM\_MASTERSLAVEMODE\_DISABLE;

}

static void MX\_DMA\_Init(void)

{

\_\_HAL\_RCC\_DMA1\_CLK\_ENABLE();

HAL\_NVIC\_SetPriority(DMA1\_Channel3\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(DMA1\_Channel3\_IRQn);

}

static void MX\_GPIO\_Init(void)

{

GPIO\_InitTypeDef GPIO\_InitStruct = {0};

\_\_HAL\_RCC\_GPIOF\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

HAL\_GPIO\_WritePin(GPIOA, GPIO\_PIN\_7|GPIO\_PIN\_8|GPIO\_PIN\_10, GPIO\_PIN\_RESET);

GPIO\_InitStruct.Pin = GPIO\_PIN\_7|GPIO\_PIN\_8|GPIO\_PIN\_10;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

GPIO\_InitStruct.Pin = GPIO\_PIN\_9|GPIO\_PIN\_11;

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_RISING;

GPIO\_InitStruct.Pull = GPIO\_PULLUP;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

GPIO\_InitStruct.Pin = GPIO\_PIN\_12;

GPIO\_InitStruct.Mode = GPIO\_MODE\_INPUT;

GPIO\_InitStruct.Pull = GPIO\_PULLUP;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

HAL\_NVIC\_SetPriority(EXTI9\_5\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(EXTI9\_5\_IRQn);

HAL\_NVIC\_SetPriority(EXTI15\_10\_IRQn, 0, 0);

HAL\_NVIC\_EnableIRQ(EXTI15\_10\_IRQn);

}

void EXTI9\_5\_IRQHandler(void)

{

AnalogIn09=0;

for (int v=0; v<=1000000;v++)//delay slightly to stop bouncing

{;}

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_9);

}

void EXTI15\_10\_IRQHandler(void)

{

PA12= HAL\_GPIO\_ReadPin(GPIOA, GPIO\_PIN\_12);

if (PA12==1)

{Count++;}

else

{Count--;}

if (Count>300)

{Count=300;}

if (Count<0)

{Count=0;}

//this code above keeps count between 0 and 300 since 300 is max value that will be counted to

for (int j=0; j<10000;j++)////delay slightly to stop bouncing

Loop\_if=1;//enable program to go through main Loop

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_11);

}

void Send\_Values\_Frequency(int Value1)

{

char buffer1[5];

itoa(Value1,buffer1,10);

ST7920\_SendString(3,1, buffer1);

}

void Send\_Values\_Amplitude(float Value2)

{

char buffer2[5];

gcvt(Value2,3,buffer2);

ST7920\_SendString(1,1, buffer2);

}

void Send\_Units(void)

{

ST7920\_SendString(1,4, "V");

ST7920\_SendString(3,4, "Hz");

}

void Send\_Menu\_Item(void)

{

uint8\_t Stateofmenu=SubMenuCount\*100+Button[0]\*10+Button[1];//converts 3 variables into //a 3 digit number to be used for state

uint8\_t menuStates[]={000,100,110,200,210};// SubMenuCount,Button0,Button1..EX: 110 is //Sub=1,B0=1,B1=0

for (ArrayElement=0;menuStates[ArrayElement]!=Stateofmenu;ArrayElement++)

{;}

Stateofmenu=ArrayElement;//this will convert previous example into a number to be used as //state which is then used for switch case below. Prev example:menustate=110, sp Array //Element =2 since 3rd element in array

if (Scroll!=ScrollOld ||ArrayElement!=ArrayElementOld||Loop==1)// if the menu needs to scroll //(not relevant for this project since only 2 elements to menu), or if the state changes or if case //determines another loop necessary

{

if (Clear==1)//gives option to sometimes not clear display on state change if clear set to //0 in switch case

{ST7920\_Clear();}

Clear=1;

Loop=0;

switch (Stateofmenu)//SubMenuCount,Button[4],Button[3],Button[2].Button[2],Button[0]`

{

case 0://000 main menu start

ST7920\_SendString(0,0, MAIN[Scroll]);

ST7920\_SendString(2,0, MAIN[Scroll+1]);

New\_Sin\_Amplitude(); //get the new array of values to be sent //to DAC

New\_Sin\_Frequency(); //get the new ARR value for timer to //change frequency

Send\_Values\_Amplitude(Amplitude); //display value of Amplitude

Send\_Values\_Frequency(Frequency); //display value of //Frequency

Send\_Units(); //send units Hz and V

Element=2; // this is used for arrow positioning and scrolling

break;

case 1://100//edit Amplitude

ST7920\_SendString(0,0, MAIN[Scroll]);

ST7920\_SendString(2,0, MAIN[Scroll+1]);

if ((Count==0)&&(CountAmplitude[0]!=Count))//if Count is 0 but //AmplitudeCount isnt, use old value

{Count=CountAmplitude[0];}

CountAmplitude[0]=Count; //store value of Amplitude as whatever //count is

New\_Sin\_Amplitude();

Send\_Values\_Amplitude(Amplitude);

Send\_Values\_Frequency(Frequency);

Send\_Units();

Element=1;

Arrow=0;//set arrow postion

Loop=1;

break;

case 2://110//edit Frequency

ST7920\_SendString(0,0, MAIN[Scroll]);

ST7920\_SendString(2,0, MAIN[Scroll+1]);

if ((Count==0)&&(CountFrequency[0]!=Count))//if Count is 0 but //FrequencyCount isnt, use old value

{Count=CountFrequency[0];}

CountFrequency[0]=Count;

New\_Sin\_Frequency();

Send\_Values\_Amplitude(Amplitude);

Send\_Values\_Frequency(Frequency);

Send\_Units();

Element=1;

Arrow=1;

Loop=1;

break;

case 3://200//send back to main menu

SubMenuCount=0;

Button[0]=0;

Button[1]=0;

Loopreturn=1;

Element=2;

break;

case 4://210//send back to main menu

SubMenuCount=0;

Button[0]=0;

Button[1]=0;

Loopreturn=1;

Element=2;

break;

}

ScrollOld=Scroll;

ArrayElementOld=ArrayElement;

}

}

void Loop\_Return\_Check(void)

{

if (Loopreturn==1)

{Send\_Menu\_Item();

Send\_Arrow();

Loopreturn=0;}

}

void Send\_Arrow(void)//sets arrow postion

{

if (Arrow==0){

ST7920\_SendString(0,7, &Six);

ST7920\_SendString(1,7, &Blank);

ST7920\_SendString(2,7, &Blank);

ST7920\_SendString(3,7, &Blank);

}

if (Arrow==1){

ST7920\_SendString(2,7, &Six);

ST7920\_SendString(0,7, &Blank);

ST7920\_SendString(1,7, &Blank);

ST7920\_SendString(3,7, &Blank);

}

}

void Calculate\_Arrow\_and\_Scroll()//int Element =number of States, 2 in this project so not really //too relevant but coded so that if more than 4 elements menu can scroll and keep track of arrow

{int possibleScrollStates = Element-4;

if(possibleScrollStates<0){possibleScrollStates=0;}

if (State>PrevState)

{Arrow++;}

if (State<PrevState)

{Arrow--;}

if (Arrow>3)

{Arrow=3;

Scroll++;}

if (Arrow<0)

{Arrow=0;

Scroll--;}

if (Scroll>possibleScrollStates)

{Scroll=possibleScrollStates;}

if (Scroll<0)

{Scroll=0;}

}

void Calculate\_Menu\_Element\_Count()//this c1 is used for menu Scrolling, input 2n-1

{int possibleStates=2\*Element-1;

if (Count>PrevCount)

{MenuElementCount++;}

if (Count<PrevCount)

{MenuElementCount--;}

if (MenuElementCount>possibleStates)

{MenuElementCount=possibleStates;

}

if (MenuElementCount<0)

{MenuElementCount=0;

}

}

void Populate\_Menu\_Settings\_Names\_For\_LCD()

{//main menu State 0

MAIN[0] = "Amplitude";//State0

MAIN[1] = "Frequency";//State1

}

void Get\_State(void)//State corresponds to how many Elements in menu screen. For 2 //Elements, 2 States, but

//Count will take on 4 values so encoder needs to be rotated two pulses to change States.

//State0; MenuElementCount=(0 or 1) State1; MenuElementCount=(2 or 3) StateN: Count=(2N //or 2N+1)

{

for (int x =0;x<4;x=x+2)//2 Elements (AMP or FREQ) so 3 is the cutoff

if (MenuElementCount>=x && MenuElementCount<=(x+2))//figure out which range Count is in, then divide by 2 to get State,

//since two pulses for click

{State=(x)/2;}

}

void Reset\_Variables(void)

{

Count=0;

MenuElementCount=0;

PrevCount=0;

Scroll=0;

Arrow=0;

PrevState=0;

State=0;

}

void Save\_Variables(void)

{

CountOld=Count;

MenuElementCountOld=MenuElementCount;

PrevCountOld=PrevCount;

ScrollOld=Scroll;

StateOld=State;

PrevStateOld=PrevState;

ArrowOld=Arrow;}

void New\_Sin\_Amplitude(void)

{

Amplitude=CountAmplitude[0]\*.01;

if (Amplitude<.01)

{Amplitude=0;}

if (Amplitude>1.16)

{Amplitude=1.16}//Max RMS value 1.16

for(int z = 0; z <= 399; z++)

{

DACvalue[z]=(int)((DACvaluePlaceHolder[z]\*Amplitude)/1.16);

}

}

void New\_Sin\_Frequency(void)

{

Frequency=CountFrequency[0];

ARR=60000/Frequency;

TIM4->ARR = ARR;

}

**B: MBED Code for Data Acquisition and Control System:**

/\*

\* Copyright (c) 2017-2020 Arm Limited and affiliates.

\* SPDX-License-Identifier: Apache-2.0

\*/

#include "mbed.h"

#include <cstdint>

#include <string>

#include "Kernel.h"

#include "HBridgeDCMotor.h"

HBridgeDCMotor m1(D9, D8);

HBridgeDCMotor m2(D6, D7);

Serial pc(SERIAL\_TX, SERIAL\_RX);

DigitalOut led(LED1);

InterruptIn \*phasein1;

InterruptIn \*button;

InterruptIn \*freq;

Timer P1;

Timer P2;

EventQueue phase;

EventQueue freqmeasure;

volatile int\_fast32\_t x;

volatile int\_fast32\_t trise, tfall,i=0;

volatile float theta,theta2,theta3, theta23;

volatile float setpoint = 120.0;

volatile float phaseangle;

volatile float phase2;

volatile float phase3;

volatile bool p = false;

volatile bool running = false;

volatile bool startup = false;

volatile bool freqcheck = false;

volatile bool update = false;

volatile int\_fast32\_t period = 0;

volatile float per = 0;

void phase12Rise(){

P1.start();

}

void phase12Fall(){

x = P1.read\_us();

P1.stop();

P1.reset();

phasein1->disable\_irq();

p = true;

}

void buttonSwitch(){

running = !running;

}

void freqMeasureRise(){

P2.start();

}

void freqMeasureFall(){

period = P2.read\_us();

P2.stop();

P2.reset();

freq->disable\_irq();

freqcheck = true;

}

void measurePhaseTask(){ //+120

led = 1;

wait\_us(1);

InterruptIn phasein1(PB\_15);

phasein1.rise(&phase12Rise);

phasein1.fall(&phase12Fall);

i =0;

while (i<1) {

if((x<50) && p){

p = false;

phasein1.enable\_irq();

}

else if(p && (x > 50)){

theta = 360.0\*x/(per\*1000.0);

p = false;

i = i +1;

}

}

led =0;

phasein1.fall(NULL);

phasein1.rise(NULL);

phasein1.enable\_irq();

update = true;

phase.break\_dispatch();

}

void measurePhaseTask12(){ //+120

led = 1;

wait\_us(1);

InterruptIn phasein1(PB\_15);

phasein1.rise(&phase12Rise);

phasein1.fall(&phase12Fall);

i =0;

while (i<1) {

if((x<50) && p){

p = false;

phasein1.enable\_irq();

}

else if(p && (x > 50)){

theta2 = 360.0\*x/(per\*1000.0);

p = false;

i = i +1;

}

}

led =0;

phasein1.fall(NULL);

phasein1.rise(NULL);

phasein1.enable\_irq();

update = true;

phase.break\_dispatch();

}

void measurePhaseTask13(){ //-120

led = 1;

wait\_us(1);

InterruptIn phasein1(PB\_14);

phasein1.rise(&phase12Rise);

phasein1.fall(&phase12Fall);

i =0;

while (i<1) {

if((x<50) && p){

p = false;

phasein1.enable\_irq();

}

else if(p && (x > 50)){

theta3 = 360.0\*x/(per\*1000.0);

p = false;

i = i +1;

}

}

led =0;

phasein1.fall(NULL);

phasein1.rise(NULL);

phasein1.enable\_irq();

update = true;

phase.break\_dispatch();

}

void measurePhaseTask23(){ //120&240

led = 1;

wait\_us(1);

InterruptIn phasein1(PB\_13);

phasein1.rise(&phase12Rise);

phasein1.fall(&phase12Fall);

i =0;

while (i<1) {

if((x<50) && p){

p = false;

phasein1.enable\_irq();

}

else if(p && (x > 50)){

theta23 = 360.0\*x/(per\*1000.0);

p = false;

i = i +1;

}

}

led =0;

phasein1.fall(NULL);

phasein1.rise(NULL);

phasein1.enable\_irq();

update = true;

phase.break\_dispatch();

}

void freqMeasureTask(){

led = 1;

wait\_us(1);

i =0;

InterruptIn freq(PB\_1);

freq.rise(&freqMeasureRise);

freq.fall(&freqMeasureFall);

while(i<1){

if(period < 50 && freqcheck){

freqcheck = false;

freq.enable\_irq();

}

else if(freqcheck && period >50){

per = 2\*period/1000.0;

freqcheck = false;

i++;

}

}

led = 0;

freq.rise(NULL);

freq.fall(NULL);

freq.enable\_irq();

// pc.printf("Phase1 angle is %f ms \n", 2\*x/1000.0);

// pc.printf("Phase1 angle is %f degrees \n", theta);

freqmeasure.break\_dispatch();

}

int main()

{

int w=0;

pc.baud(115200);

float sampleTime = 50e-3, switchingFrequency = 25e3, rampTime = 1;

m1.configure(sampleTime, switchingFrequency, rampTime, rampTime);

m2.configure(sampleTime, switchingFrequency, rampTime, rampTime);

while(1){

while(w<2){ // wait 3 seconds after powerup for start up transient to die down

wait(1);

pc.printf("waiting \r \n");

w++;

}

led=!led; // blink onboard led

freqmeasure.call(freqMeasureTask);

freqmeasure.dispatch();

wait\_us(100);

phase.call(measurePhaseTask13);

phase.dispatch();

wait\_us(100);

phase.call(measurePhaseTask12);

phase.dispatch();

wait\_us(100);

if(update){

pc.printf("Period in ms is %.4f \r \n", per);

pc.printf("Frequency in Hz is %.4f \r \n", (1/(per/1000)));

pc.printf("Phase 1-2 angle is %.4f degrees \r \n", theta2);

pc.printf("Phase 1-3 angle is %.4f degrees \r \n", theta3);

if(theta3 < (0.998\*setpoint)){

m1.setDutyCycle(1); //motor right

}

else if(theta3 > 1.002\*setpoint){

m1.setDutyCycle(-1); //motor left

}

else{

m1.coast();

}

if(theta2 < (0.998\*setpoint)){

m2.setDutyCycle(-1); //motor left

}

else if(theta2 > 1.002\*setpoint){

m2.setDutyCycle(1); //motor right

}

else{

m2.coast();

}

wait\_us(100);

update = false;

ThisThread::sleep\_for(100);

}

}

}