ELEC 145 Ultrasonic sensor-controlled synthesizer

Module code: ELEC 145

Module description: Electronic design & Build

Module tutor: Shakil Awan

Date: 22/5/2020

Group name: The **PicoFarads**

Group member IDs:

Jacob Howell 10656206

Isaac Kulimushi 10671216

Guy Ringshaw 10651581

Contents

1. Introduction

This report will analyse and explain the process of creating an ultrasonic sensor operated synthesizer. The synthesizer will be constructed using 3 main components:

- ST Microelectronics F429ZI micro-controller board
- HC-SR04 ultrasonic ranging module
- D class amplifier circuit

These components will be used to create a synthesizer that outputs a frequency audible via an amplifier circuit between 220 and 880Hz produces sine, square, triangle and sawtooth waves, and that can have its frequency output adjusted with gestures controlled by an ultrasonic ranging module.

The report will seek to explain every important step in using these components to build a synthesizer with hand-gesture based frequency adjustment. This includes research, theory, theoretical calculations, equation derivations, circuit design and programming.

The first section (2-2.6) will cover the programming required to turn a frequency derived from an ultrasonically measured distance into one of multiple output waveforms. These waveforms include sine, square, triangle and sawtooth waves. Furthermore, this section will show how the F429ZI board must be programmed to create a structure driven by a single processing thread that can handle 2 time sensitive functions successfully; polling the HC-SR04 and outputting a waveform to an amplifier circuit.

The next section (3-3.4) will cover the HC-SR04 sensor specifications, the programming required to measure distance with the sensor, and how to turn the measured distance into a usable frequency. Each important step of this distance-to-frequency transformation process will be covered, from sensor calibration programming, distance measuring programming to equation derivation.

The final section of the main report body (4-4.X) will cover the D-class audio amplifier design. The report will analyse the research that influenced the choice of amplifier, the theoretical principles behind the circuit design, component value calculations and the function of the circuit and how it uses the F429ZI output waveform to create an audible output.

Section 2: Controlling The Synthesizer

Section 2.1: Design Considerations:

Fig 2.1.1: Image of NUCLEO-F429ZI

The STM32 Nucleo-144 board provides an affordable and flexible way for users to try out new concepts and build prototypes with the STM32 microcontroller, choosing from the various combinations of performance, power consumption and features. The ST Zio connector, which is an extension of Arduino™ Uno V3, provides access to more peripherals and ST morpho headers make it easy to expand the functionality of the Nucleo open development platform with a wide choice of specialized shields. The STM32 Nucleo-144 board does not require any separate probe, as it integrates the ST-LINK/V2-1 debugger/programmer and it comes with the STM32 comprehensive software HAL library, together with various packaged software examples, as well as a direct access to the ARM®mbed™online resources.

Reference; mbed.com(1)

Design aspects of the NEUCLEO-F429ZI to consider before programming:

1. This board has a single-core ARM Cortex M4 chip – This means tasks cannot be run in parallel and therefore threading or scheduling need to be adopted for time sensitive processes.
2. The Chip doesn't support 64-bit long data types as it is a 32-bit chip – this means overflow bugs must be considered while dealing with large numbers.
3. Each Digital/Analogue Input or Output can only be accessed by one "thread" (it is parent thread) – output or input pins can't be used while a non-parent thread is running
4. Analogue outputs peak at 3.3V DC – any output needs to be well within 0V – 3.3V to prevent distortion or clipping.

Program requirements:

1. Audio should be outputted via an Analogue Out pin from between 0V – 3.3V
2. Digital inputs should be connected via Digital In pins
3. Inputs should be updated fast enough for the user not to notice
4. Must be able to change the frequency depending on inputs
5. User needs to be able to select different wave types (e.g. Sine, Square, Triangle, Sawtooth)
6. Output sample rate must be over 40KHz to avoid distortion at high audio frequency's
7. Blocking of the output needs to be as low as possible
8. Section 2.2: Program Structure

This software is written as an Object Orientated program (OOP) as it allows for the use of private attributes, so all member functions within a class can access a shared memory pool without using global variables. OOP also allows multiple instances of the same class. This provides the option of duplicating

certain features like UpdateOutput (in the case the synth needed to become polyphonic rather than monophonic). Project source code: GitHub (2)

The program consists of:

One four header files: responsible for including –

- synth-os.h
    - Definitions to be used throughout the program

- mbed.h
    - input/output pin objects
    - Timer objects
    - Serial outputs (for communicating with a PC)
    - etc…

- sensorData.h
    - declaring all member functions and attributes for the SensorData class

- updateOutput.h
    - declaring all member functions and attributes for the UpdateOutput class

The 'main' function: responsible for –

- Creating an instance of each class
- Setting the initial wave type and
- Polling the inputs and outputs

Two classes:

- UpdateOutput: responsible for –
    - Tracking and synchronising the current output sample
    - Producing a new sample depending on the wave type and frequency

- SensorData: responsible for –
    - Gathering inputs from all connected sensors
    - Calibrating the sonar
    - Updating the frequency using the sonar
    - Updating the wave type using the onboard button

Figure 2.2.1: - Table of SensorData member functions and attributes

Member functions of SensorData Class		
Type	Name	Brief Description
Private float	getFrequency	Uses a distance measurement to set the desired frequency of the output
Private float	calibrateSonar	Calculates the time delay of the circuitry of the HC-SR04 sonar sensor
Public void	waveTypeSelector	Allows user to select which type of wave the synth produces (OFF, SINE, TRIANGLE, SAW, SQUARE) by pressing the onboard button
Public void	updateFrequency	Converts frequency to number of samples per period and passes it to the setWavePeriod function of the UpdateOutput class
Private DigitalOut	trigger	Sets the value of the trigger on the sonar
Private DigitalIn	echo	Detects the value of the echo transducer on the sonar
Private Timer	debounce	Timer used to debounce the onboard button when pressed
Private Timer	sonar	Timer used to measure the time difference of trigger and echo pulses from the sonar
Attributes		
Private int	currentWaveType	Uses an enum defined in synth_os.h to keep track of wave type
Private int	correction	Set by calibrateSonar – used to subtract delay from the sonar it is self
Private float	frequencyRange	The difference between the TOP_FREQUENCY and BOTTOM_FREQUENCY (defined in synth-os.h) – used to help calculate which frequency should be set
Private Float	samplesInPeriod	Used in updateFrequency to pass the number of samples in the period of the wave at the current frequency

The sensor data class includes the following header files:

- mbed.h
- synth-os.h
- updateOutput.h

Figure 2.2.2: - Table of UpdateOutput member functions and attributes

Member functions of UpdateOutput Class		
Function type	Function name	Brief Description
Public void	setWavePeriod	Receives wave time period and sets it as a private attribute of the class for other functions to use
Public void	setWaveType	Receives the wave type integer code(0 – 4) as parameter and sets it as a private attribute of the class for other functions to use
Public void	createSample	Calculates which sample should number (0 – 43,478) should be created by reading from a clock running in micro-seconds. Produces a sample once every 23us while adjusting for jitter caused by other tasks running for varying times. This prevents the wave from becoming distorted.
Private void	type	Uses the private attribute 'int waveType' to select which function to run for the next sample. And sets the AnalogOut 'dac' to the float that gets returned
Private float	sineWave	Returns either a value between 0 and 1 to 'type' by running the 'currentSampleNo' and 'samplesInPeriod' through a sine wave process
Private float	triangleWave	Returns either a value between 0 and 1 to 'type' by running the 'currentSampleNo' and 'samplesInPeriod' through a triangle wave process
Private float	sawWave	Returns either a value between 0 and 1 to 'type' by running the 'currentSampleNo' and 'samplesInPeriod' through a sawtooth wave process
Private float	squareWave	Returns either 0 or 1 to 'type' by running the 'currentSampleNo' and 'samplesInPeriod' through a square wave process
Private Timer	runTimeTest	Timer for run time test logging
Private Timer	sampleClock	Used to synchronise output samples and calculate jitter
Private Timer	sampleTriggerClock	Used to ensure a sample is produced once every 23 us
Private DigitalOut	clipLed	Used to show the user if the analog output has clipped
Private AnalogOut	dac	Used to set the voltage of analog output pin D13
Attributes		
Int	runTime	Set by reading the 'runTimeTest' Timer to calculate the run time of processes if logging is turned on
float	Vout	Used to store what voltage will be set to AnalogOut 'dac'
Int	waveType	Used to keep track of current wave type. This means the 'type' function can select the correct wave function
Int	currentSampleNo	Used to keep track of the current sample number so the wave functions know which part of the wave to calculate

Int	samplesInPeriod	Used to keep track of the current number of samples in one wave period. This allows the wave functions to output the correct values for the desired frequency

The UpdateOutput class includes synth-os.h and mbed.h

Section 2.3: Scheduling Tasks

Figure 2.3.1: screenshot of main function

```
29 int main() {
30   pc.printf("\r\n\r\ninside main\r\n\r\n");
31
32   led1 = 1;          //turn on led1 to show main is running
33
34   wave.setWaveType(SINE);
35
36   trigger.reset();
37   trigger.start();
38   //UPDATING OUTPUT
39   while(true){
40     while (trigger.read() < UPDATE_INPUT_RATE){
41       //updating output while inputs arnt being updated
42       wave.createSample();
43     }
44     trigger.reset();
45
46     //getting inputs at specified 'UPDATE_INPUT_RATE'
47     inputs.waveTypeSelector();
48     inputs.updateFrequency();
49   }//end of while
50 }//end of main
```

The main function contains a simple polling loop that schedules input and output tasks depending on when the inputs need to be updated (as they are updated much less often than the output).

1. Start an infinite while loop to keep running while the synth is on
2. Enter a second while loop that that loops creates new samples (lines 40 - 42)
3. Exit the loop if the trigger timer if it is time to update the inputs (line 40)
4. Reset the trigger timer to re-enter the output loop after the inputs have updated (line 44)
5. Update all inputs (lines 47 -48)
6. Start again (line 39)

This method was preferable over a Ticker function to update the inputs at a specified rate, as the blocking problems that occur whilst updating inputs (mentioned in **Section 2.5**) can be solved.

This is because the AnalogOut ‘dac’ (see **Figure 2.2.2**) can only be accessed from its parent thread.

Ticker runs as a separate thread and causes Mutex errors within the mbed library if a pin defined outside its own thread is written to.

Section 2.4: Gathering input data

As mentioned in **Section 3**, the inputs are called within the 'main' function at a specified rate. The inputs are less time sensitive than updating the outputs in this case. This is because the user can only do so much in one second.

The inputs are updated once every 0.04 seconds (this is defined in synth-os.h as UPDATE_INPUT_RATE – used in 'main' line 40) as the incremental changes in frequency seem smooth at this rate.

Figure 2.4.1: updateFrequency function

```
6  //constructor
7  SensorData::SensorData(): trigger(D6), echo(D7){
8  }
9  //Serial pc(USBTX, USBRX);
10
11
12 void SensorData::updateFrequency(){
13
14   //setting period in terms of samples per second
15   samplesInPeriod = SAMPLES_PER_SECOND/(int)getFrequency();
16   wave.setWavePeriod(samplesInPeriod);
17
18 }
```

Figure 2.4.2: snippet of updateFrequency function

```
12      trigger = 1; //signal 'high' to trigger pin
13
14      //produce one wave sample to keep trigger on for 25us
15      wave.createSample(); //this function takes about 25us to execute
16      sonar.reset(); //reset timer
17      trigger = 0;//signal 'low' to trigger pin
18      //wait for echo high
19
20      //----The next two while loops cause a 1-3 ms blocking problem//
21      while (echo==0) {
22        wave.createSample();    //this line keeps creating new samples while wa
23      }; //loops empty body until echo is 'high'
24      //echo high, so start timer
25      sonar.start();
26      //wait for echo low
27      while (echo==1) {
28        wave.createSample();    //this line keeps creating new samples while wa
29      }; //loops empty body until echo is 'low'
30      //stop timer and read value
31      sonar.stop();
32      //subtract software overhead timer calculate distance in cm
33      distance = ((sonar.read_us()-correction)/58.8f);
```

List of events to update the time period of each wave:

1. Enter updateFrequency (line 47 of main)
2. Enter getFrequency (line 15 of updateFrequency)
3. Send a pulse to the trigger pin connected to the sonar (line 12 and 17 of getFrequency)
4. Enter a while loop that creates new samples until echo pin is 0 - this should pass instantly (line 21 of getFrequency)
5. Start the timer 'Sonar' (line 25 of getFrequency)
6. Enter a while loop that creates new samples until the echo pin is 1 (line 27 of getFrequency)
7. Stop 'Sonar' timer and read the time delay and adjust with pre calculated correction value (line 31 and 32 of getFrequency)
8. Calculate distance using modified Distance = speed*time (line 33 of getFrequency)
9. Calculate desired frequency and return it to update frequency (reference; Section 3.3)
10. Convert frequency value to number of samples per period and update the variable "samplesInPeriod" (line 15 of updateFrequency)
11. Pass samplesInPeriod to setWavePeriod to allow the UpdateOutput class to use it (line 16 of updateFrequency)
12. Return to 'main' function

Figure 2.4.3: waveTypeSelector function

```
5  void SensorData::waveTypeSelector(){
6    enum {RELEASED = 0, PRESSED};
7    while(button == 1){
8      debounce.reset();
9      debounce.start();
10
11     //debouncing rising edge
12     while (debounce.read_ms() <= 50){
13     }
14     debounce.stop();
15     debounce.reset();
16     //check if button still = 1 after debounce buffer
17     if (button == PRESSED){
18       currentWaveType = currentWaveType + 1;
19         //make sure we stay only select existing wave types
20       if (currentWaveType > SQUARE){
21         currentWaveType = OFF;
22       }
23
24       //wait for button to be released
25       while(button == PRESSED){
26
27       } //end of wait for release
28       wave.setWaveType(currentWaveType);
29     }
30     //if button was off before debounce buffer ended
31     else{
32       return;
33     }
34
35   }//end of while
36 }//end of waveTypeSelector
```

List of events to select a new wave type:

1. Enter waveTypeSelector (line 47 of main)
2. Check if enter while loop if button was pressed (line 7 of waveTypeSelector) if not return to ‘main’ to prevent unnecessary blocking
3. Debounce button press with a 50ms empty while loop (line 12 of waveTypeSelector)
4. Check if button is still pressed to complete debounce (line 17 of waveTypeSelector) if not it was just noise, so return to ‘main’
5. Add 1 to the private attribute currentWaveType to set the next wave type (line 18 of waveTypeSelector)
6. Reset currentWaveType to OFF when it exceeds the last currentWaveType enum code (line 20 of waveTypeSelector)
7. Wait for button to be released by sitting in a while loop until it is no longer pressed (line 25 of waveTypeSelector)
8. Pass currentWaveType to setWaveType of the UpdateOutput class so it can use it
9. Return to ‘main’

Section 2.5: Synchronizing output sample and preventing blocking

A Synthesizers main function is to output audio, so by nature it must have a relatively high frequency synchronised output. The output must have a sample rate of at least twice the audible sound spectrum (20Hz – 20KHz) in order to output a minimum of two samples per wave period.

As the processor only has a single core, the processing required for updating the instantaneous voltage of the wave needs to be synchronised to a clock to prevent distortion.

Figure 2.5.1: screenshot of createSample function

```
1  #include "synth_os.h"
2  #include "mbed.h"
3  #include "updateOutput.h"
4
5  void UpdateOutput::createSample(){
6
7
8    int currentTime = sampleClock.read_us();      //read sampleClock
9
10   //figure out which sample we need to create depending on sampleClock
11   currentSampleNo = currentTime/SAMPLE_PERIOD_us;
12
13   //adjust time offset to syincronise sample and avoid jitter
14   //caused by unknown runtime of other code
15   int jitterAdjust = currentTime % SAMPLE_PERIOD_us;
16
17   //start
18   int sampleTrigger = sampleTriggerClock.read_us();
19
20   if (sampleTrigger >= SAMPLE_PERIOD_us - jitterAdjust){
21     //update output when sampleTriggerClock equals
22     //SAMPLE_PERIOD_us (23us)  minus the time offset (to subtract the jitter)
23     type();
24
25     if (currentSampleNo > SAMPLES_PER_SECOND){
26       sampleClock.reset();  //reset sampleClock each second to prevent overflow
27     }
28     sampleTriggerClock.reset();
29
30   }// end of if
31 }
```

Figure 2.5.1 shows the function “createSample” which is responsible for calculating the current sample number and avoiding jitter caused by processes running between “createSample” calls.

Calculating the current sample and jitter:

1. The function reads the “sampleTriggerClock” in microseconds sets it to “currentTime” (line 8)
2. Divide “currentTime” by “SAMPLE_PERIOD_us” to find the quotient (line 11). This is the current sample number.
3. Divide “currentTime” by “SAMPLE_PERIOD_us” to find the remainder (line 15).
4. Reset sample clock each second to prevent integer overflow (line 26)

Synchronising to 43,478 Hz sample rate:

1. Set sampleTrigger to the returned value of sampleTriggerClock in microseconds (line 18)
2. Check if it is time to create a new sample while adjusting for jitter(line 20)
3. If true run “type” function to select wave type which will update output according to frequency and the current sample
4. If false, return out of the function. This ensures fast run time of this function with no waits; preventing it blocking other functions like updateFrequency.

Note: CurrentSampleNo is a private attribute of the UpdateOutput class and therefore doesn’t need to be returned for other member functions to use it.

Also the sample rate is 43,478 Hz because it is close to the standard 44.1 kHz used by most audio equipment and more importantly it has an integer period of almost exactly 23 us which can be stored as an integer and prevents the need for imprecise floats for such a time sensitive application.

More Blocking Prevention:

Figure 2.5.2: - code snippet from getFrequency

```
20        //----The next two while loops cause a 1-3 ms blocking problem//
21        while (echo==0) {
22          wave.createSample();    //this line keeps creating new samples w
23        }; //loops empty body until echo is 'high'
24        //echo high, so start timer
25        sonar.start();
26        //wait for echo low
27        while (echo==1) {
28          wave.createSample();    //this line keeps creating new samples w
29        }; //loops empty body until echo is 'low'
```

Here is an example where blocking could occur.
whilst the sonar is waiting to receive values from “echo” the program sits in while loops (lines 21 and 27). If empty, this would block createSample by up to 3 ms if the user’s hand is 50 cm away from the sensor.

This length of block produces audible distortion in the output as createSample should’ve created over 130 new samples in this time.

$$SAMPLE\_PERIOD\_us/BlockingTime \;=\; number\ of\ samples\ that\ \ should've\ been\ produced$$

$$23 * 10 - 6\ /\ 3 * 10 - 3 \;=\; 130.43\ samples$$

To avoid this problem, createSample is called to run within the otherwise empty while loops to continue updating the output. (lines 22 and 28).

Section 2.6: Calculating and setting the Analog output

Note: Everything in this section is executed from within the UpdateOutput class

Updating the output is much more time sensitive than updating the inputs as it needs to be executed at 43KHz. This means no wait loops and no accidental implicit conversions from a float to double as this can cause unnecessary blocking.

Once createSample has decided it is time to start updating the analog output, we enter the 'type' function.

The type function is quite long as it uses a 'switch case' to quickly select which function to use, so only small snippets of code will be shown here to give an idea on how it works. Please refer to the source code attached to see the full code.

Figure 2.6.1: Snippet of 'type' function

```
14  void UpdateOutput::type(){
15
16
17      switch(waveType)
18      {
19        case OFF:
20          Vout = 0.0f;
21          break;
22
23        case SINE:
24          #ifdef RUNTIME_LOGGING
25          runTimeTest.reset();
26          runTimeTest.start();
27          #endif
28          Vout = sinWave();
```

```
50        case SQUARE:
51          #ifdef RUNTIME_LOGGING
52          runTimeTest.reset();
53          runTimeTest.start();
54          #endif
55
56          Vout = squareWave();
57          break;
58
59        default:
60          waveType = OFF;
61          break;
62
63    }//end of wave type switch case
64    dac.write(Vout);
```

This method does two things. Selects which wave type to create a sample of and update the analog output 'dac'.

List of events in 'type':

1. First it uses the private attribute waveType - set earlier by updateFrequency of the SensorData class (refer to Section 4 - list of events) – to quickly select which function to create a sample from using a switch case (line 17)
2. It will then enter the get set the float Vout to the returned float (between 0 and 1) from the selected function (see line 28 for an example)
3. It then writes the passes this value to the AnalogOut object 'dac' (see line 64 of 'type')

Figure 2.6.2: Producing a sample from within sinWave():

```
3  float UpdateOutput::sinWave(){
4    float omega = (2.0f*PI/samplesInPeriod)*(float)currentSampleNo;
5    float wave = sin(omega);
6    //translate wave to work between 0 and 1 rather than -1 and 1
7    float Vout = (wave/(2.0f*WAVE_DEVIDOR))+0.5f;
8
9    if (Vout > 1.0f){
10     clipLed = 1;
11     Vout = 1.0f;
12   }
13   else if (Vout < 0.0f){
14     clipLed = 1.0f;
15     Vout = 0.0f;
16   }
17   //printf("Vout = %5.3f\n\r", Vout);
18
19   return Vout;
20 }//end of SIN
```

The sine wave function uses the private attributes “samplesInPeriod” and “currentSampleNo” to produce a value between 0 and 1 for Vout. This will bet set to the AnalogOut ‘dac’ as mentioned earlier in this section.

The float ‘wave’ (line 5) is set to the return value of the sin(omega) function from “math” (an mbed library).

Since $Omega = (2\pi/T) * t$

We can say:

- T (time period of one wave) = samplesInPeriod
- t (current time) = currentSampleNo

e.g.) if there are 100 samples per time period of the wave, and the current sample is 20 ‘wave’ would be set to 0.951.
if the time period was the same but the current sample was 80, ‘wave would be set to -0.951

This is too close to the upper threshold of the AnalogOut object ‘dac’ and half of the wave is below the lower threshold of 0. This will cause lots of clipping.

The wave needs to be made totally entirely positive and squashed to fit well within the thresholds of AnalogOut.

Note: only the sine wave function will become negative, so twice the reduction is needed when compared to other wave types

Line 7 fixes these issues.
the float Vout is set to ‘wave’ divided by 2 times the WAVE_DIVIDOR constant
(defined in synth-os.h value = 2) and then 0.5 is added. This reduces the magnitude of the sample Vout by 4, and translates it to the mid-point of the thresholds.

After, Vout passes through boundary checks to show the user if it clips due to some error, and gets returned to the ‘type’ function to be written to ‘dac’.

Figure2.6.3: triangleWave function

```
3  float UpdateOutput::triangleWave(){
4
5          float Vout = 0.0f;
6          float wave;
7          wave = (2*(float)(currentSampleNo % (int)samplesInPeriod)/samplesInPeriod);
8          //decide if wave should rise or fall
9          if (wave > 1.0f){
10           wave = 1.0f + (1.0f - wave);
11         }
12
13         //translate wave to work between 0 and 1 rather than -1 and 1
14         Vout = (wave/WAVE_DEVIDOR)+(0.25f);
15         if (Vout > 1.0f){
16           clipLed = 1;
17           Vout = 1;
18         }
19         else if (Vout < 0.0f){
20           clipLed = 1;
21           Vout = 0.0f;
22         }
23         return Vout;
24 }//end of triangle
```

Deriving the equation:

Note % means remainder

t = currentSampleNo, T = samplesInPeriod

and

X = (t%T)/T

A triangle wave function can written as:

Y = 2*X for Y < 1 (formula refers to line 7)

And

Y= 1 + (1 - 2*X) for Y > 1 (formula refers to line 9 and 10)

To implement this line 7 uses currentSampleNo and samplesInPeriod to create the function Y = 2*X. X needs to have a coefficient of 2 or it will produce a wave of double the frequency. This stretches the wave by half on the X (currentSampleNo) axis.

The boundary conditions are set using an if statement that reverses the function when it exceeds 1. The resulting value is set to ‘wave’

Before ‘Vout’ is set to ‘Wave’ , it must be divided by the wave divider constant and translated by 0.25 in order to sit nicely within the thresholds of the AnalogOut ‘dac’ (line 14).

Finally Vout is set to wave and can be returned to type to update dac.

Figure 2.6.4: sawWave function

```
3  float UpdateOutput::sawWave(){
4    float Vout = 0.0f;
5    float wave = 0.0f;
6
7    wave = (0.5f*(float)(currentSampleNo % (int)samplesInPeriod)/samplesInPeriod);
8
9    if (wave > 1.0f){
10     clipLed = 1;
11     wave = 0.0f;
12   }
13
14   //translate wave to work between 0 and 1 rather than -1 and 1
15   Vout = (wave/2.0f*WAVE_DEVIDOR)+(0.25f);
16   if (Vout > 1){
17     clipLed = 1;
18     Vout = 1.0f;
19   }
20   else if (Vout < 0){
21     clipLed = 1;
22     Vout = 0.0f;
23   }
24   return Vout;
25 } //end of SAW
```

A sawtooth wave is a very similar function to a triangle wave function. The only difference is , it doesn't need boundary conditions and needs to be stretched by a factor of 0.5 on the X axis to maintain the correct frequency.

Figure: squareWave function

```
3  float UpdateOutput::squareWave(){
4    float Vout = 0.0f;
5    float wave;
6
7    float x = (float)(currentSampleNo % (int)samplesInPeriod)/samplesInPeriod;
8
9    if (x > 0.5f){
10     wave = 1.0f;
11   }
12   else{
13     wave = 0.0f;
14   }
15   Vout = (wave/WAVE_DEVIDOR)+(0.25f);
16   if (Vout > 1){
17     clipLed = 1;
18     Vout = 1.0f;
19   }
20   else if (Vout < 0.0f){
21     clipLed = 1;
22     Vout = 0.0f;
23   }
24   return Vout;
25 }//end of squareWave
```

The squareWave function uses a sawtooth value for x as it has a liner relationship with what we need Vout to return.

The sawtooth wave will start at 0 at t = 0 and will finish at 1 when t = T

To create a square wave from this function, 'wave' needs to be high when x > 0.5 and low when x < 0.5 .

This is done using if and else statements on lines 9 and 15.

'Vout' is then set to 'wave' divided by the wave divider and translated by 0.25 to keep it within the threshold of dac.

Vout is then returned to 'type' and written to 'dac' and the output update is complete.

Section 3: Programming the HC-SR04 to measure distance and creating a resultant frequency

3.1: Ultrasonic sensor specifications and use overview

The HC-SR04 is an ultrasonic ranging module used for detecting object distances. The module has two transducers, one emits 40kHz ultrasonic pulses while the other transducer produces an output when it receives the ultrasonic pulse reflected from an object. Using 5V DC at a typical consumption of 15mA, the module can detect objects within a range of 3 to 400cm in a 15-degree pulse beam width. The module has 4 pins; VCC, Trig, Echo and GND shown from left to right in **Fig** 3.1. The VCC pin connects to the 5V DC supply required, the Trig pin receives the 'high' signal causing it to emit the 40kHz pulse, the Echo pin sends an output when a reflected signal is received and the GND pin acts as a connection to ground. (mbed.com)

Fig 3.1.1: Photograph of the HC-SR04 ultrasonic ranging module showing the two transducers, input and output pins.

Reference; Sparkfun(3)

Once the device has been connected to control electronics such as a micro-controller board, it can be used to measure distance. To begin the measurement process, a 'high' signal must be sent to whichever pin or output lead the trigger is connected to on the microcontroller board. The microcontroller must be programmed to record the time between when it starts receiving an input from the Echo pin to when it stops receiving this input (Echo pin signal pulse width). This process is illustrated in **Fig** 3.2, once this time has been measured, it can be used with the known value for the speed of sound to calculate distance.

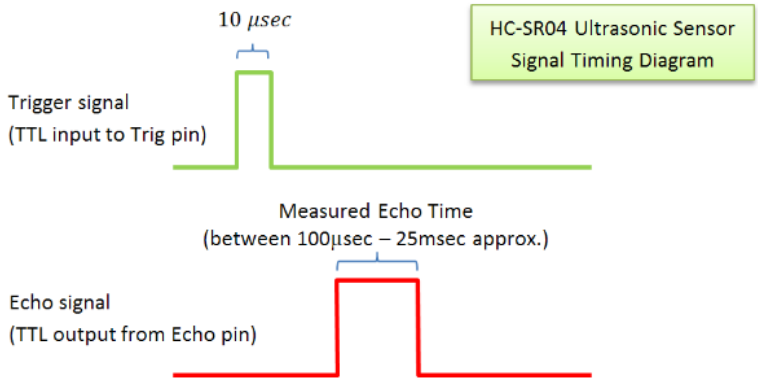


Fig 3.1.2 HC-SR04 trigger and echo timing diagram.

Reference; mbed(4)

3.2: Programming the HC-SR04 to measure distance

```
#include "synth_os.h"
#include "sensorData.h"
DigitalOut led(LED3);

float SensorData::getFrequency()
                {
                        led = !led; //shows how often this function is running
                        float resultingFrequency = 0.0f;
                        float frequencyMultiplyer = 0.0f;
                        float distance = 0.0f;
                        // trigger sonar to send pulse for 10us
                        trigger = 1; //signal 'high' to trigger pin

                        //produce one wave sample to keep trigger on for 25us
                        wave.createSample(); //this function takes about 25us to execute
                        sonar.reset(); //reset timer
                        trigger = 0;//signal 'low' to trigger pin
                        //wait for echo high

                        //----The next two while loops cause a 1-3 ms blocking problem//
                        while (echo==0) {
                                wave.createSample();            //this line keeps creating new samples while waiting for echo to become 1 to prevent blocking
                        }; //loops empty body until echo is 'high'
                        //echo high, so start timer
                        sonar.start();
                        //wait for echo low
                        while (echo==1) {
                                wave.createSample();            //this line keeps creating new samples while waiting for echo to become 0 to prevent blocking
                        }; //loops empty body until echo is 'low'
                        //stop timer and read value
                        sonar.stop();
                        //subtract software overhead timer calculate distance in cm
                        distance = ((sonar.read_us()-correction)/58.8f);

                        if(distance >= OFF_DISTANCE){
                                resultingFrequency = OFF_FREQUENCY;
                                return resultingFrequency;
                        }
                        else if (distance >= UPPER_THRESHOLD){  //check if distance is beyond desired operating threshold
                                resultingFrequency = BOTTOM_FREQUENCY;  //set to min frequency
                                return resultingFrequency;
                        }
                        else if (distance <= LOWER_THRESHOLD){  //check if distance is below desired operating threshold
                                resultingFrequency = TOP_FREQUENCY;             //set to max frequency

                                return resultingFrequency;
                        }
                        frequencyMultiplyer = (distance - LOWER_THRESHOLD)/(UPPER_THRESHOLD - LOWER_THRESHOLD); //calculate what frequency the distance corresponds to

                        if (frequencyMultiplyer < 0.0f){ //check if multiplier is less than 0 (occurs when distance is below lower threshold or above upper threshold)
                                frequencyMultiplyer = 0.000001f;
                        }
                        else if (frequencyMultiplyer > 1.0f){  //check if multiplier is above 1
                                frequencyMultiplyer = 1.0f;  //set multiplier to 1 if it is above 1, as 1 is the max value
                        }

                        //This line sets frequency: the further away your hand
                        //is from the sensor, the lower the frequency
                        resultingFrequency = ((frequencyRange) - ((frequencyRange)*frequencyMultiplyer))+ BOTTOM_FREQUENCY;
                        //resultingFrequency = 440.0f;

                        return resultingFrequency;
                //
                }
```

Distance measuring code section based on demo from mbed. Reference; mbed(4)

Fig 3.2.1 C++ Code for measuring distance with the HC-SR04 and returning a usable frequency value.

To examine how the HC-SR04 can be programmed to measure distance, the **Figures** 3.2.1, 3.2.2, 3.2.3, 3.2.4 and 3.2.5 will be used as a reference when discussing the C++ code that drives the HC-SR04 in this project.

```
#ifndef SENSOR_DATA_H
#define SENSOR_DATA_H

#include "synth_os.h"
#include "mbed.h"
//#include "rtos.h"
#include "updateOutput.h"

extern UpdateOutput wave;
extern Serial pc;

class SensorData{
        public:

                //constructor
                SensorData();

                void updateFrequency();



        private:
                //private member functions
                float getFrequency();
                float calibrateSonar();

                //private attributes
                DigitalOut trigger;
                DigitalIn  echo;
                Timer sonar;
                int correction = calibrateSonar();

                float samplesInPeriod = 100.0f;

};
#endif
```

Fig 3.2.2 class for inputs, outputs, timers and constants related to the process of measuring distance.

```
#include "sensorData.h"

float SensorData::calibrateSonar(){
                                Timer time;

                                float distance = 0;
                                int correction = 0;
                                sonar.reset();
                                // measure actual software polling timer delays
                                // delay used later in time correction
                                // start timer
                                sonar.start();
                                // min software polling delay to read echo pin
                                while (echo==2) {};
                                // stop timer
                                sonar.stop();
                                // read timer
                                correction = sonar.read_us();

                                time.reset();
                                time.start();
                                int tmr = time.read_ms();
                                while (tmr >= 2000){}

                        return correction;      //returns correction of
                }//end of calibrate sonar
```

Fig 3.2.3 function to calibrate HC93-SR04 sensor

(demo code from mbed (4) as template)

```
#include "synth_os.h"
#include "sensorData.h"



//constructor
SensorData::SensorData(): trigger(D6), echo(D7){
}
//Serial pc(USBTX, USBRX);


void SensorData::updateFrequency(){

        //setting period in terms of samples per second
        samplesInPeriod = SAMPLES_PER_SECOND/getFrequency();
        wave.setWavePeriod(samplesInPeriod);


}
```

Fig 3.2.4 CPP file showing the which pins Trigger and Echo have been assigned to on the Nucleo F429ZI

```
#ifndef SYNTH_OS_H
#define SYNTH_OS_H

//enum to assign types of wave
enum waves {OFF = 0, SINE, TRIANGLE, SAW, SQUARE};

#define PI 3.14159265f
#define WAVE_DEVIDOR 2.0f //keeps wave well within 0 - 3.3v output voltage
#define SAMPLE_PERIOD_us 23 //us
#define SAMPLES_PER_SECOND 43478 //Hz - 1/(SAMPLE_PERIOD_us)

//////////sensor data/////////

//distance thresholds
#define UPPER_THRESHOLD 60.0f   //cm
#define LOWER_THRESHOLD 10.0f   //cm
#define OFF_DISTANCE 100.0f     //cm

//frequency thresholds
#define TOP_FREQUENCY 880.0f    //Hz
#define BOTTOM_FREQUENCY 220.0f //Hz
#define OFF_FREQUENCY 1.0f      //Hz


#define UPDATE_INPUT_RATE 0.05f //s


//#define LOGGING
#endif
```

Fig 3.2.5 header file showing defined values for upper and lower distance thresholds, distance required to deactivate output and frequency thresholds.

3.2.1 Calibrating the HC-SR04

The HC-SR04 module requires calibration since it takes a significant amount of time to read the state of the Echo pin. Therefore, this time must be removed from the measured pulse width time to allow for accurate distance calculation.

Before the calibration begins, a value for software delay must be created. Since this value will be removed from the measured Echo pulse width later as a correction, it can be appropriately named "correction". The timer "sonar" created in **Fig** 3.2.2 can be used to measure the delay. To begin the calibration sequence, the timer must be started ("sonar.start()" in **Fig** 3.2.3). After starting the timer, the Echo pin must be read ("while (echo==2) {}" in **Fig** 3.2.3) directly after to obtain a value for the software delay. After the Echo pin is read, the timer must be stopped immediately ("sonar.stop()" in **Fig** 3.2.3) Once this process is complete, the value for the timer sonar will be equal to the software delay in reading the Echo pin. The value must be read and set to the previously mentioned value "correction" ("correction = sonar.read_us()"), now the value can be used to remove the software delay when calculating distance.

3.2.2 Programming the HC-SR04 to correctly pulse, and measuring the Echo pulse width

To begin the distance measuring process, the three values "resultingFrequency", "frequencyMultiplyer" and "distance" must be set to 0 so that they can be calculated later on in the program. These values represent different steps in the process of taking a distance and transforming it into something useful for the project, in this case a frequency. "distance" represents the measured value for distance, "frequency multiplier" represents the value that will modify the frequency via the measured distance and set thresholds and "resultingFrequency" represents the final frequency obtained when these processes are complete.

Once the values are set, the initial 'high' signal can be sent to the trigger ("DigitalOut trigger" in **Fig** 3.2.2) to activate it ("trigger = 1" in **Fig** 3.2.1), around this time the timer used to measure the echo pulse width ("Timer sonar" in **Fig** 3.2.2) can be reset. Since the activation requires a pulse for an arbitrary amount of time, another task can be performed to make this time useful. Furthermore, a blocking problem can be avoided by continuing to produce wave samples during this time, as the waveform output does not have to wait for the distance to be fetched and transformed into a frequency to proceed ("wave.createSample();" in **Fig** 3.2.1). Once 25 microseconds (approximate time taken executing wave sample creation) have elapsed the trigger signal can be set back to 'low' ("trigger = 0" in **Fig** 3.2.1). Since the pulse width of the Echo signal is needed, the program must wait until the rising edge of this signal is detected ("while (echo==1){" in **Fig** 3.2.1). While waiting for the Echo signal to be received, a blocking problem can once again be avoided by creating a wave cycle inside the while loop body until the Echo is received. Once the Echo signal is received, breaking the empty loop set by "while (echo==1) {}" (**Fig** 3.2.1), the allocated timer for measuring the pulse width must be activated ("sonar.start()" in **Fig** 3.2.1). To then correctly measure the pulse width, the program must wait for the Echo signal's falling edge ("while (echo==1) {}" in **Fig** 3.2.1) before proceeding to stop the timer. Once

the Echo signal's falling edge is detected and breaks the empty loop set by "while(echo==1) {}", the timer can be stopped ("sonar. stop" in **Fig** 3.2.1).

3.2.3 Using the Echo pulse width to calculate a value for distance

Once the value for Echo pulse width has been measured, a value for distance can be calculated. In **Fig** 3.2.1 this achieved with:

$$distance \ = \ ((sonar.read\_us() - correction)/58.8)$$

This equation is a derivation of the equation for velocity with some adjustments (notably the value 58.8) related to the specific case created in using ultrasonic ranging. To derive this equation, three things must be first be considered:

The equation for velocity, the speed of sound and units used for the output.

The equation for velocity:

$$v = \frac{d}{t} \tag{1}$$

however, the desired output is distance, so to be useful the equation should be rearranged (both sides multiplied by t) to

$$d \ = \ v \ \times \ t$$

Taking the speed of sound in air to be 340 m/s , the final thing to consider is the output units. Since the distance is wanted with centimetre precision and the timers function in microseconds, the value for the speed of sound in air needs to be converted from metres per second to centimetres per microsecond.

To convert to metres per microsecond, the value is multiplied by 10^{-6} ,

$$340 \ \times \ 10^{-6} \ = \ 3.4 \ \times \ 10^{-4} \ m/\mu s$$

To convert this result to centimetres per microsecond, the value is multiplied by 10^2,

$$3.4 \ \times 10^{-4} \times 10^2 = 0.034 \ cm/\mu s$$

While this is the correct speed, the sound emitted from the module has to travel the distance between the module and a user's hand twice (to and from) therefore, for the final equation to be effective either the speed or the time must be halved. Since the value representing the speed of sound in the final equation is a constant whereas the value for time can change with each cycle of the program, the value for speed will be halved. This results in a usable value of 0.017 cm/µs . Since multiplying by v and dividing by 1/v are the same action, the value can be turned into:

$$1/0.017 \ = \ 58.8 \ \mu s/cm$$

Currently the equation is in the form:

$$distance = \frac{t}{58.8}$$

Therefore, the value for t must be considered. Since it has been established that the time required is the Echo pulse width minus the software delay produced when reading the Echo pin, t can be represented by "sonar.read_us()-correction" as the sonar read statement reads the measured value for Echo pulse width and "correction" holds the value for software delay measured in **Fig** 3.2.3. Substituting this into the equation gives

$$distance = (pulsewidth - delay)/58.8$$

Which is equal to in code form

$$distance = ((sonar.read\_us() - correction)/58.8)$$

3.3 Calculating a usable frequency from distance

Once the value for distance has been calculated, the frequency calculation process can begin. This starts in **Fig** 3.2.1 with the initial check on if the distance exceeds a preset upper threshold (threshold defined in 3.2.5) with an if statement ("if (distance >= UPPER_THRESHOLD){" in **Fig** 3.2.1). If the distance exceeds the threshold, the frequency is set to its minimum ("resultingFrequency = BOTTOM_FREQUENCY" in **Fig** 3.2.1). This is check is repeated to determine if the distance is below a preset lower threshold (threshold defined in 3.2.5) with another if statement ("else if (distance <= LOWER_THRESHOLD){" in **Fig** 3.2.1). In the same fashion as the previous check, the resulting frequency is set to a maximum ("resultingFrequency = TOP_FREQUENCY") if the distance is below the lower threshold.

Once the initial threshold checks have been passed, the frequency multiplier can be calculated. The desired outcome is a frequency multiplier that reflects the comparison between the distance and the difference between the upper and lower thresholds, as if it were a percentage of maximum distance in decimal form. The distance as a percentage in decimal form would be represented by this equation:

$$decimal\ result = \frac{(distance - minimum\ distance)}{Maximum\ distance} \quad (2)$$

In this case, the decimal result is the frequency multiplier, the minimum distance is the lower threshold and the maximum distance is the difference between the upper and lower thresholds. Substituting this gives

$$frequency\ multiplier = \frac{distance - lower\ threshold}{upper\ threshold}$$

This is the equation used in 3.2.1 for calculating the frequency multiplier, in code form:

frequencyMultiplyer = (distance - LOWER_THRESHOLD)/(UPPER_THRESHOLD - LOWER_THRESHOLD)

Once the value for the frequency multiplier has been calculated, the final frequency calculation process can begin. The frequency multiplier must pass through 2 checks in the form of an if-else statement similar to the checks on the value for distance in calculating the frequency multiplier. The first check ("if (frequencyMultiplyer < 0.0f){frequencyMultiplyer = 0.000001f;}" in **Fig** 3.2.1) checks if the frequency multiplier is below 0, this occurs when the distance is below the lower threshold, if the frequency multiplier is below 0, the it is set to a low value. The next check ("else if (frequencyMultiplyer > 1.0f){ frequencyMultiplyer = 1.0f;}"in **Fig** 3.2.1) checks if the frequency multiplier is above 1, this occurs when the distance is above the upper threshold, if the frequency multiplier is above 1, it is set to 1 as this is the maximum value.

Once these checks have been passed, the resulting frequency can be calculated. Since the resulting frequency has a range of values with a maximum frequency of 880Hz and the minimum 220Hz, the product of the frequency multiplier and the range will provide the output. However, the desired result requires that smaller distances produce higher frequencies and larger distances produce lower frequencies. Considering this, the multiplier-range product must be subtracted from the range to create the desired effect. However, this method only calculates the magnitude of the resulting frequency, it is still required to sit between the maximum and minimum frequencies and thus, must be shifted by 220Hz. This gives an equation of:

$$resulting\ frequency = frequency\ range - (frequency\ multiplier\ \times frequency\ range) + 220$$

In code form this is:

"resultingFrequency = ((frequencyRange) - ((frequencyRange)*frequencyMultiplyer))+

BOTTOM_FREQUENCY;"

3.4 Problem with ultrasonic ranging

Since ultrasonic ranging relies on the propagation of sound, there can be significant delay when measuring distances as sound travels slow relative to other waves. This delay . To demonstrate this delay, the equation for velocity can be used;

$$v = \frac{d}{t} \tag{1}$$

Rearranged for time:

$$t = \frac{d}{v}$$

Since this project uses distance scaled in centimetres, a value for d such as 0.4m is appropriate. Using the known value of the speed of sound 340m/s, multiplying the distance by 2 as the sound travels to and from an object and substituting the values in;

$$t = \frac{0.8}{340} = 2.35ms$$

In the context of a time sensitive program, 2.35ms is a significant amount of time to spend waiting for one operation. This delay can easily become the source of blocking problems when constant updates to the output waveform are required with as little time spent between updates as possible.

4. Creating the audio amplifier circuit

4.1 Research

For designing the circuit It was required to do some research beforehand about what class of amplifier to use, suggestions were to look up on what class type of amplifier should be used for our project. some time was allocated to doing research and discovered for the project the best class type of amplifier to be used is a class D amplifier, this is because the class D amplifier is the most efficient amplifier at greater 90% efficiency unlike other classes from A to class F which vary from 25%-90%. During research on the class D amplifier, the website that provides information about the different classes was: Electronic design, reference; Electronic design(5)

After looking at this the group had a collective agreement to use the class D audio amplifier . When it was established to use the class D audio amplifier the procedure continued to look at other resources to help understand and get a more reliability on the concept of the circuit to use whilst basing it around a LM741 op amp. Another source from a YouTube video provided a circuit and explanation on how to build the class D audio amplifier (reference; DIY Class D Audio Amplifier)(6) it was shown in a different video (one which was a more complex version) of the first one that was seen as It was used to compare between different versions of how to make the circuit and understand the concepts behind them (reference; Class D Audio Amplifier (Part 1 – Design and Simulation)(7)). It was found out through this that the build for the class D audio amplifier is very similar to one another so To continue it was decided to use the same concept when designing the circuit for our project and also received feedback from my peers as when It was seen, however some alterations were made to it due to the Nucleo board. Reference; AnalogDialogue(8) this source provided external information on the topic and included some extra information that was not given in the other vied sources such as the range of values to use for the voltage supply into the power stage.

4.2 Circuit build & design

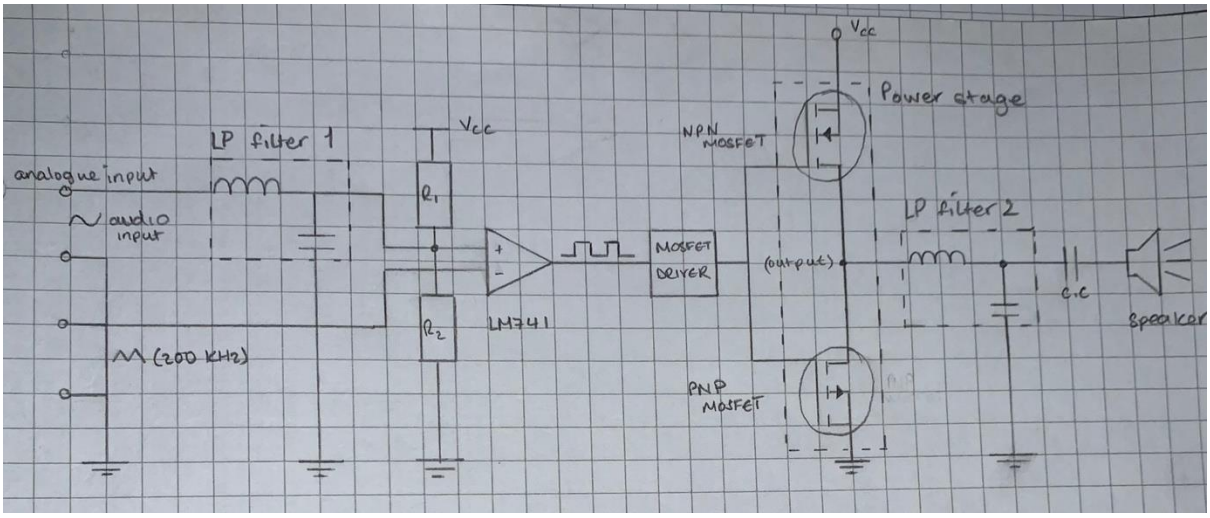


Figure 4.1

When looking at the second video It was discovered that after 3:30 they broke down the circuit into more depth of how it was being used to control bass, treble, balance as this has already been achieved in the coding this was unnecessary so it was needed to add the extra bits into the circuit, nonetheless a low pass filter was added at the analogue input before it goes into the comparator (shown in **Figure** 4.1) to have a clear signal going in as the Nucleo board produces high frequency noise, and it is a hindrance to have noise above 1kHz, an extra 7kHz is required to let the synth produce harmonics but anything higher than the total 8kHz isn't necessary therefore regarding this the values needed of the components to create the low pass filter with a cut off frequency of 8kHz was :

- 10nF capacitor
- 2kΩ resistor

This was done through calculation as the cut off frequency:

$$fc = \frac{1}{2\pi RC} \tag{3}$$

therefore the resistance needed as the capacitor was given is:

$$R = \frac{1}{2\pi(fc)C} = \frac{1}{2\pi(8\times10^{3})(10\times10^{-9})} = 1.989436789\times10^{3}\sim2k\Omega$$

For the comparator, the resistors need to be the same so to receive a 50% voltage drop against the op amp but also need to consider the current of 2.5-5A being produced at the output of the power stage.

The current was calculated given the specs of the speaker having 4Ω impedance and a power output of 25-50w therefore by using the formula:

$$P = I^2R \tag{4}$$

Therefore, $I = \sqrt{\frac{P}{R}} = \sqrt{\frac{25}{4}}$ =2.5A minimum

Max current is when:

$$I = \sqrt{\frac{50}{4}} = 3.53553906 \approx 3.5A$$

And by using this information the voltage output from the power stage can vary from 10-20v.

For this circuit, the Vcc will be 5v for the comparator with resistor values of 10kohm and since R1=R2 both resistors are 10kΩ. The voltage supply (Vcc) for the power stage (which is also known as the switching output stage) being used is 14v.

4.3 Circuit operation

1. Firstly, the analogue input which provides the sine wave signal (audio input) comes from the Nucleo board at pin D13
2. The triangle wave has a frequency of 200KHz which enters the inverting input of the comparator whilst the sine wave enters the non-inverting input
3. As both waves enter the comparator a square wave is generated
4. As the square wave is generated it enters the MOSFET driver which is then produced to the powers stage which consists of an NPN and a PNP MOSFET
5. The enhanced square wave passes through the low pass filter which will turn the square wave signal generated by the power stage into a sine signal that speech will be able to interpret
6. The coupling capacitor (shown as C.C in **Figure** 4.1) is added to block the DC signal as it will affect the speaker as the speaker tends to not favour constant DC signals
7. Then the signal reaches the speaker and the audio is amplified without background noise interfering

5. Working prototype demonstration

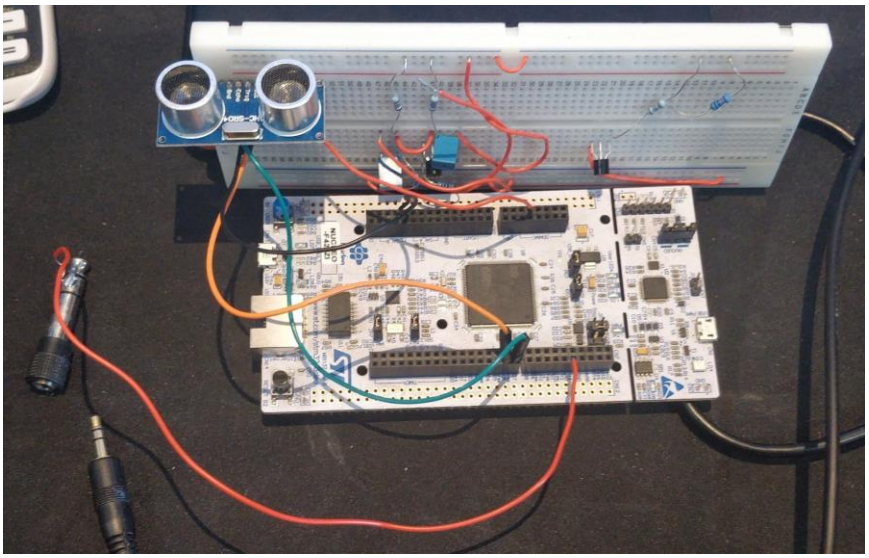


Figure 5 synthesizer prototype

A working prototype of the synthesizer can be seen in **Figure** 5 without output connected to the amplifier. A demonstration of this prototype can be seen in reference 9; Demonstration of Synthesizer – The PicoFarads Group project(9).

6. References

(1) mbed https://os.mbed.com/platforms/ST-Nucleo-F429ZI/

(2) GitHub- link to synth source code https://github.com/Jake-Howell/Synth-OS?fbclid=IwAR34vHkoUXYkE1nX0LHvLXg9Ynzv8k7TnOM8JSYCUJcQ3z9lomejH9j4VFs

(3) Sparkfun https://www.sparkfun.com/products/15569

(4) mbed https://os.mbed.com/components/HC-SR04/

(5) ElectronicDesign https://www.google.com/url?q=https://www.electronicdesign.com/technologies/analog/article/21801654/understanding-amplifier-operating-classes&usg=AFQjCNFKjq6leFFi-8N-cRFS9mzm2fDqoQ

(6) DIY Class D Audio Amplifier https://www.youtube.com/watch?v=3dQjIeYoIdM&feature=youtu.be&ab_channel=GreatScott%21

(7) Class D Audio Amplifier (Part 1 – Design and Simulation)

https://www.youtube.com/watch?v=vWPax49nWic&feature=youtu.be&ab_channel=AllThingsEngineering

(8) AnalogDialogue https://www.analog.com/en/analog-dialogue/articles/class-d-audio-amplifiers.html#

(9) Demonstration of Synthesizer – The PicoFarads Group project

https://www.youtube.com/watch?v=7eH6Q4KMtnQ&feature=youtu.be&fbclid=IwAR04cYIBKmyb3z80zsqwFFJUQnyyMcZxyEdNq5edeTC4uE-Mn0WK9q65znU&ab_channel=JakeHowell (Please read video description for more information)

7. Equations

Section 3:

(1) velocity equation

$$v = \frac{d}{t}$$

(2) distance percentage of maximum distance in decimal form

$$decimal\ result = \frac{(distance - minimum\ distance)}{Maximum\ distance}$$

Section 4:

(3) RC network cutoff frequency

$$fc = \frac{1}{2\pi RC}$$

(4) Equation for power in an electrical circuit

$$P = I^2R$$

8. Workload distribution

Jake Howell; Team leader, lead programmer for F429ZI, writer of section 2 and tester of synthesizer prototype (see **Figure 8** for contributed programs).

Guy Ringshaw; Group project report editor, programmer for F429ZI control of HC-SR04 and writer of section 3 (see **Figure 8** for contributed programs).

Isaac Kulimushi; D-Class audio amplifier circuit designer and writer of section 4.

File name	File type	Author
calibrateSensor	CPP	Guy Ringshaw
createSample	CPP	Jake Howell
getFrequency	CPP	Guy Ringshaw
main	CPP	Jake Howell
mbed_conFig	header	Jake Howell
sawWave	CPP	Jake Howell
sensorData	CPP	Jake Howell
sensorData	header	Jake Howell
setWavePeriod	CPP	Jake Howell
setWaveType	CPP	Jake Howell
sinWave	CPP	Jake Howell
squareWave	CPP	Jake Howell
synth_os.h	header	Jake Howell
triangleWave	CPP	Jake Howell
updateOutput	CPP	Jake Howell
updateOutput	header	Jake Howell
waveType	CPP	Jake Howell
waveTypeSelector	CPP	Jake Howell

Fig 8 table showing files from GitHub (reference (2)), file type and author.

9. Plymouth shell individual appendixes

9.1 Plymouth shell build and test appendix – Guy Ringshaw

Introduction

In this appendix my construction of the Plymouth shell will be reviewed, comparing the construction and build quality of the shell to an ideal case along with an overview of the device combined with the Nucleo F429ZI board. The review will be split into 3 parts; construction review, improvements and device implementation.

Throughout the review, **Fig** 1 will be used as a reference when discussing the name and location of components on the shell.

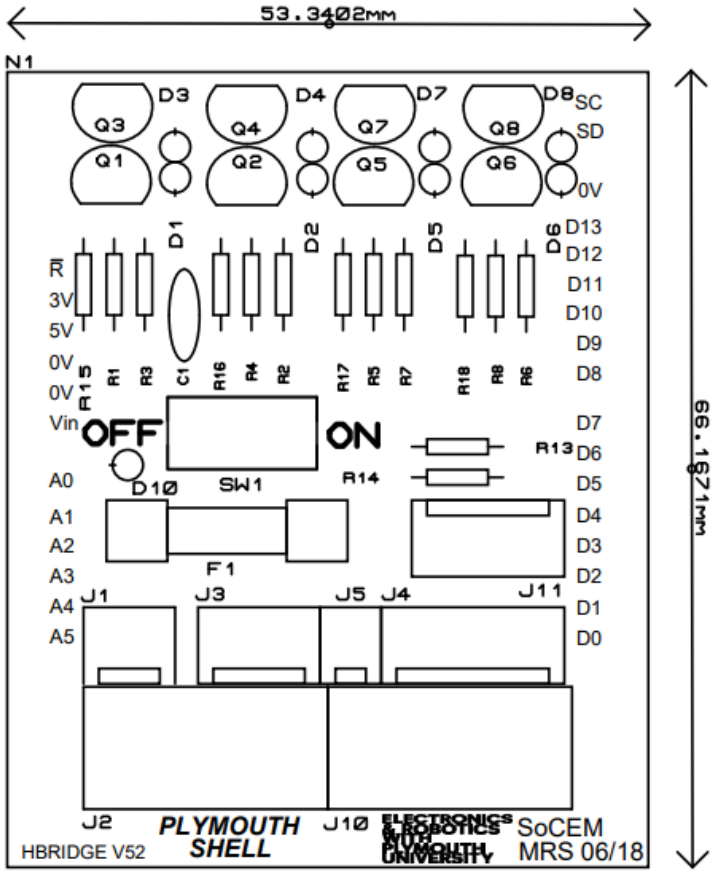


Fig 1: Plymouth shell schematic

Part 1: Construction review

Overall the construction of my Plymouth shell is satisfactory when compared to an ideal case where each component is applied and soldered perfectly. There are some areas where the construction is well executed and others where there are clear improvements to be made. **Fig** 2 shows my construction of the Plymouth shell from a top down view and **Fig** 3 shows the underside of the board.

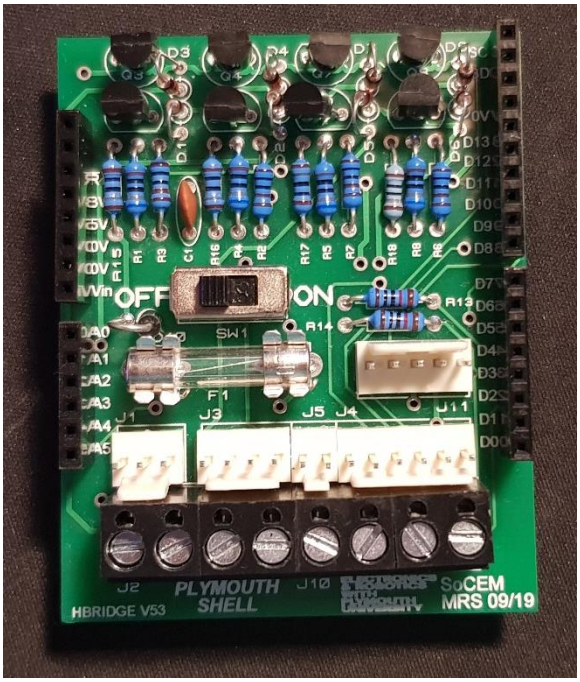


Fig 2 top-down view of Plymouth shell

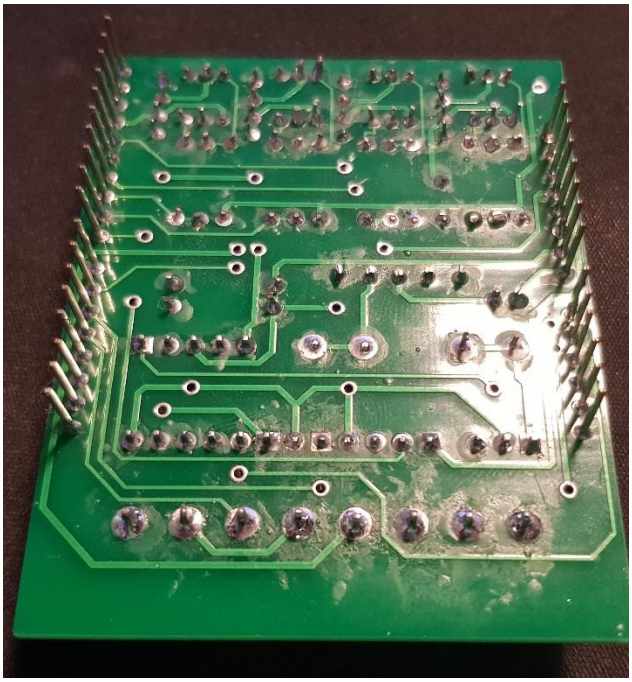


Fig 3 underside view (**Fig** 2 flipped horizontally)

The construction of the resistor networks is satisfactory overall, with the resistors seated flush to the board in between their respective pairs of pins. Furthermore, the soldering underneath on the resistors is generally efficient and tidy, not leaving too much solder on the leads or too much excess wire behind. However, the abundance of grey markings around the pins is evidence of multiple re-solders of the same pin as initially inserted some of the resistors into the wrong points.

The input and output pins have also been seated well, without any of the black rows bent severely. However, there is more evidence of multiple re-solders around the J2, J3, J5 and J10 pin sections on the board. This was due to the J5 pins being initial inserted incorrectly, as it was not level, interfering with the other components around it. Furthermore, the J2 section was re-soldered as initially I failed to realise J2 could be connected to J10 via a slot accessed from below and soldered as one component, keeping the entire section level, as I had attempted to insert J2 first.

Another set of components that were not constructed as well are the transistors Q1-8. Q3 and Q5 have bent leads most likely due to poor handling, with Q5 almost being bent enough to lean into the diode D2 and not fit its contacts. Furthermore, on the underside of the board a few of the pins have excess solder and wire protruding from the contact.

Part 2: Improvements

Upon reviewing my construction, it appears there are a few key flaws that appeared multiple times throughout the construction process. The most significant being misplaced components requiring re-soldering attempts that risk damage to the board, components with deformed leads and misinterpreting the construction process with certain components.

To correct these errors in a second attempt of the construction, I could introduce new methods to ensure that the same oversights do not re-occur. Since the most significant consequence of my flawed

methods was the constant re-soldering as that risked damage, in a second attempt I would thoroughly analyse and pre-plan how each component is supposed to be inserted, checking before and after placement whether the correct component has been used. This would ensure that in cases where there are multiple similar components such as the row of resistors, the correct component is used. Furthermore, with the checking system, it further reduces the chance of the incorrect component being used and reduces the number of components that need to be re-soldered as the mistake is more likely to be identified earlier.

Another significant flaw in my construction was the transistor Q5. To rectify this, the components would be examined before construction starts. This way any damaged components can be identified early on providing plenty of time to seek out replacements. Furthermore, with the transistors, using solder more moderately would result in a tidier, less wasteful construction also leaving more of the wire exposed to be cut cleanly.

Part 3: Device implementation

Fig 4 and 5 show the Plymouth shell attached to the Nucleo board. Once the correct input and output pins are lined up the board can be easily slotted into place. Furthermore, **Fig** 4 shows the piezo buzzer attached to the J5 output pins and the battery terminal connected to the J1 pins.

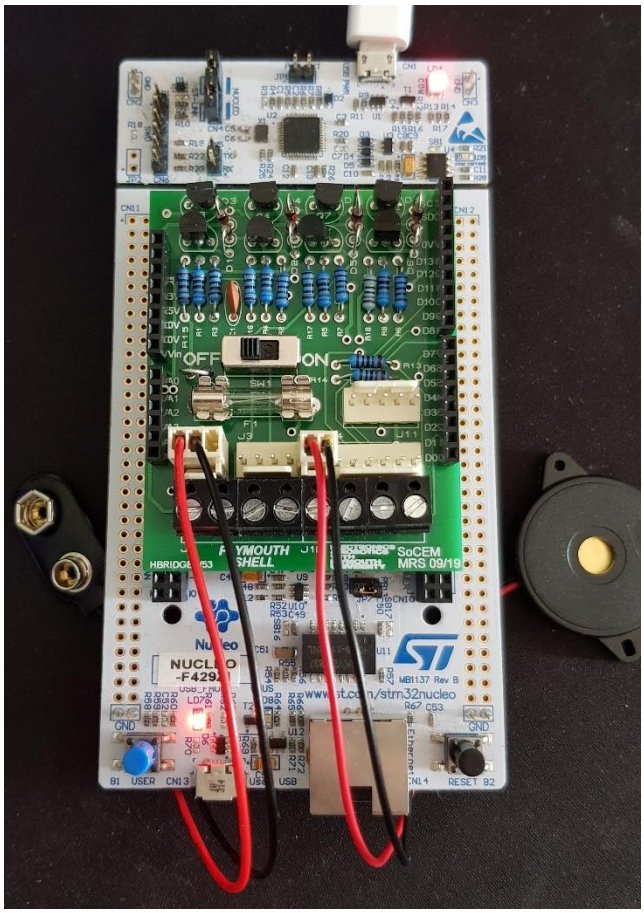


Fig 4 Plymouth shell attached to Nucleo board

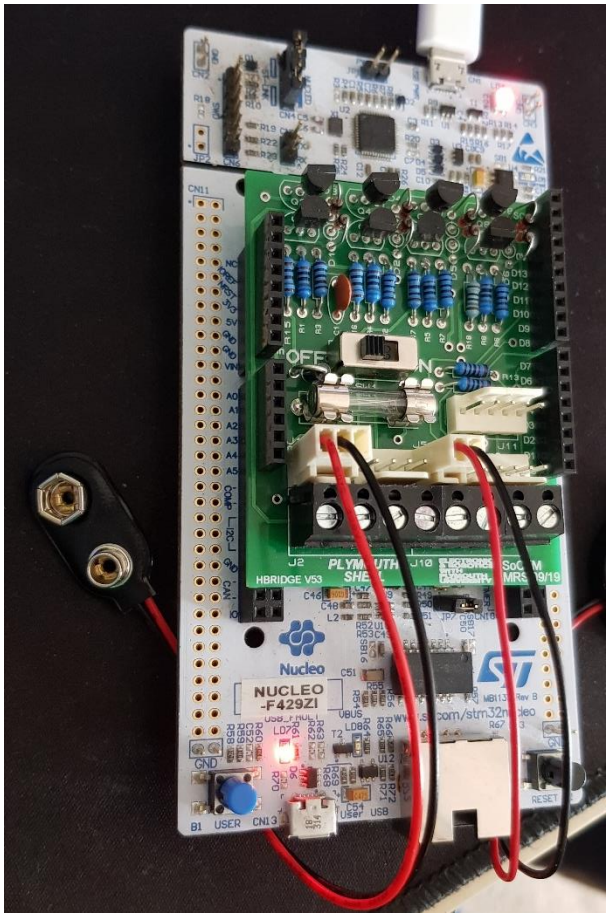
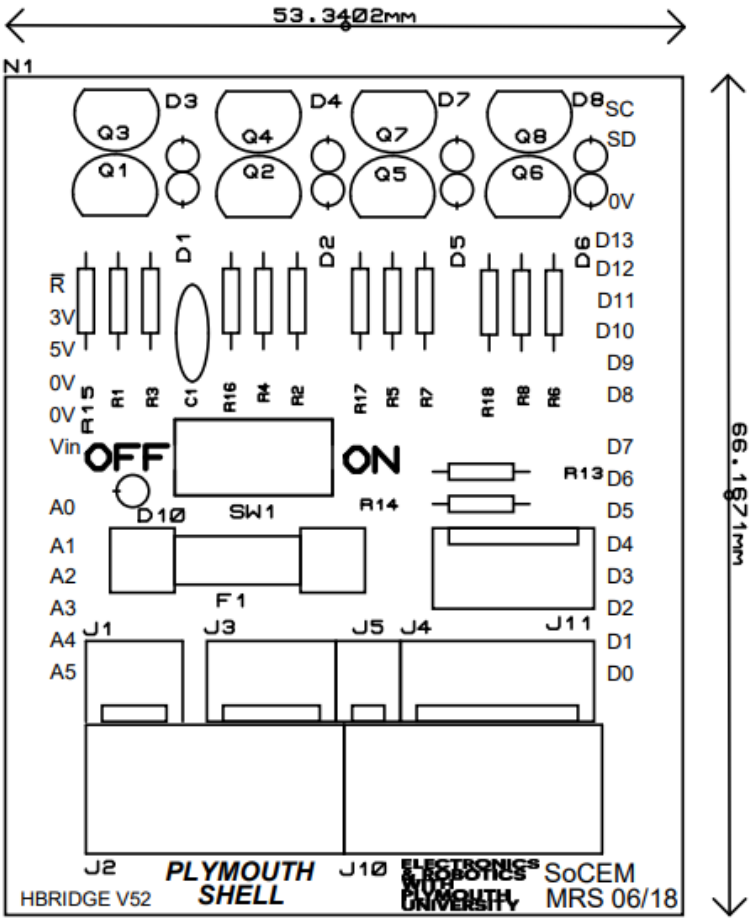


Fig 5 Showing the I/O pins lined up

9.2 Plymouth shell build and test appendix – Isaac Kulimushi

Introduction

In this appendix I am explaining the making of Plymouth shell that I did and have given a schematic of the PCB board that I'll be using. In the process of creating the Plymouth shell I have given the stepson what I did to create and explanations on what I did though the assembling of the shell, and I'll be using the schematic shown in the diagram below as a reference to where I'd be placing certain components.

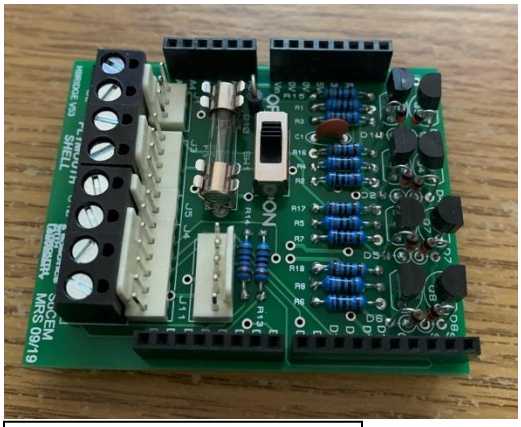


Assembly

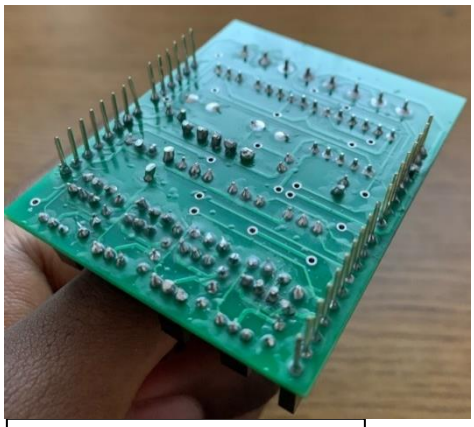
When assembling the shell, we had had to solder components onto the printed circuit board (PCB)

1. First was the R1 resistor, I had to place the 680ohm resistor in the slot R1 then had to the flip the board and bend the legs of the resistor outwards so it would stay in place when soldering it in to the PCB.
2. I had to repeat this step for all the other resistor pins that was in the same column as the R1 pin,
3. After I soldered all the resistors in that column if the PCB I had to solder two more resistors into pin R13 and R14 and with pin R14 since it had three holes it was given that we had to use the outer two holes.
4. Next was to place a 100nF capacitor and place in the pin allocating C1 and solder it on.
5. Next was to place and solder switch SW1
6. After this was all done, I had to use the long pin connectors 10 way, two 8 ways and a 6 way where I'd need to place them on the edges of the PCB. When soldering the long pin connectors, it was advised to solder one pin into position to make it stable and easier to solder all the other pins later

7. After this was achieved, I cut the legs off all components except from the long pin connectors because they were going to be used later to be joint with the NUCLEO-F4291 board
8. Next I had to select two 4 way screw PCB terminals and slide them together into J2 and J10 to form an 8 way block then had to flip the board over and solder them at right angle to the board so it would be connected properly the soldering is done similar to the long pin connectors by soldering one pin first and once adjusted correctly solder on the rest
9. Next The NPN transistors were place on the opposite end of the board to the two 4 way screw PCB terminals and we had to make sure the orientation indicated how the transistor should be positioned and I had to repeat this for all the transistor slots and solder them on afterwards
10. Next, I had to select the 1N4148 diodes and bend the leg at where the BLACK banded ends this was achieved by using a snipe nosed pliers to protect the diode body as it's made of glass. The positioning of the diode Shippuden be the Amos places into the pad with the white circle printed in the PCB and they all have to be levelled with the two 4 way screw blocks.
11. After this was completed I had to select a 6 way moles pin header and place it into J4 I had to ensure that the connector was flat and proceed with the same soldering pattern as done earlier, then I had to repeat this process with J5,J3,J1 and J11 slots
12. Next, I had to use the 1N5819 diode and place it at D10 however with the cathode in the square pad and then proceed with soldering
13. After all this was achieved I removed a the excess legs on the PCB leaves only the legs of the long pin connectors, I then got a supervisor to check my board and once it was approved I continued on with the assembly of the cable to Molex connectors



The overview of the PCB



The bottom of the PCB

Improvements

In this case I could of done some improvements with the soldering as it's shown on the PCB there's quite a few grey marks showing that I did in fact re-soldered the pin legs a couple times which doesn't make the presentation seem very nice and clean, also with the soldering there are a few pins that have a lot of soldering done to them which isn't really the greatest thing to have as it could damage some of the components that are being soldered onto the PCB board.

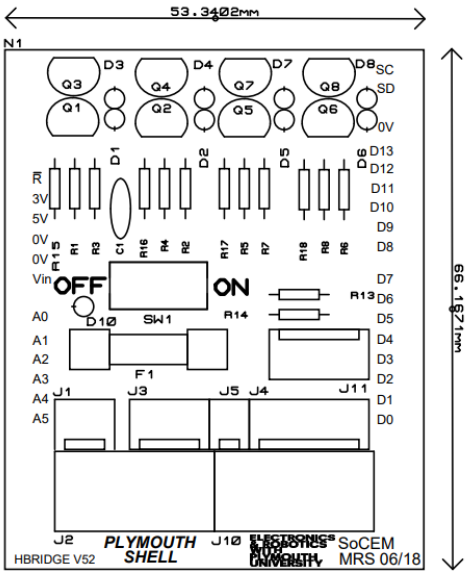
9.3 Plymouth Shell build and test – Jacob Howell Student ID:10656206

Introduction

This appendix will discuss the build process of my Plymouth shell board. I will comment of the build quality of my work and improvements that could be made with a corresponding method.

Images of the final product will be included with closeups of important talking points.

Figure 0.1: Plymouth Shell Schematic



Section 1: Construction Process and build quality

My construction of the Plymouth shell overall was good. I took plenty of time to carefully read the step by step instructions, measure each resistor value before soldering and check the model number of each transistor before soldering each component.

I made one minor mistake where I didn't properly seat the J3 connector (see **Figure 0.1**)header (in the red box in **Figure 1.5**). This just made it slightly harder to connect and disconnect the SC-SR04 Ultrasonic Sensor.

Figure 1.1: view of all components

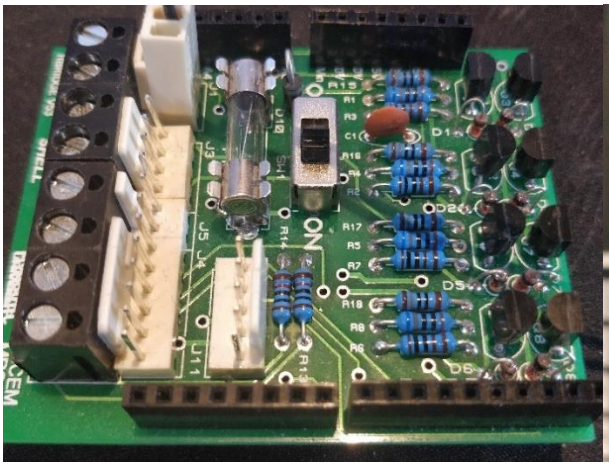


Figure 1.2: view of all solder joints

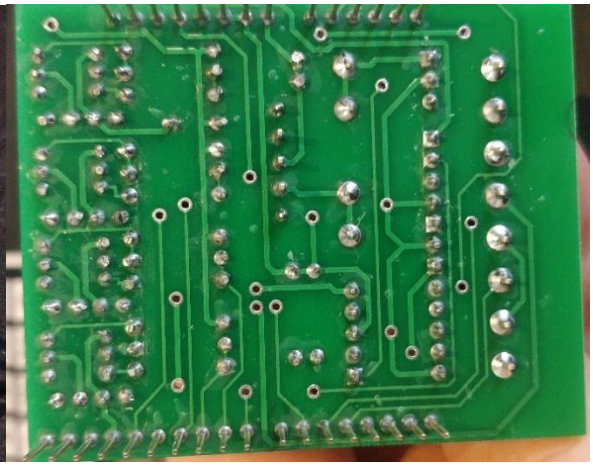


Figure 1.3: Close up view of solder joints

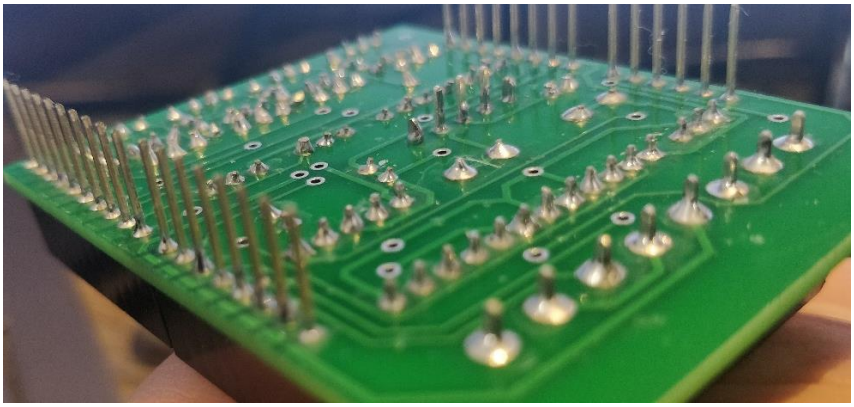


Figure 1.4: Close up view of H-Bridge circuit and resistor network

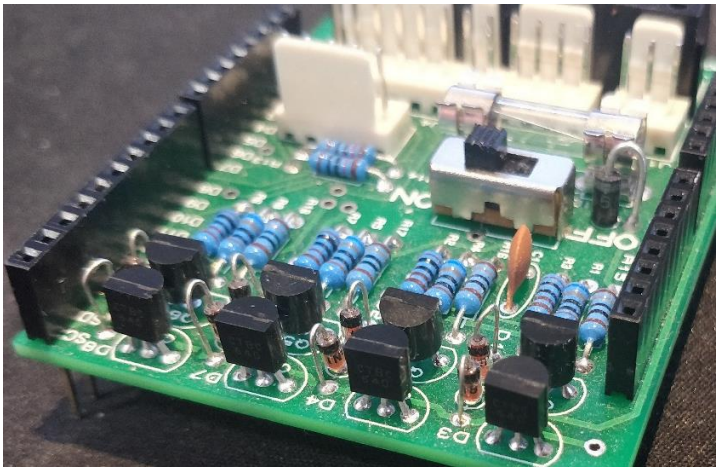
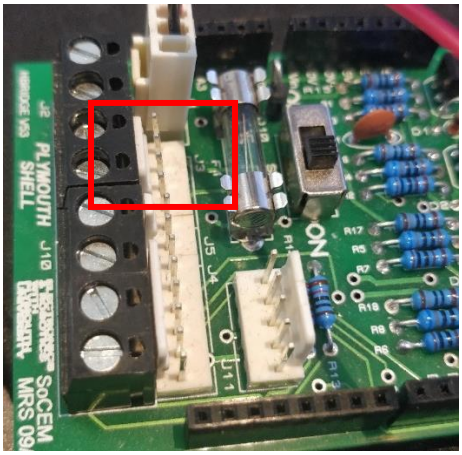


Figure 1.5: Poorly seatet connector J3



Construction of the input and output pins:

As mentioned above, this is where I made a mistake seating the connector J3. As you can see in **Figure 1.5** the backside of the connector is touching the connector block J2. I is visible that the pins of J3 are not in line with the rest of the input/output pins. This was very difficult to correct as I failed to de-solder all the pins on the rear of the board and therefore could not re seat them.
the rest of the input/output pins were seated and soldered to a satisfactory level. (See **Figure 1.2** and **Figure 1.3**)

Construction of the Resistor Networks:

I took care while selecting each resistor to shape and solder next, by measuring the value with a multi-meter. This is because they are easily mistaken for the wrong value at a glance.

I bent the leads at a 90 degree angle using some plyers from the lab, threaded them through the corresponding junctions, flipped the board over while ensuring the resistor was still tightly seated, Bent the leads out by 45 degrees to allow me to solder them in place. I was careful not to use too much or too little solder to avoid creating a dry joint, or creating a short circuit with another junction. Finally I clipped the excess lead off with plyers too keep them out the way for my next solder job

Construction of the H-Bridge:

For the H-Bridge, I was instructed to take care selecting the correct transistor. And place it the correct way around.

I used a similar process as above to seat and solder the transistors to the board.

Soldering the Diodes:

The diodes prevent current from running the wrong way through the H-Bridge so it was essential that they were orientated correctly. To save space on the board, the diodes were seated vertically with the exposed lead bent 180 degrees back into the junction. I tried my best to not leave any excess lead before the joint as this looks untidy and would allow the diode to wobble.

Fitting the Switch and fuse connectors:

These components allow the board to be turned off, and prevent an spike in current from frying the board, so it is important that they are fixed with a good quality solder joint, as they handle more current that most other components on the board. A dry joint here could make the circuit temperamental or not work at all. The switch and fuse connector do not require soldering in a specific orientation, as long as they are seated and fixed well.

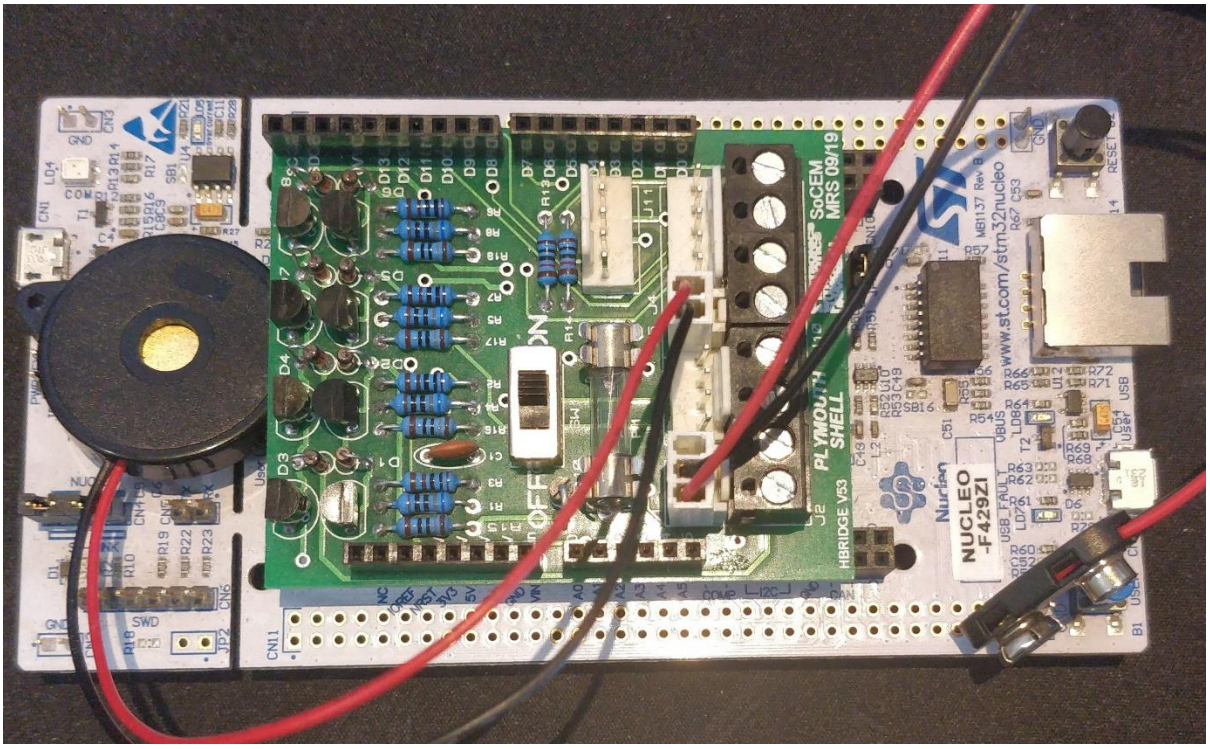
Section 2: Improvements

The main improvement required for my build implementation would be improving the seating of components.
Some were almost perfect, most were satisfactory, but as mentioned above, the J3 connector was unsatisfactory.

To ensure this error doesn't happen I will only solder one pin into place, flip the board to check if it is seated properly, make any adjustments by reheating the single solder joint and finally solder the rest of the pins in place when the seating is satisfactory.

Section 3:- testing the Plymouth shell using the provided sample code

Figure 3.1: Plymouth shell seated on NUCLEO-F429ZI controller board



Once the shell was checked by a technician, I got the go-ahead to connect it to my Nucleo board and test the Piezo output with the sample code ROCO104_base_template.This played various tunes through the Piezo confirming that this component was correctly wired up.