|  |
| --- |
| The University of York |
| Electronics Department |
| Design & Construction |
| Final Report |
|  |
| **Jonathan Caine**  **Sam Beedell**  **Alexander Brown**  **Liyu Ding** |
| **06/05/2014** |

# Group Summary of Project

The group feels that overall the project was a success, as we managed to meet the vast majority of the level 2 and level 3 requirements outlined in our initial specification. We managed to exceed them in some places, such as with the maximum achievable frequency for sine and square wave generation. In places where we were unable to achieve the desired specification such as with amplitude and frequency modulation, we replaced them with alternative functionality for example frequency shift keying modulation, or added additional functionality such as an arbitrary function generator.

The software team managed to generate both sine and square waves using the DDS on the AD9850 module, which exceeded the 1MHz requirement outlined in the initial specification. This is because the DDS chip itself has an upper limit for the generation of waves at 40MHz, although they become extremely distorted above roughly 5MHz. As a software team we choose to limit the output frequency of the DDS chip to 35MHz to protect the chip itself, and to leave the decision up to the user of the system whether they would like to use the waveforms at high frequencies, even though the distortion becomes very significant the higher the frequency selected.

The software team also successfully managed to create a random noise generator, however it did not use the internal PR sequence generator. Instead while implementing the triangle wave generation using the DAC, we noticed in the data sheet [1] that the DAC has the ability to generate small amplitude noise signals that can be added to another waveform. We based our code around the example code from the peripheral examples by the ST Micro Electronics application team, provided with the CMSIS peripheral library. This resulted in a nice random noise signal that could be easily turned on or off, by enabling or disabling the appropriate channel of one of the DACs.

As a result of the software team’s success in generation of the signals the main tasks for the hardware team were to create circuitry to feed an appropriate input to the board and also to take the output from the board and transform it to a much more useful level. This was achieved using a combination of circuits based around operational amplifiers: a comparator, frequency divider, amplifier, integrator and a DC-offset circuit.

## Full System Review

After combining the software and hardware sections of the project, the overall product was pretty successful at its intended functionality. It provided sine and square wave generation from approximately 0.03Hz up to 35MHz, with the ability to adjust the frequencies “on the fly”, and triangle wave generation again from approximately 0.03Hz up to about 1MHz. An accurate frequency meter across a range of frequencies from 0.1Hz up to 10MHz and above, although the accuracy is noticeably reduced above 10MHz, and which displays the frequency and duty cycle on the LCD screen. Also a predefined “arbitrary” waveform in the form of a sinc function, a small amplitude noise generator, a pulse generator with the ability to vary the duty cycle and frequency range, and frequency shift key modulation (FSK) for a selected range of carrier frequencies.

Any input signals to the product were prevented from causing damage to either the STM32F4 board or the carrier board, by passing them through a crossing detector circuit, which shifted the amplitude of the signal to be between 0 and 3 volts, and provided an equivalent frequency square wave to input to the carrier board. This meant that theoretically any signal could be able to be used as an input to the product, including sine waves and random or arbitrary waveforms. The product also has the ability to adjust certain parameters of output signals, such as the frequency and duty cycle from the software implementation, and the voltage range and amplitude from the hardware implementation. The carrier board can only output signals in the 0 to 3 volt range, but by passing these through an amplification circuit with a potentiometer to vary the amplitude, followed by a level shifting circuit, the output waveforms can be varied from -12 to +12 volts.

The group feels that the “arbitrary” waveform could have done with more work, so that ideally the user could have downloaded any waveform into the system from a connected computer running MatLab. MatLab was used to generate an array to represent the desired waveform to the precision required by the DAC. If the software team had had extra time, they would have liked to implement a serial over USB protocol, also known as a Communications Device Class (CDC), to be able to use a console window on the connected PC, to send data to the product. This would have allowed the user to define a true arbitrary waveform of their choosing in MatLab, and then “send” it to the board ready to be output.

Also the user interface that was available on the board for the demonstration afternoon, was not very user friendly and appeared buggy. This is mainly due to the lack of hardware de-bouncing provided on the STM32F4 board for the blue user button. This will be expanded on in the “Problems Encountered” section, however the group would really have liked to have had a much more suitable user interface for the product.

Overall the product was a great success, but the group would have liked to have more time available to further enhance the product, and develop it further to a full prototype stage rather than leaving it in mid-development.

## Problems Encountered and still Outstanding

The most noticeable problem that the group encountered was the poor user interface available on the board during the demonstration afternoon. In conjunction with the hardware team, the software team had ordered a 1-pole 12-output rotary switch, which was intended to be used as an alternate method to select the desired functionality of the product. This is because using the blue user button the STM32F4 board resulted in it being very difficult to select the desired functionality, as there was no hardware de-bouncing associated with the input. The software team tried to implement a form of software de-bouncing, by not clearing the interrupt pending flag until the very end of the IRQ handler routine, which means that a rapid second press or prolonged press of the blue user button, cannot generate another interrupt until the IRQ handler routine has completely finished executing first time around. However as the M4 processor on the STM32F4 board executes instructions so quickly, and the IRQ handler routine for the blue user button was so short, it finished executing the routine so quickly that a second or third interrupt was able to be generated immediately after the first one had finished executing, resulting in rather than stepping through the different product functionalities one at a time, 2 or 3 would be jumped through on each button press. The solution to this was to use the rotary switch and write an IRQ Handler for each output of the switch, eg one for each different functionality provided by the product. Therefore when the rotary switch was turned to a desired output, the desired output would always be set correctly, as each IRQ handler routine would only set 1 functionality rather than all of them like using the blue user button required. Unfortunately the company the group ordered the rotary switch from made a mistake, and shipped some accessories for the rotary switch such as the cap and a plastic ring to identify the switch positions, but failed to ship the actual switch. This meant that the switch finally arrived about 30 minutes before the demonstration afternoon was due to start, and although the software team had tried to write the appropriate IRQ handler routines and configure the EXTI lines appropriately, when we tried connecting the switch it didn’t work as expected, and we did not have time to trouble shoot the problems before the demonstration afternoon was due to start. We therefore reverted to the blue user button as a user interface for the demonstration afternoon, as at least it worked, and have since got the rotary switch working as desired.

Another problem the software team encountered was that, when they initially wrote the IRQ handler for the EXTI\_Line0 (blue user button) calls were made in the IRQ handler to LCD screen functions, which in turn called the Delay() function. This caused us serious problems that took several weeks before an acceptable work-around was finally found. The problem itself was never fully resolved, although it was partially caused by the fact the in the early stages the software team had tried to change the system core clock frequency by simply changing one of the #define values in the system file. This causes problems for the board, as it makes the software program think its running at the new frequency, however it is not as the clock itself hasn’t actually been changed. The solution to this was to download the cock configuration tool provided by ST MicroElectronics, and regenerate the system file with the desired system core clock frequency set accordingly. The work-around that was suggested by Dr Andy Pomfret, wasn’t so much of a work around as more advice on best practice for coding. He outlined to the software team why calling blocking functions (functions that stop the CPU executing any other instruction while it “waits” for a period of time to elapse), even if implicitly, while inside an IRQ handler was very bad practice, and should be avoided at all cost. This meant the software team had to implement a flag system in order to update the LCD screen. Rather than calling the LCD functions in the IRQ handler, they set a variable called UpdateFlag to 1, and then test if this is equal to 1 every time the code enters a new clause inside the while loop of the main function. If it is equal, they then call the appropriate LCD functions, which has worked in their favour in the end, as it made updating the LCD screen with the appropriate text when buttons were pressed significantly easier.

The hardware problems are detailed in the hardware team’s section of this report, their major issues tended to be operational limitations towards higher frequencies and noisy signals when multiple circuits were combined.

# Hardware Team Design/Implementation Report

This section details the hardware team’s work since the first report. It has been broken down into two main sections: the input stage and the output stage. Each section details the relevant circuits along with any problems encountered by the team. In the first report we said that we would continue to look at signal generation in hardware, this has not been the case as our software team have been capable of producing signals which meet the specifications set. The only signal they haven’t created is a triangle wave, which has instead been realised by running the square wave output through an integrator circuit.

## Input stage:

The input stage could be divided into three parts, Comparator, Potential Divider and Frequency Divider. They are designed to transfer analogue waveforms into digital binary signal so that the software team would be able to detect the frequency of the input signals by their board. Meanwhile the hardware team must guarantee the voltage of input signals are lower than 3V in order to protect their PCB board. The following sections will individually introduce each ‘part’, detailing the design, theory, purpose and limitations, before a short section introduces how they are combined.

## Comparator:

|  |  |  |  |
| --- | --- | --- | --- |
| Component used in the final circuit | Number of components | Value | Components be replaced |
| LM311 | 1 | N/A | LM393 |
| Resistor | 1 | 3.3kΩ |  |

A comparator circuit would be able to transfer square, triangle and sine wave into a square wave digital signal as output. It is designed to produce well limited output voltages that easily interface with digital logics.

As our team decided to implement a mixed digital approach, the Appendix includes the circuit diagram of comparator. Initially we used ½ of LM393 circuit (shown in the lecture) but found the output to be particularly noisy and the up frequency limit was quite low. As an alternative, we tried comparator with Hysteresis, with the intention of improving the noise performance, however the results of testing showed it made things worse than before. In order to improve the performance, we designed a circuit as figure 2 shows. LM393 is replaced by LM311, with two 100nF decoupling capacitors are added at VEE and VCC to reduce the noise effect. A 3MΩ potential meter is used to vary the output voltage in order to make it fit with other circuits’ requirements.

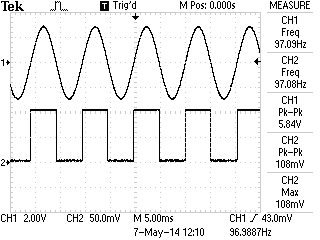


Figure 2: Output of Comparator

Figure 2 is the oscilloscope’s screenshot of our comparator’s output. Channel 1 is a 5.84V sine wave input in 97.1Hz. Channel 2 is the output of comparator circuit. As the figure shows, frequencies of two signals are almost the same, which proved that comparator was working as per expectations.

Limitations:

* Amplitude: Power supply: +/- 15V Input voltage: Maximum of twice as power supply

According to the datasheets [2] power consumption of LM 311 is +/- 15V. If the power supply to it is lower than that level, the circuit still could work as expected but we found that when the voltage of the input signal was larger than the twice of comparator’s working voltage an extra period of square wave appeared at the output of comparator. Figure 3 shows that effect. Comparator got working voltage around 4V at that time when voltage of input signal is higher than 8.1V comparator didn’t operate as before.

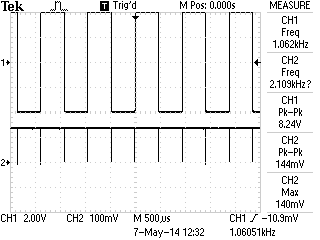
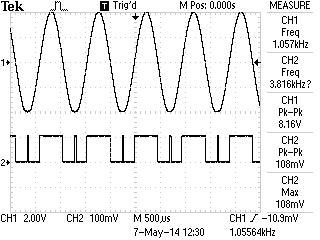


Figure 3. Extra period effect

* Frequency: Maximum: 8MHz

According to the slew rate effect (figure 4) of circuit when the frequency of input increase to a high range, the voltage of output signal will start to decrease. In our design comparator is followed by a frequency divider circuit, that circuit has a minimum voltage requirement about input signal which should be higher than 1.4V, otherwise counters would ignore that signal.

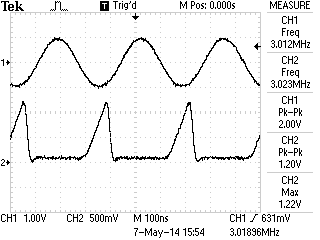


Figure 40: Slew rate effecting comparator’s output

In this case we connected the output of comparator to the input of frequency divider during testing, to find the maximum significant frequency that comparator can produce. Figure 4 is the maximum frequency.



Figure 5: Highest significant frequency comparator can achieve

Channel 1 is the output of a decay counter Channel 2 is the output of comparator. Comparator’s working voltage at that time was 14.3V. Signal input to comparator is a 1.08V sine wave. The image shows Channel 2’s frequency as 10 times larger than Channel 1’s, meaning the frequency divider could still detect the comparator’s output.

## Frequency divider:

|  |  |  |  |
| --- | --- | --- | --- |
| Component used in the final circuit | Number of components | Value | Components be replaced |
| 74LS90N | 3 | N/A |  |

In order to help the software team to measure high frequency waves, we designed a frequency divider circuit as shown in the Appendix. All of 74LS90N are connected in the way of decay counter. In this case connecting 3 counter circuits in series gives us a frequency divider which could divide input digital signal by 1000. Figure 5 shows how one 74LS90N works as a decay counter. Channel 2 is the TTL digital signal input and Channel 1 is the output. Measurement at right side of the diagram proves that dividing function is working as anticipated.

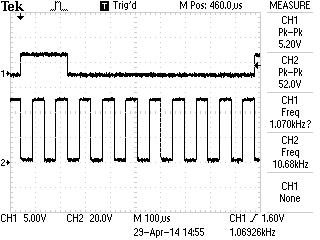


Figure 6: Frequency divider ÷ 10

Limitation:

* Amplitude:

According to the datasheet of 74LS90N, both the power supply and the input signal voltage must be equal or lower than 7V. [3] This is the reasoning behind placing a potential meter at the output of the comparator, in order to protect the frequency divider.

* Frequency:

In the datasheet the ideal highest counting frequency of 74LS90N is 42MHz. Due to our circuit design, the frequency divider only receives input from the comparator’s output. In this case the highest frequency that the divider circuit can produce is enslaved to the comparator’s output. The maximum frequency we recorded in that situation is shown in Figure 4.

Problem:

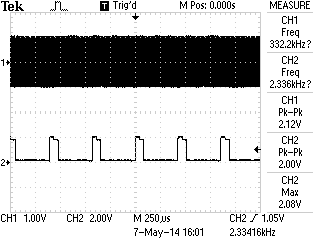


Figure 7: Frequency divider ÷ 1000

Figure 7 shows the performance of when 3 decay counters were connected together. The input frequency (CH1) from comparator is 2.3MHz, for some reason that scope can’t measure it in that scale. CH2 is the output of frequency divider, measurement of ch2’s frequency proves that counters are working as expected. Problematically, the outputs of the divider are not in a perfect square wave form. There is a slight step at the peak of square wave. Whilst being unsure of what caused this, it didn’t affect our result of frequency measurement.

## Potential divider:

|  |  |  |  |
| --- | --- | --- | --- |
| Component used in the final circuit | Number of components | Value | Components be replaced |
| Potential meter | 1 | 3MΩ |  |
| Potential meter | 1 | 2MΩ |  |

The main aim of using potential divider circuits is to reduce the voltage of each circuit output and protects the circuit following it.

3MΩ potential meter is placed in comparator circuit to make sure that the output of it is always lower than 7V. 2MΩ potential meter is placed at the output of frequency divider to protect software team’s board which only could take 3.3V input signal as maximum.

## Output stage

The output stage consists of three circuits: an integrator, an amplifier and a DC offset. This stage has been designed to transform the low level signals output from the board to a useable level. Each of the three sub-circuits was first built and tested separately before being integrated into the master circuit. All of the op-amps used in each circuit have de-coupling capacitors at the power inlet for the IC to maintain a smooth supply voltage and reduce noise interference. The following section details the work carried out to create each individual circuit. This is followed by a short section describing the integration of each sub-circuit into the overall output stage circuit. See the appendix for all circuit diagrams.

## Integrator

The first circuit in the output stage is an integrator. While its operation may seem obvious from its name, it is one of the more subtle properties for which we have chosen to use this circuit; when a square wave is input into the integrator the output will be a triangle wave. We required this circuit as the software team found they were unable to generate triangle waves at the frequencies required by the specifications laid out, and we found that the hardware solutions also failed to operate at the required frequency (see the previous report for details of this). With the software team able to generate very high frequency square waves this solution seemed most appropriate as the only immediately obvious limitation is the slew rate of the op-amp used. This circuit’s output is controlled by the capacitor charging and discharging which, with a square wave input, forms the triangle wave output desired. For the initial prototype of the circuit we used a 741 op-amp and a 1µF capacitor; however during testing we realised that the RC time constant became a limiting factor for the frequency range over which the integrator could operate and provide a clean triangle wave output. We decided that switching the capacitor would be the best solution for this and found that a 100pF capacitor allowed operation up to the highest frequency range while a 2.2µF capacitor was required to get a good triangle wave at 1Hz. Table 1, below, shows the useable frequency range for a selection of capacitors.

|  |  |
| --- | --- |
| **Capacitor** | **Frequency Range** |
| 100pF | 30kHz – 3MHz |
| 15nF | 350Hz - 220kHz |
| 1µF | 10Hz - 1kHz |
| 2.2µF | 0.2Hz – 20Hz |
| 47µF | 0.2Hz – 1Hz |
| *Table 1: Operational frequency range of various capacitors* | |

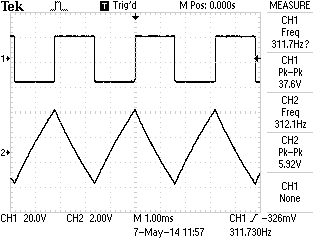
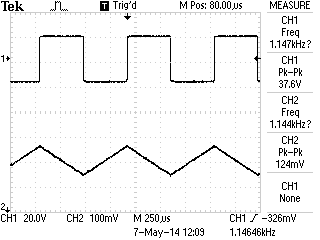


Figure 8: The integrator with a 47nF capacitor on the left and on the right the integrator with a 68uF capacitor

These results show that no single capacitor is able to operate of the entire range of frequencies desired. Larger capacitors are required for lower frequencies to ensure that the signal doesn’t saturate, while smaller capacitors are needed with increasing frequency due to the voltage drop encountered. To achieve the full range of frequencies laid out in the specification different capacitors are required to be switched into the circuit, with the user selecting the appropriate capacitor for the desired frequency range. The circuit diagram in the appendix shows how, by using a 1 to 4 switch, this functionality is achieved.

There were two major limitations to the operation of this circuit. The first was the slew rate of the LM741 op-amp (0.5V/µs [4]); it was unable to handle the highest frequencies. We changed it out for a LM318 op-amp which overcame this problem due to its higher slew rate of 50V/µs [5]. The second major limitation is the fact that as the frequency approaches the upper limit for the capacitor being used there is a significant voltage drop. Fortunately there is a large overlap between each capacitor’s operating ranges which counters this problem, except at the highest frequencies where there is no overlap. This proved particularly problematic as the output from the board was already a very low level signal. Our solution was to run the output from the board through a gain stage first, followed by the integrator then running it through a final gain stage.

## Amplifier

The amplification is carried out using a relatively standard set up for a non-inverting operational amplifier. The output from the board is limited to a maximum level of 3.3V, hence the need for the amplifier circuit. To allow for the gain to be adjusted the pair of resistors, that form a potential divider in the feedback loop, have been replaced with a 200kΩ potentiometer and a 10kΩ resistor to ground. Adjusting this pot changes the gain of the amplifier. Table 2 shows the frequency response of the amplifier with a fixed gain of 2.

|  |  |  |
| --- | --- | --- |
| **Frequency** | **Vin** | **Vout (Gain = 2)** |
| 0.2Hz | 2.00 | 4.00 |
| 1Hz | 2.00 | 4.00 |
| 10Hz | 2.00 | 4.00 |
| 100Hz | 2.00 | 4.00 |
| 1kHz | 2.00 | 4.00 |
| 10kHz | 2.00 | 4.00 |
| 100kHz | 2.00 | 4.00\* |
| 1MHz | 2.00 | 2.00\* |
| 10MHz | 2.00 | 0.28\* |
| *Table 2: Frequency response of the amplifier* | | |

\*At these frequencies the slew rate of the op-amp became a limiting factor.

As can be seen up to 100kHz the amplifier worked as expected however beyond this there were several problems. Figure 2 shows the scope capture of the circuits output.

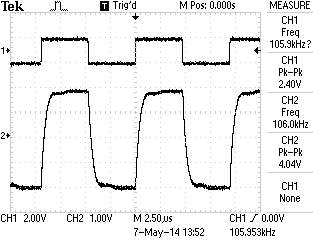


Figure 9: Scope capture from the amplifier at 100kHz

Theoretically the LM318 should be able to handle a signal of about 2MHz for a 4V output.

Where:

* f is the frequency in Hz
* SR is the slew rate per second
* V is the peak to peak voltage of the output

It is therefore surprising to see that the output signal is showing signs of distortion that can be caused by having too low a slew rate. Inputting higher frequencies the distortion became worse.

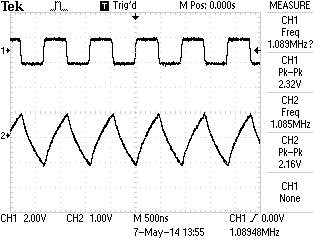


Figure 10: Op-amp fed with a 1MHz signal

We tried adjusting the gain of the amplifier circuit, which improved the shape of the output signal, see figure 4.

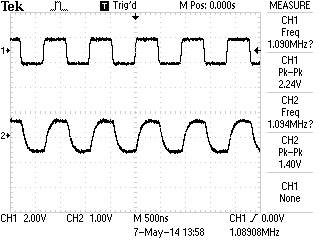


Figure 11: The result of changing the amplifiers gain

Unfortunately we have been unable to work out what is causing this problem.

## DC offset

The DC-offset circuit’s operation is fairly simple, taking the input signal and adding a constant DC voltage to it, allowing the user to shift the signal to the required operating level. This circuit also makes use of a LM318 op-amp with negative feedback. The offset voltage input is taken from the power supply and then run through a potentiometer to set its level. Our circuit combines an averager and an amplifier. First the input signal and the DC-offset signal are averaged, then amplified. When the gain of the op-amp is set to two the output is that of a true adder circuit.

|  |  |  |
| --- | --- | --- |
| **Frequency** | **Vin** | **Vout (Gain = 2)** |
| 0.1Hz | 4.00 | 8.00 |
| 1Hz | 4.00 | 8.00 |
| 10Hz | 4.00 | 8.00 |
| 100Hz | 4.00 | 8.00 |
| 1kHz | 4.00 | 8.00 |
| 10kHz | 4.00 | 8.00 |
| 100kHz | 4.00 | \* |
| 1MHz | 4.00 | \* |
| 10MHz | 4.00 | \* |
| *Table 3: Frequency response of the DC offset circuit* | | |

\*Function generator possibly causing problems (instek GFG-8210)

One of the limiting factors for the DC-offset is that it’s upper and lower limits are set by the supply voltage to the op-amp; if it reaches this point the signal saturates and becomes a constant DC level. We also encountered similar problems with this circuit to those that we had already seen in the amplifier circuit; this time, however, the input signal from the function generator also became distorted.

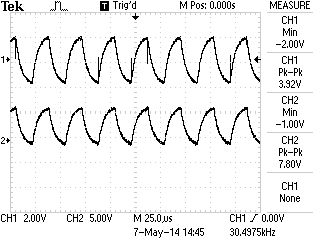


Figure 12: DC-offset output showing distortion

## Combining the circuits

When it came to combining the circuits we realised that we could use our DC-offset circuit as an amplifier as well by replacing one of the feedback resistors with a potentiometer. We decided to keep the amplifier we had already created and use it to feed the integrator circuit, with the DC-offset/amplifier circuit being placed last in the chain. A simple bypass, consisting of a toggle switch, allows for the user to select whether the output from the board is run through the integrator stage or not. We encountered an increased noise level when the signal passed through both op-amps.

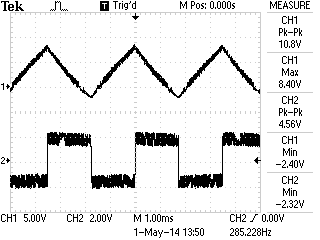


Figure 13: The output signal became very noisy in the final circuit

## Improvements

There are a few issues with individual circuits which have already been discussed that would need to be solved before the function generator and frequency meter reach their final implementation stage; in addition to this there are a couple of other points for improvement of the overall package. With more time we would like to have been able to provide software control over all of the adjustable components (switches and potentiometers). To do this would require that all of the components be replaced with their digital counter-parts and for the software team to provide the relevant control signals. We would also have liked to create our own power supply, similar to the lab supply, which would have outputs for +/- 15V, 5V DC and 10V DC.

# Software Team Design/Implementation Report

## Wave Generation Using DDS

After the hardware team had set up the EIModule containing the ADS9850 (DDS) on a breadboard, we started by looking at the EIModule datasheet [6] so see what external signals would be required to drive the module. Obviously it required voltage and ground supplies, but additionally required a pin for the data signals, one for the frequency update signals, and one to clock the module. Looking at the Board Edge Connector document [7] provided on the internal webpage, we realised we would need to choose one of the GPIO peripheral ports that had connections to the board edge connectors. We therefore opted to use GPIO ports E, to control the DDS (see table below).

|  |  |
| --- | --- |
| **DDS Pin** | **Board Edge Pin** |
| 1 (VCC) | J7-1 (VCC) |
| 9 (D7) | NOT REQUIRED! |
| 10 (GND) | J7-8 (GND) |
| 12 (Sine Out1) | OUTPUT |
| 14 (Square Out1) | OUTPUT |
| 15 (GND) | J7-8 (GND) |
| 16 (RST) | NOT REQUIRED! |
| 17 (DATA) | J7-2 (E3) |
| 18 (FQ\_UD) | J7-4 (E5) |
| 19 (W\_CLK) | J7-3 (E4) |
| 20 (VCC) | J7-1 (VCC) |

After configuring the GPIO E ports appropriately, we next looked at what control signals are required by the DDS. The hardware team had configured the DDS in serial mode for us, so following the timing diagram on page 12 of the Analog Devices datasheet [8] for the DSS, the initialisation sequence for the DDS is pulse the clock pin, pulse the frequency update pin, then write 40 bits of frequency, phase and control data to the data pin one at a time pulsing the clock on each one, followed by a final pulse of the frequency update pin. For ease of readability we created separate methods to pulse the clock pin, pulse the frequency update pin, take the data pin low, and to write a single bit of data to the DDS. The 40 bits of programming data comprised of 32 bits for the frequency, 3 bits of control data, and 5 bits for the phase. We decided that the easiest way to visualise this was by writing the 40 bits of data into an array, which could then be interpreted and send one bit at a time to the DDS. This allowed us to create our DDS\_Default\_Init() method, with an array for a 1KHz wave with 0° phase shift. After verifying that this method initialised the DDS successfully, and that a 1KHz square and sine wave were available on the outputs (using the oscilloscope), we were then able to write our DDS\_Set() method. This broadly followed the same format as the DDS\_Default\_Init() method, but also had to interpret a requested frequency for the DDS, and construct the 40 bit “tuning word” (array) required to program the DDS. The Analog Devices datasheet [8] gave the formula for calculating the tuning word as;



Once the logic for calculating the 40 bit “tuning word” was implemented we had to write a way to be able to set and adjust the frequency of the DDS. This meant writing a simple user interface for the board. In the main function after all the initialisation functions for the switches, LEDs, LCD screen, and now the DDS had been called, in the while loop we called the SWT\_Get() method, and then tested to see which switch had been pressed with a series of if-else statements. Having only 8 buttons to play with, we decided to use 6 of them to set varying increment sizes, and 2 of them to actually increment or decrement the DDS frequency based on the increment already chosen. After looking at the current signal generators in the 4th floor electronics labs, we decided that appropriate increments for us would be 0.1, 1, 100, 1000, 10000, and 1000000, as our specification stated that the waveforms should range from 0.1 Hz up to at least 1M Hz. After running our new program, and verifying the DDS\_Set() method was successful, and our simple user interface worked we refactored the main method to make it more generic, and added code to allow the blue user button to generate interrupts to change the “functionality” of the product, in preparation for implementing the frequency meter.

## Frequency Meter

Revisiting the concept of using input capture to measure the frequency of waves, we found through research that our initial worries that the time delay between detecting a rising edge of a waveform, and triggering the interrupt event were found to not be an issue. The solution was to set the frequency of the timer as high as possible (84M Hz), and adjust the prescaler value for the timer to slow down the frequency, when lower frequency waves were required to be measured.

So far we had used a “direct register access” method, meaning that we cleared and set bits, and wrote values in either binary or hexadecimal format directly to the specified registers in the ST RM0090 reference manual [9]. However configuring timers in this way became quite complex, as you had to ensure that exactly the right values were written to exactly the right registers, otherwise undesirable behaviour ensued. Therefore we decided to change to using the CMSIS standard peripheral library for ARM processors, as this allows the programmer to set all relevant parameters for the peripheral they want to use using a data structure, which is then passed to an Init() method which handles the complicated part of writing the parameters to the correct registers. As part of downloading and linking the CMSIS library to our project, we also found a set of peripheral example projects provided by the ST Microelectronics application team, using the CMSIS library.

Looking at the TIM\_PWM\_Input example project, we found that this did almost exactly what we needed our frequency meter to do. It calculated the frequency and duty cycle of a waveform input on GPIO port B7, by using the Pulse Width Modulation (PWM) input mode of the timer. The timer is configured to start counting, once an interrupt event has been generated after detecting a rising edge on the input pin. It counts continues counting until the next rising edge is detected, however by knowing how many counts happened when the input wave was “high”, and how many when it was “low”, and the frequency the counter was set to, the frequency and duty cycle can be calculated when the interrupt handler routine is called. The IRQ handler, retries the value from the capture compare register, and provided it wasn’t empty (no rising edges detected since IRQ handler last called) calculates the duty cycle and frequency as follows;

DutyCycle = (Capture-Compare Register1 value \* 100) / Capture-Compare Register2 value

Frequency = (counter frequency / 2) / Capture-Compare Register2 value

This looked very promising, so we started by copying their code into our project, and adding a call to our Freq\_Meter\_Init() method inside of our main method, so we could check their code still worked once integrated with our existing project. The code did work, however the accuracy of the frequency meter left something to be desired. The comments stated that the code provided would work over the rough range of 1.28K Hz up to 1M Hz. The reason for this is twofold. Firstly if the counter frequency is set too high, and the waveforms on the input are at a low frequency relative to the counter frequency, the counter will overflow by reaching its max value before the next rising edge occurs, and therefore when the capture compare register values are retrieved they will be a long way out. Secondly if the counter frequency is set too low or too high relative to the frequency of the input waveform, it is likely to miss count in the very small amount of time taken to notify the counter that the next rising edge has occurred, it will continue counting. Another problem if the counter frequency is too low, is that in order to calculate the frequency and duty cycle the counter value has to be rounded to the nearest integer when it is retrieved from the capture compare register. This means that it suffers from rounding errors, and the lower the value in the capture compare register to be rounded, the greater the error proportion will be. This means it’s effectively a trade-off between the counter frequency and the accuracy of the frequency meter (error proportion).

In order to implement the prescaler adjustments, we realised we would need to have some sort of frequency ranges, one of which would need to be selected based on the frequency of the input waveform, so the prescaler could be set accordingly, and consequently the frequency and duty cycle values would be much more accurate. We implemented this in the same way as selecting increments for the waveform generation works, by looping over if-else statements checking which switch had been pressed. We decided that our frequency ranges should roughly be 0.01-1Hz, 1-100Hz, 100-10000Hz, and 10000-1M Hz. By selecting input frequencies that were very close to the boundaries of the ranges we had selected, we were able to obtain values for the prescaler using a trial and error method. The frequency ranges and prescaler values we finally arrived at are as follows;

0.06 – 1 Hz -> 61440 - 1

1 – 100 Hz -> 3840 – 1

100 – 10000 Hz -> 15 – 1

10000 Hz + -> 1

Unfortunately we couldn’t get the frequency meter to accurately measure waveforms less than 0.06Hz, so the lower limit of 0.01Hz had to be abandoned, but we managed to get pretty close. Also we chose not to specify an upper limit for the 10000 + Hz range, as although above 1M Hz the accuracy of the readings started to progressively degrade, it was still capable of measuring them, and wasn’t too far out until it hit about 10M Hz.

## Triangle Wave Generation

After completing the frequency meter, and having a conversation with the hardware team, who informed us that they had still not been able to achieve a triangle wave with a maximum frequency above 500 HZ. We therefore started looking into implementing a triangle wave using the DAC, which would use a counter to count up to a predefined maximum value, at which point it would generate an overflow event, and then start decrementing its count until it hit 0, when an under-run event would be generated, and it would start incrementing again. The DAC allowed us to output an analogue representation of the count onto GPIO A pin 5, which has an internal connection to the output of the second DAC channel.

After referring to the ST RM0090 reference manual [9], and understanding the relationship between the counter overflow value and the amplitude of the triangle wave, we looked at the ST Microelectronics Application Teams example DAC project to see how the DAC is enabled. We found that this gave examples of both triangle wave generation and noise generation using the 2 independent channels available on the DAC. After adding this code into our current project, then adding the appropriate initialisation calls to the start of our main method, we were able to verify that the example code did indeed give us a small amplitude triangle wave.

However the amplitude of the triangle wave generated was very small indeed, around 10 millivolts, also the maximum frequency of the generated wave was around 12.5K Hz, which although is significantly better than the hardware team had managed to achieve, is still nowhere near the 100K Hz we stated in our specification. Therefore we started adjusting the DAC\_LFSRUnmask\_TriangleAmplitude from the 1023 given in the example code, working through the range of values available. We discovered that a rough relationship was that by doubling the amplitude setting we would half the maximum frequency of the generated waveform. Therefore we had another trade off, this time between voltage amplitude and frequency of the waveform. The optimum setting we discovered was setting the amplitude (effectively the counter overflow value) to 255 which gave us a maximum waveform frequency of roughly 50K Hz. Although this was only half the desired frequency, we felt that going any further below this amplitude setting, resulted in the output voltage of the waveform being so low it would be almost impossible for the hardware team to do anything with it, such as level shifting and amplifying it.

Therefore after a discussion with the hardware team, we decided to try using an integrator circuit fed from the square wave output of the DDS, which would hopefully be able to provide a wider frequency range while maintaining an acceptable voltage amplitude.

## Noise Generation

While looking at the DAC signals generation example project from the ST Microelectronics Application Team, we also noticed example code to generate noise. Although we stated in our specification we were going to use the internal PR sequence generator, after testing the example code and confirming that it did generate random noise, we decided it would be simpler to use thee example code. After copying it into the existing project, and adding the appropriate calls in our main method to add it as a new “function”, we inspected the result on the oscilloscope. We noted that because it was another signal generated from the DAC, its voltage amplitude was again quite small. As the amplitude of the noise couldn’t be adjusted in software and it was a case of being on or off, we consulted the hardware team, who agreed that we would be able pass the signal through the same level shifting and amplifying circuit they were already planning on using for the sine and square waves from the DDS.

## Arbitrary Function Generation

Although we had not originally planned to include an arbitrary function as part of our product, when looking through the DAC signal generation example project, we also noticed that it used the DAC to produce a sine wave. After inspecting this code we initially didn’t fully understand how it worked. After conducting some research on the internet, we found an article by someone called Sergey Ostrikov [10], which explained in detail how the example code worked, and provided some small further modifications. Rather than hardcoding values in the initialisation steps for the output frequency, wave resolution, and period of the timer used, the article suggested making them #define statements so that they could be easily modified, and have the potential further down the line to allow them to be adjusted “on the fly”.

The code works setting a timer to generate interrupt events at predefined intervals. Every time an interrupt event is triggered, the DAC would request the next set of data to be loaded into its data holding register, by a Direct Memory Access (DMA) request. DMA requests allow blocks of data to be transferred to either peripherals or other memory locations, at high speed without using any CPU resources. The reason the DAC uses DMA requests in this case is that the frequency at which it requests new data to be loaded is so high, that it could potentially cripple the system by using all the available CPU time, hence blocking other actions from happening such the SysTick\_Handler which controls the timing aspects of the Delay() method. The contents of the DAC data holding register are then transferred to its data output register, 3 APB1 (peripheral) clock cycles later. The value stored in the data output register is then converted to an analogue voltage by the DAC, and output on the appropriately configured GPIO pin (GPIO A pin 4 in this case).

We loaded up MatLab and modelled a sinc function, which we then cast to a 12-bit data array. Pasting this data array into the WaveForm[] array in our arbitraryFunc.c file, we were then able to run the arbitrary function after again adding the appropriate calls into our main method. After some very small refinements we opted to leave the arbitrary function as it was, and come back to it at the end of the project if we had time, as we still had a pulse generator, amplitude modulation and frequency modulation that needed implementing. However we planned two stages of further development, firstly adapting the code to allow the output frequency, counter frequency, wave resolution, and counter time period to be adjusted. This would provide full flexibility to be able to output any waveform we liked via the arbitrary function. The second stage was to make the waveform truly arbitrary, so that a PC could be connected to the ARM processor board, and any waveform that could be modelled in MatLab could subsequently be downloaded onto the board and output. This would however require us to implement one of the available communication protocols such as I2C, UART (serial), or USB, all of which require considerable time to implement.

## Pulse Generator

In our initial report we stated that we were going to use the DDS to create our pulse generator, as at the time our initial research showed that the duty cycle of the DDS could be varied. This is true however it can’t be controlled via software, instead it’s a pre-set potentiometer located on the DDS module, so requires manual intervention. Therefore we had to look for an alternative way to implement the pulse generator, as it wouldn’t be much use if the duty cycle couldn’t be varied.

Skimming through the ST RM0090 reference manual [9], we noticed that the timers also have a pulse width modulation output compare mode. This mode allows the programmer to control the frequency and duty cycle of a waveform they are generating. It works by setting the period (ARR) of the associated timer as follows;

ARR = (Timer counter clock / Timer output clock) - 1

And is then able to compute the required duty cycle by doing;

Duty cycle = (capture-compare register value / ARR ) \* 100

Using GPIO C pin 6 in alternate function mode, the output compare initialisation structure allows you to set both the output state (high or low), and the value at which the associated counter will change the output polarity. The counter is started and the output waveform goes high, using the capture compare functionality it timer constantly checks if the counter has reached the value pre-loaded into the capture compare register. When do match, it stops and clears the counter, and flips the polarity of the output waveform until the ARR value (period) of the timer has elapsed. Once it has elapsed it again takes the output waveform high, restarts the counter and starts checking the capture compare register value again for a match.

We found an example project that did something similar to what we required, again by the ST MicroElectronics Application Team. However this code used pre-set duty cycles that weren’t variable, and had been calculated manually and hardcoded into the capture compare register. Therefore it required significant changes, based around calculating and changing the duty cycle “on the fly”. We were able to copy a portion of the code that initialised the GPIO port, and the timer into output compare mode. We created #defines for the timer ARR value, timer clock frequencies and a 50% duty cycle value. This meant it was easy to make the additional changes required. We initialised the timer to a 50% duty cycle, and created PWM\_SetDC() method which took in the desired duty cycle as a parameter, and calculated the value to write to the capture compare register as follows;

newDutyCycle = (dutycycle \* ARR) / 100;

## Frequency Shift Keying Modulation

In our initial report we stated we would implement frequency modulation by multiplying the input waveform by a fixed frequency sine wave. After further investigation it became apparent that this would not be possible in software, without taking the sine wave output from the DDS and the input waveform, running both through separate ADC channels simultaneously, then multiplying the two resultant analogue signals together, then running the result of this through the DAC to get back to an analogue waveform. Which would in turn still require level shifting and amplifying by the hardware team. As we were running very short on time before the demonstration afternoon by this stage, realistically this wasn’t going to be feasible in the time left. However in the last Design & Construction lecture, Dr Dave Chesmore mentioned using Frequency Shift Keying (FSK) as a simpler alternative to frequency modulation. As we had never come across FSK before, some research ensued. The result of which was that we discovered FSK meant taking an input waveform, and when it was high outputting a sine wave with one frequency, and when it was low outputting a sine wave with a second frequency.

To do this initially seemed reasonably simple, as the code we used for our frequency meter used input PWM to detect when rising edges appeared on the relevant GPIO port. Therefore we thought this would be a good place to start, so started by copying the frequency meter code wholesale into our FSK.c file. The first obvious change to make was that we needed to be able to detect both rising edges and falling edges, not just rising ones. This was a simple change and involved changing the TIM\_ICPolarity in the initialisation structure to detect both edge types. However we were faced with a difficult decision to make, as using the input PWM mode appeared to require the use of the same timer and GPIO pin (B7) in alternate function mode. We spent a while experimenting with trying to use an alternate timer, and input pin, namely GPIO B pin 8, as this was also available on the board edge connectors mentioned earlier. However after quite some time attempting this, it became apparent that it simply wasn’t going to be possible. Speaking to a fellow student, they pointed us in the right direction by directing us to look in ST MicroElectronics user manual UM1472 [11]. Looking through the manual, table 5 lists which GPIO ports have internal connections to which peripherals, and what alternate functions are available for each GPIO port. On page 25, we noted that the only combination that would be of use is Timer 4 Channel 2, with GPIO B pin 7. Due to the fact that pin B7 was the only pin with the correct alternate function, whose associated timer was not already in use (excluding frequency meter). Which left us with the problem that both the frequency meter and FSK functionality wanted to use the same timer and input pin.

A solution to this was to combine parts of the frequency meter and FSK code. The initialisation for timer 4 was exactly the same, so was only required to be called once, and was best suited in the frequency meter section. The initialisation for the PWM and interrupts needed to be different, due to the need to detect both rising and falling edges for the FSK code. Obviously the actions taken by the interrupt handler routine for timer 4 would also need to differ depending on which functionality was being run at the time. At first this seemed like a problem, however we already had a global variable called “function” which held a reference to the current functionality of the product. By testing the value of this variable inside the IRQ handler, it allowed us to execute the 2 different paths appropriately.

However now we had to find a way of changing the output frequency of the sine wave on the DDS, whenever the IRQ handler was called, and it was in the FSK functionality mode. This turned out to be fairly simple, by using a variable that was initialised to 1, and then executing an if-else statement which checked the current value of the variable. If it was equal to 1, then the higher sine wave frequency was set, and the variable was set to 0, so that the next time it passed through the IRQ handler the variable equalled 0, and the lower sine wave frequency was set, with the variable being set back to 1. This worked lovely in theory, except that the DDS frequencies couldn’t be set directly inside an IRQ handler, as these implicitly called the Delay() method which caused the lock-up problems described earlier in the introduction section. The solution to this was to set another global variable inside the IRQ handler, the value of which could then be tested back in the main method, and the DDS frequency set accordingly.

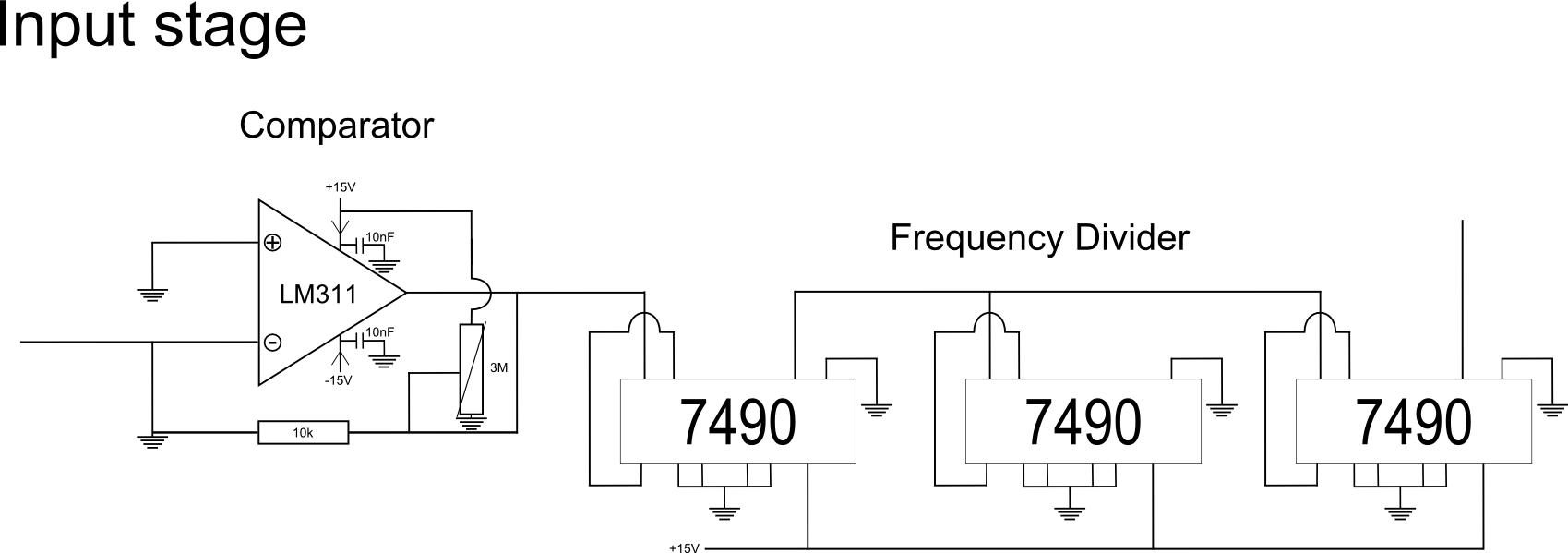
This brought around another problem for us, as in our main method once all the initialisations had occurred and we were inside the while loop, we then tested which functionality we were executing with a series of if-else statements. This caused the problem as while the CPU was testing the if-else conditions, the time 4 IRQ handler had been called and supposedly changed the DDS frequency accordingly. However back in the main method, by the time processor had entered into the FSK section, the DDS frequency had been changed back, meaning that the user never actually saw the frequency change on the oscilloscope. The solution to this was twofold, firstly changing the if-else statement inside the while loop in the main method to be individual while loops. This was possible because the functionality was set using the interrupt handler associated with EXTI\_Line0 (the blue user button). This meant that even though the processor would appear to be in an endless while loop (the condition would never become false without external intervention), it could exit and almost immediately as when the blue user button was pressed the value of the “function” variable would be changed, meaning the condition of the while loop would no longer be true, causing the main method to re-evaluate which while loop to enter. Secondly was to create another global variable called FSK\_Change which was set to true when timer 4 IRQ handler was called. Back in the FSK while loop this was then checked, and only if it equalled true were either of the DDS frequencies set, followed by setting it back to false ready to enter into the IRQ handler again.

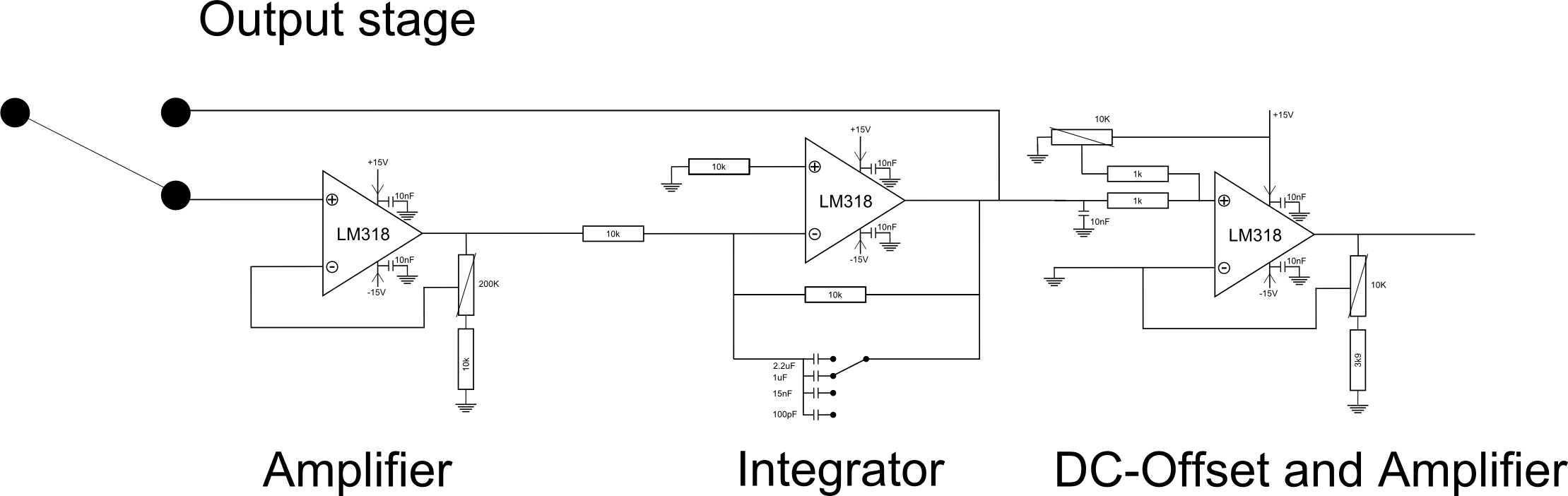
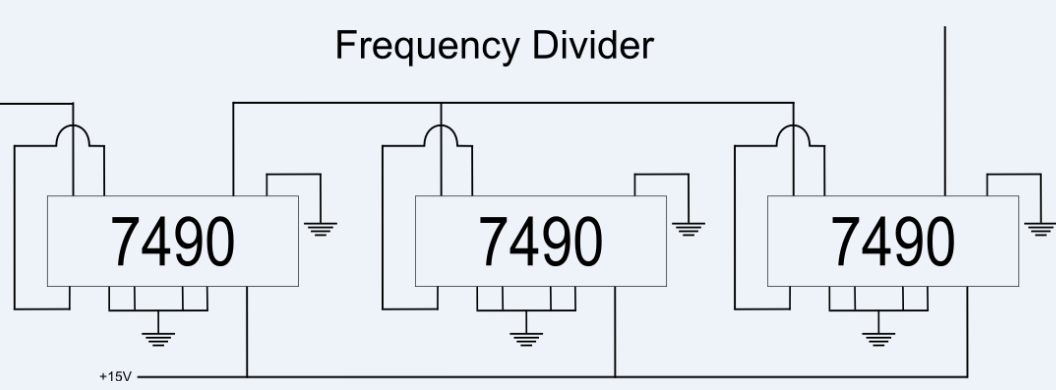
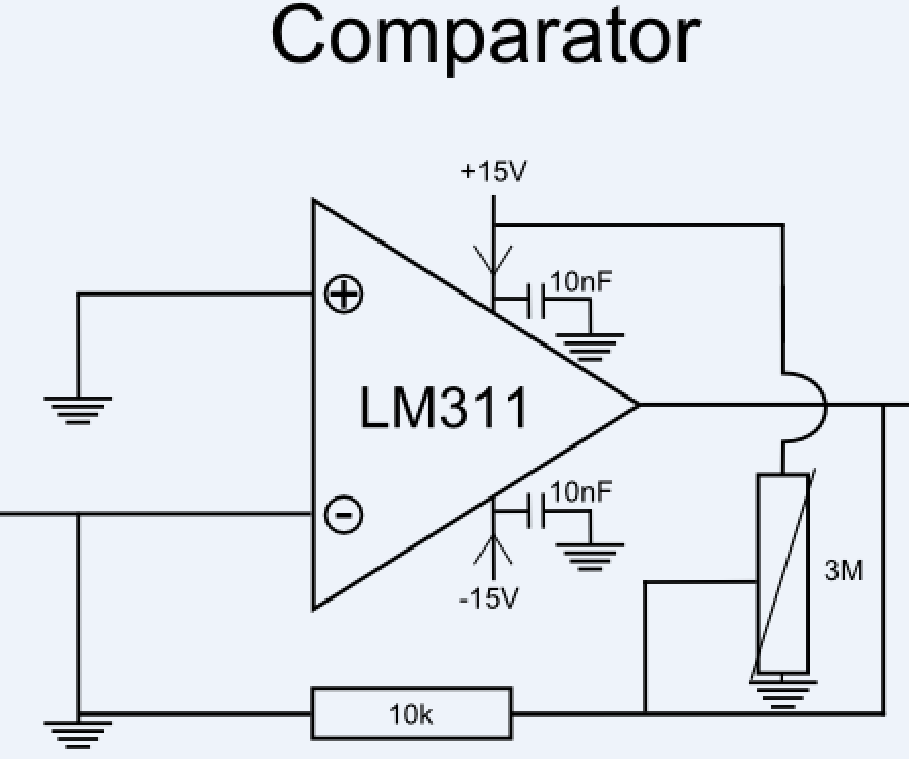
The result of this was that FSK actually appeared to work on the scope, however it had taken so long to fix the problems we had encountered with it, we had run out of time before the demonstration afternoon, and were unable to refine it, so that a wide range of input frequencies could be used. The result of this was that it only worked at around 9-10Hz, and the two sine wave frequencies were not ideal at all. The low one was set to 1K Hz and the high one to 1M Hz. The contrast between the 2 being far too great, so that when the lower frequency was selected it would appear almost like a straight line on the oscilloscope, as it changed back to the higher frequency before even 1 cycle had completed at the lower frequency. Ideally the two frequencies would have been much closer together, at around 2.5M Hz and 1M Hz, but we simply ran out of time to make any further adjustments or refinements.

# References

|  |  |
| --- | --- |
| [1] | D. Chesmore, “Design and Constuction,” January 2014. [Online]. Available: http://www.elec.york.ac.uk/internal\_web/meng/yr3/modules/Design\_Construction/Design%20and%20Construction%20Lab%20Script%202014.pdf. [Accessed 8 February 2014]. |
| [2] | Instruments, TEXAS, “LM111-N/LM211-N/LM311-N Voltage Comparator,” March 2013. [Online]. Available: http://www.ti.com.cn/general/cn/docs/lit/getliterature.tsp?genericPartNumber=lm311-n&fileType=pdf. [Accessed 8 May 2014]. |
| [3] | Naitonal Semiconductor, June 1989. [Online]. Available: www.datasheetarchive.com/74LS90N-datasheet.html. [Accessed 08 May 2014]. |
| [4] | Texas Instruments, “LM741 Datasheet,” March 2013. [Online]. Available: http://www.ti.com/lit/ds/symlink/lm741.pdf. [Accessed 7 May 2014]. |
| [5] | Texas Instruments, “LM318 Datasheet,” March 2013. [Online]. Available: http://www.ti.com.cn/cn/lit/ds/symlink/lm118-n.pdf. [Accessed 7 May 2014]. |
| [6] | EIModule, “Module ELE00003H : Design and Construction,” [Online]. Available: https://www.elec.york.ac.uk/internal\_web/meng/yr3/modules/Design\_Construction/EIM377\_AD9850\_Signal\_Generator\_Module\_V01.pdf. [Accessed 8 May 2014]. |
| [7] | D. D. Chesmore, “Module ELE00003H : Design and Construction,” [Online]. Available: https://www.elec.york.ac.uk/internal\_web/meng/yr3/modules/Design\_Construction/D&C%20board%20edge%20connector.pdf. [Accessed 8 May 2014]. |
| [8] | Analog Devices, “Module ELE00003H : Design and Construction,” 2004. [Online]. Available: https://www.elec.york.ac.uk/internal\_web/meng/yr3/modules/Design\_Construction/AD9850.pdf. [Accessed 11 02 2014]. |
| [9] | ST MicroElectronics, “Module ELE00003H : Design and Construction,” 2013. [Online]. Available: https://www.elec.york.ac.uk/internal\_web/meng/yr3/modules/Design\_Construction/RM0090%20ref%20man.pdf. [Accessed 11 02 2014]. |
| [10] | S. Ostrikov, “STM32 F4 DAC DMA Waveform Generator,” [Online]. Available: http://00xnor.blogspot.co.uk/2014/01/6-stm32-f4-dac-dma-waveform-generator.html. [Accessed 8 May 2014]. |
| [11] | ST MicroElectronics, “UM1472 User Manual,” 29 Januray 2014. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user\_manual/DM00039084.pdf?s\_searchtype=keyword. [Accessed 8 May 2014]. |

Hardware Team Appendixes





# Software Team Appendixes