# 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 [REF] 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 [REF] 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 [REF] 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 [REF] 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 [REF]. 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 [REF], 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 [REF], 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 [REF], 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 [REF]. 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.

## Additional Considerations

Control interface for amplitude done in analogue not digital pot.

AM too complex and no time.