IRISHWAVY MAINTAINER

Spring 2014

Thomas Irish
ENEE440 – Final Project

Table of Contents

1. Abstract

IrishWavy is a highly versatile signal generator, which utilizes various IO devices, produces a selection of waveforms, and allows for several modes of operation. IrishWavy is capable of producing Sinusoid, Triangle, Square and user-defined waves with a frequency range of 1Hz to 1000Hz. These waveforms can be output continuously, a single period at a time, or based on a trigger input. The amplitude of these waveforms can also be modulated between 100% and 1% of the DAC's maximum output voltage. The program has been designed using high level data structures for maximum modularization, which breaks the program into manageable components that utilize the other components' APIs. All hardware communication and configuration has been completed at the assembly level for increased efficiency and clarity, while the overall data organization has been created using C level data structures.

2. Program Operation

2.1 Introduction

The program utilizes various IO devices and interfaces to allow for versatile signal generation. All 13 switches of the primary switches and the rotary encoder on the board are leveraged. All six of the primary LEDs and the seven segment display are used to indicate program status and modes, as well as reading and writing variable values. The program also uses serial communication to enable custom waveforms that can be created and removed on the fly. The following sections will discuss each input and output medium available for this program.

2.2 Input Overview

The rotary encoder and each switch have a dedicated usage that is consistent regardless of the program state. The usage of the rotary encoder and these switches is illustrated in Figure 1.

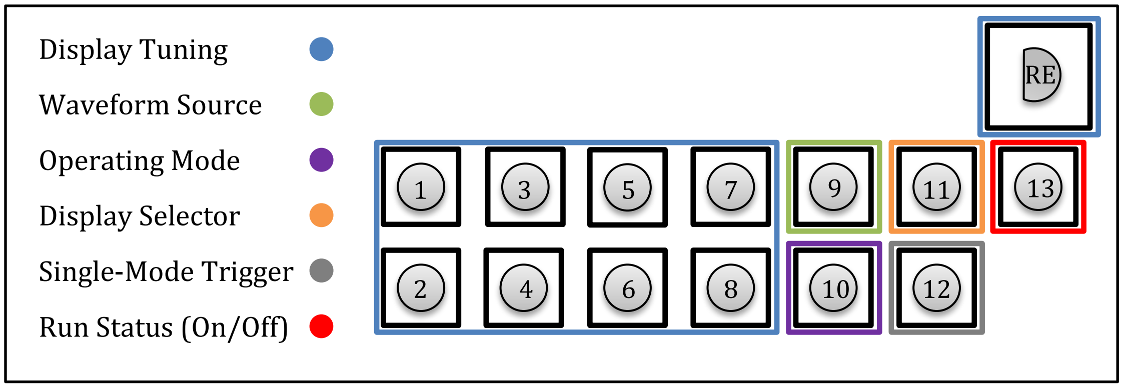


Figure 1: Input devices

The rotary encoder and switches 1-8 are used to modify the value being displayed on the seven segment display (if it is not a read-only value). Switch 9 is used to step through the waveform sources (Sinusoid, Sawtooth, Square, & USART2). Switch 10 is used to step through the operating modes (Repetitive, Single, Trigger). Switch 11 steps through various values for the seven segment display (Frequency, Amplitude, Read-Only Values 1-4). Switch 12 is used as a trigger when the program is in Single-Mode, each press triggers a single period to be output (the switch cannot be held down). Switch 13 is used to toggle the run status of the program between On and Off.

The behavior of the display selector switches and rotary encoder is described in Table 1, below. The rotary encoder has several "notches" in which it requires a bit of force to move it. Between each "notch" are three intermediate steps. Each "notch" change corresponds to ±4 to the display value and each step change corresponds to ±1 to the display value. Turning clockwise will increase the display value while turning counterclockwise will decrease it.

Switch	Effect
1	+1000
2	-1000
3	+100
4	-100
5	+10
6	-10
7	+1
8	-1
RE Step (CW)	+1
RE Step (CCW)	-1
RE Notch (CW)	+4
RE Notch (CCW)	-4

Table 1: Switch/Rotary Encoder Behavior
RE=Rotary Encoder CW = Clockwise Turn
CCW = Counterclockwise Turn

2.3 Serial Communication Overview

The program can be interfaced using a serial communication terminal on a PC. The proper configuration for the serial communication terminal is shown below in Table 2.

Parameter	Value
Baud rate	9600
Data Size	8 bit
Parity	None
Stop Bits	1 bit
Flow Control	None

Table 2: Serial Communication Configuration

Once connected, the USART will accept any character sent to it, and will transmit the character back as confirmation. This connection can be used to build a waveform period that can be sent to the DAC. Since the DAC accepts 12-bit values, the USART will buffer and combine three valid hex characters into a single 12-bit number to be added to the waveform's period. The hex characters sent to the USART do not need to be consecutive, as all non-hex characters will only be transmitted back (but not buffered). As confirmation of a buffered 12-bit number, the USART will transmit back a newline character followed by a carriage return character. For example, if you send the USART the string "1h2j3", that exact string will be sent back to you and the terminal cursor will now appear on the next line. The result of the previous string is that the number 0x123 will be added to the USART waveform buffer, and the effect will be immediately visible in the waveform. This functionality is further discussed in the section for **Waveform Source**, below.

2.4 Output Overview

The seven segment display is used to display various values, some of which are read-only. The LEDs are used to indicate the program state, such as the current operating mode and waveform source. Figure 2 illustrates the uses of the six LEDs. Each LED has three states: Off, Green and Red. The possible LED states will be discussed in the following sections.

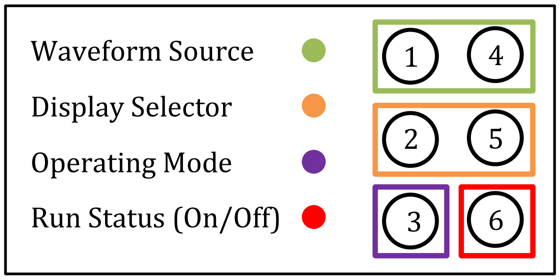


Figure 2: LED usage

The primary output for the program is the waveform output, which can be probed from pin PA5. All changes made via the input devices should have an immediate effect on the waveform produced.

2.5 Waveform Source

The waveform can come from two sources (USART and RAM) and produce four different signals (User-Defined, Sinusoid, Triangle, and Square). For the sake of simplicity, the four possible signals are treated as four different sources, to be stepped through by a single switch (Switch 9). Table 3, below, illustrates how the LEDs indicate which waveform is selected. The order of the states in the figure is also the order they appear by stepping through them using the switch.

Waveform	LEDs	
Sinusoid	○ 1	○ 4
Triangle	○ 1	🟢 4
Square	🟢 1	○ 4
USART	🟢 1	🟢 4

Table 3: Waveform LED Indicator

The USART waveform is created by user input via a serial communication terminal. The USART handler will accept all characters, and transmit them back as confirmation, but only stores valid hex characters. After storing three valid hex characters, the characters are combined to form a single 12 bit number, which is added to the USART's waveform buffer. This buffer is what is used by the DMA to output a waveform. The period size is determined by the amount of values given to the USART, for example if you give the USART two valid 12-bit hex values, those two values will be the entire period that is output. In order for the frequency used for the other waveforms to be valid for the USART data, the max sample amount of 256 must be given to the USART to create a complete waveform. The waveform amplitude is applied to the USART waveform as usual.

When the USART waveform buffer is full (at 256 12-bit numbers), the USART will send back a "!" character after all future attempts to add more numbers. To clear the USART data buffer, simply send the "x" or "X" character. The amount of 12-bit numbers in the USART waveform buffer can be viewed from the first read-only value that can be displayed on the seven segment display, see the **Display Selector** section, below, for more information.

2.6 Display Selector - Variable Selection

The display selector switch (Switch 11) is used to step through various 32-bit BCD values to be shown on the seven segment display. Internally, this display selection is referred to as variable selection. Both writeable and read-only values are stepped through by this switch. The writeable values that are displayed are Frequency and Amplitude. The frequency can be any value (inclusive) between 1 and 1000, and the amplitude can be any value (inclusive) between 1 and 100. The frequency value shown is in Hz, and the amplitude is in percentage of the maximum output voltage. There are four read-only values that can be displayed, of which only two are currently configured. The first read-only value is the amount of samples currently stored in the USART waveform buffer, the second read-only value is a counter that is being incremented by a service call (as a demonstration value) and the remaining two read-only will always be zero, as they have not been connected to any values. Table 4, below, illustrates how the LEDs indicate which display value is selected.

Display Value	LEDs	
Frequency	🟢 2	⚪ 5
Amplitude	⚪ 2	🟢 5
Read-Only 1:RO1	⚪ 2	⚪ 5
Read-Only 2:RO2	⚪ 2	🔴 5
Read-Only 3:RO3	🔴 2	⚪ 5
Read-Only 4:RO4	🔴 2	🔴 5

Table 4: Display Value LED Indicator
RO1=USART waveform buffer size
RO2=SVC demonstration
RO3=RO4=No connected value

2.7 Operation Mode

The operation mode can be one of three values: Repetitive, Single and Triggered. For repetitive mode, the signal is output continuously, each time a period ends to begins another one. For single mode, only one period is output, then the program waits for the user to press Switch 12 before outputting the period again. Each time a period finishes, a flag is set, and pressing Switch 12 simply clears this flag so that another period can be output. Holding the switch or pre-emptively pressing it will have no effect. For triggered mode, the signal will only output if an input pin is being triggered. By default, the input pin is PB15 and is set to Pull-Down, so a high signal on PB15 will trigger the signal. Configuring this input pin will be discussed in a later section. Table 5, below, illustrates how the LED indicates which operation mode is selected.

Mode	LED
Repetitive	🟢 3
Single	⚪ 3
Triggered	🔴 3

Table 5: Operation Mode LED Indicator

2.8 Run Status

The run status of the program (either On or Off) is toggled by Switch 13 and indicated by LED 6. By setting the run status to Off, the program simply disables the Timer (TIM6) used to send values from the DMA to the DAC. When toggling the run status to On, the program simply enables the Timer (TIM6). Table 6, below, illustrates how the LED indicates the run status.

Status	LED
On	🟢 6
Off	🔴 6

Table 6: Run Status LED Indicator

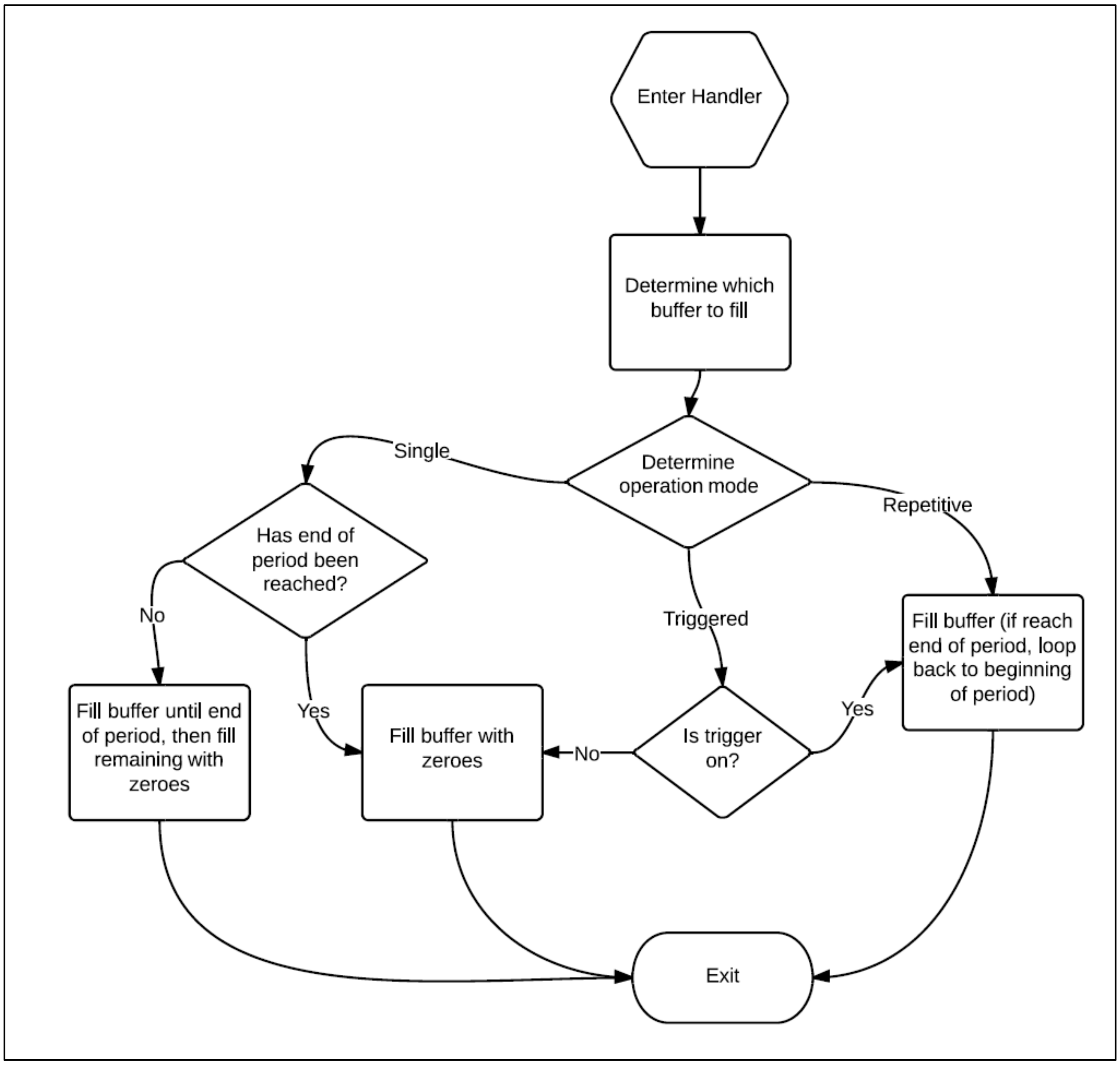
3. Interrupt-Driven Program Flows

3.1 Introduction

IrishWavy is an interrupt-driven program that uses several different interrupts to achieve pseudo-parallel processing. The interrupts that are used are: Timer 6, DMA, Timer 4, Systick, SVC, PendSV and USART. Each of these interrupts have their own individual program flows which will be illustrated and discussed and their respective sections.

3.2 DMA Global Interrupt

This interrupt is triggered by hardware when either the ping or pong buffers are finished being read. When the interrupt is triggered, the handler determines which of the two buffers needs to be refilled then fills it based on the currently selected waveform and operation mode. The control flow is illustrated by Flowchart 1, below.


Enter Handler
Determine which buffer to fill
Determine operation mode
Single
Repetitive
Triggered
Has end of period been reached?
No
Yes
Fill buffer until end of period, then fill remaining with zeroes
Fill buffer with zeroes
No
Is trigger on?
Yes
Fill buffer (if reach end of period, loop back to beginning of period)
Exit

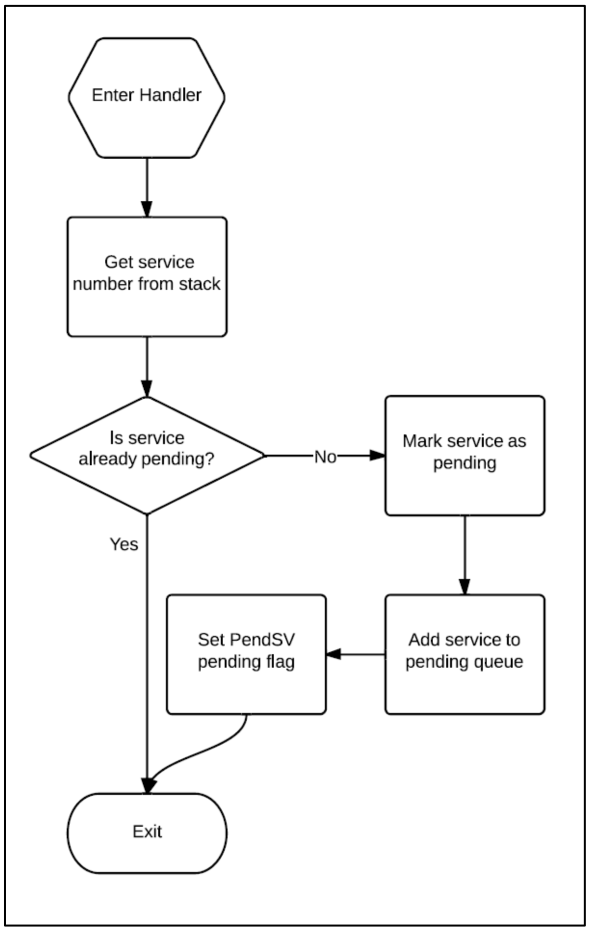
Flowchart 1: DMA Interrupt Handler

3.3 SysTick Interrupt

This interrupt triggered by hardware and is used to demonstrate the functionality of the SVC and PendSV interrupts. Every millisecond, the SysTick triggers and makes a service call. This service call actually triggers a SVC interrupt. After the service call is made, SysTick simply exits.

3.4 SVC Interrupt

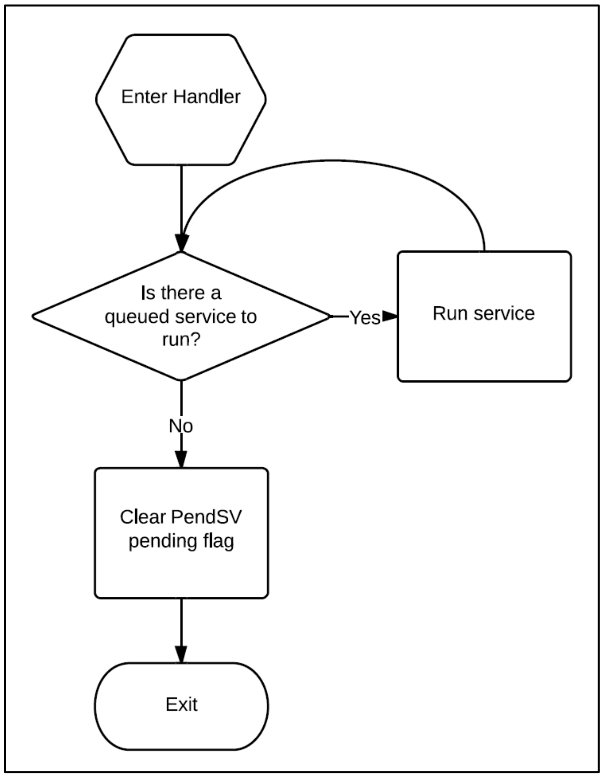
This interrupt is triggered by software in order to set the pending flag for the PendSV interrupt and store the service number of the service for PendSV to run. The software trigger is the "svc" assembly instruction, which is followed by a constant number (the service number). The control flow is illustrated by Flowchart 2, below.


Enter Handler
Get service number from stack
Is service already pending?
No
Yes
Mark service as pending
Add service to pending queue
Set PendSV pending flag
Exit

Flowchart 2: SVC Interrupt Handler

3.5 PendSV Interrupt

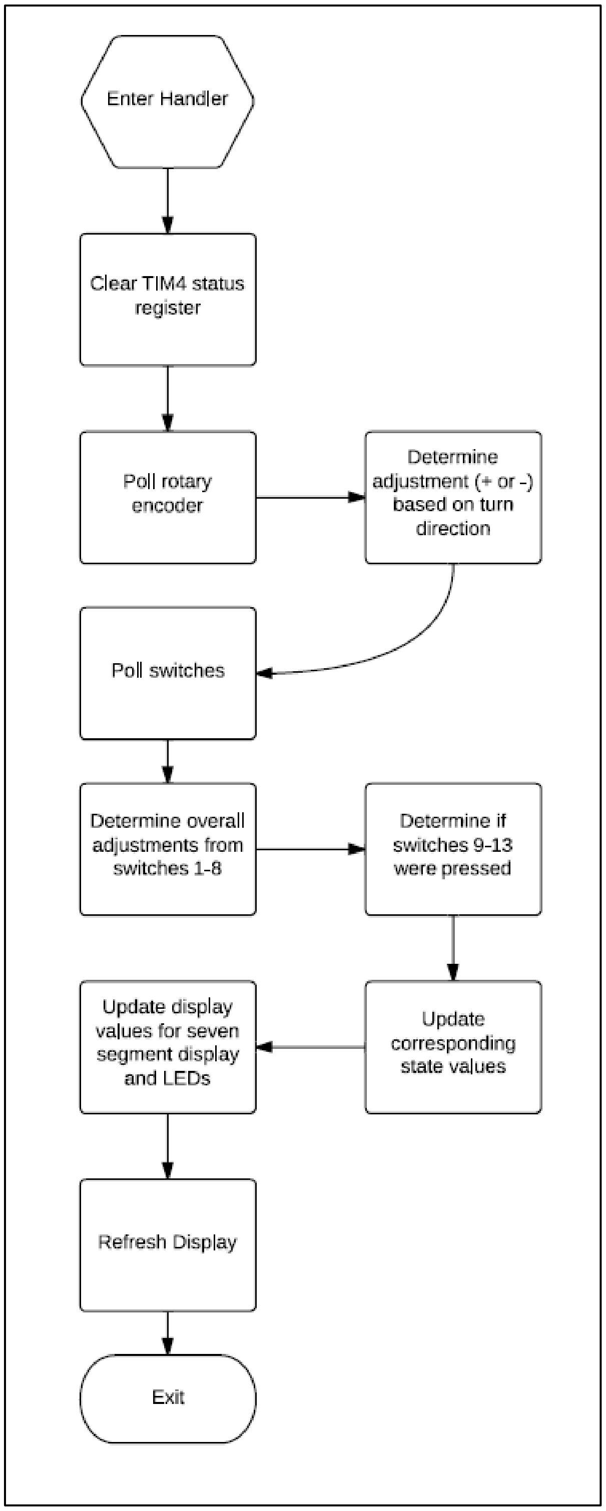
This interrupt is a hardware interrupt that is triggered by a flag set in software. This interrupt runs all pending service calls that have been queued by the SVC interrupt. The control flow is illustrated by Flowchart 3, below.



Flowchart 3: PendSV Interrupt Handler

3.6 TIM4 Global Interrupt

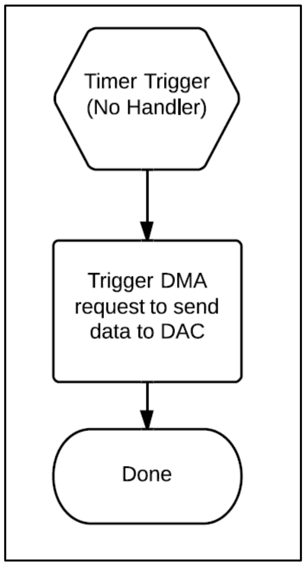
This timer interrupt is used to drive the program's IO devices. During the interrupt handlers, all input devices are polled, state variables are adjusted accordingly, and then the output display values are updated. All of these steps take place during the same interrupt due to the reads and writes to the anode and cathode during the input and output phases. The control flow is illustrated by Flowchart 4, below.


Enter Handler
Clear TIM4 status register
Poll rotary encoder
Determine adjustment (+ or -) based on turn direction
Poll switches
Determine overall adjustments from switches 1-8
Determine if switches 9-13 were pressed
Update corresponding state values
Update display values for seven segment display and LEDs
Refresh Display
Exit

Flowchart 4: TIM4 Interrupt Handler

3.7 TIM6 Global Interrupt

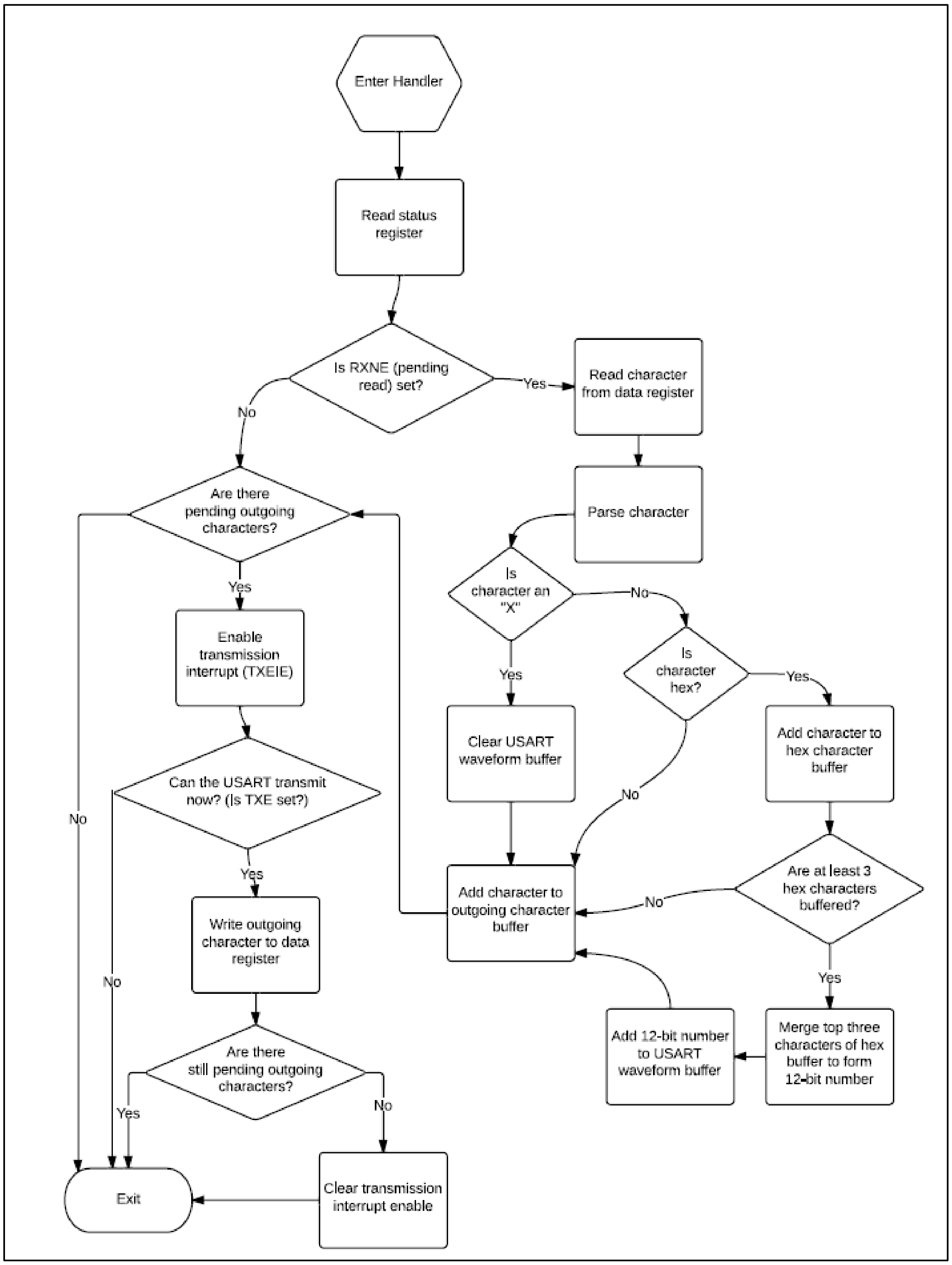
This interrupt is triggered by hardware and does not have an interrupt handler, it is all set up in configuration. Every time Timer 6 triggers, it has the DMA send a new value to the DAC. The hardware flow is illustrated by Flowchart 5, below.


Timer Trigger (No Handler)
Trigger DMA request to send data to DAC
Done

Flowchart 5: TIM6 Interrupt

3.8 USART2 Global Interrupt

The USART interrupt handler handles both pending incoming and outgoing characters. It first reads the status register to determine if there are pending incoming characters, if so it reads the character, parses it and adds it to the outgoing character buffer (to be sent back to the user). After this, it checks if there are pending outgoing characters, if so it turns on the transmission interrupt enable (which causes an interrupt to trigger if a character can be sent) then sends a character. After sending the character it checks if there are more characters to send, if not it turns off the transmission interrupt enable and exits. The control flow is illustrated by Flowchart 6, below.


Enter Handler
Read status register
Is RXNE (pending read) set?
Yes
No
Read character from data register
Parse character
Is character an "X"
No
Yes
Is character hex?
Yes
No
Clear USART waveform buffer
Add character to hex character buffer
Are at least 3 hex characters buffered?
No
Yes
Add character to outgoing character buffer
Merge top three characters of hex buffer to form 12-bit number
Add 12-bit number to USART waveform buffer
Are there pending outgoing characters?
Yes
No
Enable transmission interrupt (TXEIE)
Can the USART transmit now? (Is TXE set?)
Yes
No
Write outgoing character to data register
Are there still pending outgoing characters?
Yes
No
Clear transmission interrupt enable
Exit

Flowchart 6: USART2 Interrupt Handler

4. Program Organization

4.1 Intro

The program has been broken down into subsections, each of which utilizing C data structures to keep the program organized and modularized. These subsections are: Program State, IO, Waves, SVC/PendSV and USART.

4.2 Program State Component

This subsection contains all of the variables used to maintain the overall state of the program, such as the operation mode and waveform source. The entire program state is stored in a variable called **WavyState**. This variable is the only instance of the **Program_State** data structure. See Snippet 1 for the exact details of this data structure.

```
typedef struct {

      State_Variable frequency;
      State_Variable amplitude;
      State_Variable variable_selection;
      State_Variable source;
      State_Variable oper_mode;
      State_Variable run_status;

      struct WavySignal_s * wav;


} Program_State;

Program_State WavyState;
```

Snippet 1: Program_State data structure
Source: _ti_stm32f4_state.h

The **Program_State** data structure contains six primary variables, of the type **State_Variable**. These six variables are: Frequency, Amplitude, Variable Selection, Waveform Source, Operation Mode and Run Status. Also contained within the **Program_State** data structure is a pointer to a **WavySignal** data structure (which is part of the Waveform component). This pointer will always be set to the currently selected waveform.

The **Program_State** data structure is largely a wrapper for these **State_Variable** values, simply for the sake of organization. However, the **State_Variable** data structure features much more useful values within in. See Snipper 2 for the exact details of this data structure.

```
typedef struct {


      uint32_t value;
      uint32_t displayValue;

      uint8_t LED_CLAIM_COUNT;
      int LED_CLAIM[MAX_LED_CLAIMS];
      int LED_VAL[MAX_LED_CLAIMS];

      int TOGGLE_SWITCH;

      State_Variable_Getter Getter_Func;
      State_Variable_Setter Setter_Func;
      State_Variable_Display_Updater Display_Update_Func;
      State_Variable_LED_Encoder LED_Func;
      State_Variable_Next_State Next_State_Func;



} State_Variable;
```

Snippet 2: State_Variable data structure
Source: _ti_stm32f4_state.h

The **State_Variable** data structure allows for the program state component to configure the six primary state variables in a completely contained manner, while allowing the IO component to easily read/write the state variables without worrying about the exact implementation.

This data structure is especially useful for outputting data the LEDs. The **LED_CLAIM_COUNT** value indicates how many LEDs are linked to this state variable, for example the Waveform Source variable has two LEDs linked to it, the Run Status variable has one, and the Frequency variable has zero. This value tells the IO component how many LED values to check for when looking at any given variable. The **LED_CLAIM** array contains the LED indices of any LEDs it has claimed. For example, if a variable wanted to claim LED 2 and 4, index zero of this array would have the value 1 (for LED2) and index 1 of this array would have the value 3 (for LED4). In this example, LED2 would be considered CLAIM1 (for index zero) and LED4 would be considered CLAIM2 (for index one). The indices within the **LED_CLAIM** array are what link the LED claims to the values stored in the **LED_VAL** array. Using the example above with LED2 and LED4 claimed, by putting a two into index zero of **LED_VAL** and putting a three into index one of **LED_VAL**, you have told the IO code that this state variable is assigning a two to LED2 and a three to LED4.

The way this is utilized in the code is that you only need to configure the LED claims for a state variable during initialization, and it will not be hardcoded anywhere else, allowing for easy configurability. The **LED_Func** function pointer, points to a function that encodes the value of the state variable into the **LED_VAL** indices, effectively converting the value into an LED pattern. In order for the IO code to get the values for the LEDs, it simply calls the **LED_Func** for each state variable, then gets the values out the state varible's **LED_VAL** array, using the **LED_CLAIM** array to determine which LEDs the values correspond to. Since the total amount of LEDs to expect from a state variable is stored in **LED_CLAIM_COUNT**, the IO count can easily use the same code to parse through each state variable. The code to do this is shown in Snippet 3.

```
void State_To_LED(State_Variable * var){
      int i;
      int claim_count;
      int led_index;
      int led_val;

      (*var->LED_Func)();//update led encoding
      claim_count = var->LED_CLAIM_COUNT;
      for(i=0; i < claim_count && i < MAX_LED_CLAIMS; i++){
            led_index = var->LED_CLAIM[i];
            led_val = var->LED_VAL[i];
            WavyDisplay.LED_Value[led_index] = led_val;
      }

}
```

Snippet 3: Using State_Variable to update LEDs
Source: _ti_stm32f4_IO.c

The **TOGGLE_SWITCH** value is used to indicate if there is a switch linked to that state variable, if the value is non-zero, it is considered enabled. When IO code reads in the switches, it uses these **TOGGLE_SWITCH** values to determine if that variable needs to be updated. In the event that the switch indicated by **TOGGLE_SWITCH** is pressed, the IO code calls the state variable's **Next_State_Func**, which tells the state code to advance that state variable to its next state. Much like the LED claiming, simply by initializing the state variable with a value in its **TOGGLE_SWITCH**, it will not need to be hardcoded anywhere else, allowing easy configurability.

The state variables that do not have a corresponding toggle switch (Frequency and Amplitude) utilize the **Getter_Func** and **Setter_Func** values to adjust their state. The IO determines which of the two state variables is currently in focus (using the variable selection **State_Variable**) then uses its **Getter_Func** and **Setter_Func** to update its value. The **Display_Update_Func** points to a function that simply re-syncs the **displayValue** with the **value**. The **displayValue** is what gets pushed to the seven segment display, while **value** remains internal, as the two values may not always be exactly the same.

The state component of the program contains the implementations of the functions pointed to within the **State_Variable** structure, which serves as an API for the IO code to utilize.

4.3 IO Component

(You have given me permission to omit these similar sections, for the sake of time. The sections for IO, Waveforms, SVC/PendSV and the USART would be crafted much like the section for Program State, including code snippets, etc.)

4.4 Waveform Component

(You have given me permission to omit these similar sections, for the sake of time. The sections for IO, Waveforms, SVC/PendSV and the USART would be crafted much like the section for Program State, including code snippets, etc.)

4.5 SVC/PendSV Component

(You have given me permission to omit these similar sections, for the sake of time. The sections for IO, Waveforms, SVC/PendSV and the USART would be crafted much like the section for Program State, including code snippets, etc.)

4.6 USART Component

(You have given me permission to omit these similar sections, for the sake of time. The sections for IO, Waveforms, SVC/PendSV and the USART would be crafted much like the section for Program State, including code snippets, etc.)

5. Alternate Configurations

5.1 Introduction

The program is highly configurable thanks to the vast amount of #define and .equ statements throughout the code. This section is dedicated to indicating where these various tweaks are located and how to use them for various situations.

5.2 Trigger-Mode Configuration

The configuration for the Triggered Operation Mode is located in two files: *_ti_stm32f4_io_constants.asm* and *_ti_stm32f4_IO.h*. The constants for setting what GPIO, what pin number and what pull to give it are located in the *.asm* file, and the active-high versus active-low behavior is configured in the *.h* file. At the very end of the *.asm* file there are three .equ statements for the constants **TRIG_GPIO_BASE**, **TRIG_PIN_NUM**, and **TRIG_PIN_CONFIG**. The **TRIG_GPIO_BASE** constant can set to any of GPIOx_BASE constants, defined near the top of that file. The **TRIG_PIN_NUM** constant can set to any number from 1 to 15. The **TRIG_PIN_CONFIG** constant can set to either **PULLUP_INPIN** or **PULLDOWN_INPIN**.

In the *.h* file, towards the end of the file, is a #define for **TRIG_ON_VAL**. This value is used by the program to determine if the current value being read from the input pin is to be considered On or Off for the trigger. The table below shows how to configure these values.

	Pull	TRIG_ON_VAL
Default-Off	Up	0
	Down	1
Default-On	Up	1
	Down	0

5.3 Switch Debouncing

The configuration for the switch debouncing is located near the end of the file: *_ti_stm32f4_io_constants.asm*. From here, you can change how many consecutive reads of zero are needed for a switch to be considered pressed. A read is done every millisecond, so the configuration essentially changes how many milliseconds the switch must be held to be detected. Only one value should be changed, **SHIFTREG_CHUNK_SIZE**. This value can be at most 31, because the 32nd bit is needed to detect a transition.

5.4 Frequency and Amplitude Range

The maximum and minimum values for these two variables are controlled by four #define statements in *_ti_stm32f4_state.h*. The values are called **FREQ_MIN**, **FREQ_MAX**, **AMP_MIN**, and **AMP_MAX**. The initial values for frequency and amplitude can also be changed using **FREQ_INIT** and **AMP_INIT**.

6. Further Expansion & Known Problems

6.1 Preface

Due to the time constraints of this project, in conjunction with other end of the semester (and pre-graduation) commitments, I was not able to include and fine tune everything to my complete satisfaction. As a result, I decided to implement a large portion of the project in C, but was sure to use assembly for any code that talked directly to the hardware. The following sections briefly touch on some known problems and features I was not quite able to implement in time.

6.2 Alternate USART Functionality

In my implementation of the USART handler, I had the numbers being buffered and creating a waveform period to be played back like the other three pre-made signals. In class, it sounded like a desired functionality for the USART was to have the data streamlined from the USART to the DAC, without major buffering and without recording it into a period. This direction for the USART was not taken but would be a feature to be added given the time.

6.3 Alternate SVC/PendSV Implementation

The implementation for SVC and PendSV in this program is for SVC to register a service needs to be run, then PendSV runs that service when all higher priority interrupts give it a chance. Essentially, SVC indicates that something needs to be run in the near future and PendSV eventually runs it. This implementation was originally a misunderstanding of the project requirement, but ultimately became a more practical use for SVC/PendSV. However, given the time, task switching would be implemented using SVC and PendSV. Another further improvement for this is to properly integrate SVC/PendSV into the primary program functionality, rather than simply using it to demonstrate that it works.

6.4 High Frequency Problems

The frequency cap imposed on the program is due to necessary frequency for the DMA to refill the ping-pong buffers quick enough. As a fix for this issue, the amount of samples used by higher frequencies is decreased (at various cutoffs). As a result, the lower frequency signals look very crisp, using the full 256 period samples, while the higher frequency have visible steps, using only 64 samples per period. If too many period samples are used at the higher frequencies, the DMA interrupt will overwhelm the system, and the displays will flash/dim due to being a lower priority than the DMA interrupt.