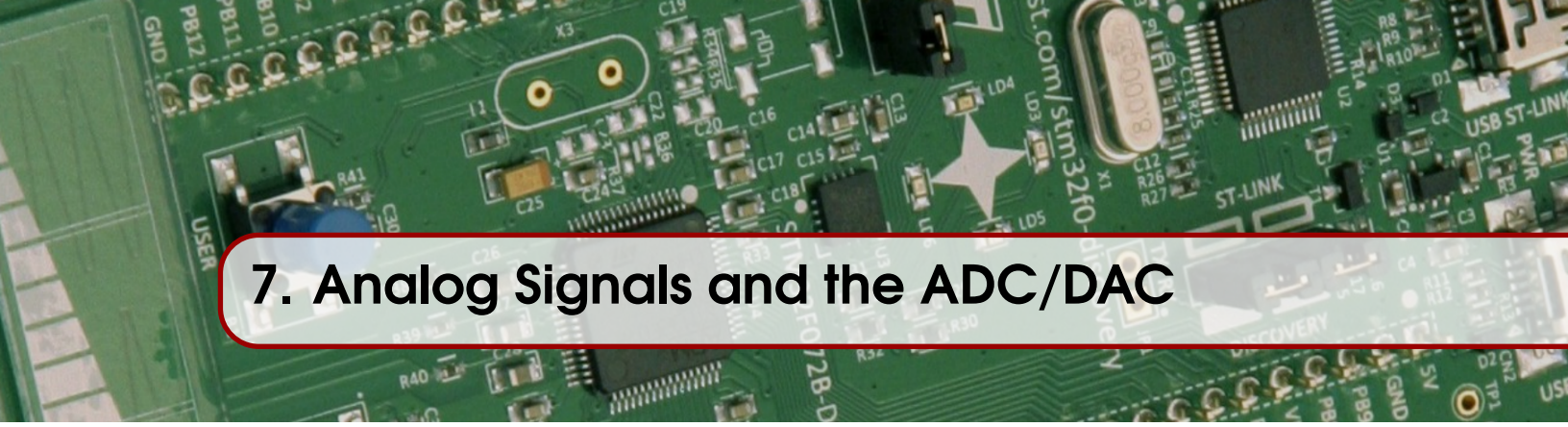

7. Analog Signals and the ADC/DAC

This lab manual is an early revision, the content will change as it is improved and expanded.

7.1 The Analog Truth Behind Digital Signals

All of the signals we have been dealing with in the previous labs have been digital, in that we've been considering them as either "high" or "low". However all real-world signals are actually analog. Regardless of how cleanly and crisp a signal is generated it is essentially a voltage on a wire that has an analog value such as 3V or 0V. As digital signals travel their sharp edged transitions begin to soften and blur and a signal will often gain odd voltage spikes/dips introduced by external factors.
Because of this most digital inputs contain circuitry to recover the intended digital meaning from the messy analog signal received. One commonly used circuit is known as a *Schmidt Trigger*. Figure 7.1 shows the schematic symbol for a Schmidt Trigger and its operation.

7.1.1 Simple Analog Waveforms

Many everyday devices use analog signals, examples include most sensors, speakers, motors, and the AC power in the wall. Some analog signals such as a battery have very simple and (hopefully) predictable characteristics. Others such as audio, have complex waves which are non-repetitive and consist of combinations of many frequencies. For this lab we'll be working with a few of the fundamental wave shapes: sinusoidal, triangle, sawtooth and square. Figure 7.2 shows examples of each of these.

7.2 Representing Analog Signals Digitally

For most analog signals we aren't interested in the basic 1-bit (high/low) analog-to-digital conversion that Schmidt trigger protected inputs have. We'll be using dedicated hardware which can represent a voltage range as a series of binary numbers. However, the conversion between analog and digital is not perfect, there are a few sources of error that you need to know about.
Since this isn't a digital signal processing class, we'll be covering just the barest minimum of theory needed to understand the following concepts.

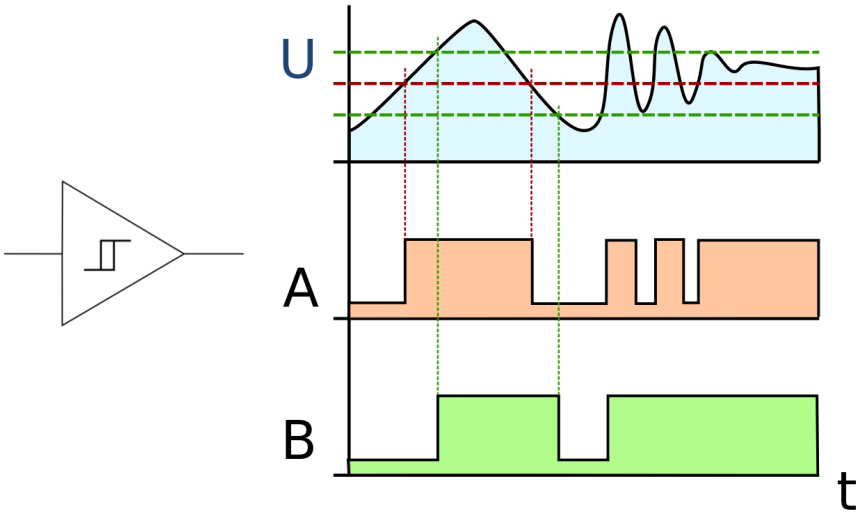

Figure 7.1: Symbol and operation of a Schmidt Trigger.

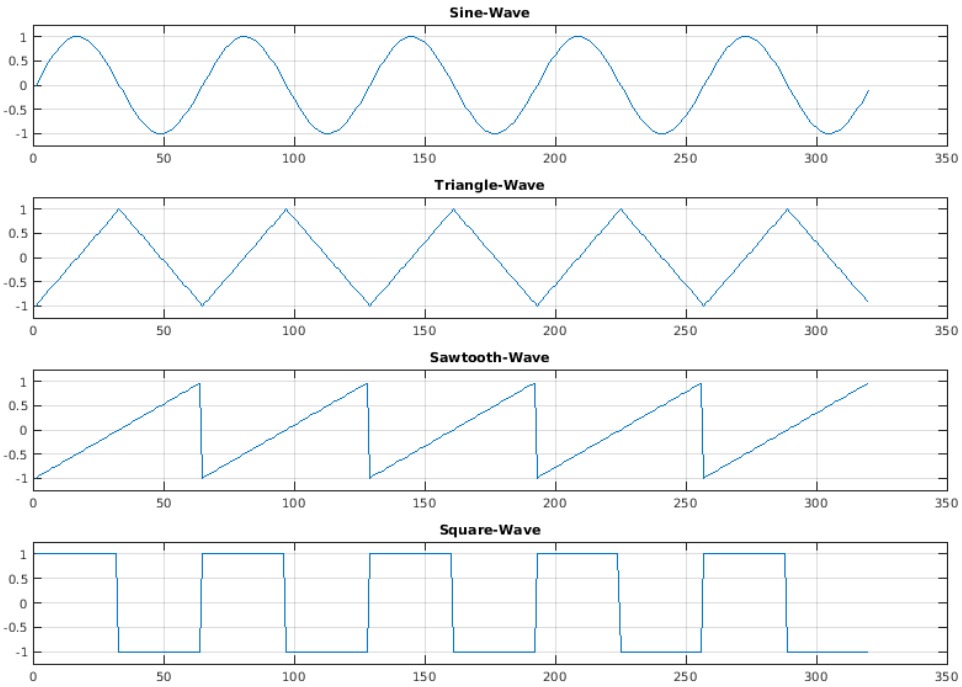

Figure 7.2: Fundamental wave shapes.

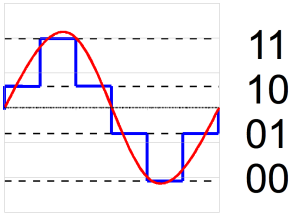

Figure 7.3: 2-Bit quantized sine-wave cycle.

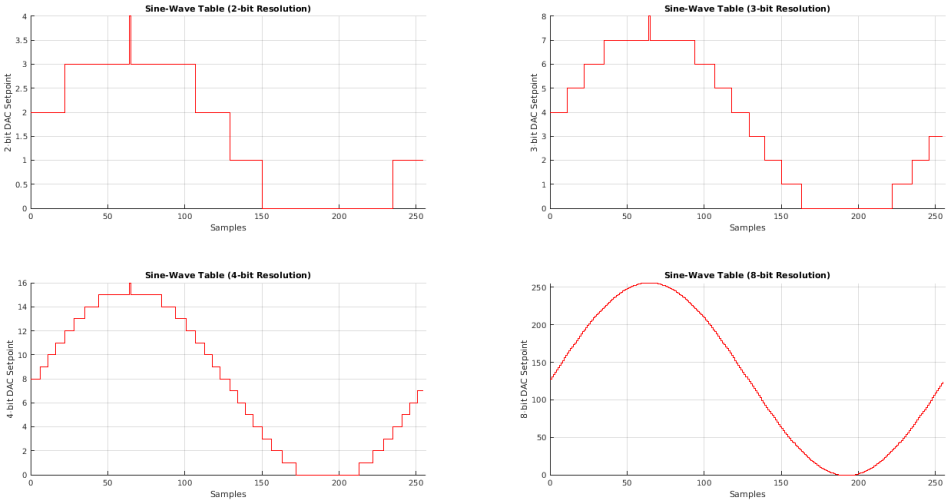

Figure 7.4: Sine-wave cycle under different quantization levels.

7.2.1 Quantization

Essentially an analog-to-digital conversion is like setting up a number of different threshold values and then counting how many the input signal crosses. Because of this the fewer threshold values you have, the less information you actually receive about the input signal. An example is that your input signal might actually be higher than the last triggered threshold, (but not high enough to trigger the next) but you can't tell. Likewise if your steps are too big you might miss changes in an input signal since they never quite reach between the trigger values.

This problem is the basis of quantization error, you are essentially "flattening" pieces of the input signal to make it representable in a certain number range. The more numbers you have the better the resolution of the result. Figure 7.3 shows a (red) analog sine wave being *quantized* into a (blue) 2-bit range or four level digital representation.

Figure 7.4 shows a sine-wave being converted into a range of resolutions. As the number of bits increases, the number of possible voltage steps increases. As the number of steps increases, the difference in voltage between them becomes smaller. As the voltage differences decrease we are able to show more accurately the real analog value at that point in time.

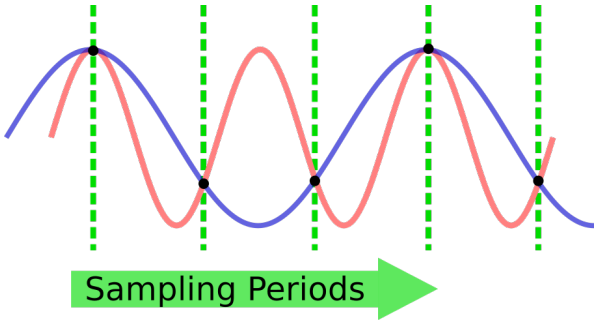

Figure 7.5: Aliasing sine-waves.

7.2.2 Sampling Rate and Nyquist Theory

Unfortunately the number of voltage steps (quantization) isn't the only thing we have to worry about when converting between analog and digital. Take a look at figure 7.5, you'll notice that every time the device samples the analog signals, both of the two different waves have the same value. This problem is the core of something called Nyquist theory.

Nyquist theory explains the relationship between how often you sample an input signal and whether or not you'll be able to tell what it is afterwards. In figure 7.5 the sampling rate was far too slow/infrequent to be able to represent either of those signals accurately. Try to connect the black dot's where the green sampling lines cross the input signals, you'll find that the result looks like a very poor representation of the lower-frequency blue signal, and doesn't represent the higher-frequency pink one at all.

Nyquist theory states that in order to represent an input signal by sampling its value periodically, the sampling rate MUST be at least twice the frequency of the fastest signal. If it isn't then you'll either not be able to recognize the output at all, or you will have the higher-frequency signals "aliasing" and looking like a slower signal. (see the previous image for an example of aliasing)

Typically the faster you can sample an input signal the better results you'll get. Figure 7.6 shows a sine wave that has been sampled a number of times over a single cycle. You'll notice that these look very similar to the quantization graphs, but that they are all using 8-bit (256 step) resolution. If you check the quantization graphs, you'll notice that they are all sampled at 256 samples/cycle.

In order to get a good quality representation of an analog signal you'll need to have sufficient resolution and sampling frequency.

7.2.3 Representing Analog Signals in Software

The in-lab assignment uses a series of wave tables which contain a digital representation of a single wave cycle. These tables are arrays which are indexed by a counter that increments as the data is moved into the DAC. The rate at which the waveform samples are fed into the DAC determines the resulting output frequency. The following code snippet shows the sine-wave table used in the lab assignment.

```
// Sine Wave: 8-bit, 32 samples/cycle
uint8_t sine_table[32] = {127,151,175,197,216,232,244,251,254,251,244,
    232,216,197,175,151,127,102,78,56,37,21,9,2,0,2,9,21,37,56,78,102};
```

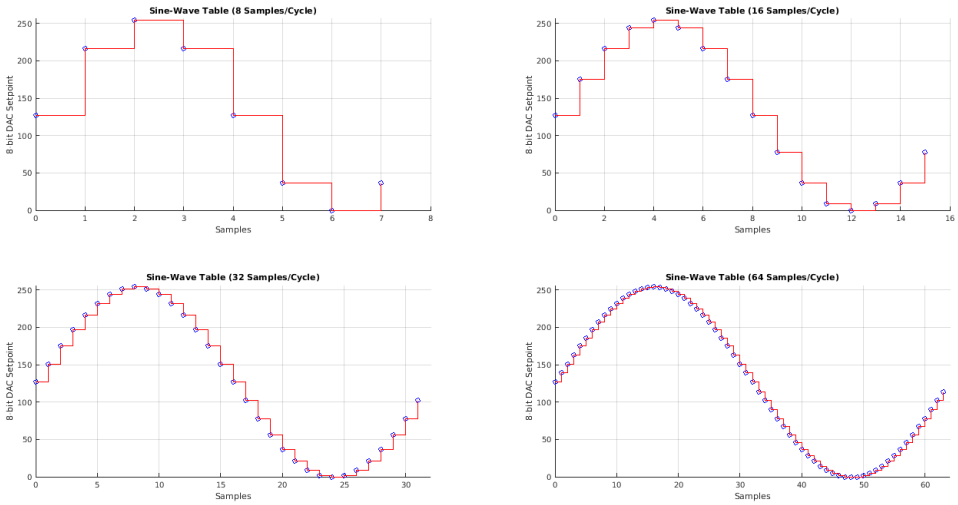

Figure 7.6: Sine-wave cycle under different sampling rates.

Figure 7.7 shows the tables used for each wave type. Each table uses 8-bit data (256 quantization levels) with 32 samples per wave cycle. Increasing the number of samples per cycle would improve the output of the system. However, this lab will use a timer interrupt to transfer data and the number of samples is limited to prevent excessive amounts of interrupts occurring. A more refined approach would be to use the direct-memory-access peripheral (DMA) to automatically move data from memory into the DAC without using an interrupt.

7.3 Analog Input/Output Sources

Voltage Dividers

There are a number of ways to get a variable analog voltage. One of the easiest is to use a circuit called a voltage divider. A voltage divider is two resistors in series that are placed between two different voltages. (for example 3V and GND) By changing the resistance ratio between the two resistors it is possible to generate any voltage in the range between the two endpoints. The following equation shows the behavior of a voltage divider:

$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

Instead of placing two separate resistors and then swapping them out to change the output voltage, we'll be using a device called a potentiometer. A potentiometer is a three-terminal device that is essentially a long resistor with a movable "tap" or connection in the middle. By adjusting the dial on the top of the device the tap moves along between the resistive material in the potentiometer and the resistance between it and each of the endpoints changes.
Potentiometers are excellent for building voltage dividers because they look exactly like a pair of resistors in series. By adjusting the location of the tap, the ratio of the resistors changes and you get a different analog voltage on the tap pin.

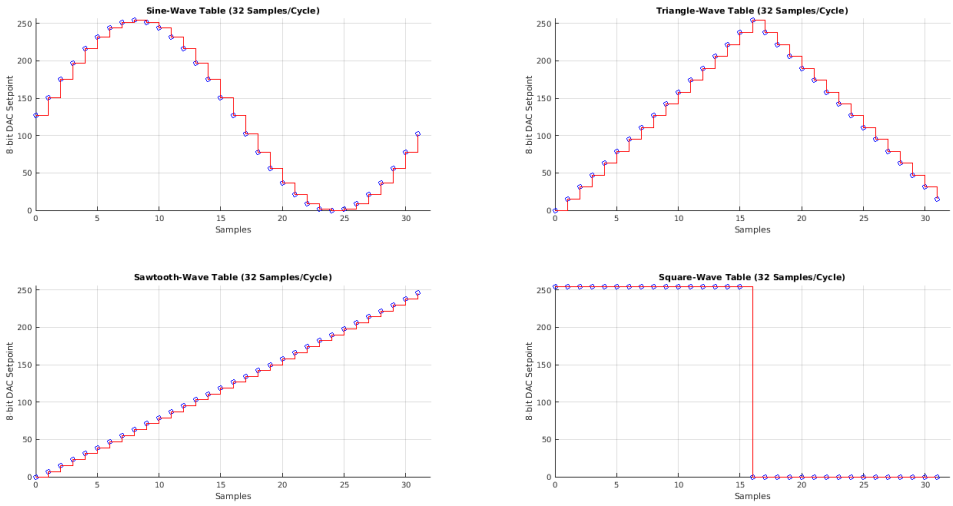

Figure 7.7: Graphical representation of the wave tables used in the lab assignment.

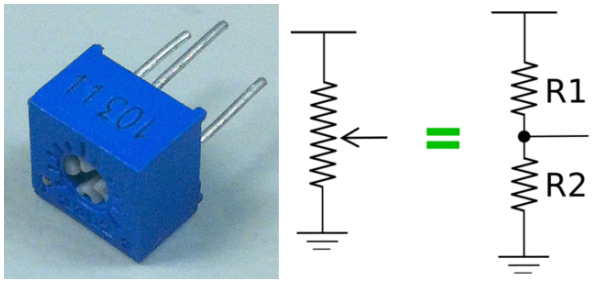

Figure 7.8: Potentiometer used as a voltage divider.

For this lab you'll want to use jumper wires to connect one of the end of the potentiometer to 3V and the other to GND. The center (tap) pin will be the analog voltage output, you'll need to connect that to whatever pin you are using as the ADC input.

7.4 The Analog to Digital Converter (ADC)

Similarly to the USART the analog-to-digital converter of the STM32F0 is a remarkably powerful and flexible peripheral. The on-chip 12-bit successive approximation (SAR) ADC has 19 multiplexed measurement channels of which 16 are connected to external pins and the others measuring internal peripherals and sensors. The ADC can be configured to scan enabled channels in a variety of patterns and conversions are either triggered by hardware events (such as a timer update event) or the ADC can be configured to continuously run.
The output of the ADC can be configured into a few different bit-width's depending on the conversion time, storage capacity and resolution requirements of the user application. The ADC can output data in either 12-bit, 10-bit, 8-bit or 6-bit resolution. As the output resolution decreases the amount of time required for each analog conversion drops.
Because there are delays between starting a conversion cycle and getting an output, there are lots of status flags that the ADC uses to notify the user when it's ready to start a new conversion, whether the ADC is busy, what input in a conversion sequence is currently active and more. Similarly to the USART there are a lot of advanced modes and features that we won't be using for this lab, you will have to look through the configuration registers and ignore things that don't look like they are needed for our requirements.

The ADC Control Register (ADC_CR)

Because the ADC is an analog peripheral, it is very important that you allow it to perform a self-calibration every time you first turn it on. Additionally the ADC has a warm-up period after you enable it before you can start a conversion process. These functions are all located in the main ADC control register (ADC_CR). The largest confusion when using the ADC is that most of the bits in the control register are not only modified by the user to select functions but that they are also modified by the hardware to signal different conditions that the ADC is in.
An example of this is the self-calibration, once the user triggers a calibration cycle by setting the appropriate bit, the ADC will clear that same bit when calibration is complete. Most of the bits in the control register have strict requirements for when the ADC will allow you to set them. You will need to read the bit descriptions to know what you are allowed to configure before/after each step.

7.5 The Digital to Analog Converter (DAC)

In contrast to the ADC, the digital-to-analog converter is one of the simplest peripherals to configure on the STM32F0. It can have either one or two output channels (the STM32F051 has one) with either 8 or 12-bit operation. After configuration, you only need to write a value into the appropriate data register and the DAC will automatically update to the analog representation of that value.
The DAC has three different data registers that are used depending on the data format that has been selected. The three possible operation modes are:

- 8-bit right-aligned (data in bits 0-7)
- 12-bit right-aligned (data in bits 0-11)
- 12-bit left-aligned (data in bits 15-4)

										EVENTOUT	
24	H5	-	-	-	-	PC4	I/O	TTa	-	EVENTOUT	ADC_IN14
25	H6	-	-	-	-	PC5	I/O	TTa	-	TSC_G3_IO1	ADC_IN15
26	F5	18	F3	14	14	PB0	I/O	TTa	-	TIM3_CH3, TIM1_CH2N, TSC_G3_IO2, EVENTOUT	ADC_IN8
27	G5	19	F2	15	15	PB1	I/O	TTa	-	TIM3_CH4, TIM14_CH1, TIM1_CH3N, TSC_G3_IO3	ADC_IN9

Figure 7.9: Section of table 13 with analog functions of PB0 highlighted.

The left-aligned mode is typically used for selecting the upper bits of a 16-bit number, allowing the DAC to act on 16-bit data without any conversion or shifting. (with some minor loss in precision provided by low-order bits) You can read more about the DAC's data formats on page 271 of the peripheral reference manual.

7.6 Analog Functions of GPIO Pins

Similar to the alternate function system which connects pins to digital signals within the chip, the STM32F0 has an analog bus that connects pins to internal analog peripherals.
For this lab we want to connect a couple of pins to ADC_INx (ADC input) and DAC_OUTx (DAC output) signals.

Finding a Pin's Analog Functions

Open the STM32F072 chip datasheet to chapter 4 - *Pinouts and pin descriptions*. Because we're dealing with chip specific pin routing, we can't get the information we need directly from the GPIO peripheral. Take a look at table 13, you'll notice that it maps pin names such as PA0 to physical package pin numbers as well as specifying input/output characteristics, alternate functions and additional functions. Scan through the table looking at the additional functions available for the GPIO pins. Find PB0 and examine it, hopefully you'll notice that its additional function is "ADC_IN8" one of the ADC input channels. Figure 7.9 shows a portion of table 13 with the analog functions of PB0 highlighted.

Setting a Pin's Additional (Analog) Function

Unlike the alternate function system the analog bus on the STM32F0 isn't configurable. Pins are permanently assigned analog functions and beyond setting the GPIO peripheral to analog mode there isn't any additional configuration required. Setting a pin to analog mode disconnects it from the GPIO peripheral, this means that the other GPIO settings such as output speed or pull-up/down don't have any effect.

7.7 Lab Assignment

The lab assignment will be included after review and testing.