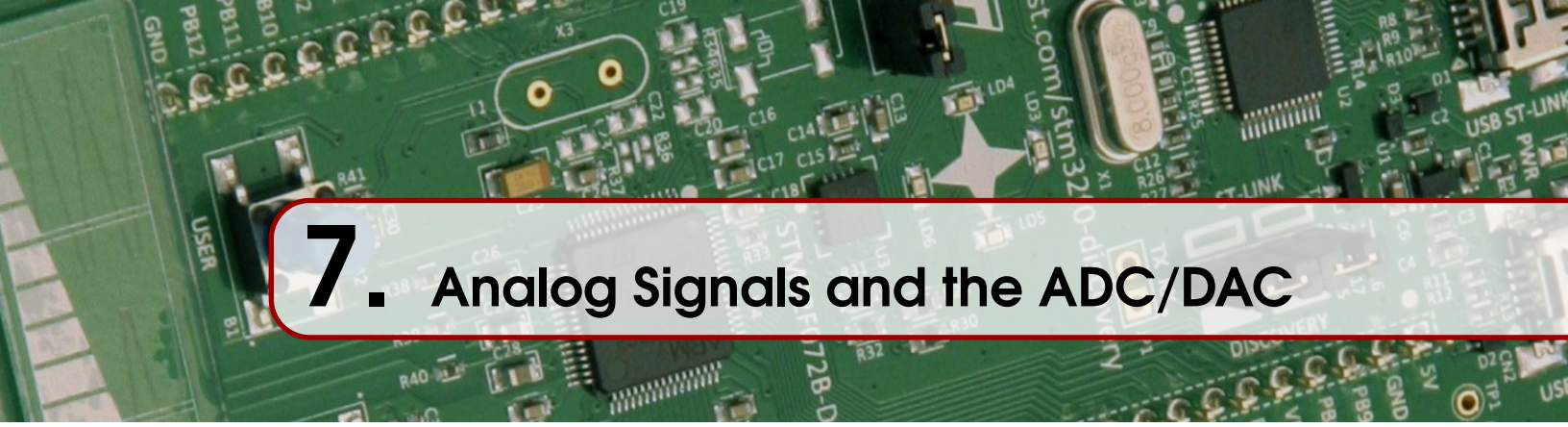

7. Analog Signals and the ADC/DAC

7.1 Section

some separating text

7.1.1 Subsection

more separating text

Subsubsection

even more separating text

Problem 7.1 testing 123...

7.2 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 when generated, it is essentially a voltage on a wire with an analog value such as 3V or 0V. As digital signals travel their sharp edged transitions soften and blur. Often, external factors introduce signal noise in the form of voltage instability/ripple or spikes and dips.

Because of this, digital inputs contain circuitry to recover the original digital meaning from the messy analog signal received. The simplest method of recovering a digital signal is using a voltage threshold. Any voltage greater than the threshold value is considered a digital high, anything less than the threshold is a digital low.

Thresholding only works well when the input signal is relatively clean and avoids values near the threshold. An input signal with voltages close to the threshold have the tendency to cause rapid transitions since even small noise ripples can move the value above and below the trigger point. An improvement on simple thresholding is to use *hysteresis*.

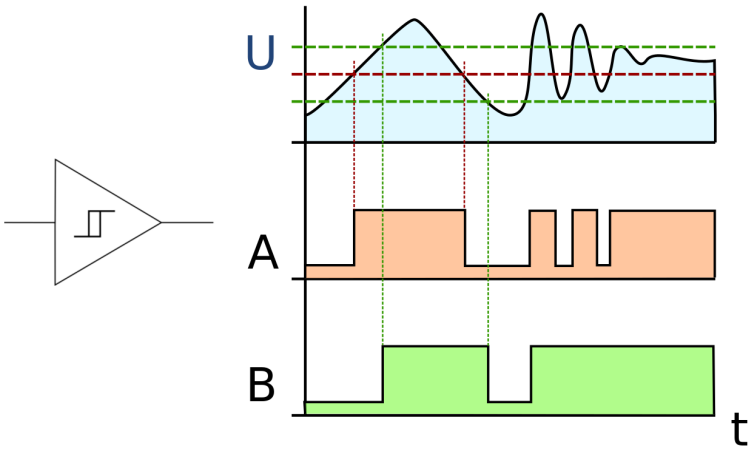

Figure 7.1: Symbol and operation of a Schmidt Trigger.

Hysteresis

Hysteresis changes the voltage threshold depending on the currently detected digital state. This makes it impossible for a signal to consistently hang around the trigger point. For example, once a signal is interpreted as digital high, the threshold for transitioning to digital low moves downwards, and requires a very low value to move to the low state. Even if the analog value remains near the original threshold voltage for detecting a logic high signal, small ripples won't cause unwanted transitions since the threshold moved. One commonly used hysteresis circuit is known as a *Schmidt Trigger*. Figure 7.1 shows the schematic symbol for a Schmidt Trigger and its operation. Signal 'U' is the analog input to be converted into digital signals. Signal 'A' shows the output of a simple threshold indicated by a dashed red line. Signal 'B' shows the output of a system with hysteresis, the two positions of the moving threshold are indicated by green dashed lines.

Simple Analog Waveforms

Many everyday devices use analog signals. Common examples include: 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.

7.3 Representing Analog Signals Digitally

When interpreting an analog signal directly as a high/low digital signal, basic hysteresis circuits such as a Schmidt trigger are sufficient. However, many systems such as sensors, produce analog signals where the specific voltage of the output represents information. For these signals we must use dedicated hardware that can represent a voltage range as a sequence of binary numbers. Unfortunately, the conversion between analog and digital is never perfect and there are a few limitations and sources of error you need to understand.

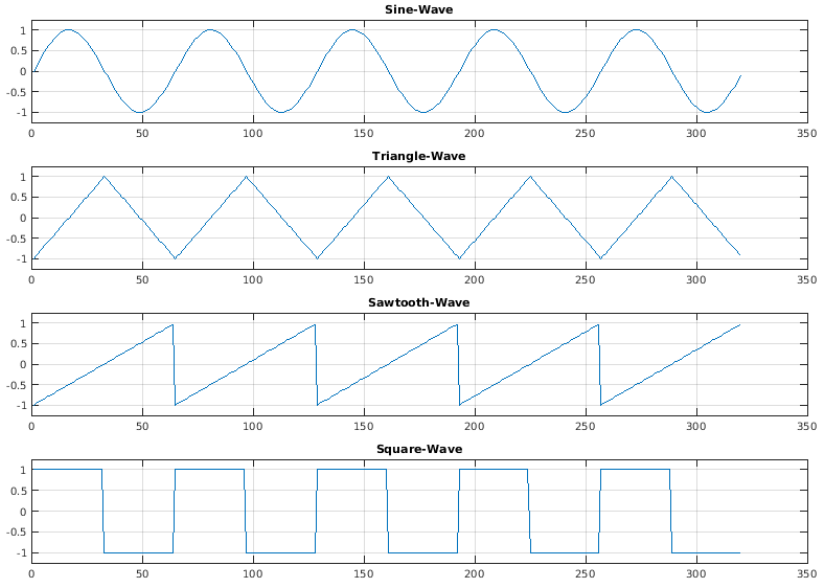

Figure 7.2: Fundamental wave shapes.

Since this isn't a digital signal processing class, we'll be covering just the barest minimum of theory required to understand the following concepts.

7.3.1 Quantization

Converting analog to digital is similar to setting up a range of increasing threshold values and counting how many the input signal crosses. Because of this, the fewer threshold values there are, the more granular the result and less information received about the input signal. This is the basis of *quantization*.

Quantization is the process of mapping a high-resolution signal to a manageable lower-resolution one. Analog signals technically have infinite resolution in their voltage levels, we would like to represent groups of these voltages as a range of digital numbers. Digital quantization is represented by the number of binary bits in its output, a 2-bit device has four values, an 8-bit system can represent 256.

When quantizing a signal you take the maximal range of the input and split it into a number of chunks (placing thresholds) determined by the range that the output supports. Basically, quantization "flattens" pieces of the input signal to make it representable in a certain number range. Figure 7.3 shows a (red) analog sine wave being *quantized* into a (blue) 2-bit range or four level digital representation.

Quantization Error

The main issue quantizing signals is that their value will always be in a state where the input value is higher than the last triggered threshold, but not high enough to trigger the next. Unfortunately, changes within an input signal are lost unless they are large enough to cross one of the threshold values. This concept is the basis of quantization error.

Quantization error describes the difference between the real analog input and the quantized output values. The larger each quantization step, the larger the error grows before the output changes to the next value and becomes closer to the input's true value.

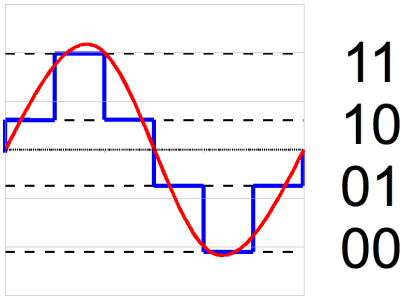

Figure 7.3: 2-bit quantized sine-wave cycle.

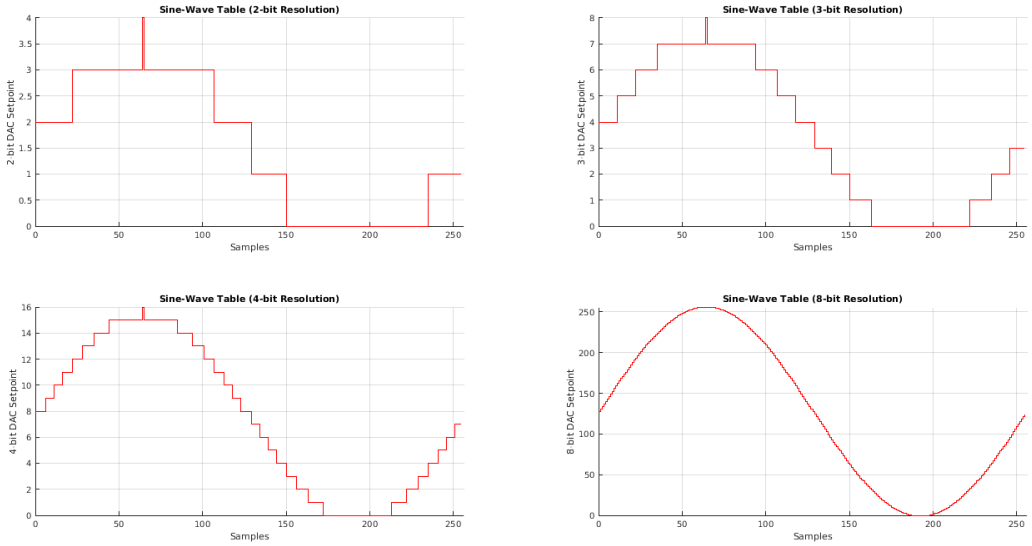

Figure 7.4: Sine-wave cycle under different quantization levels.

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

7.3.2 Sampling Rate and Nyquist Theory

Unfortunately the number of voltage steps isn't the only thing we have to worry about when converting between analog and digital. Figure 7.5, shows that at every sampling point of the analog signals, both of the two different waves shown have the same value. This problem is the core of something called Nyquist theory.

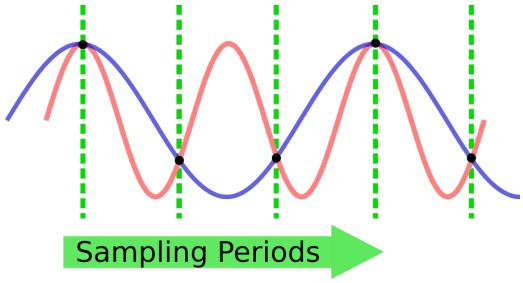

Figure 7.5: Aliasing sine-waves.

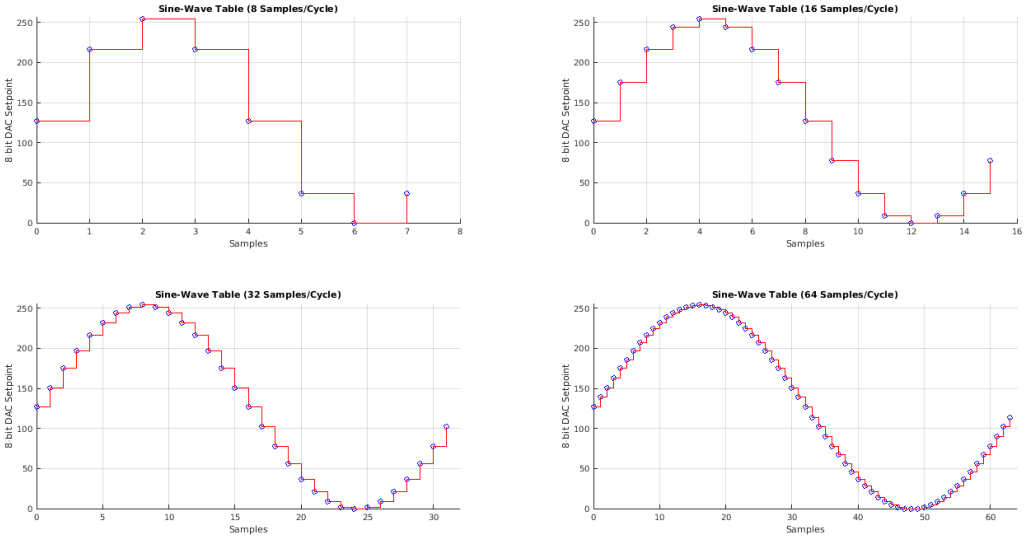

Figure 7.6: Sine-wave cycle under different sampling rates.

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. If you attempt to connect the black dots 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 higher-frequency signals *aliasing*, and falsely appearing as slower ones.

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 quantization resolution and sampling frequency.

```
// Sine Wave: 8-bit, 32 samples/cycle
const 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};


// Triangle Wave: 8-bit, 32 samples/cycle
const uint8_t triangle_table[32] =
    {0,15,31,47,63,79,95,111,127,142,158,174,

    190,206,222,238,254,238,222,206,190,174,158,142,127,111,95,79,63,47,31,15};


// Sawtooth Wave: 8-bit, 32 samples/cycle
const uint8_t sawtooth_table[32] =
    {0,7,15,23,31,39,47,55,63,71,79,87,95,103,

    111,119,127,134,142,150,158,166,174,182,190,198,206,214,222,230,238,246};


// Square Wave: 8-bit, 32 samples/cycle
const uint8_t square_table[32] =
    {254,254,254,254,254,254,254,254,254,254,
 254,254,254,254,254,254,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

Figure 7.7: Wave tables used in the lab assignment.

7.3.3 Representing Analog Signals in Software

Through quantization, analog signals can be represented in software as a list of numerical values. Part of the in-lab assignment uses a *wave table* containing a digital representation of a single wave cycle. These tables are arrays, 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. Figure 7.8 contains the c-code for tables of the basic wave types.

Figure 7.8 shows graphs of each table in figure 7.8. These tables use 8-bit data (256 quantization levels) with 32 samples per wave cycle. Increasing the number of samples per cycle would improve the smoothness of the output. However, increasing the number of samples also requires a much higher rate of moving data into the DAC to achieve the same output wave frequency.

This lab will use the main application loop to move data from a wave tables to the DAC peripheral. A more refined approach could use a timer interrupt or the direct-memory-access peripheral (DMA).

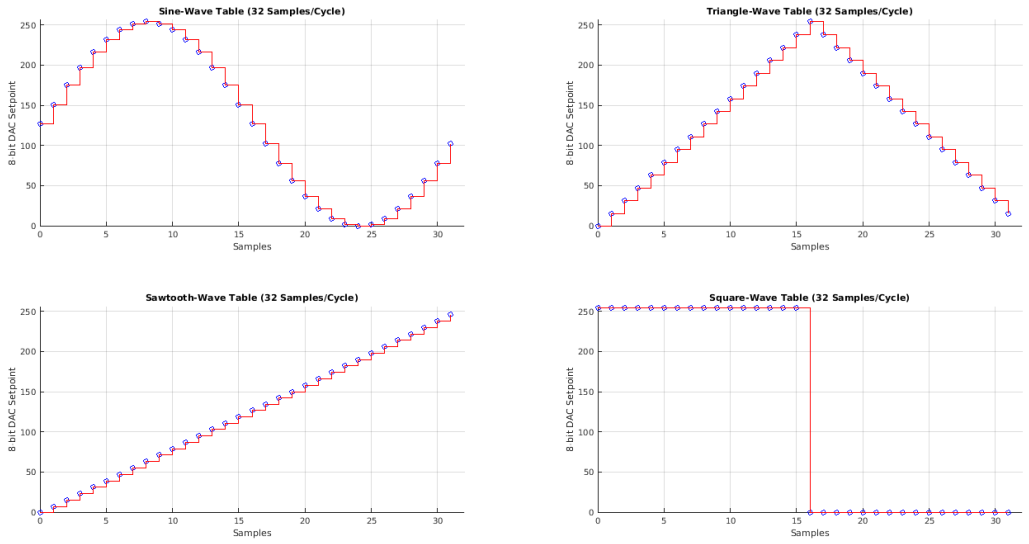

Figure 7.8: Graphical representation of the wave tables used in the lab assignment.

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. See section 13.4.8 in the peripheral reference manual for information about the ADC's operational modes.

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.

7.4.1 Peripheral Registers

Because there are delays between starting a conversion cycle and getting an output, the ADC uses status flags throughout all its registers to notify the user when it is ready to start a new conversion, when it is busy, what input in a conversion sequence is currently active and more. There are many advanced modes and features that we won't be using for this lab, you will have to look through the registers and ignore things that don't look like they are needed for the assignment requirements. The main registers you will need for this lab are mentioned here.

ADC Interrupt and Status Register (ADC_ISR)

The interrupt and status register contains interrupt flags as well as a few high-level status bit which indicate when the ADC is ready for use.

ADC Control Register (ADC_CR)

The ADC control register is used to startup, calibrate, and shutdown the ADC peripheral.

To minimize errors due to voltage offset and drift, it is important that you allow the ADC to perform a self-calibration process 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.

> One common source of confusion when first using the ADC is that many of the bits in the control register are not only used to control the peripheral, but are also modified by hardware as status flags.
>
> 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.

ADC Configuration Register 1 (ADC_CFGR1)

The configuration register sets the operational mode, data resolution, and trigger source of the ADC.

ADC Channel Selection Register (ADC_CHSELR)

Each input to the ADC has a conversion channel associated with it. The channel selection register controls which of these channels are read by the ADC when operating. You will need to enable the channel associated with the input pin you choose in the lab assignment.

ADC Data Register (ADC_DR)

The data register holds the most recently converted value. You will read the quantized value of the ADC input pin from this location.

7.4.2 Initializing the ADC

This section will list the basic procedure of calibrating, enabling, and starting the ADC without mentioning the actual flags and conditions for advancement. For details on the specific bits and flags to modify, use sections 13.4.1 (Calibration) and 13.4.2 (ADC on-off control) in the peripheral reference manual.

To initialize the ADC for use:

1. Set the desired operating mode, data resolution, and trigger source.
    - Single conversion mode performs one measurement every time the ADC is triggered. Continuous conversion mode repeatedly measures and updates the data register.
    - The trigger source selects whether software or a hardware signal starts a new conversion process.
2. Start the ADC calibration
    - Calibration can only be performed when the peripheral is stopped, don't set any enable/start bits (other than in the RCC peripheral) before attempting to start a calibration process.
3. Wait for the hardware to signal that the calibration has completed.

- The ADC will clear the calibration start/request bit when the process has completed.

4. Set the peripheral enable.
5. Wait until the ADC ready flag is set.
6. Start the ADC conversion.
    - Starting a conversion sequence is separate from enabling the peripheral.
    - Since the lab assignment uses continuous conversion mode, this bit only needs to be set once at the beginning.

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 STM32F072 has two) 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.

7.5.1 Peripheral Registers

DAC Control Register (DAC_CR)

The DAC control register selects the trigger source and enables/disables the output channels. The trigger source of the DAC determines whether software (writing to a data holding register) causes the DAC to update its output, or if the peripheral waits on a signal from a peripheral such as a timer.

Data Holding Registers

The DAC has three different types of data holding registers used depending on the desired data format. When triggered, the DAC moves the data from the appropriate holding register into the read-only output registers.

The three possible operation modes are:

- **DAC_DHR8Rx** – 8-bit right-aligned (data in bits 0-7)
- **DAC_DHR12Rx** – 12-bit right-aligned (data in bits 0-11)
- **DAC_DHR12Lx** – 12-bit left-aligned (data in bits 15-4)

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 in section 14.5 of the peripheral reference manual.

7.5.2 Initializing the DAC

There are only two steps to initializing the DAC for use:

1. Set the trigger source for the channel/output update.
2. Enable the channel used for output.

										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.

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 type don't have any effect.

7.7 Lab Assignment

The exercises in this section introduce the basic operation of the ADC and DAC peripherals. Each exercise is standalone, however they can be implemented together in the main application loop without conflict.

7.7.1 Measuring a Potentiometer With the ADC

The goal of this exercise is to use the ADC to measure the position of a potentiometer and display the result using the LEDs on the Discovery board. Each LED will have a threshold voltage/value that will cause it to turn on if the measured output of the ADC is greater. Likewise, they should turn off if the value drops below the threshold. When the potentiometer is turned so that the output (center) pin's voltage increases, the LEDs should light up in sequence.

Your lab TA should have potentiometers available for use during the lab session. For out-of-lab use, you can purchase small potentiometers from the lab stockroom. The potentiometers for sale typically require a small screwdriver to adjust and jumper wires or breadboard to connect to the Discovery board.

1. Initialize the LED pins to output.
2. Select a GPIO pin to use as the ADC input.
    - Check
    - Remember that the "ADC_INx" additional/analog function indicates which ADC input channel the pin connects to.
    - Configure the pin to analog mode, no pull-up/down resistors.
    - Connect the output (center pin) of a potentiometer to the input pin. The other two pins of the potentiometer should be connected to 3V and GND.

3. Enable the ADC1 in the RCC peripheral.
4. Configure the ADC to 8-bit resolution, continuous conversion mode, hardware triggers disabled (software trigger only).
5. Select/enable the input pin's channel for ADC conversion.
6. Perform a self-calibration, enable, and start the ADC.
    - The lab manual describes the basic procedure without mentioning the actual flags and conditions for advancement.
    - Use sections 13.4.1 (Calibration) and 13.4.2 (ADC on-off control) in the peripheral reference manual.
7. In the main application loop, read the ADC data register and turn on/off LEDs depending on the value.
    - Use four increasing threshold values, each LED should have a minimum ADC value/voltage to turn on.
    - As the voltage on the input pin increases, the LEDs should light one-by-one.
    - If the pin voltage decreases below the threshold for a LED, it should turn off.

7.7.2 Generating Waveforms with the DAC

The goal of this exercise is to generate an analog waveform that can be viewed using either an oscilloscope or the analog input of a Saleae logic analyzer. The DAC peripheral will be fed values from one of the lab wave-tables located in figure 7.8.

1. Select a GPIO pin to use as the DAC output.
    - Remember that the "DAC_OUTx" additional/analog function indicates which DAC output channel the pin connects to.
    - Configure the pin to analog mode, no pull-up/down resistors.
    - Connect an oscilloscope probe or channel 0 of a Saleae logic analyzer to the pin.
        - Remember that the old Saleae 16's available for checkout do not have analog capabilities. Unless you have a new-model logic analyzer, you will need to use an oscilloscope.
2. Set the used DAC channel to software trigger mode.
3. Enable the used DAC channel.
4. Copy one of the wave-tables in figure 7.8 into your application.
    - The wave tables are 32-element arrays of unsigned 8-bit values.
    - Don't use the square-wave. Pick one of the sine, triangle, or sawtooth waveforms.
5. In the main application loop, use an index variable to write the next value in the wave-table (array) to the appropriate DAC data register.
    - Use the one that matches closest to the value type of the wave-table.
6. Use a 1ms delay between updating the DAC to new values.
    - The resulting frequency of the waveform will be:
      1 kHz (1ms between updates) / 32 samples per cycle $\approx$ *31 Hz*
7. Submit a screen capture or photograph of the oscilloscope or logic capture in your postlab.