An ARM-Based Sequential Sampling Oscilloscope

By
Qiaodan (Jordan) Jin Stone
S.B. in E.E., Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
August 2014

Author: _____
Department of Electrical Engineering and Computer Science
Aug 22, 2014

Certified by: _____
Gerald Jay Sussman
Panasonic Professor of Electrical Engineering
Thesis Supervisor

Accepted by: _____
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

An ARM-Based Sequential Sampling Oscilloscope

By
Qiaodan (Jordan) Jin Stone

Submitted to the Department of Electrical Engineering and Computer Science
August 22, 2014
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Sequential equivalent-time sampling allows a system to acquire repetitious waveforms with frequencies beyond the Nyquist rate. This thesis documents the prototype of a digital ARM-based sequential sampling oscilloscope with peripheral hardware and software. Discussed are the designs and obstacles of various analog circuits and signal processing methods. By means of sequential sampling, alongside various analog and digital signal processing, we are able to utilize a 3MSPS ADC for a capture rate of 24MSPS. For sinusoids between 6-12MHz, waveforms acquired display at least 7dB of SNR improvement for unfiltered signals and at least 60dB of SNR improvement for aggressively filtered signals.

Thesis Supervisor: Gerald Jay Sussman
Title: Panasonic (formerly Matsushita) Professor of Electrical Engineering

Acknowledgements

My experience here would not have been possible without the generous donations of various organizations internal and external to MIT. Thank you all so much for this opportunity.

I'd like to thank the MIT Electronics Research Society (MITERS) for giving me the confidence to learn (and hack) … well anything. I'll never forget my first blinky LED, first time I turned on (and fell off) Segboard, the F21 "shopping" trip, mountains of etched PCBs, the time Babycopter bit my finger, and all the other absurd, happy memories. Thank you all for taking me in sophomore year. I hope MITERS exists for decades to come and continues to foster the spirit of engineering in each new generation.

Speaking of MITERS, I'd like to particularly thank Charles Guan and Shane Colton for their friendship, guidance, and nearly infinite patience. I wouldn't have been half the engineer (or human being) without you two. Thank you for helping me gain the courage to go build.

My gratitude to Taylor Barton and Zhen Li for my excellent circuits education. Just so you know, me being analog is your fault.

A big thanks to Joseph Colosimo whose unwavering support, encouragement, and cheer has kept me sane the last three years. You're the best Joe.

I'd like to thank Prof. Sussman for taking the role of advisor quite literally and offering me guidance not only in debugging circuits, but also in debugging life.

Lastly, thank you Dad for all the sacrifices you've made. I wouldn't have gotten this far without you. I know I don't tell you this enough, but you really are my role model. Thanks for always accepting the unfiltered, unaltered me. Shin-Kay-Low.

Table of Contents

Table of Figures and Tables

1 Introduction

1.1 Educational Lab Equipment

A recent trend in education is massive open online courses (MOOCs). Universities such as MIT, Harvard, UC Berkeley, and Stanford have begun developing large online classes based on a limited selection of their course catalogs. These classes are offered from a variety of technical fields: mechanical engineering, computer science, and mathematics. A challenge for these courses is providing hands-on experience for classes that would traditionally have a lab component.

For electrical engineering classes, basic lab equipment such as oscilloscopes and signal generators often easily cost thousands of dollars. Students enrolled in an online course may not have access to a university or hackerspace that could provide such test equipment. Most students cannot purchase such lab equipment. Furthermore, swapfests and eBay are not viable options for beginners to hunt down niche gear, nor are such vendors guaranteed to be globally available.

For my thesis, I propose to implement the hardware and initial software for a small, affordable digital oscilloscope based off an ARM architecture microcontroller. The parts are generic, ensuring students access to electronic component vendors online through which they could purchase microcontrollers and oscilloscope peripherals. Additionally, students are then capable of individually installing oscilloscope firmware in the comfort of their own homes. Thus, an oscilloscope based on a common ARM microcontroller would permit students not only to own lab equipment, but also a platform from which they could learn microcontroller programming and build other personal projects. The overall cost of the

oscilloscope and signal generator system should be relatively affordable ($10-$30) with free and accessible example code for future individuals to update and modify.

1.2 Previous Personal Computer Oscilloscopes

Several other oscilloscope microcontroller projects already exist. The majority are software projects with an oscilloscope GUI and a simple PCB to test basic circuit connections. An ADC directly contacts the point of interest with a microcontroller buffering a set of sampled information to a screen. Very little to no signal processing is applied to the collected signal, and the system is bandwidth limited by the input ADC.

While this simplified digital system is perfect for electronics enthusiasts, I intend to deliver a more sophisticated hardware system complete with protection and trigger circuits and basic signal processing. Most importantly, I intend to implement sequential sampling, allowing an input signal to be faster than our Nyquist cut-off frequency.

2 Oscilloscope Sampling

In order to achieve a bandwidth higher than specified by our local ADC, a sequential sampling technique is implemented.

Two popular signal acquisition techniques are real-time and sequential equivalent-time sampling. Each has a specific set of pros and cons.

2.1 Real-Time Oscilloscopes

The concept behind the real-time oscilloscope (RTO) is straightforward. Signal capture is performed in one contiguous stream. The signal bandwidth must be below the Nyquist sampling frequency of the input signal path (typically an ADC with a sample and hold) as seen in Fig 2-1 and Fig 2-2 [1]. On some systems, the signal is captured and stored continuously, allowing a buffer to contain signal information before and after a trigger event. As such, it is possible for an RTO to operate without a trigger circuit; we are simply loading a FIFO and displaying buffer information. A trigger level can be set externally through software if desired and needs not exist in hardware.

Another characteristic of RTOs is their "one-shot" nature. Signal information is typically stored in some sort of FIFO. As the signal is collected, the buffer is filled. This acquisition can be done specifically for a transient one-shot or repetitively. The buffer also ensures that all transient signals are preserved without additional signal processing. However, due to this buffer architecture, one problem is memory depth as demonstrated in Fig 2-3. During a single buffer acquisition, one obtains all points for that waveform. Therefore memory depth limits the time resolution of our capture signal [2]. For example, were we to have 1000 points in memory,

we could either store 1μs worth of signal at 1GSPS or 2μs worth of signal at 500MSPS. To store more time, one must either obtain more memory or operate at a slower sampling frequency. Due to these design considerations, RTOs tend to utilize faster, yet lower resolution ADCs [2].

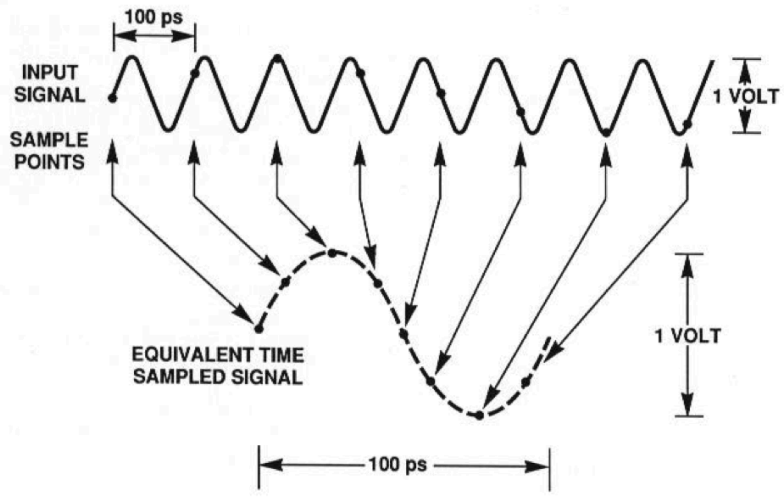


Figure 2-1. Output signal displays aliasing if input frequency higher than Nyquist cutoff [3].

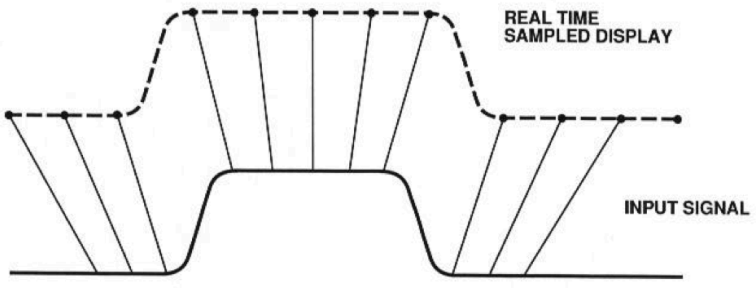


Figure 2-2. Sampling clock must be fast enough to capture edge transitions [3].

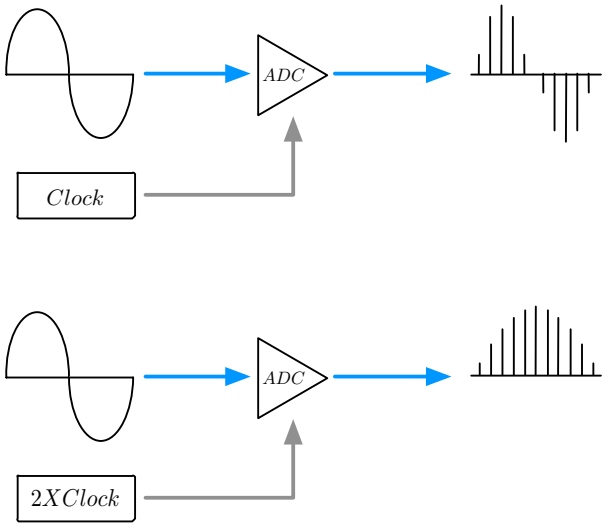


Figure 2-3 Buffer Memory Depth vs. Sampling Clock Diagram

Thus the distinct advantages of RTOs are: trigger circuits are not necessary; one may look ahead and behind a "trigger" event in the stored buffer, and transient events are easily stored and captured. Disadvantages include finite buffer memory depth and constrained sampling rates and bandwidths [1].

2.2 Sequential Equivalent-Time Sampling Oscilloscopes

Unlike real-time sampling, sequential sampling allows an oscilloscope to capture an input signal much greater than Nyquist. Outlines for techniques similar to sequential sampling exist since the 1960s [4]. By inserting a systematic phase delay in the sampling clock, one can amass a set of sampled data with a higher bandwidth than the original sample rate.

Fig 2-4 demonstrates the first procedural step of sequential sampling. The input signal is first sampled by the ADC and digitized by a sampling rate clock (that is much slower than Nyquist). As expected, aliasing of the signal occurs. This data stream is then stored in memory and the ADC configures itself to capture another stream of the same input signal but with an added phase offset to the previous sample clock.

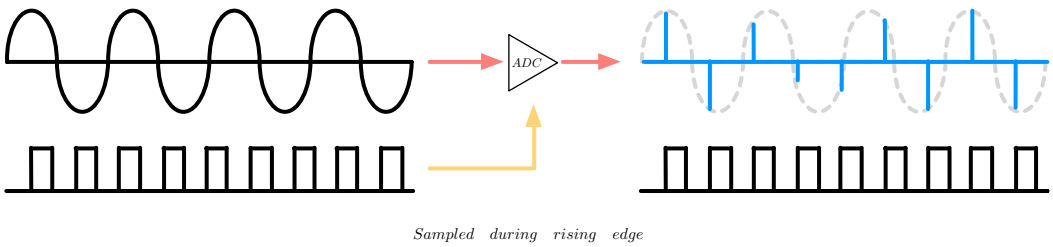


Figure 2-4

The clock delay allows us to collect another set of samples (Fig 2-5). This sample set again displays aliasing; we have not changed the clock frequency. However, this aliased data is different from our previous aliased set (Fig 2-6).

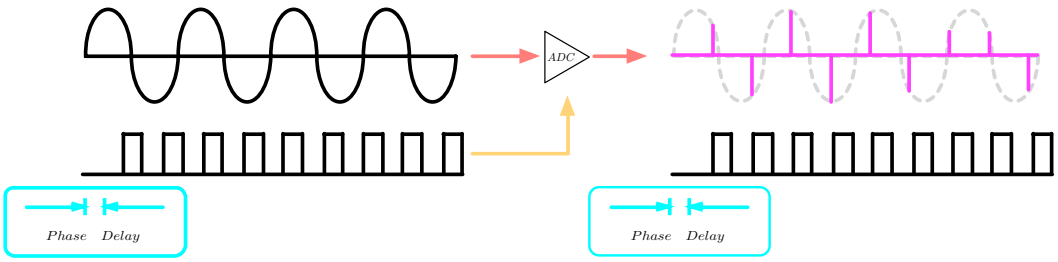


Figure 2-5

We then iterate through this process adding a constant phase delay for each trial (until our total phase is 360°). Ultimately, we merge all the individual sets of captured data into one total output signal. This master output signal then contains data with time-steps proportional to the difference of phase delays. As shown in Fig 2-6, the end effect is a sample

rate much higher than our original clock frequency (and permissible ADC bandwidth).

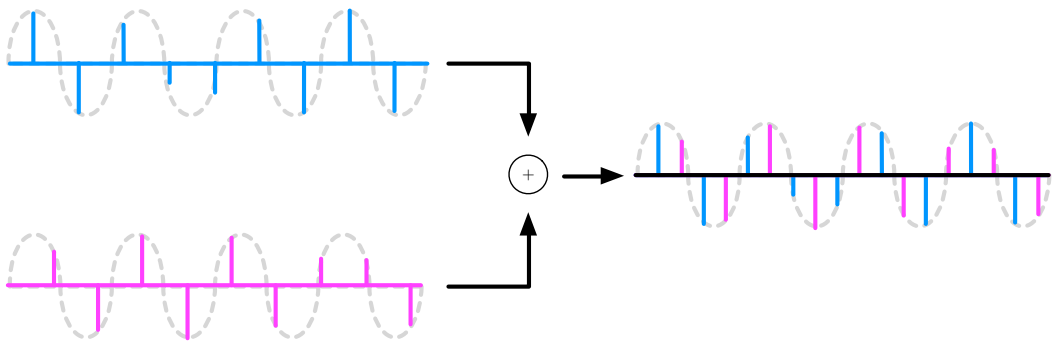


Figure 2-6

While this process significantly improves the sample rate, there are limitations. The input signal must be repetitious; we must sample several phase-delayed copies. In addition, precise timing is critical and a trigger mechanism is required. Lastly, one must have enough memory to store several data sets.

Our oscilloscope allows students to surpass the onboard ADC sampling rate of 3MSPS. The oscilloscope system is composed of four main sections: 1) input attenuator, 2) gain circuit, 3) trigger circuit, 4) firmware for sequential sampling and USB communication. Fig 2-7 is a large system block diagram and schematic depicting the connections of the systems.

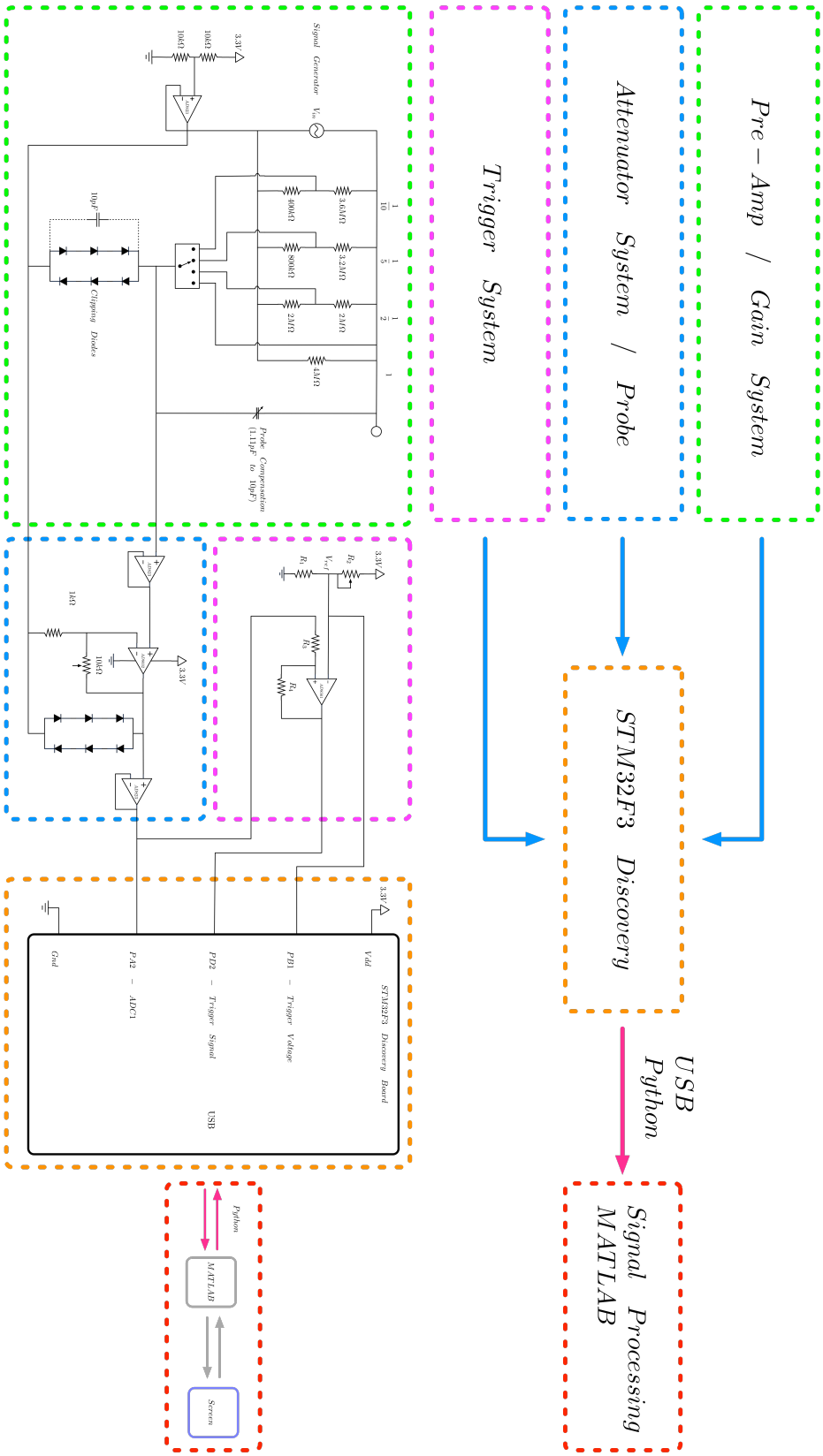

Pre – Amp / Gain System
Attenuator System / Probe
Trigger System
STM32F3 Discovery
USB
Python
Signal Processing
MATLAB
Signal Generator
Clipping Diodes
Probe Compensation
(1.1pF to 10pF)
3.3V
10kΩ
400kΩ
3.63kΩ
800kΩ
3.23kΩ
2MΩ
2MΩ
4MΩ
10pF
1MΩ
STM32F3 Discovery Board
Vdd
PB1 – Trigger Voltage
PD2 – Trigger Signal
USB
PA2 – ADC1
Gnd
Python
MATLAB
Screen

Figure 2-7

3 Input Attenuator and Gain Circuit

3.1 Input Stage

High input impedance is necessary to capture signals without excessive loading of the signal source. However, all physical attenuators exhibit some amount of reactive impedance. Therefore, it is not surprising that the input attenuators exhibit low-pass filter behavior.

3.1.1 Input Attenuator

Our attenuator circuit maintains total 1MΩ input impedance with four different levels of attenuation: 1/10, 1/5, 1/2, 1.

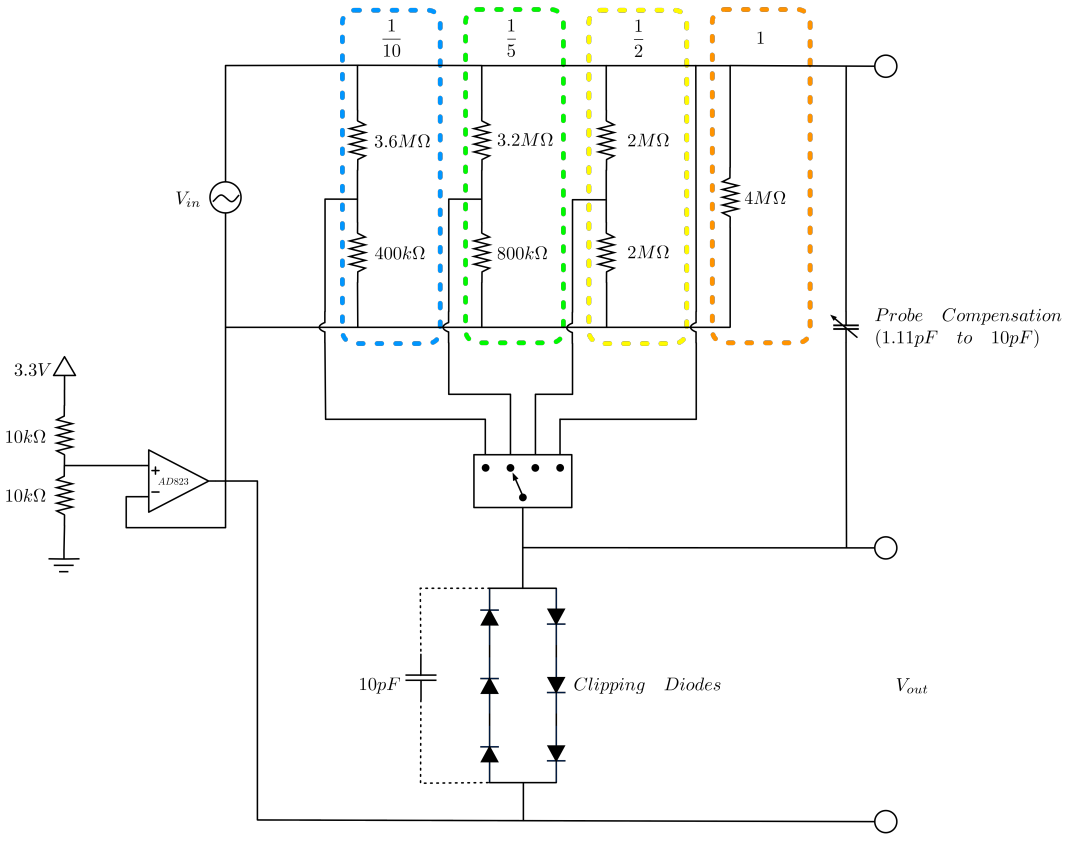


Figure 3-1

Fig 3-1 displays our attenuator system. A single-pole four-throw switch selects the attenuation level. This switch is connected to four different resistor dividers similar to traditional analog attenuators. An op-amp generates a virtual ground of half-rail. This virtual ground is connected to the output of the signal generator, giving us an additional half-rail voltage bump – critical for our system as most signal generators default center their output at 0V. Our ADC cannot accept negative values - a virtual ground is essential. Clipping diodes are connected to the pole of the switch to prevent the signal from rising above 0V-3.6V (the ADC maximum input voltage is 4V).

Unfortunately the clipping diodes present capacitance. Each 1N4001 diode has 15pF of junction capacitance, creating a low-pass filter for gains 1/10, 1/5, 1/2 (Fig 3-2). To remove this low-pass filter, a compensation capacitor must be placed between the pole of the switch and the positive line of the signal generator (Fig 3-1). This variable trimmer capacitor should only be a few pF and allows the user to tune the top impedance of each resistor divider to match the bottom impedance of each resistor divider – we are compensating the bottom impedance's pole by placing the top impedance's zero at the same location. If the capacitor is chosen correctly, the signal should pass through without filtering (Fig 3-3). In a traditional analog scope, we would calibrate our probe with a well-defined square wave. We would then carefully tune the variable cap by observing the step response of the probe. Our user could use a low-frequency square wave for such fine-tuned capacitor adjustments, or the user could use a pre-defined and well-characterized sine wave and test for the appropriate level of magnitude attenuation on the scope.

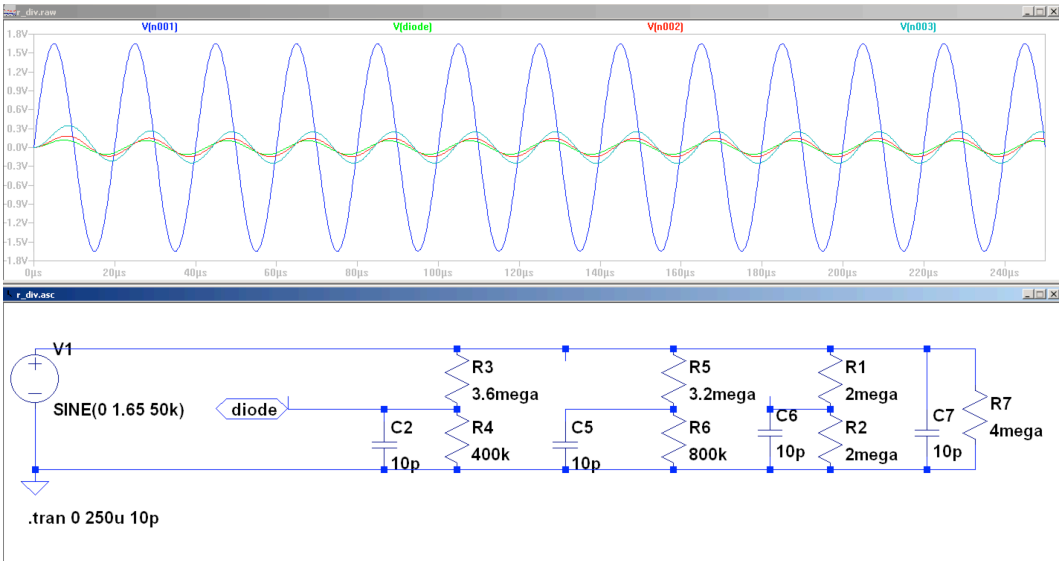


Figure 3-2. Input signal is 50kHz sine wave with 3.3Vpp. Displayed waveforms are for the four attenuator outputs. As noted, 1/2, 1/5, and 1/10 create a low pass effect (middle graphs).

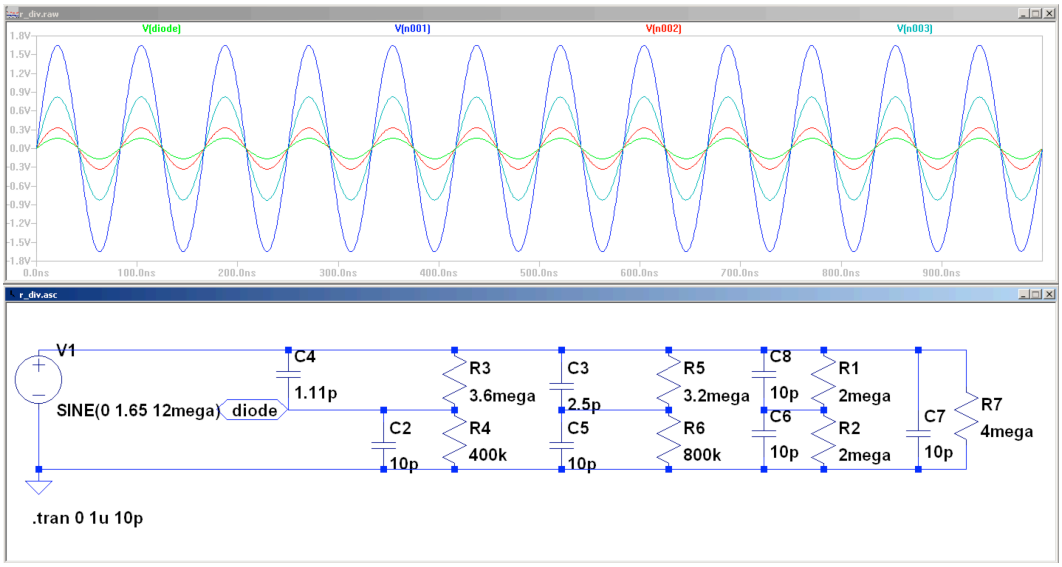
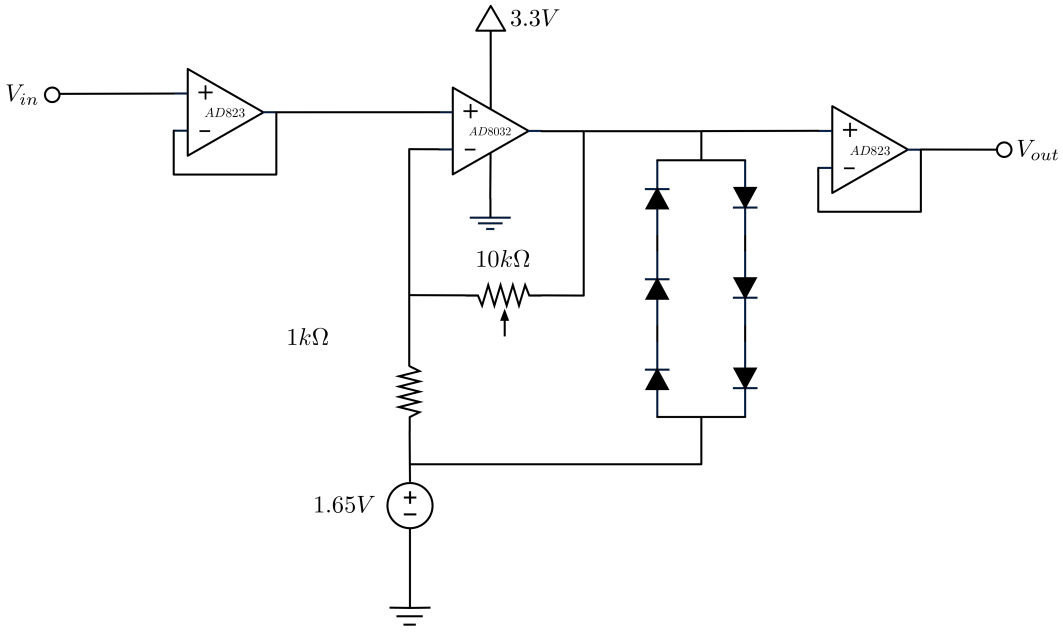


Figure 3-3. Input signal is 12MHz sine wave with 3.3Vpp. Displayed waveforms are for the four attenuator outputs. With proper tuning 1/2, 1/5, and 1/10 (middle graphs) display no filtering.

3.2 Input Gain Pre-Amp

The input pre-amplifier for our adjustable gain is a simple non-inverting op-amp gain circuit (Fig 3-4). This is to ensure the amplification of not only small signal AC waveforms, but also DC voltages. The feedback resistor is a single-pole four-throw switch with one percent resistors of 1Ω, 1kΩ, 4kΩ, and 9kΩ (offering gains of 1, 2, 5, 10). Clipping signal diodes are present to prevent the ADC from experiencing overvoltage.



$$V_{out} = \left(1 + \frac{R_{pot}}{R_{bot}}\right) V_{in}$$
$$V_{out} = \left(1 + \frac{R_{pot}}{1k\Omega}\right) V_{in}$$

Figure 3-4

AD8032 and AD823 are selected for their unity gain bandwidth and DIP packages. All circuits for this oscilloscope are dead-bugged (due to complex routing for proper high-frequency PCBs, and the technique's low parasitic capacitance and fast production time).

The AD8032 is a BJT op-amp with an 80MHz unity gain bandwidth (Fig 3-5). The AD823 is a JFET input op-amp with 16MHz unity gain bandwidth (Fig 3-6). Due to BJT inputs, the AD8032 has low differential input impedance – about 280kΩ. The minimal impedance for any of the resistor dividers of the attenuator is 400kΩ, so an AD823 is used as a buffer for the AD8032. An AD823 is also placed after the AD8032 to provide additional low-pass filtering for signals below 16MHz.

A few small gain bumps of ~1dB exist near 9-10MHz for AD823 and 11-12MHz for A8032. They produce a slight amplification towards the upper ends of our frequency range. However, more problematic are the slew rates and bandwidths for gains larger than unity.

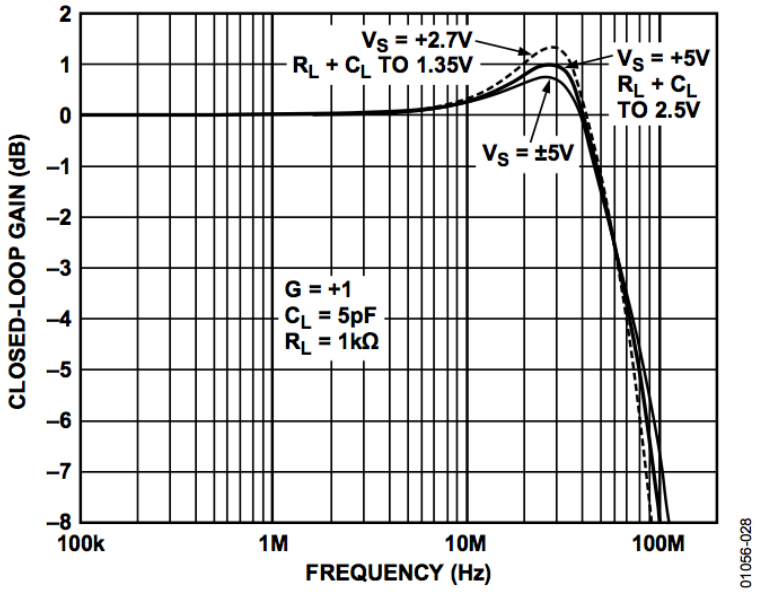


Figure 28. Closed-Loop Gain vs. Supply Voltage

Figure 3-5, AD8032 [5]

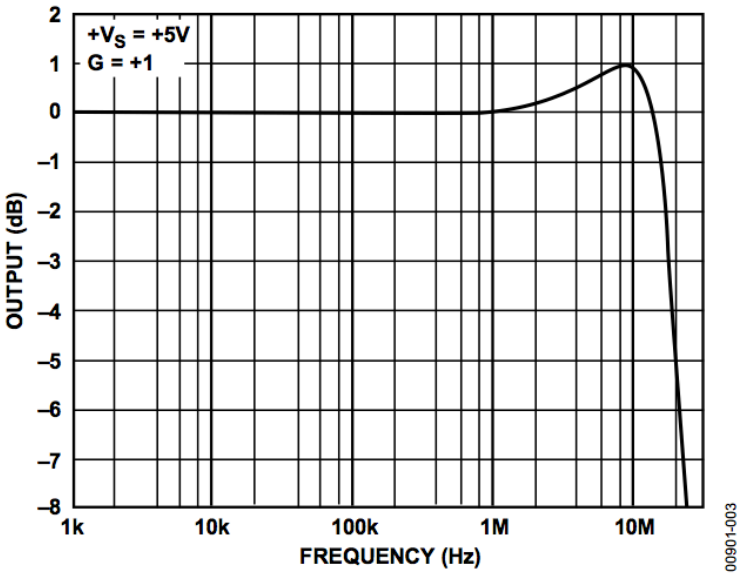


Figure 3. Small Signal Bandwidth, G = +1

Figure 3-6, AD823 Datasheet [6]

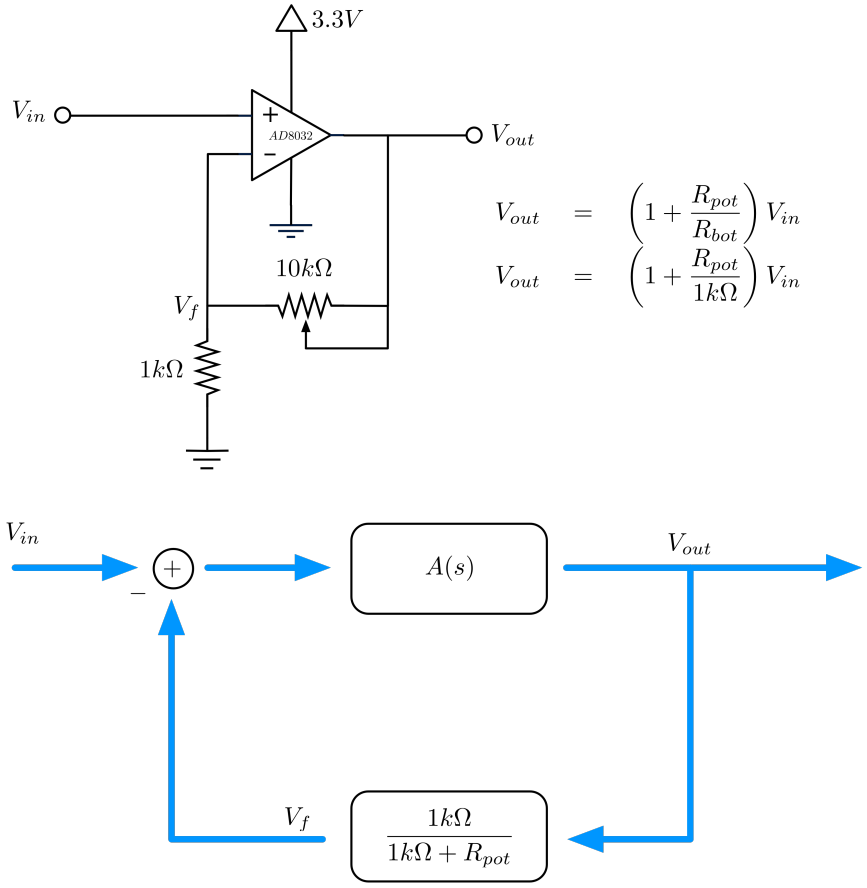
3.2.1 Bandwidth

The system bandwidth changes depending on the user selected gain. This is due to the feedback nature of our op-amp circuit.

For our gain circuit, varying the amount of feedback allows us to adjust our overall gain (Fig 3-4). Concurrently, this gain change also alters our feedback dynamics (Fig 3-7).

$A(s)$ is the representation of our op-amp in the frequency domain. Typically $A(s)$ is represented as a first-degree low-pass filter roughly characterized as $\frac{Open\ Loop\ Gain}{1 + s * \left(Gain\ Bandwidth / Open\ Loop\ Gain\right)}$. Fig 3-9 and Fig 3-10 depict the $A(s)$ bode plots for our op-amps. Altering the resistor divider controlling our feedback voltage manipulates our bandwidth (Fig 3-7). The largest bandwidth is at unity gain, for the resistor ratio is $\frac{1k\Omega}{1k\Omega + 1\Omega} \cong 1$. This is evident in the of $L(s)$, loop transmission, bandwidth of Fig 3-8. For a gain of 1, F = 1; we have the top graph (blue). However, as we increase the

overall system gain of our op-amp circuit, we reduce the size of our resistor feedback divider. For example, a gain of 10 requires a resistor divider of $\frac{1k\Omega}{1k\Omega+9k\Omega} \cong \frac{1}{10}$. This modifies $L(s)$ to now be $\frac{1}{10} * A(s)$. Our new plot is the second to top graph (green). As we increase the overall gain, our bandwidth decreases (Fig 3-8).



$$L(s) = Elements\ of\ Loop = \left(\frac{1k\Omega}{1k\Omega + R_{pot}}\right) * A(s)$$

$$V_{out} = \frac{A(s)}{\left(\frac{1k\Omega}{1k\Omega+R_{pot}}\right) * A(s) + 1} V_{in} = \frac{A(s)}{L(s) + 1} V_{in} \quad (Black's\ Formula)$$

$$Assume\ A(s)\ large\ enough\ so\ that\ \frac{A(s)}{A(s) + 1} \simeq 1$$

$$V_{out} = \frac{1}{feedback} V_{in} = \left(1 + \frac{R_{pot}}{1k\Omega}\right) V_{in}$$

Figure 3-7

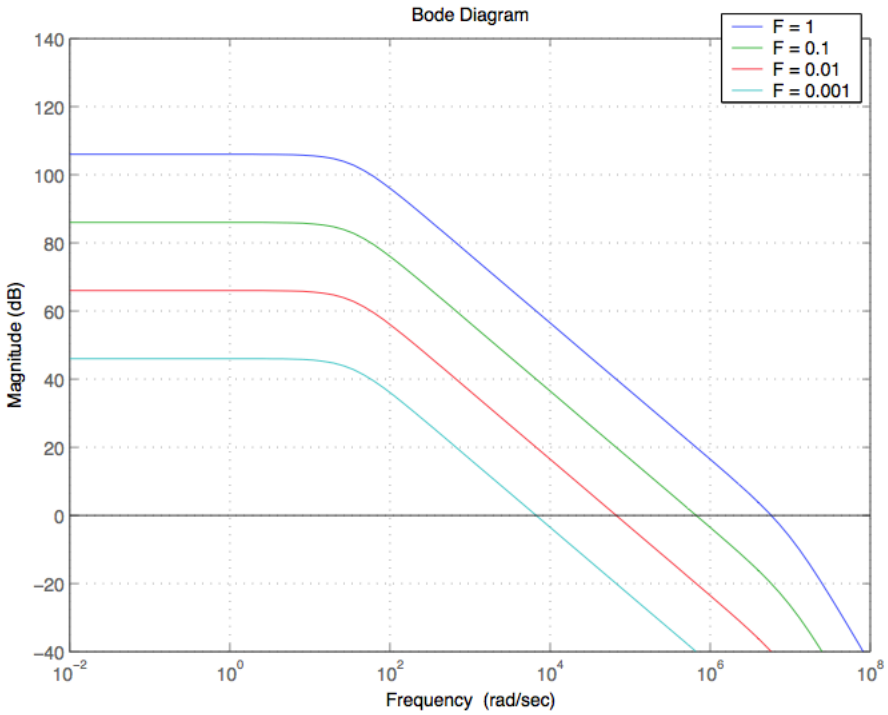


Figure 3-8 [7]. From Kent Lundberg, "Internal and External Op-Amp Compensation: A Control-Centric Tutorial ," in *American Control Conference*, vol. 6, Boston, 2004 , pp. 5197 - 5211.

This presents an obstacle in terms of bandwidth. While our gain of 1 and 2 are capable of over 12MHz, our gain of 5 barely achieves 12MHz and our gain of 10 is capable only of ~2MHz.

Gain (small signal)	Gain 1	Gain 2	Gain 5	Gain 10
3dB Frequency	> 12MHz	>12MHz	~12MHz	~2MHz

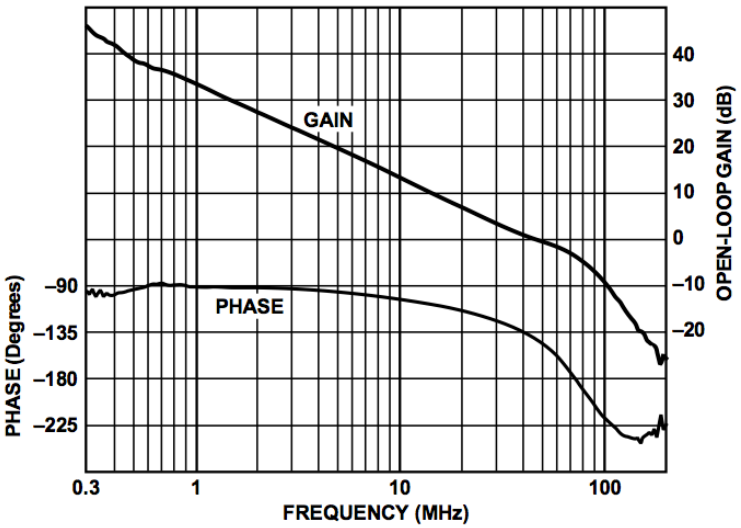


Figure 29. Open-Loop Frequency Response

Figure 3-9 AD8032 [5]

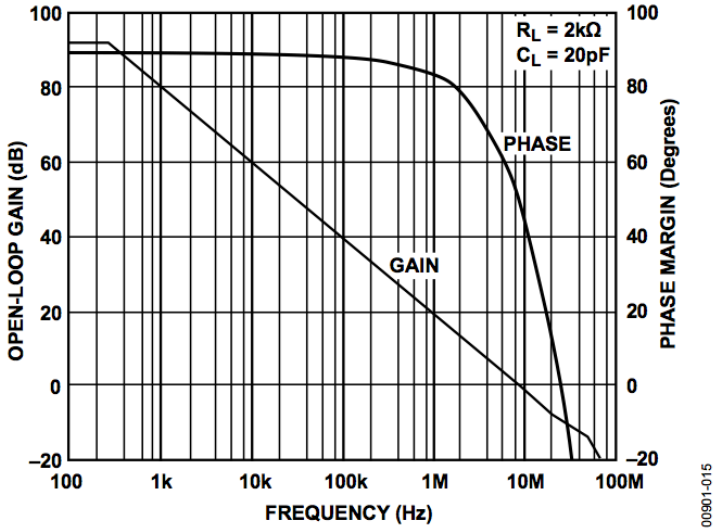


Figure 15. Open-Loop Gain and Phase Margin vs. Frequency

Figure 3-10, AD823 Datasheet [6]

3.2.2 Slew Rate

The other main issue for our input amplifier is the slew-rate that produces our maximum slope limitation.

The slew rate for the AD8032 is 30 V/μs and for the AD832 is 20 V/μs [5] [6]. Unfortunately, the slew-rate greatly constricts our signal range. We see signal distortion above these slew specs.

Figure 3-11. Input Signal =1Vpp x 2 Gain at 8MHz. Notice DC shifts for each trace.

In Fig 3-11, the input signal is originally a 1Vpp sine wave at 8MHz. The bottom signal (yellow) represents the signal after the first unity gain AD832 buffer (Fig 3-4). The middle graph (pink) is the signal with a virtual ground of half the rail, and the top graph (green) is the same signal without the virtual ground. The pink and green signals materialize as a triangle waves rather than a squares.

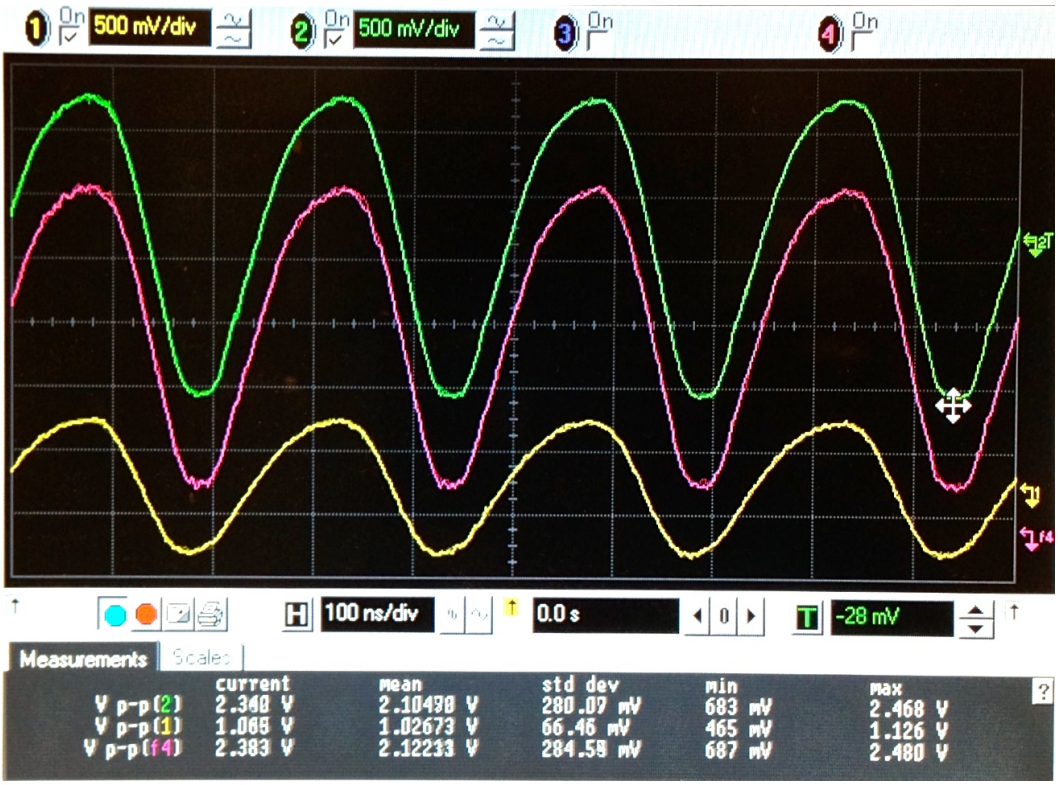


Figure 3-12. 1Vpp x 2 Gain, 4MHz

Although Fig 3-12 has a slower input signal, we still see slew distortion. Moreover, we have around a 10pF load from the clipping diodes for both the attenuator and pre-amplifier. These capacitive loads also decrease our bandwidth for large amplitude signals for we must dump and empty their charge.

4 Trigger Circuit

4.1 Overview

A trigger circuit is critical for sequential sampling. There are three main sections for the trigger: a comparator, a pre-amplifier, and hysteresis circuit.

4.1.1 Comparator

A comparator is the primary trigger mechanism. The crucial features are propagation and settling time. This is mainly to ensure each trigger occurs quickly and reliably for our sampling frequency of 3MSPS.

Input offset is of some concern as well, however if the input offset is repeatable and well characterized, then triggers will still appear correctly. The main concern for offset is repeatability.

We are using the AD8561 comparator with a 2.3mV input offset with worst-case ~10ns propagation delay at 3V supply with a small hysteresis circuit as shown in Fig 4-5.

Regrettably, the comparator is a somewhat digital device. Large logic swings produces noise during edge events. This may couple into the ADC channel. As seen in Fig 4-1, large noise spikes emerge in the signal path (purple, middle graph) during edges in the trigger signal (pink, top graph).

Figure 4-1. 9MHz sine at 200mVpp. Green is the trigger voltage.

4.1.2 Pre-Amplifier

The pre-amp is an additional (and optional) add-on for the trigger circuit. As long as the offset for the comparator remains consistent for a fixed input signal, then the overall trigger mechanism will perform adequately. Should the user wish for a more controlled trigger system, one might implement a pre-amp.

A pre-amp attenuates input referred noise and input referred voltage offset. Often pre-amps are the first input stage of a latched IC comparator as seen in the StrongArm comparator topology of Fig 4-2 [8]. However, we are not interested in IC comparator design. Rather, we wish to improve a pre-existing comparator chip. For a prearranged discrete comparator, a pre-amp will not have direct access to a latch, and therefore pre-amp is not guaranteed to positive feedback. Another challenge is the large common mode range required for use with a discrete comparator. Due to this, a

typical differential amplifier is not suggested. However, one could use a windowed instrumentation amplifier (Fig 4-3).

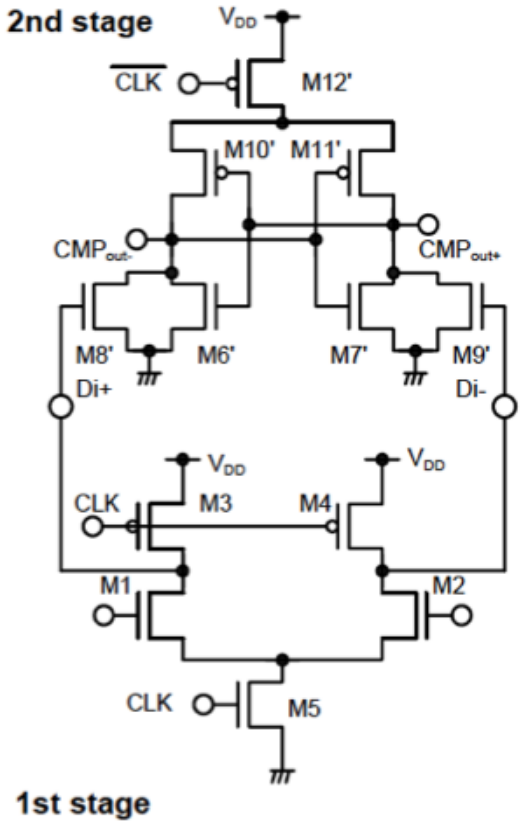
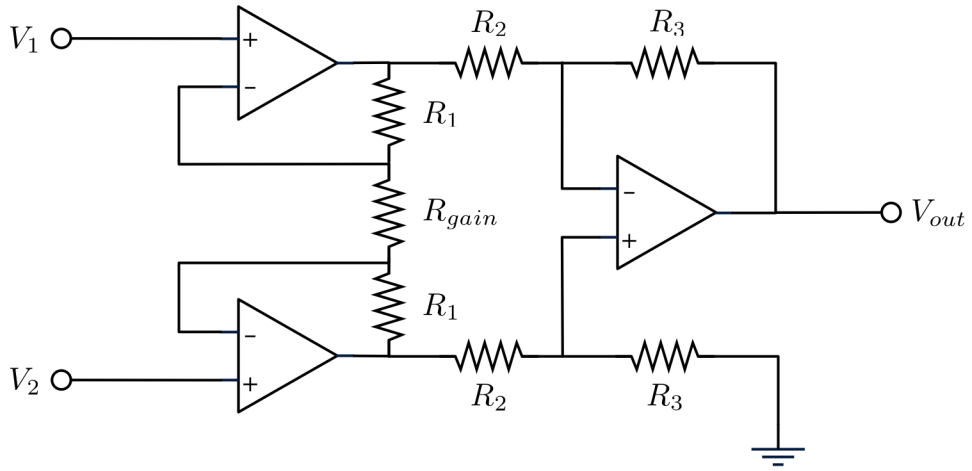


Figure 4-2, Strong Arm with Dynamic Pre-amp (first stage). M Miyahara, Y Asada, D Paik, and A Matsuzawa, "A low-noise self-calibrating dynamic comparator for high-speed ADCs," in *Solid-State Circuits Conference*, Fukuoka , 2008, pp. 269 - 272. [8].



$$V_{out} = (V_2 - V_1)\left(1 + \frac{2R_1}{R_{gain}}\right)\frac{R_3}{R_2}$$

Figure 4-3. Windowed Instrumentation Amplifier

The instrumentation amplifier is a suitable proxy pre-amp. The amplifier has very low DC input offset, low noise and drift, high input impedance, and high CMRR. The open–loop gain is incredibly large. However, due to its op-amp structure, one must select for speed and stability. The overall system, closed-loop gain of the instrumentation amplifier can be fairly small - just enough to boost input signals past the required input offset spec. One should select a conservative closed-loop gain that allows the instrumentation amplifier to remain within the linear regime. This prevents the amplifier from excessive slewing, and thus wasting time.

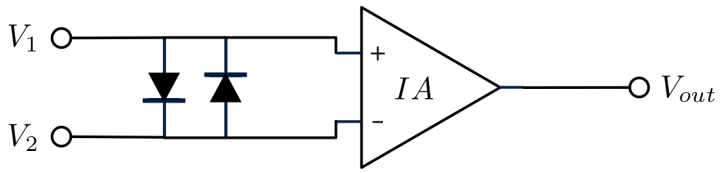
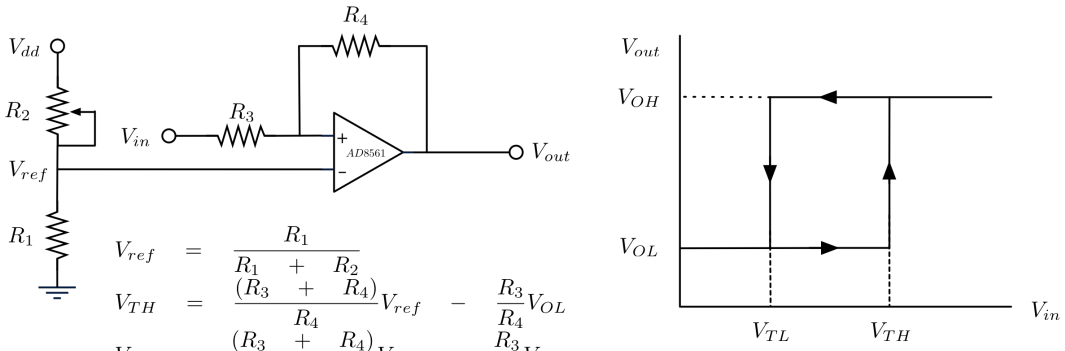


Figure 4-4. Instrumentation amp with windowing diodes.

4.1.3 Hysteresis

Hysteresis circuits decrease noise nearby trigger points Fig 4-5. Two resistors determine the hysteretic loop. The potentiometer ($\boldsymbol{R_2}$) allows the user to select a trigger voltage. The width of the hysteresis input voltage loop is $\frac{\boldsymbol{R_3}}{\boldsymbol{R_4}}(\boldsymbol{V_{OH} - V_{OL}})$. For our system $\boldsymbol{V_{OH} = 3V}$ and $\boldsymbol{V_{OL} = 0V}$ and $\frac{\boldsymbol{R_3}}{\boldsymbol{R_4}} = \frac{1}{30}$. This circuit then generates a rising edge at the user defined trigger voltage with some added noise immunity due to hysteresis.

A point to note is the trigger voltage exists mainly for signal capture, not as a voltage reference. The trigger voltage must be precise and repeatable, but not necessarily accurate; we must simply generate a rising edge for sequential sampling.



$$V_{ref} = \frac{R_1}{R_1 + R_2}$$

$$V_{TH} = \frac{(R_3 + R_4)}{R_4} V_{ref} - \frac{R_3}{R_4} V_{OL}$$

$$V_{TL} = \frac{(R_3 + R_4)}{R_4} V_{ref} - \frac{R_3}{R_4} V_{OH}$$

$$R_3 = 1k\Omega, \quad R_4 = 30k\Omega$$

$$V_{OH} = 3V, \quad V_{OL} = 0V$$

$$V_{TH} = V_{ref}, \quad V_{TL} = V_{ref} - 0.1V$$

Figure 4-5

5 Firmware Architecture

The firmware comprises of four main peripheral systems: ADC, DMA, external trigger/timer, and USB. These four pieces come together to produce two sampling options: 1) real-time (for signals below the ADC's inherent Nyquist rate), 2) sequential-sampling (increasing our native 3MSPS to 24MSPS with incremental phase shifts of 42ns). These are then both supplied to the user's computer whereupon he or she may employ MATLAB or Python to process the raw data signals.

5.1 libopencm3

A peripheral code library created by STMicroelectronics exists for this microcontroller - but a build system, such as Keil or Eclipse, is necessary. Libopencm3 is an open source alternative containing peripheral libraries complete with a build system. Firmware can be written in legible C and compiled with GNU Make. Unfortunately, some aspects of libopencm3 are metastable, but the memory mappings are correct. The register settings are performed through bit assignments mapped to the correct register in the STM32F3's reference manual.

5.2 Signal Chain (Real-Time Sampling)

For frequencies less than 1.5MHz, real-time sampling is available. In this manner of operation, the ADC is arranged in regular continuous mode, clocking the DMA. This sequence fills a sample buffer that, upon completion, is sent from memory to the USB (Fig 5-1). External triggers during this sequence are ignored. No sequential sampling is used.

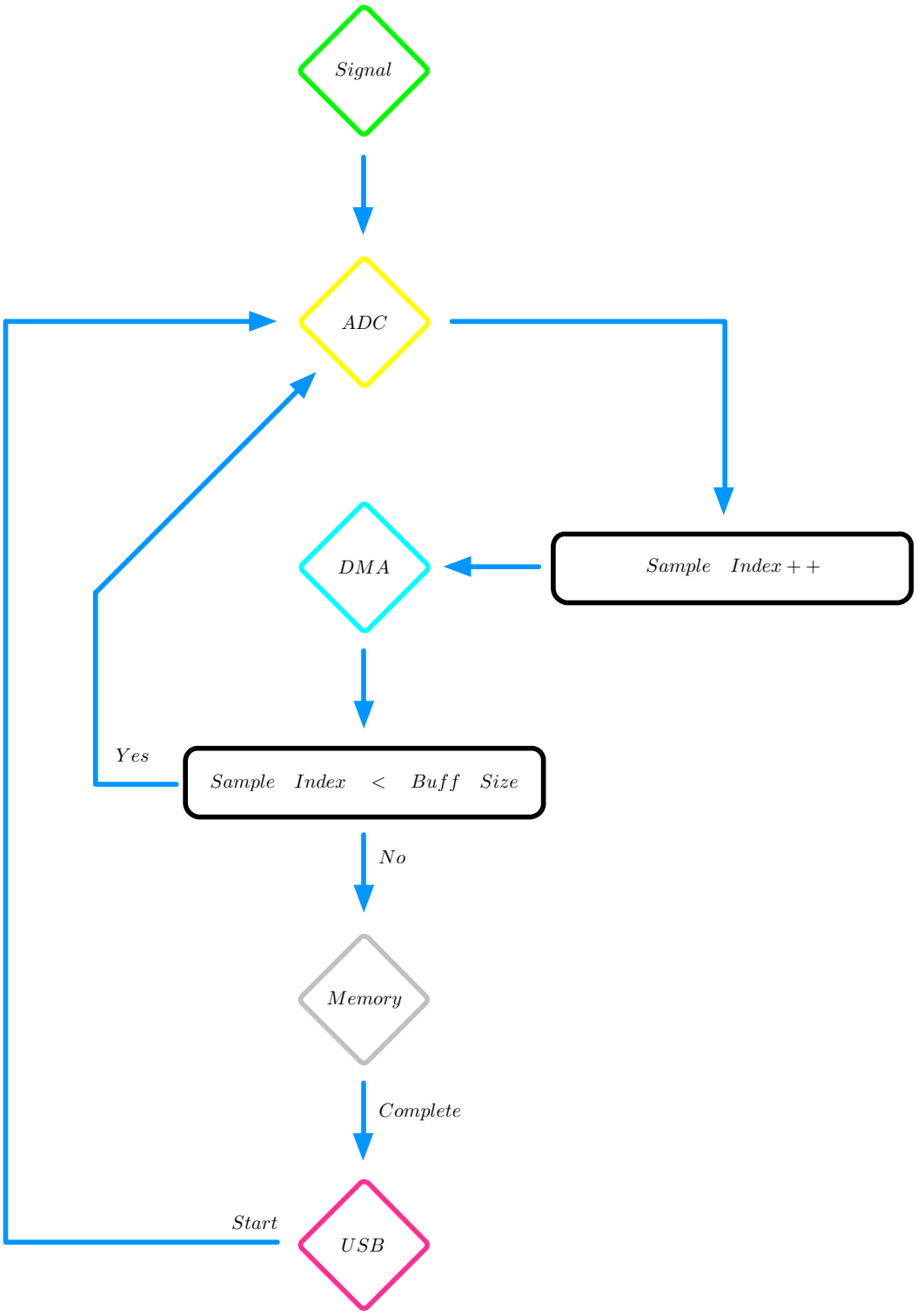

Signal
ADC
DMA
Sample Index++
Yes
Sample Index < Buff Size
No
Memory
Complete
Start
USB

Figure 5-1

5.3 Signal Chain (Sequential Sampling)

The signal chain for sequential sampling mode is illustrated in Fig 5-2. A user-defined software command arms the trigger for the first sample. An external trigger then initiates the timer that contains the variable for phase. Next, the timer activates the ADC pre-fixed in continuous mode. Continuous mode allows the ADC to clock the DMA, filling the sample buffer. Once the buffer is full, the data is sent through USB to the user. Afterward, we examine the phase stored in our phase variable. If the phase has not yet reached 360°, we increment the phase by 45° and re-arm the external hardware trigger. After 8 increments of phase, our signal is sequentially sampled at 24MSPS.

The hard-set memory buffer size (due to RAM constraints onboard the microcontroller) can be problematic. One can solve this through further optimization of the USB-laptop communication system. Note: sequential sampling is required only for frequencies above 1.5MHz. The memory buffer should be more than capable of capturing several periods of an >1.5MHz input signal.

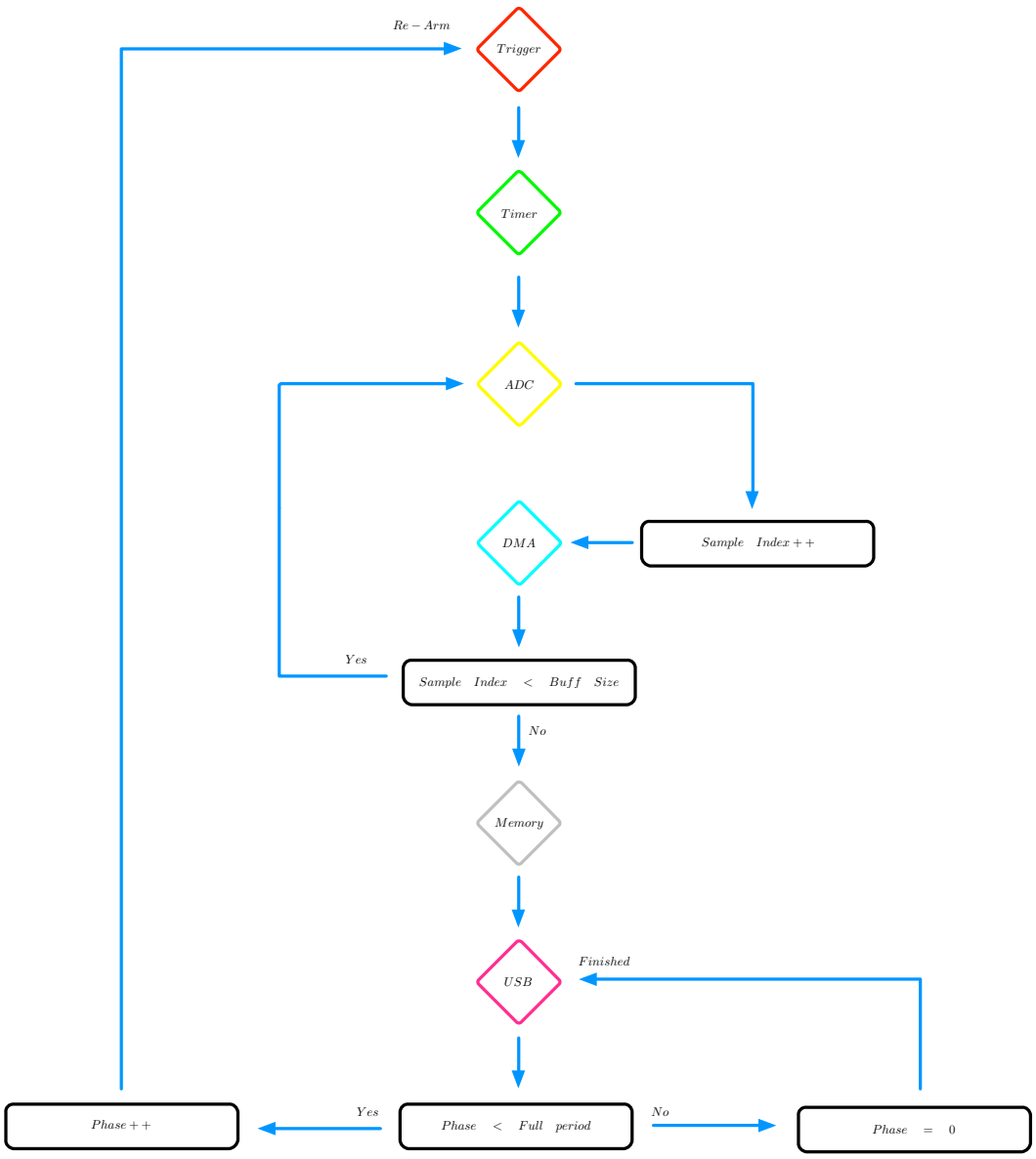

Re – Arm
Trigger
Timer
ADC
DMA
Sample Index++
Yes
Sample Index < Buff Size
No
Memory
USB
Finished
Phase++
Yes
Phase < Full period
No
Phase = 0

Figure 5-2

5.4 Trigger-Timer Subsystem

The timer and external trigger systems are interconnected, producing the phase increment mechanism for sequential sampling.

5.4.1 Timer Setup

Our timer (TIM3) must first have the correct settings. From the STM32F3 reference manual [9], we obtain the generalized block diagram of the timer system Fig 5-3. Although the register settings may be somewhat hard to decipher, we utilize only a small section of the overall timer functionality (pink).

The main points of interest are: integrating the external trigger (yellow), supplying the counter with the correct phase (green and bottom gold), and connecting the correct interrupt to the ADC (blue).

TIM3_SMCR is the slave mode control register [9]. To select an external trigger, one must turn on TIM_SMCR_TS_ETRF (External Trigger input (ETRF)) and TIM_SMCR_SMS_TM (Trigger Mode). Note: the trigger event only commands the counter register to *begin* counting. TIM_SMCR_ETPS_OFF is selected, as we do not need to filter the trigger line. The TIM_SMCR_ETP bit should be zero to ensure that the external trigger polarity is for rising-edge. All other bits in this register can be set to zero as well.

TIM3_CR1 is Control Register 1 [9]. For this register, bits TIM_CR1_CKD_CK_INT, TIM_CR1_DIR_UP, and TIM_CR1_OPM are set high. TIM_CR1_CKD_CK_INT prevents any clock division. TIM_CR1_DIR_UP directs the counter to count up from 0 to 65535. TIM_CR1_OPM places the timer in one-pulse/shot mode. The one-shot setting is especially critical - we must prevent the external trigger from disrupting the ADC-DMA system when collecting a sample buffer. One-

shot is also reinitialized upon the completion of each sample buffer, permitting the next phase to be processed.

Figure 173. General-purpose timer block diagram

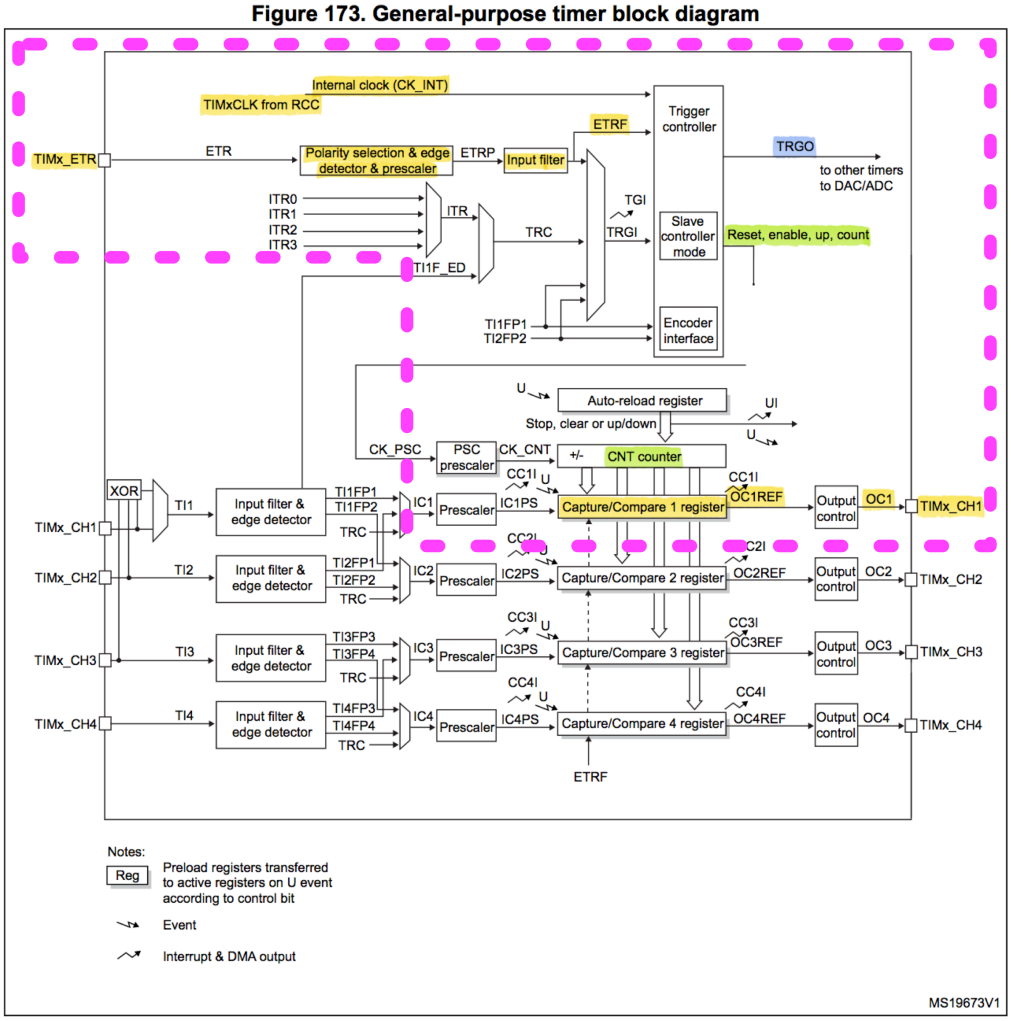


Figure 5-3, from the STM32F3 Reference Manual [9]

TIM3_CCR1 and TIM3_CNT are Capture/Compare Register 1 and the Counter. TIM3_CCR1 holds the numerical value to which one wishes to count to and TIM3_CNT increments itself until it is the value of TIM3_CCR1. These two registers hold the phase degree and the counting mechanism to reach the phase.

TIM3_CR2 is control register 2 [9]. In this register we only need TIM_CR2_MMS_COMPARE_OC1REF. This setting brings OC1REF

high when TIM3_CNT equals TIM3_CCR1. Fig 5-4 is part of the "Trigger Controller" block before the TRGO line in Fig 5-3. We see that "the master mode controller" is connected directly to OC1REF. The "master mode controller" is actually a mux controlled by the TIM_CR2_MMS_COMPARE_OC1REF. This mux allows OC1REF to attach directly to TRGO, creating a TRGO event when the OC1REF pin goes high (remember, this occurs when TIM3_CNT equals TIM3_CCR1, indicating we have reached our phase, whether it be 0°, 45°, 90°, 135°, etc.). The only path to the ADC from the timer is through the TRGO line, hence connecting OC1REF to TRGO allows us to control when the ADC conversion sequence begins.

TIM3_CCMR1 is Capture/Compare Mode Register 1. In this register we select TIM_CCMR1_OC1M_PWM2, which specifies that OC1REF is low as long as TIM3_CNT < TIM3_CCR1 and high once TIM3_CNT >= TIM3_CCR1.

TIM3_ARR is the Auto-Reload Register and controls the overall period of the PWM (which we set to the max of 0xFFFF).

TIM3_PSC controls the prescaler for the timer (seen as CK_PSC in Fig 5-3). We have set the prescaler to the default value of one.

TIM3_CCER is the Capture/Compare Enable Register. This can be set to TIM_CCER_CC1E should one wish to see TIM3_TRGO on an external pin for debugging purposes (as seen in Fig 5-4, CC1E sets the subsequent muxes connecting OC1, that can be tied to an external pin, to OC1REF). A pin to use on the STM32F3 Discovery board is PB4 (note: one must then initialize the PB4's GPIO setting as alternate function two, e.g. GPIO_AF2).

Figure 200. Output stage of capture/compare channel (channel 1)

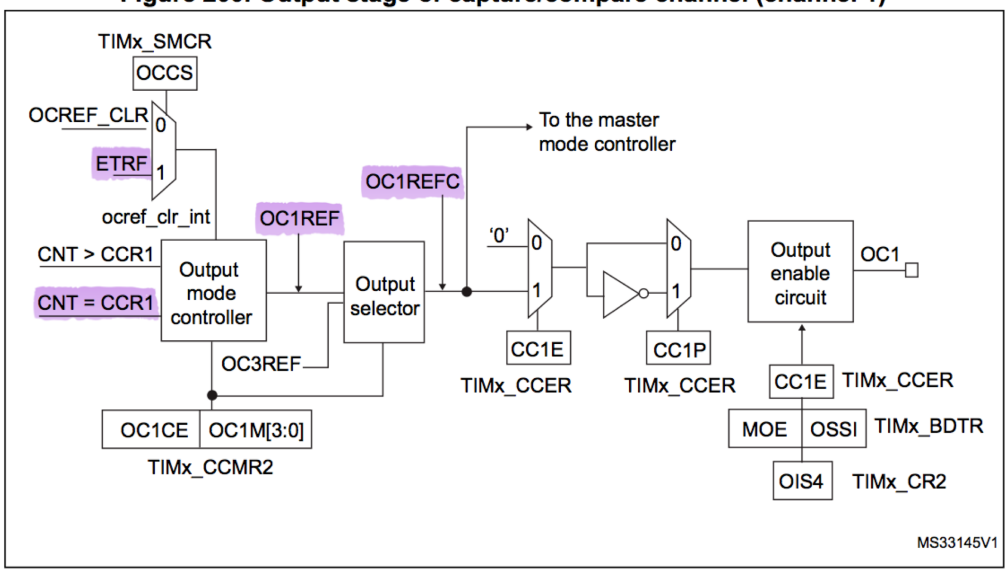


Figure 5-4

5.4.2 Phase Subsystem

The phase increment for sequential sampling is implemented through the timer counter register. The master clock (RCC) for the STM32F3 is 48MHz. The ADC sample rate is 3MSPS. Given these two conditions, we sample eight phases: 0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°, and 360° - each phase is an additional 42ns. The timing diagram for the phase increments is shown in Fig 5-5 (note not drawn to scale).

Timer 3 (TIM3) is tied to an external trigger circuit that produces falling edges (not shown on diagram) when the input signal reaches the trigger voltage (green Fig 5-5). This event then activates TIM3's PWM2 feature (a 16-bit upward counter). Register TIM3_CNT increments until its value matches the number in register TIM3_CCR1. When the two registers are equal, a TIM_TRGO event occurs and a rising edge is produced on EXT4 (red Fig 5-5). This then activates the ADC-DMA subsystem that fills a sample buffer (pink dashed box Fig 5-5). Note, should a trigger occur during the ADC-DMA sampling state, it is ignored.

External trigger interrupts are considered invalid until the sample buffer is full.

In the code, each sample buffer is called a "gel" (inspired by the color gels used in theater lighting). Each gel correlates with a phase. When eight gels (and hence eight phases) are collected, we have finished our sequential sampling. To systematically increase the phase, TIM3_CCR1's value is incremented when the DMA has collected the last sample for a given gel (end of each pink dashed box Fig 5-5). We first begin with TIM3_CCR1 = 0x0h. As $\frac{\mathbf{48}\boldsymbol{MHz}}{\mathbf{3}\boldsymbol{MHz}} = \mathbf{16}$ and we have eight gels, each of 42ns, the TIM3_CCR1 register is incremented by 2 until the last gel.

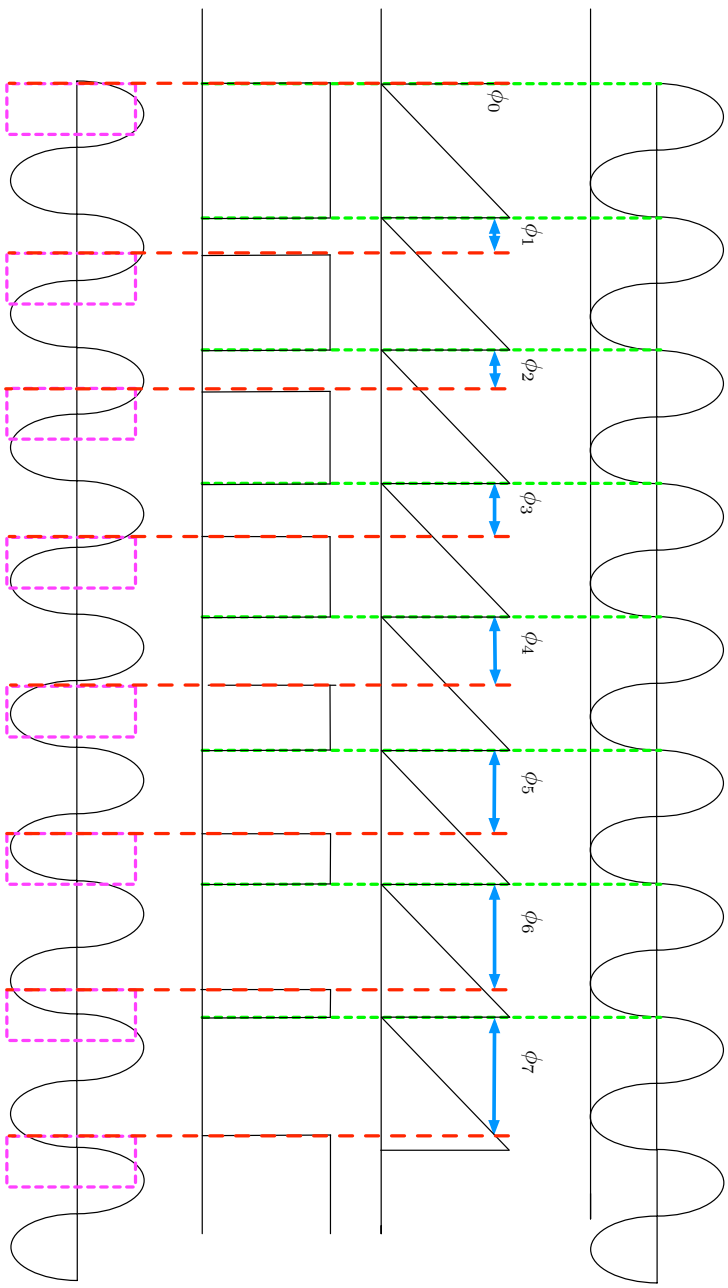

Input Signal with Trigger
TIM3_PWM2
TIM3_TRGO event on EXT4
ADC Captured Windows
ϕ_0
ϕ_1
ϕ_2
ϕ_3
ϕ_4
ϕ_5
ϕ_6
ϕ_7

Figure 5-5

5.5 ADC-DMA Subsystem

The second large subsystem is our ADC-DMA block. This block captures samples and places them in a buffer, later transmitting them to the user through USB. Sending one ADC sample to the USB at a time is exceedingly slow. Connecting the ADC to the DMA allows us collect a buffer of samples at the maximum ADC sample rate. We then later send them as blocks of data to the USB (which transfers data at a slower rate). We are using ADC1 in regular continuous mode with the DMA in one-shot mode.

5.5.1 ADC

The ADC setup from the reference manual is somewhat difficult to understand and particularly complicated. The ADC is the most complex section of this whole oscilloscope firmware system. Here are some setup elaborations.

5.5.1.1 ADC Off

First, the ADC must be off (not paused) when establishing registers settings. ADC1_CR (control register) controls the bits that activate and disable the ADC. Before attempting any register settings, one must disable ADC_CR_ADEN in ADC1_CR and wait until that bit is cleared by hardware. Afterwards, set ADC_CR_ADDIS and wait until the bit is cleared. Note: this is the procedure for shutting the ADC off, not pausing the ADC during sampling. The specific procedure is outline in Fig 5-6.

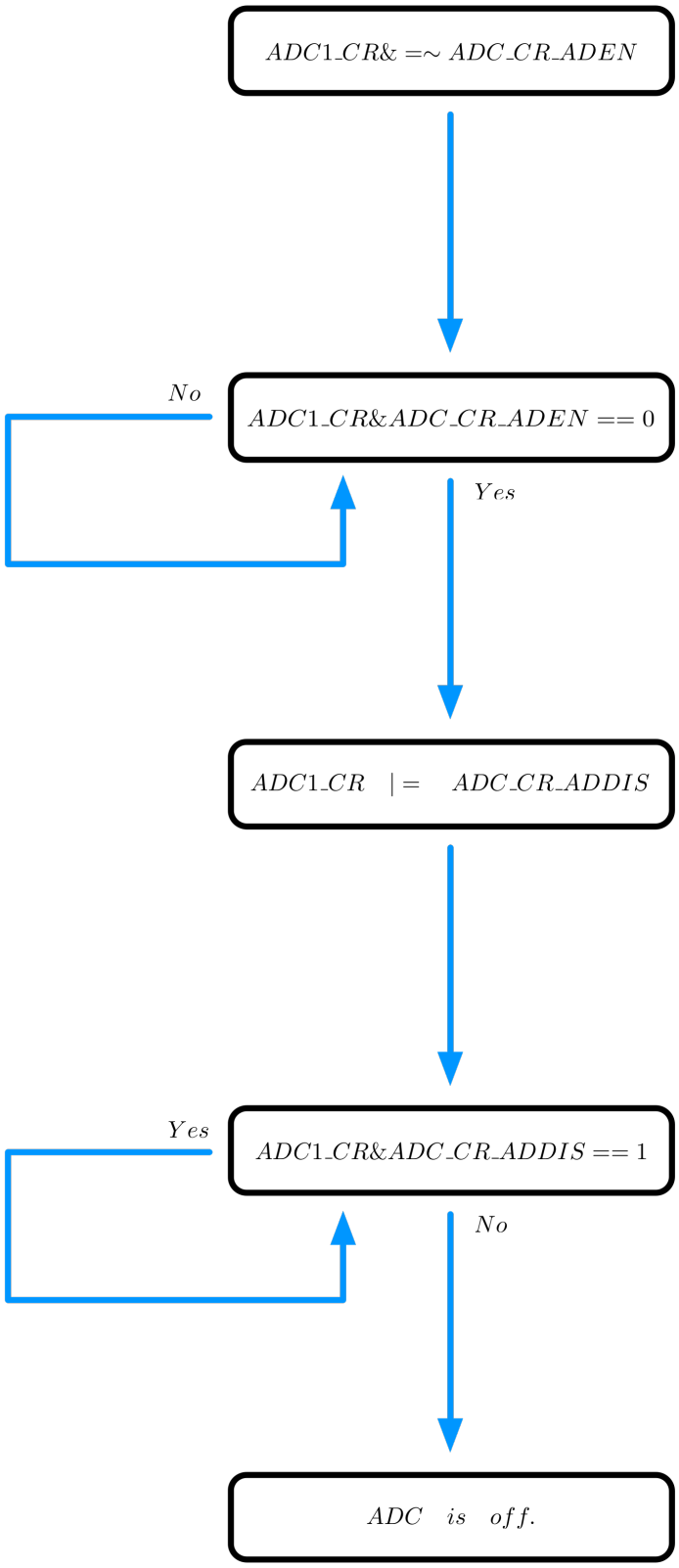

ADC1_CR& =~ ADC_CR_ADEN
No
ADC1_CR&ADC_CR_ADEN == 0
Yes
ADC1_CR |= ADC_CR_ADDIS
Yes
ADC1_CR&ADC_CR_ADDIS == 1
No
ADC is off.

Figure 5-6

5.5.1.2 ADC On/Unpaused

The ADC on procedure is also very specific (Fig 5-7). The ADC will not activate unless the commands are given in this particular manner.

One must first enable ADC_CR_ADEN then wait until ADC_ISR_ADRDY is high. After this, the ADC will not capture samples until ADC_CR_ADSTART is set high by the user.

Note: one may pause the ADC during regular continuous mode. In order to un-pause the ADC, this routine must be performed.

Calibration must be completed before the ADC is activated. Simply set ADC_CR_ADCAL in ADC1_CR, and wait until the microcontroller sets the bit in hardware.

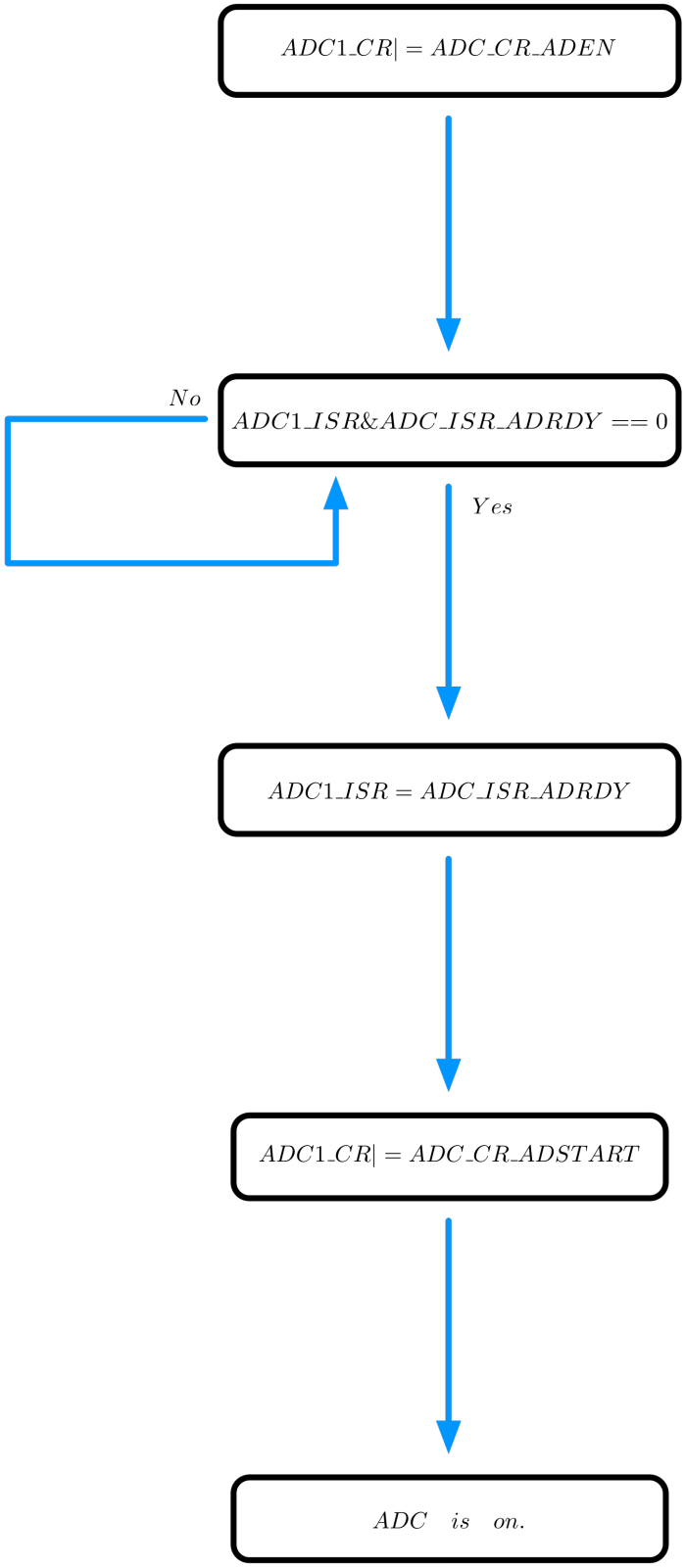

ADC1_CR| = ADC_CR_ADEN
No
ADC1_ISR&ADC_ISR_ADRDY == 0
Yes
ADC1_ISR = ADC_ISR_ADRDY
ADC1_CR| = ADC_CR_ADSTART
ADC is on.

Figure 5-7

5.5.1.3 ADC Paused

Pausing the ADC and disabling the ADC are two different procedures. One powers off the ADC while the other temporarily suspends ADC capture. Turning the ADC on and off repetitively will erase previous ADC calibrations. Pausing the ADC saves the calibration values from startup.

Pausing is absolutely necessary for our phase offsets. Each time TIM3_CCR1 is incremented (essentially, each time our phase is incremented), we must pause the ADC so as to not collect samples during a phase shift.

First, check that the ADC has already been started (Fig 5-8). Then set the ADC_CR_ADSTP bit and wait until the microcontroller clears it in hardware. Next, ADC_CR_ADDIS must be set, and we must again wait for the ADC_ADEN bit to be cleared by the microcontroller before we can safely assume the ADC is paused.

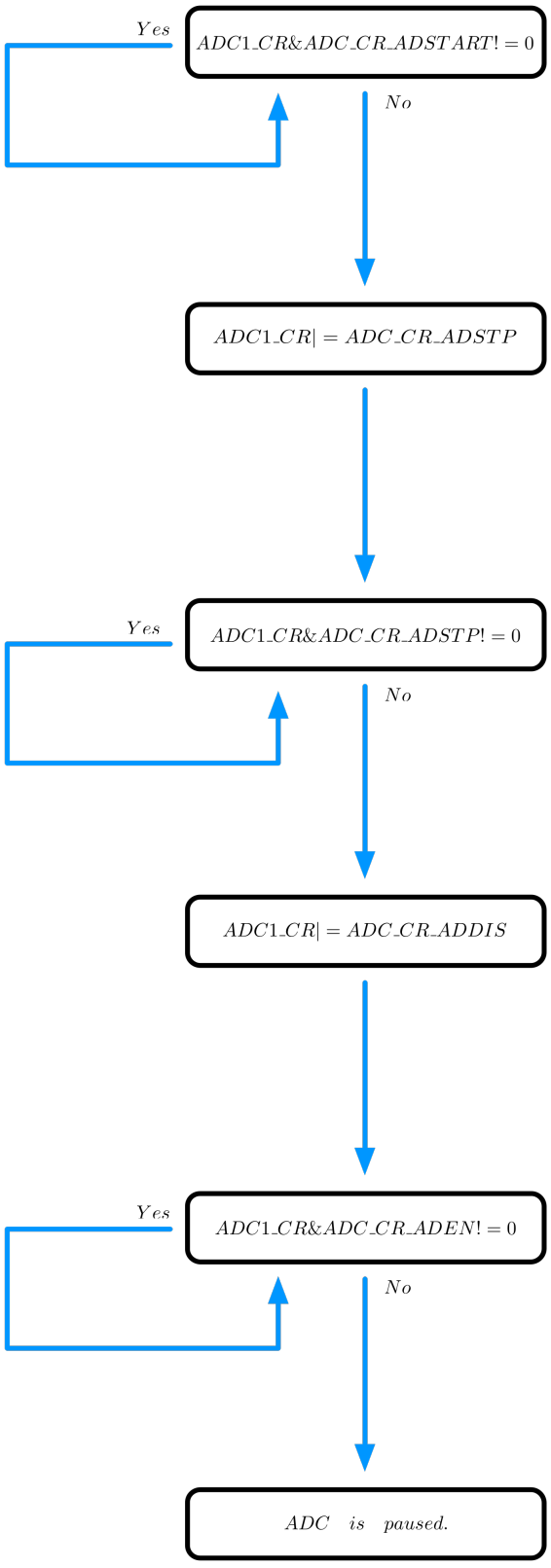

Yes
ADC1_CR&ADC_CR_ADSTART! = 0
No
ADC1_CR| = ADC_CR_ADSTP
Yes
ADC1_CR&ADC_CR_ADSTP! = 0
No
ADC1_CR| = ADC_CR_ADDIS
Yes
ADC1_CR&ADC_CR_ADEN! = 0
No
ADC is paused.

Figure 5-8

5.5.1.4 ADC Register Settings

The STM32F3 has four sophisticated ADCs (Fig 5-9). Unfortunately, this causes registers settings to be quite complex.

Figure 28. ADC block diagram

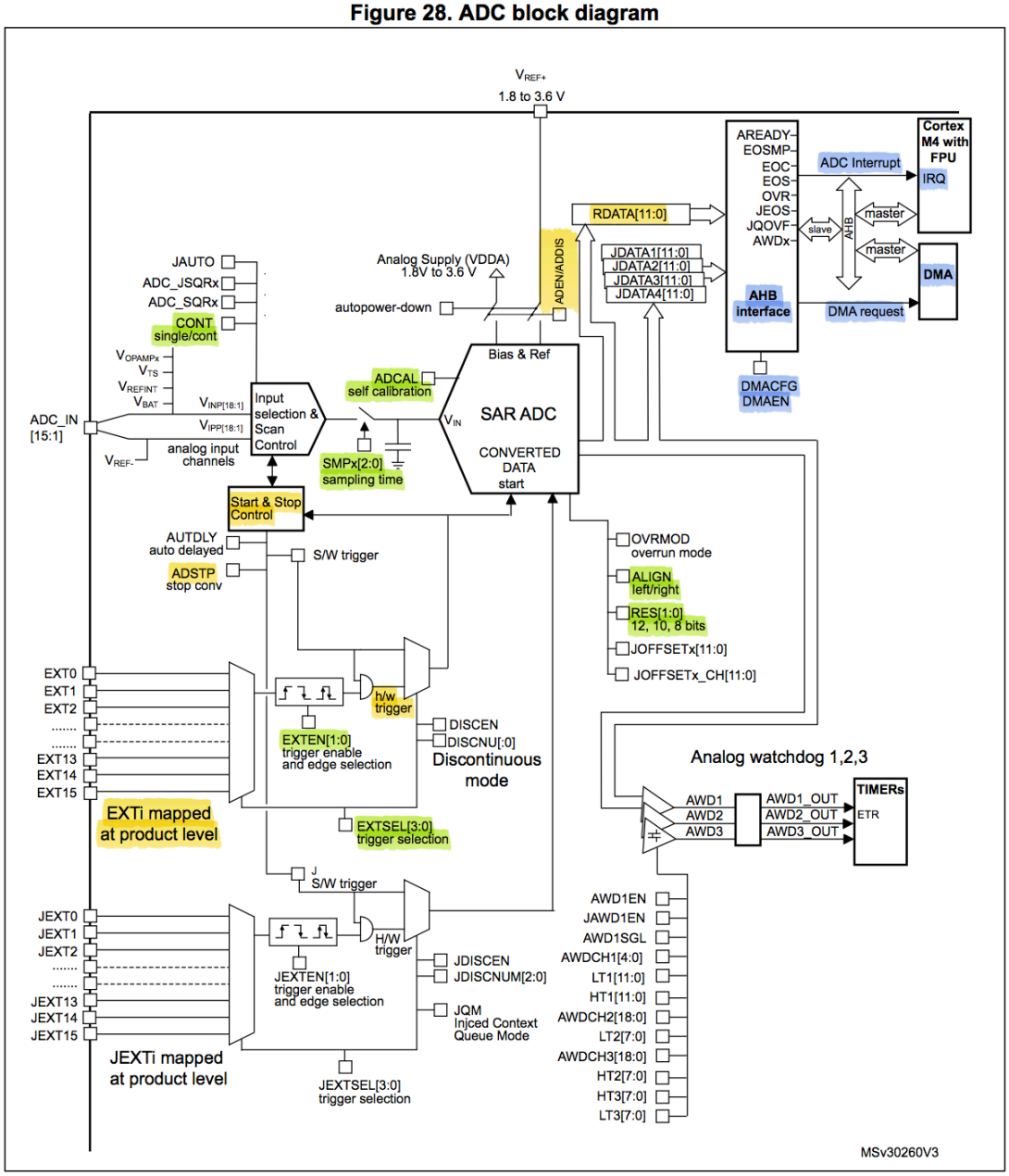


Figure 5-9

ADC1_CFGR is the main register for ADC1 settings unrelated to activation/deactivation or interrupts. In this register, we must set ADC_CFGR_CONT (continuous mode), ADC_CFGR_DMAEN (DMA enabled NOT in circular buffer mode), and ADC_CFGR_RES_8_BIT (8-bit ADC data).

We must also be careful in configuring the ADC trigger. We use TIM3, therefore we must remap TIM3's TRGO event to the ADC. From Fig 5-9, we see that the ADC can be triggered off of a selection of EXTi (external pin triggers). Listed in the reference manual is a set of external interrupt mappings (Fig 5-10). For this design, we used ADC_CFGR_EXTSEL_EVENT_4 configured with a rising edge (ADC_CFGR_EXTEN_RISING_EDGE), for TRGO and OC1REF are connected together and go high when the counter has reached its specific phase.

Table 39. ADC1 (master) & 2 (slave) - External triggers for regular channels

Name	Source	Type	EXTSEL[3:0]
EXT0	TIM1_CC1 event	Internal signal from on chip timers	0000
EXT1	TIM1_CC2 event	Internal signal from on chip timers	0001
EXT2	TIM1_CC3 event	Internal signal from on chip timers	0010
EXT3	TIM2_CC2 event	Internal signal from on chip timers	0011
EXT4	TIM3_TRGO event	Internal signal from on chip timers	0100
EXT5	TIM4_CC4 event	Internal signal from on chip timers	0101
EXT6	EXTI line 11	External pin	0110
EXT7	TIM8_TRGO event	Internal signal from on chip timers	0111
EXT8	TIM8_TRGO2 event	Internal signal from on chip timers	1000
EXT9	TIM1_TRGO event	Internal signal from on chip timers	1001

Figure 5-10

ADC1_SMPR1 is the sample time register that controls the time between each ADC sample in continuous mode (e.g. sampling rate). Note: there are actually two sample time registers for each of the four ADCs (SMPR1 and SMPR2). This is due to the fact that each ADC has several multiplexed inputs (up to 14 on ADC1 for the STM32F3 Discovery Dev

Board). Other ADCs can have up to 18 multiplexed inputs. One must index into the correct bits for the channel(s) one intends to use. Hence settings for SMPR must be bit-shifted for the correct channel.

The general formula for sample rate is thus:

$$\boldsymbol{T_{conv} = \frac{(SMPR + BITS + 0.5)}{RCC\ Clock}}$$

Equation 1

For our 8-bits with an RCC clock of 48MHz, and an SMPR value of 7DOT5CYC, our $\boldsymbol{T_{conv} = \frac{(7.5+8+0.5)}{48MHz} = 0.33\mu s}$, or 3MSPS.

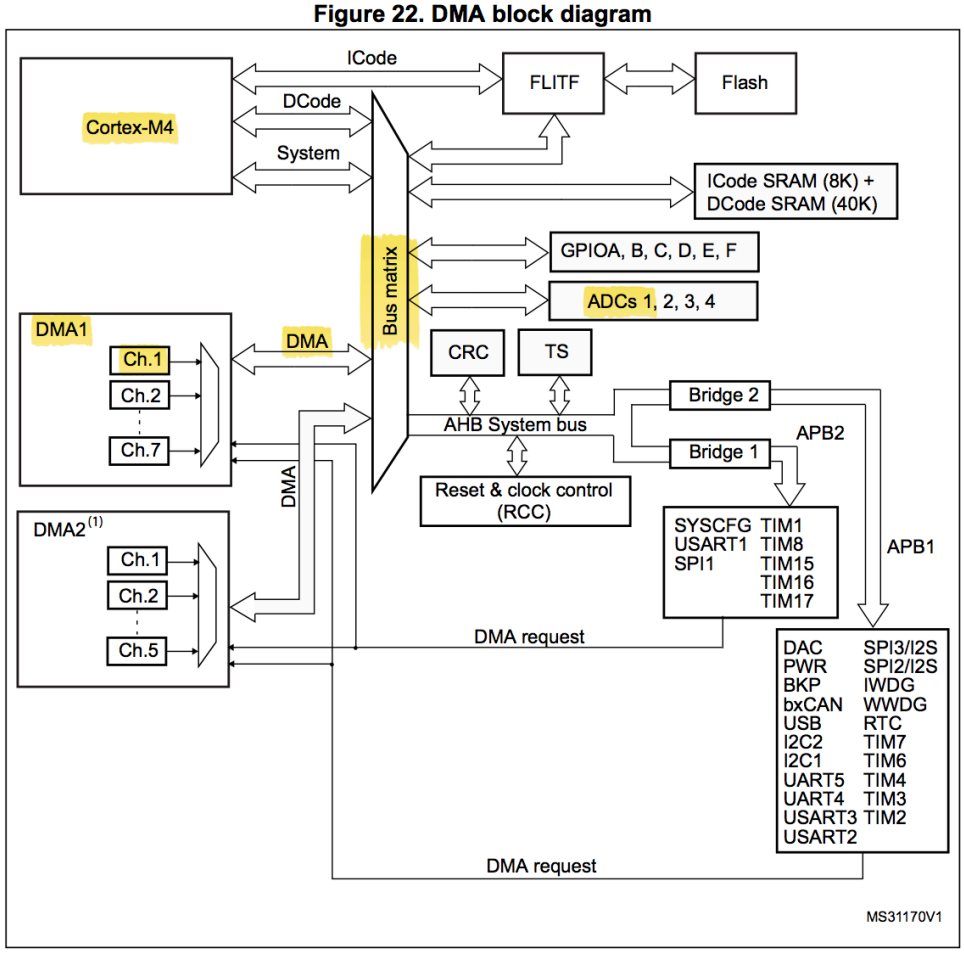
ADC1_SQR1 and ADC1_SQR2 select the port pin joined to ADC1. Before choosing a pin, check both the STM32F3 and STM32F3 Discovery datasheets to ensure the pin is not connected to a strange peripheral (such at as the gyroscope or accelerometer), for those have a tendency to interfere with the ADC.

ADC_CCR is the ADC Common Control Register and manages aspects such as optional battery usage and clock division. In this register, we only need to specify the clock division as ADC_CCR_CKMODE_DIV1.

5.6 DMA Register Settings

The DMA module allows us to capture large sections of data at a time, and then send them as a stream over USB. There are two DMA modules in the microcontroller. We are only using DMA1 channel 1 for this oscilloscope system. The signal path follows the yellow sections of Fig 5-11.

Figure 22. DMA block diagram



1. DMA2 is not available in STM32F303x6/8 and STM32F328x8 devices.

Figure 5-11

DMA1_CCR1 is the channel control register for DMA1. We set the priority as high with both the memory and peripheral size as 8-bit (as the ADC is 8-bit). We also must set the memory increment feature (DMA_CCR_MINC) and the transfer complete IRQ (DMA_CCR_TCIE).

DMA1_CNDTR1 dictates the size of the data buffer for one DMA sweep (after which the DMA transfer complete IRQ fires). This controls the size of the sample buffer for *one* gel.

DMA1_CPAR1 is the peripheral address register for channel 1 of DMA1. This must be set to the address of the ADC data register. Data is taken from this address and placed in the data buffer.

DMA1_CMAR1 is the memory address register for channel 1 DMA1. This must be set to the address of the data buffer.

Lastly, we must enable the DMA NVIC interrupt, which will go high whenever a sample buffer is filled. We must also enable the DMA module *after* the register settings have been completed.

5.6.1 DMA – ADC Module

If all the ADC and DMA settings are correct, the system should behave as described in Fig 5-12.

Similarly to Fig 5-2, the trigger activates the timer that then initiates the ADC. The DMA and the ADC have a feedback loop that completes once the memory buffer for data is full. This then generates the DMA transfer complete IRQ. If all 8 gels of data are collected, the data buffer will be sent through USB, otherwise the phase increments and the trigger is re-armed to capture another gel of data.

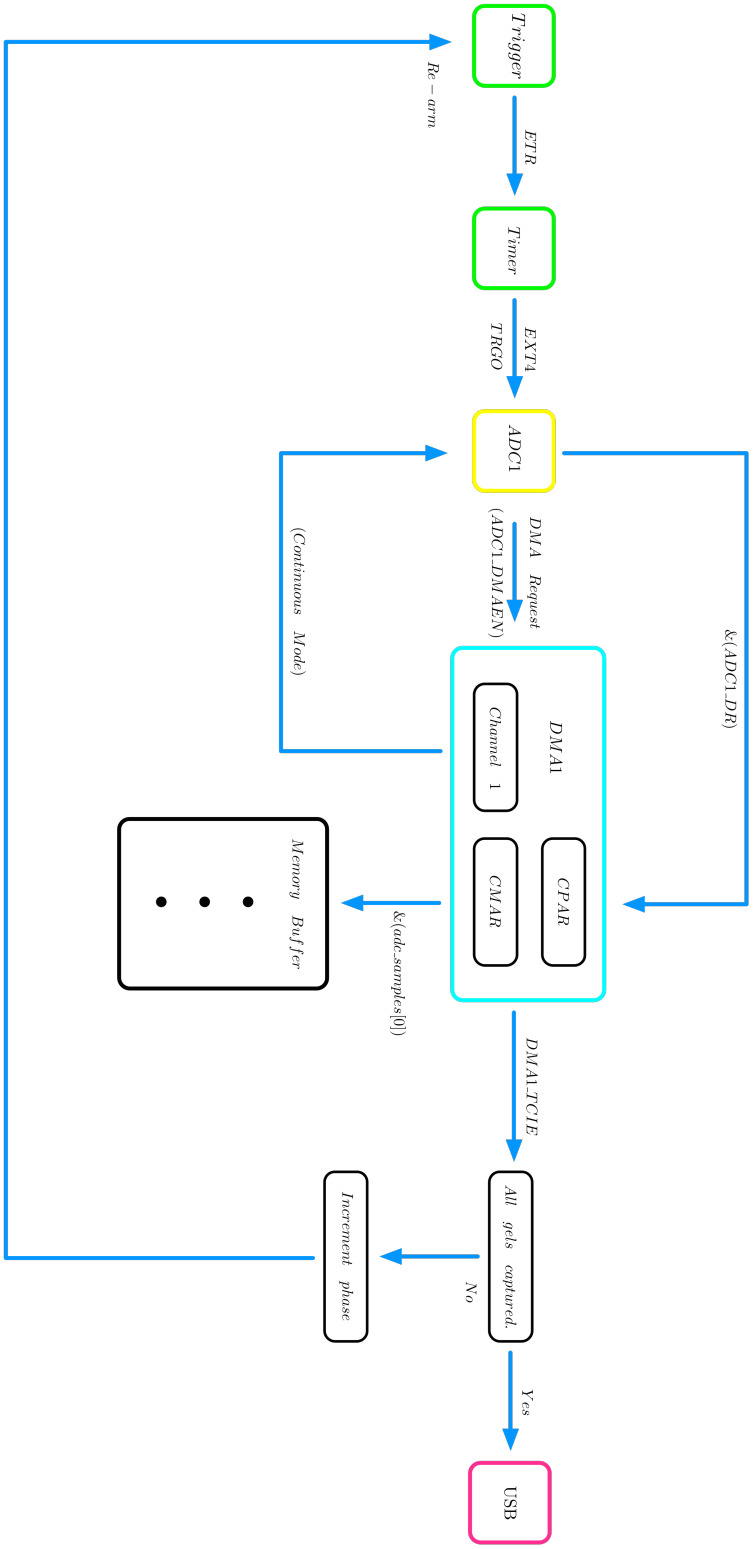

Trigger
ETR
Timer
EXT4
TRGO
ADC1
DMA Request
(ADC1_DMAEN)
&(ADC1_DR)
(Continuous Mode)
DMA1
Channel 1
CMAR
CPAR
Memory Buffer
&(adc_samples[0])
DMA1TCIE
All gels captured.
No
Increment phase
Re − arm
Yes
USB

Figure 5-12

5.6.2 USB

The USB library employed is native to libopencm3. There are two primary functions: cdcacm_data_tx_cb and cdcacm_data_rx_cb. The cdcacm_data_tx_cb function is called whenever the usbd_ep_write_packet function is called. Likewise, the cdcacm_data_rx_cb function is called whenever the usbd_ep_read_packet function is called (essentially whenever the laptop sends data).

5.6.3 Summary

In short, the four main firmware pieces of this system are the timer, ADC, DMA, and USB. Together all contribute to the sequential sampling system with phase increments. Careful register settings must be executed for the hardware modules on the microcontroller.

6 Signal Processing

The input signal is processed in MATLAB and displayed as a graphical figure.

6.1 Signal Chain

Ideally the signal chain should appear as in Fig 6-1. There are three main sections: 1) the pre-filtered analog signal and input filtering hardware (green), 2) the sampling from the ADC of the STM32F3 (yellow), 3) the reconstruction in MATLAB (pink).

Ideally, the input signal would be filtered with a low-pass filter with a ½ LSB magnitude at the cutoff of 12MHz (Fig 6-1 green-dashed box). The low-passed input would then be sampled by the STM32F3 (modeled as a delta train of 24MHz in the yellow-dashed box of Fig 6-1).

The input signal and the delta train would then produce a series of copies in the frequency domain (Fig 6-2). A brick wall in the frequency domain would afterwards process this train of copied signals; we would return our original low-pass filtered signal (Fig 6-1, pink-dashed box).

External Hardware / Outside World
Input Signal
X(f)
W = 12MHz
Low – Pass Filter
G(f)
X'(f)
x'(t)
ADC/Microcontroller
Sampling Train
s(t)
Laptop/MATLAB
H(f)
X''(f)

Figure 6-1

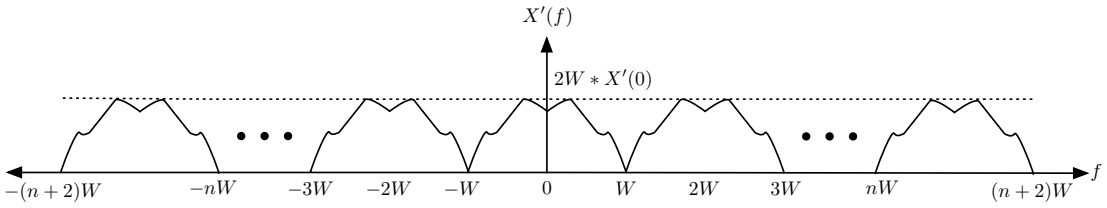


Figure 6-2

A few liberties were taken with this ideal signal chain. The input low-pass filter was somewhat omitted. The input amplifier of our scope provides some inherent filtering to 16MHz. However, one hopes all input signals transmitted to the scope are less than 12MHz. This may be somewhat problematic for signals such as triangle or square waves, for their Fourier coefficients continue infinitely. Such waveforms would result in high-order coefficients aliasing onto our input waveform. This would manifest as distortion in lower frequency bands if these coefficients were not of small enough amplitude.

In addition, the 24MHz sampling is approximated through sequential sampling as 8 gels of 3MSPS. We must not forget we technically only have 3MSPS for our ADC and noise may be picked up cyclically from during sequential sampling.

6.2 MATLAB Reconstruction

In order to reconstruct our signal, we pass our raw input data through MATLAB code. This code can later be imported to python so as to be free for potential students and users.

6.2.1 MATLAB Code Overview

The MATLAB code must do two things: 1) deconstruct the data packet into its separate components, 2) perform signal reconstruction on these components.

6.2.1.1 Data Packet

The data packet sent by the STM32F3 contains all eight gels of phase information and a beginning packet with eight sampled values of the trigger voltage. Within the microcontroller, the USB send function (usbd_ep_write_packet) transmits a buffer of 65 characters. We have chosen to use 62 characters in case future modifications require identifying the buffer. We then have 75 sets of 62 characters per gel (Fig 6-3). Hence, the total number of 8-bit samples *per gel* is 4650.

Data Packet
[0]
[1 : 8]
[9 : 61]
[62 : 471]
[472 : 936]
[936 : 1401]
[1402 : 1866]
[18662 : 23311]
[23312 : 27961]
[27962 : 32611]
[32612 : 37261]
Zero
Trigger-Samples
Zeros
Gel 0
Gel 1
Gel 2
Gel 3
Gel 4
Gel 5
Gel 6
Gel 7
Amplitude
Samples
Gel 1[0],Gel 2[0],Gel 3[0],Gel 4[0],Gel 5[0],Gel 6[0],Gel 7[0]
Gel 1[1],Gel 2[1],Gel 3[1],Gel 4[1],Gel 5[1],Gel 6[1],Gel 7[1]
Gel 1[2],Gel 2[2],Gel 3[2],Gel 4[2] ...
Index "0" of all gels.
Index "1" of all gels.
Index "2" of all gels.

Figure 6-3

After the data packet is decoded accordingly, we then re-order the gels to obtain the 24MHz-sampled waveform. Each gel packet is of length 4650. To construct the waveform, we first take all index "0"s of each gel (gel0[0], gel1[0],…gel7[0]). These samples become the first 8 samples of the 24MHz-sampled signal. To obtain the next eight samples we use gel0[1], gel1[1],…gel7[1], and to acquire the next eight, we use gel0[2], gel1[2],…gel7[2], and so on and so forth to gel0[4649], gel1[4649],…gel7[4649].

In the re-ordered sine wave in Fig 6-3, we see how sections from each gel (labeled by its color) are then re-ordered. The first rainbow set of samples is index 0 of the eight gels from the data packet. The next rainbow set is index 1 of the eight gels from the data packet. The next rainbow set is index 2 of the eight gels from the data packet. This continues until index 4649 of all eight gels.

6.2.1.1.1 Fast Fourier Transform

To obtain the frequency decomposition of the input signal, we employ a Fast Fourier Transform (FFT) to attain the values of relevant frequency coefficients.

MATLAB's fft(X,n) command returns the n-point Discrete Fourier Transform of the input vector X. A word of caution – this process is inherently a circular convolution. If performed carelessly, the input signal is presumed infinite in time and repetitious in waveform, which may lead to distortion.

One point of interest is that this FFT function must later be implemented in Python.

6.2.1.1.2 Sinc Interpolation

Our primary reconstruction method derives from the Nyquist-Shannon Sampling Theorem:

"Suppose that $\boldsymbol{X(f)}$, the Fourier transform of $\boldsymbol{x(t)}$, is identically zero for all $\boldsymbol{|f| > W}$ and has no singularities at $\boldsymbol{|f| = W}$. Then $\boldsymbol{x(t)}$ has exactly the representation:

$$\boldsymbol{x(t) = \sum_{n=-\infty}^{\infty} x\left(\frac{n}{2W}\right) \frac{\sin 2\pi W\left(t - \frac{n}{2W}\right)}{2\pi W\left(t - \frac{n}{2W}\right)}}$$

Equation 2

That is, samples of $\boldsymbol{x(t)}$ at points $\boldsymbol{t = \frac{n}{2W}}$ (samples that are $\boldsymbol{\frac{1}{2W}}$ apart in time) completely determine $\boldsymbol{x(t)}$ at all points in time" [10].

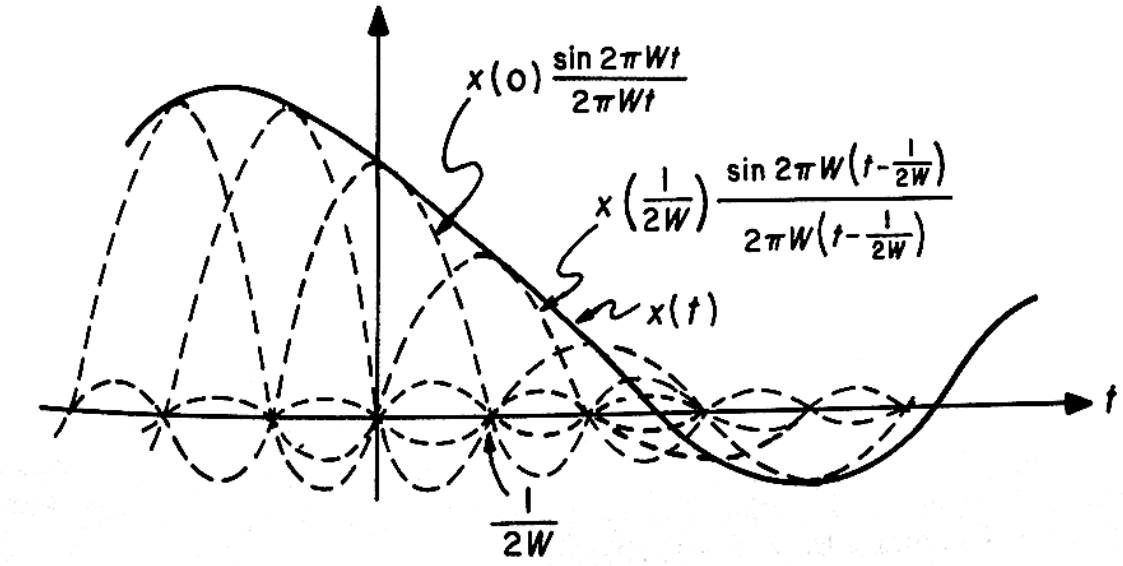


Figure 6-4, William Siebert, *Circuits, Signals, and Systems*. Cambridge, MA, USA: The MIT Press, 1986. [10]

One should recognize that this sinc-interpolation method (Fig 6-4) is simply the time-domain representation of filtering the sampled input signal with an ideal frequency domain brick-wall (pink dashed box in Fig 6-1).

From this, it is possible to reconstruct a continuous signal from a discretized signal. However, the interpolation summation limits would then have to be infinite. This is too time-consuming, so we must be cautious with the selection of our summation limits.

The sinc-interpolation function used, $\mathbf{y} = \mathbf{sinc\_interp}(\mathbf{x}, \mathbf{s}, \mathbf{u})$, is credited to Ted Pavlic [11]. A Python version exists as well [12]. In this interpolation function, the inputs are: "**s**" – a vector of the input signal length spaced 1/24MHz apart, and "**u**" – a vector of the desired up-sampled time steps (for example a vector of 1/48MHz spaced apart time steps). The input vector, **x**, is then remapped to the higher frequency time samples while simultaneously sinc-interpolated. Connecting back to Equation 2, the original input time step vector controls $\boldsymbol{n}$ and the desired up-sampled time step vector controls $\boldsymbol{t}$. $\boldsymbol{x}\left(\frac{\boldsymbol{n}}{\boldsymbol{2W}}\right)$ is the input vector **x**, and our output **y** is $\boldsymbol{x}(\boldsymbol{t})$.

This function takes considerably less time than attempting to brute force Equation 2.

Important to note, this function is a finite element method procedure rather than a circular convolution procedure. A brick wall filter is often implemented through the relationship of convolution and multiplication in the frequency domain.

$$\boldsymbol{x} * \boldsymbol{y} = \boldsymbol{F^{-1}}[\boldsymbol{F}(\boldsymbol{x}) \cdot \boldsymbol{F}(\boldsymbol{y})]$$

Equation 3

In a typical circular convolution approach, we would FFT our input and our sinc, producing a Fourier transformation of our signal and a brick wall. We would then multiply the two and inverse FFT (IFFT) the product.

In one sense, this is easier to visualize. However, these styles of FFT and IFFT procedures are typically performed in MATLAB as circular convolutions that assume the signal is continually periodic in time. Regrettably, our signal is finite. Should we attempt to interpolate with a circular convolution, we would witness the appearance of slow time-domain waveforms that do not physically exist in our signal - Moiré patterns (explained in later section).

Furthermore, an FIR filter with a sinc kernel does not yield as high of an SNR as finite element sinc-interpolation. A limit exists for the FIR sinc kernel window length – some initial values of the convolution between the input and the sinc-filter must be discarded. If the kernel size is too large, we must reject the majority of our convolution. However, decreasing the sinc kernel length creates a less precise and less smooth output signal due to the fact that unlike interpolation, we do not create a higher sample point density with an FIR filter. While the FIR method is faster, it still exhibits Moiré patterns in our frequency range, even with a large kernel window. Ultimately the finite time method was selected.

6.2.1.2 Signal Reconstruction Overview

The main procedure in MATLAB is outlined in Fig 6-5. The reconstruction depends on the input waveform shape. Should a user select a sine wave input, the code will specifically hunt for the highest frequency component and interpolate with a frequency window. Should the user select a square or triangle wave, the user should not opt for any pre-conditioning filter (so as to avoid accidental deletion of higher order components).

The reconstruction begins by first averaging the trigger values. The reconstructed signal (the main waveform of Fig 6-3) is then sinc-interpolated by a factor of two (to reduce noise in the sampled signal so that the subsequent FFT can locate the center frequency). Depending on the input signal, the user can either select to window the frequency around the center frequency by $1/8^{th}$ of the Nyquist frequency (limit the band around the center frequency to 1.5MHz) or to allow the interpolated signal to be processed without additional frequency windowing. After this, the signal is again sinc-interpolated by a factor of 10 (to smooth out any additional Moiré patterns) and the waveform is displayed with the original trigger value.

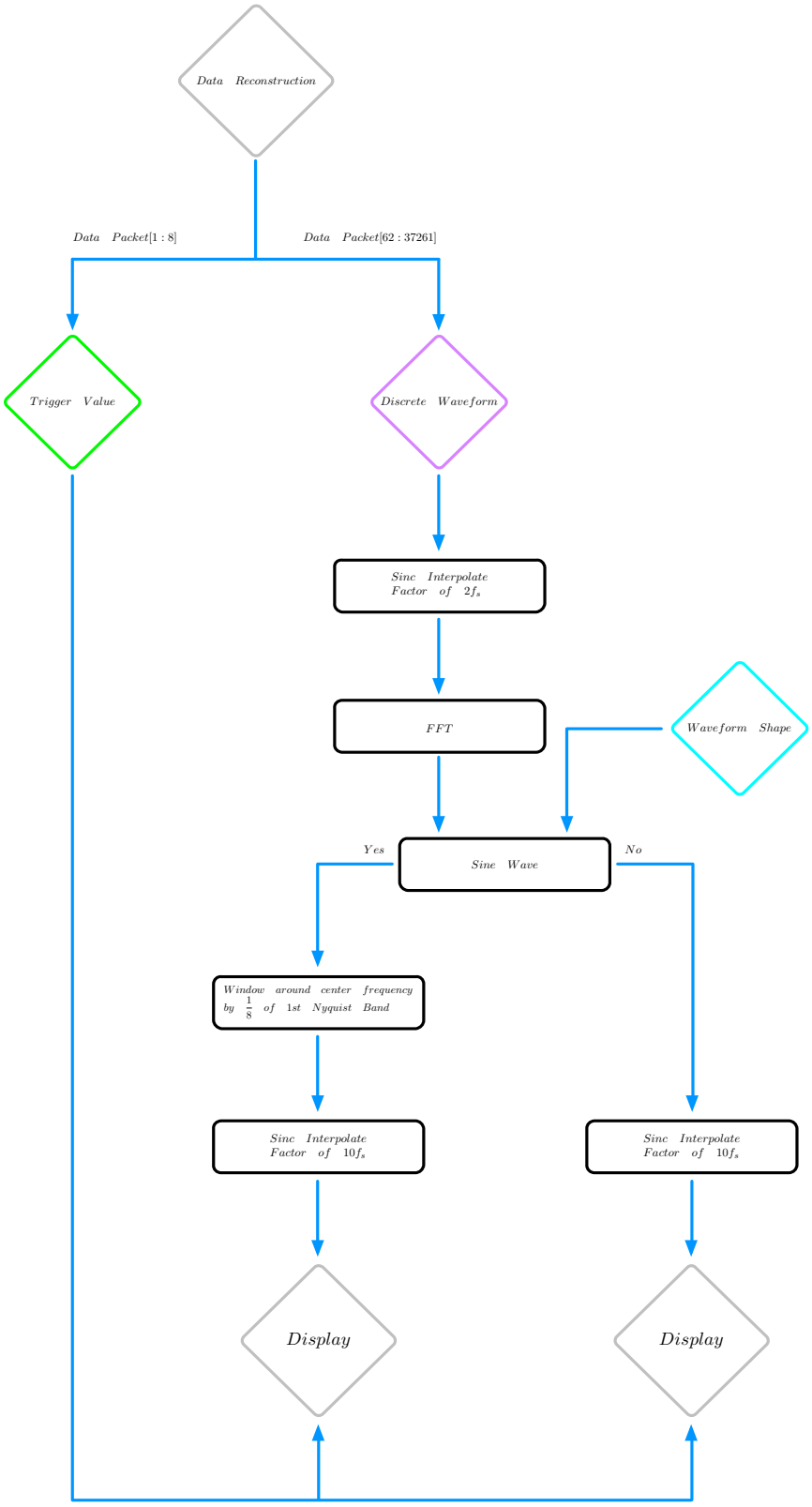

Data Reconstruction
Data Packet[1 : 8]
Data Packet[62 : 37261]
Trigger Value
Discrete Waveform
Sinc Interpolate Factor of $2f_s$
FFT
Waveform Shape
Yes
Sine Wave
No
Window around center frequency by $\frac{1}{8}$ of 1st Nyquist Band
Sinc Interpolate Factor of $10f_s$
Sinc Interpolate Factor of $10f_s$
Display
Display

Figure 6-5

6.2.1.3 Signal Reconstruction Issues

There were a few unexpected issues with the signal reconstruction mainly due to Moiré patterns. Particular waveforms, such as square waves and triangles, present obstacles due to the higher frequency components aliasing into lower bands.

6.2.1.3.1 Moiré Patterns

During the digitization process, a pseudo-aliasing phenomenon called the Moiré effect occasionally occurs [13]. This effect emerges within the first Nyquist band and appears as a slow envelope atop the input signal (Fig 6-6).

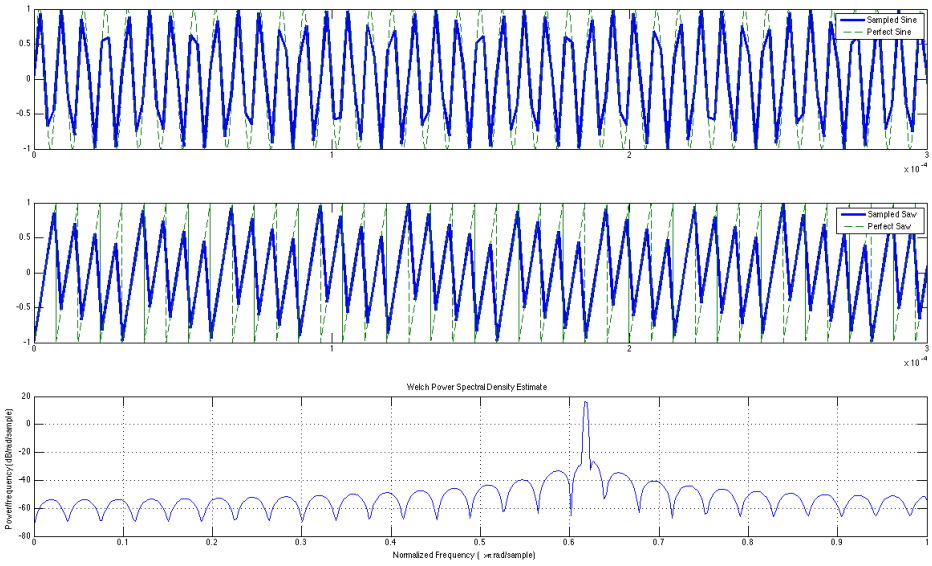


Figure 6-6

The waveforms in Fig 6-6 are MATLAB simulations of an input signal captured within first Nyquist. The original input signal is green and the sampled digitized signal is blue. The digitized signal is sampled at a rate of $\frac{f_{signal}}{2\times(1.618)}$ for both the sine and sawtooth waveforms. The FFT of the

digitized sine waveform (bottom graph) displays the highest amplitude coefficient at the correct normalized frequency, $\frac{\mathbf{1}}{(\mathbf{1.618})} = \mathbf{0.618}$ of $\frac{\boldsymbol{f_{signal}}}{\mathbf{2}}$ with rounded amplitudes for the other coefficients, but the time-domain digitized signal has a slow envelope.

This envelope and FFT shape is not due to external noise or aliasing, rather due to digitization and a lack of sinc-interpolation. The actual input signal is windowed and finite in the time-domain. However, as discussed earlier, when we perform an FFT through circular convolution, we imagine the signal as infinite and repetitious. This allows the Moiré patterns to show, for if the signals were infinite and repetitious, the "slow-envelope" signal energy would average to zero. This explains the lack of Fourier coefficients for the envelope in the frequency domain.

6.2.1.4 Signal Reconstruction Later Improvements

In the future, the reconstruction could also be optimized for speed and refresh rate. Memory allocation between the microcontroller and user laptop could also be improved. A better input filter could be designed with proper impedance matching.

7 Data and Specs

7.1 Overall Performance (Input Stage + ARM)

In total, we have an oscilloscope with 24MSPS and a gain pre-amp with various bandwidths and a 2.5 V/µs (very, very conservative estimate) slew rate.

7.1.1 Normal Mode

7.1.1.1 Signal Path Description

In normal mode, we return a stream of samples from the ADC. No trigger is required. In our software, a small interpolation by a factor of ten is performed. This reduces most Moiré effects. Although one can theoretically capture waveforms up to 1.5MHz, we advise against this. Tektronix does not use the full bandwidth - only $\frac{f_s}{2.5}$ to allow for some sinc interpolation margin [14]. Thus, we suggest operating in normal mode only up to 1.2MHz.

Keep in mind the number of coefficients necessary to view a waveform. Should the user attempt interpolating a square or triangle wave in this mode, do not employ frequencies above or near 500kHz.

In the following figures, a characterization of real-time mode was performed. Sinusoids of frequencies 100kHz – 1.4MHz were captured with 1Vpp amplitudes (all below 2.5 V/µs slew-rate). As we can see, the 1.4MHz sinusoid displays artifacts. In each figure, the top is the raw data, middle is sinc-interpolated data, bottom is FFT of sinc-interpolated data.

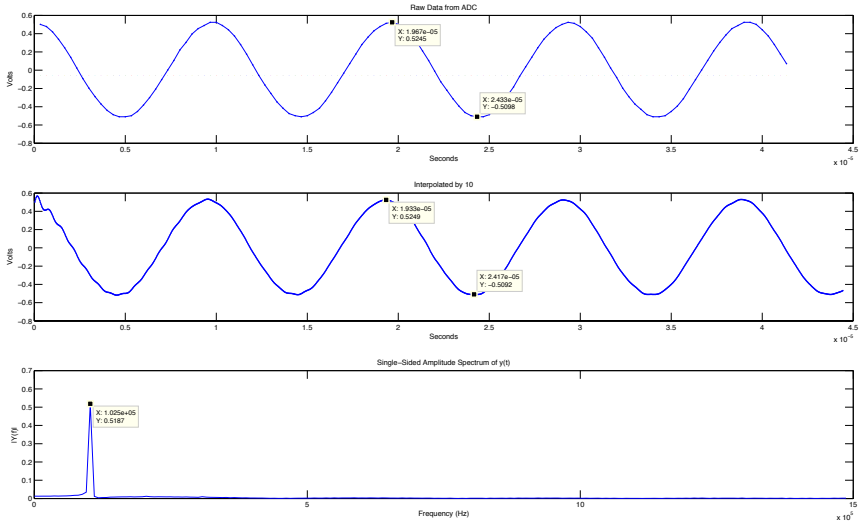


Figure 7-1. 1Vpp sine wave at 100kHz. SNR raw data = 33.45dB. SNR interpolated = 35.07dB.

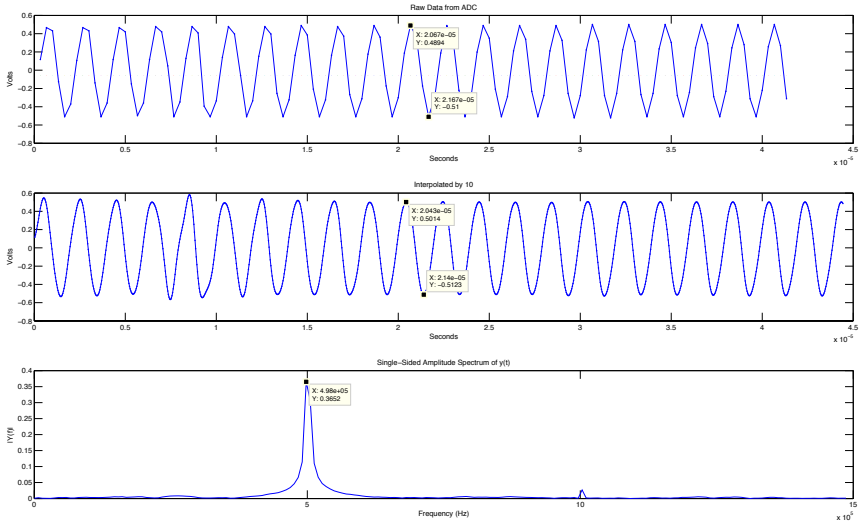


Figure 7-2. 1Vpp sine wave at 500kHz. SNR raw data = 24.61dB. SNR interpolated = 33.34dB.

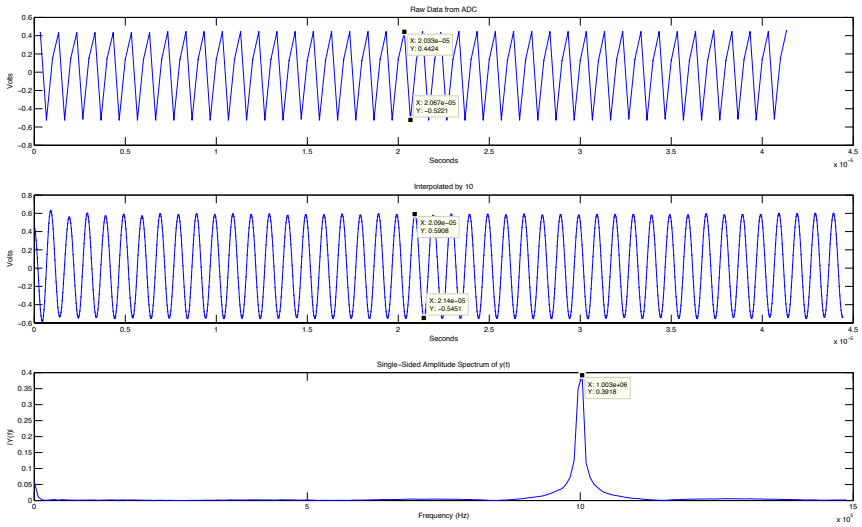


Figure 7-3. 1Vpp sine wave at 1MHz. SNR raw data = 17.72dB. SNR interpolated = 42.69dB.

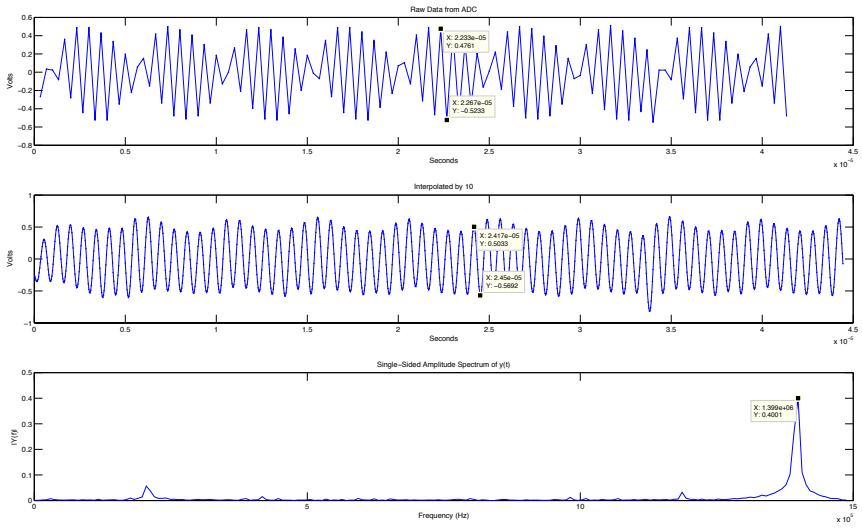


Figure 7-4. 1Vpp sine wave at 1.4MHz. SNR raw data = 14.19dB. SNR interpolated = 14.68dB.

For lower frequency waveforms, the signal captured by the ADC already contains a sufficient sample/point density that there is no significant change in SNR. As we approach higher frequencies (and the point density decreases) we still see a factor of ~0.5dB improvement. This reduces the

slow envelope in Fig 7-3, but as previously stated, we suggest capturing signals only to 1.2MHz.

The following set of figures is for triangle and square waves captured in normal mode. Again, in each figure: top is the raw data, middle is sinc-interpolated data, and bottom is FFT of sinc-interpolated data.

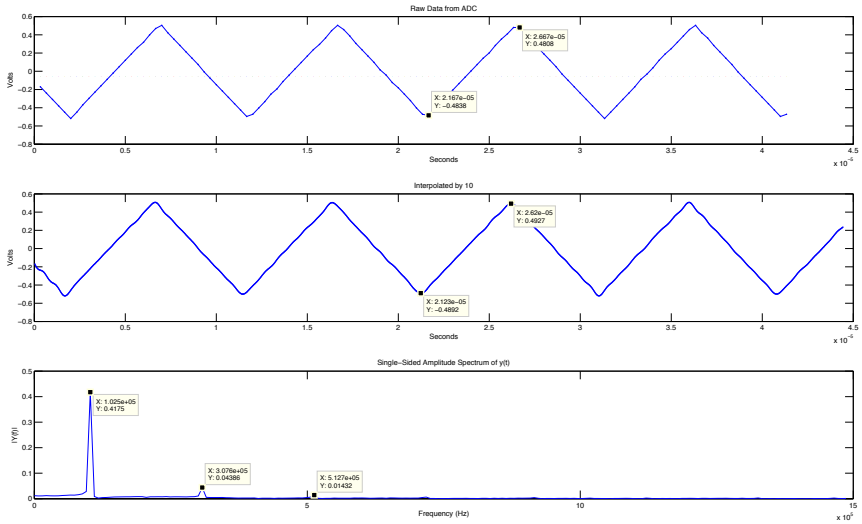


Figure 7-5. 100kHz triangle 1Vpp.

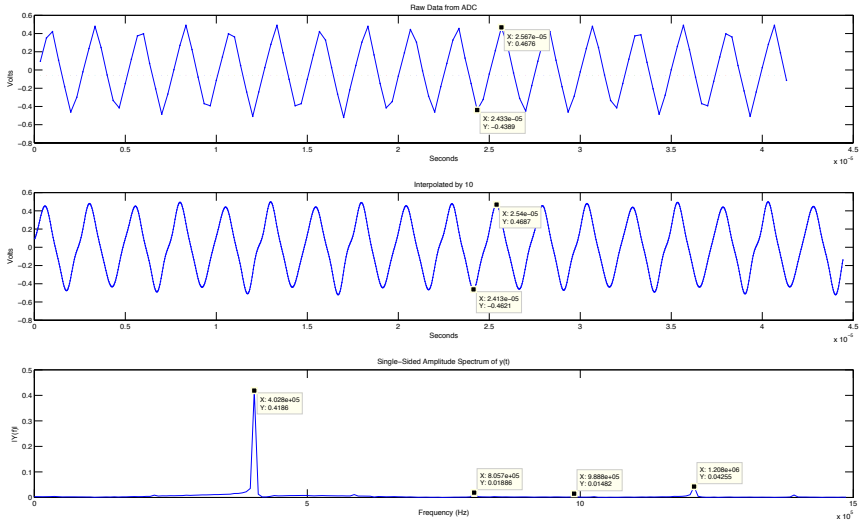


Figure 7-6. 500kHz triangle 1Vpp.

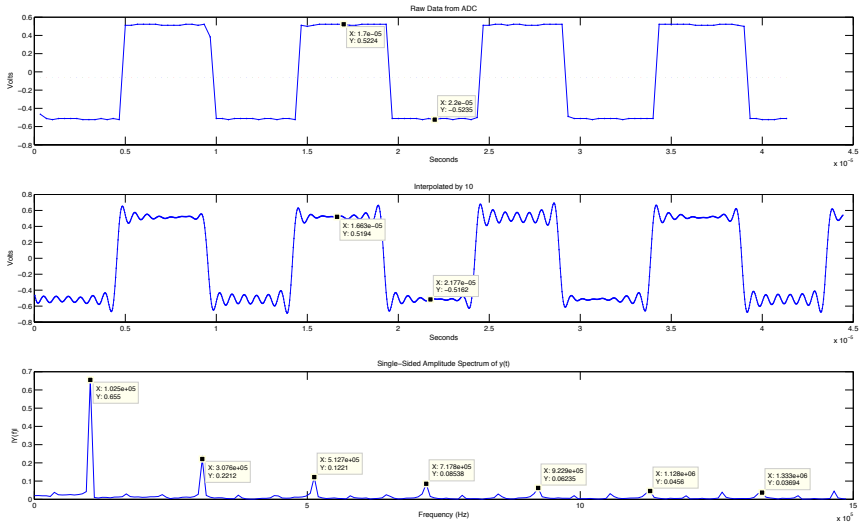


Figure 7-7. 100kHz square 1Vpp.

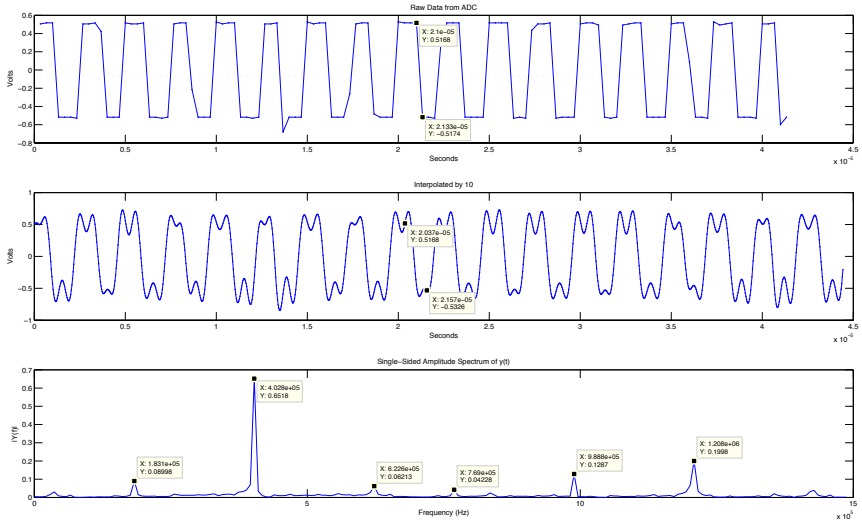


Figure 7-8. 500kHz square 1Vpp.

7.1.1.2 Issues

For other shapes, such as triangles and squares, they appear for, at max, 1.5MHz/3. Even so, Gibbs phenomenon prevents a square wave from being completely smooth, so some ringing will occur.

7.1.2 Sequential Mode

7.1.2.1 Signal Path Description

For sequential mode, we are theoretically able to sample waveforms up to 12MHz. However, due to our pre-amplifier, we are limited in both bandwidth and slew rate. If one choses to bypass the pre-amp, we are still able to sample at 24MSPS. However, again we suggest in practice only capturing frequencies up to $\frac{f_S}{2.5}$, or 9.6MHz [14].

Data is processed as described in Section 6.2.1.2. Fig 7-9 displays each gel captured by the 3MSPS ADC. The input waveform is a 5MHz sine wave with 400mVpp.

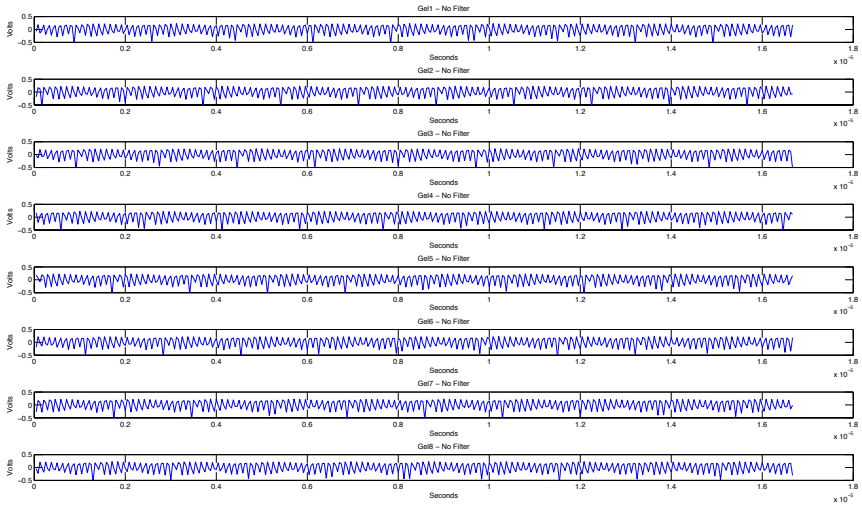


Figure 7-9. Eight gels of 5MHz unfiltered sine wave with 400mVpp.

The recombined gels are then passed through a fixed element sinc interpolation of two, and then through an FFT (Fig 7-11). Afterwards, the user has the optional choice of sending the data through an additional sinc/brick wall filter centered around the highest frequency coefficient with a passable bandwidth of 0.75MHz on either side of this center frequency (Fig 7-10). This optional brick wall is specifically designed to decrease noise for sine waveforms. If no sinc/brick wall filtering is selected, the FFT data is passed through with no alteration (Fig 7-11), and afterwards sinc-interpolated once more by a factor of 10 (Fig 7-12, middle two graphs). The 2, then 10 times interpolated FFT is shown as the top graph of Fig 7-12. The top middle graph is the 2, then 10 times interpolated signal but only the first quarter of the waveform. The graph below that is the complete interpolated waveform. The bottom is the original waveform without any interpolation.

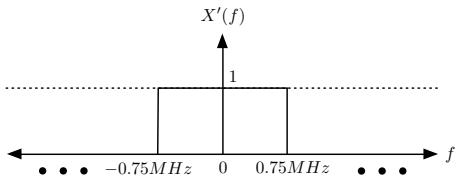


Figure 7-10. Brick wall filter moves depending on center frequency.

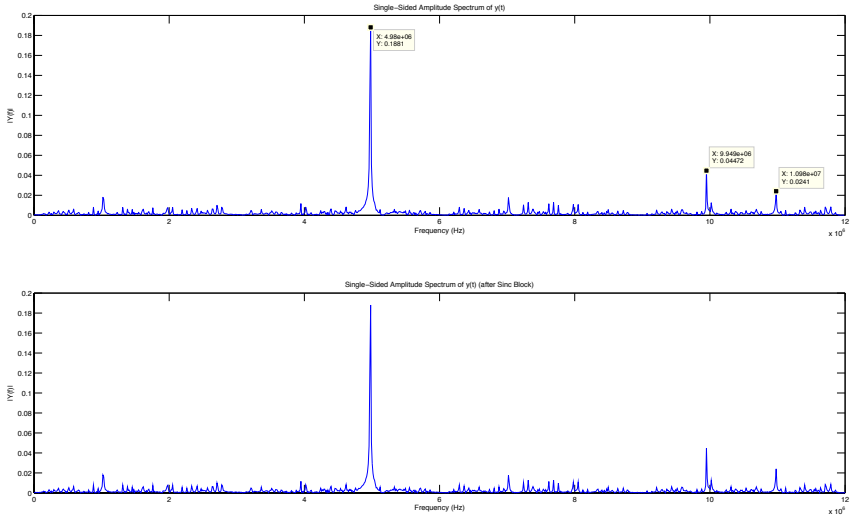


Figure 7-11. 5MHz-unfiltered sine wave with 400mVpp. FFT after 2 times interpolation.

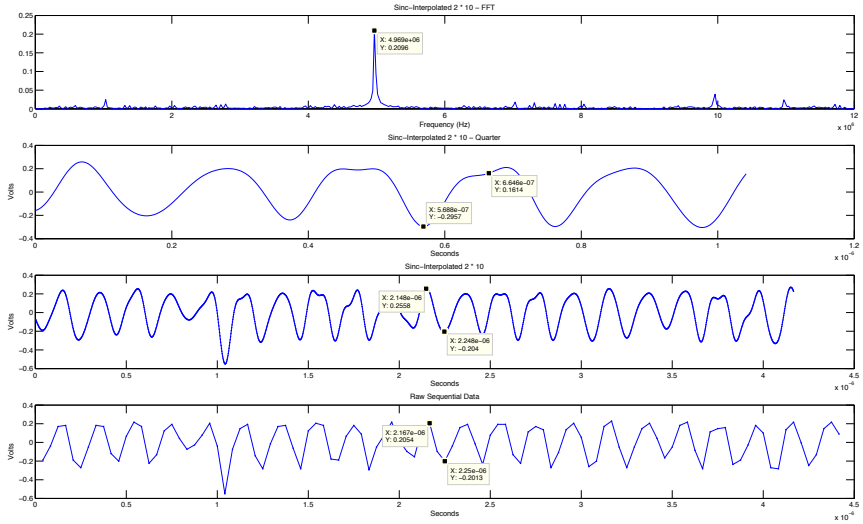


Figure 7-12. 5MHz-unfiltered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -0.55dB. Sinc-Interpolated SNR = 9.9dB.

If sinc/brick wall filtering is selected, the previous interpolated by 2 FFT data is filtered as shown in Fig 7-13. This filtered data is then 10 times interpolated and then reconstructed in Fig 7-14.

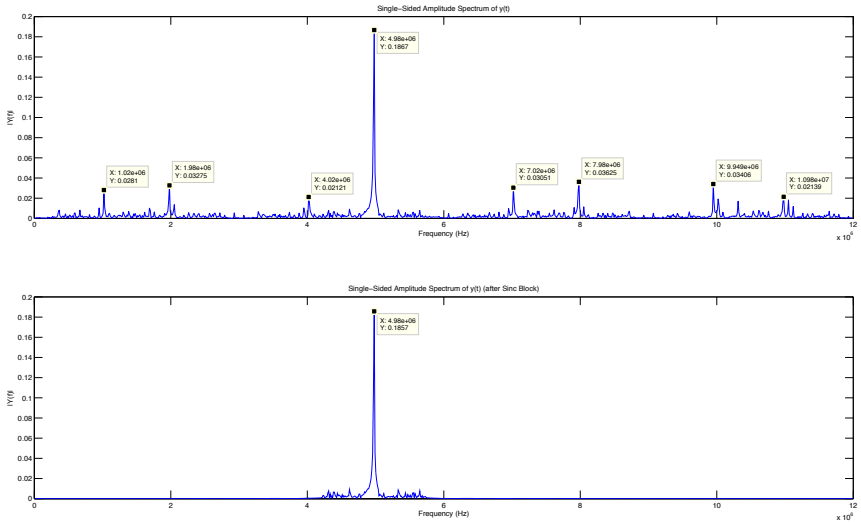


Figure 7-13. 5MHz-unfiltered sine wave with 400mVpp. FFT after 2 times interpolation. Bottom FFT is after sinc-filtering with brick wall at center frequency (frequency of largest coefficient) and passable bandwidth of .75MHz on either side of the center frequency.

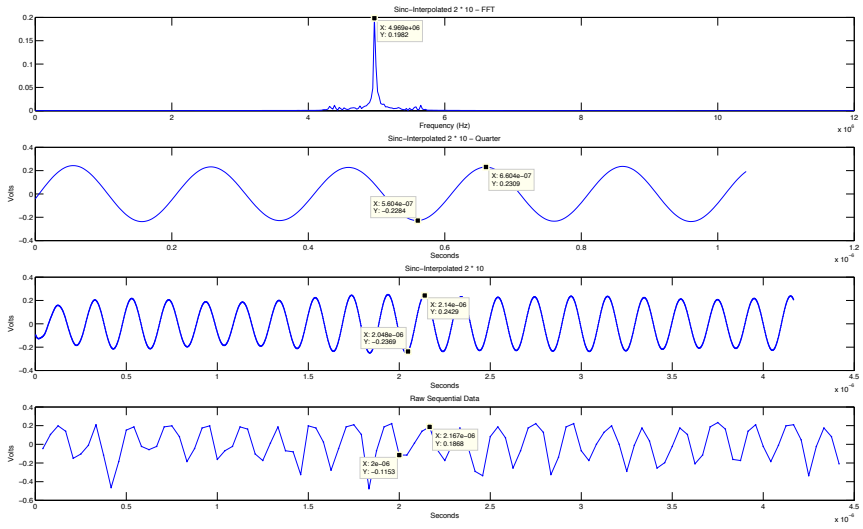


Figure 7-14. 5MHz Filtered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 1.94dB. Sinc-Interpolated SNR = 66.02dB.

Below are selected waveforms captured first without then with the sinc/brick wall filter. Note: signals 3MHz and below produce an unfiltered SNR of at least 20dB. Signals from 3MHz - 5MHz produce an unfiltered SNR of at least ~10dB. Afterwards, 2 then 10 interpolation increases the raw SNR by at least ~7dB. Some distortions in amplitude exist for the unfiltered signals. However, if passed through the brick wall, the original sine wave and amplitude are mostly preserved. Ultimately we suggest this: if the user is specifically searching for a change of sine amplitude to test a filter or amplifier, use the brick wall setting. The differential change in a filtered sine can still be of value. If the user is testing a step response or integration, do not use the brick wall but use as slow of a square or triangle wave as possible.

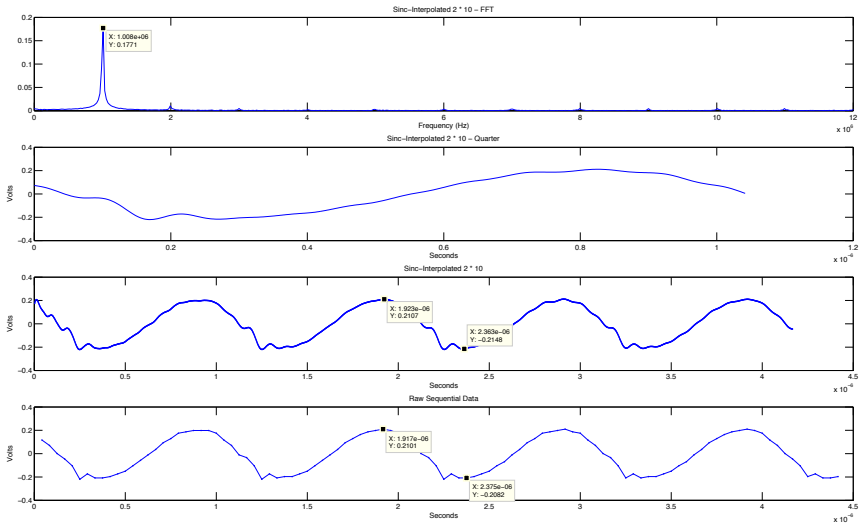


Figure 7-15. 1MHz Unfiltered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 11.07dB. Sinc-Interpolated SNR = 21.09dB. Blips are from trigger noise.

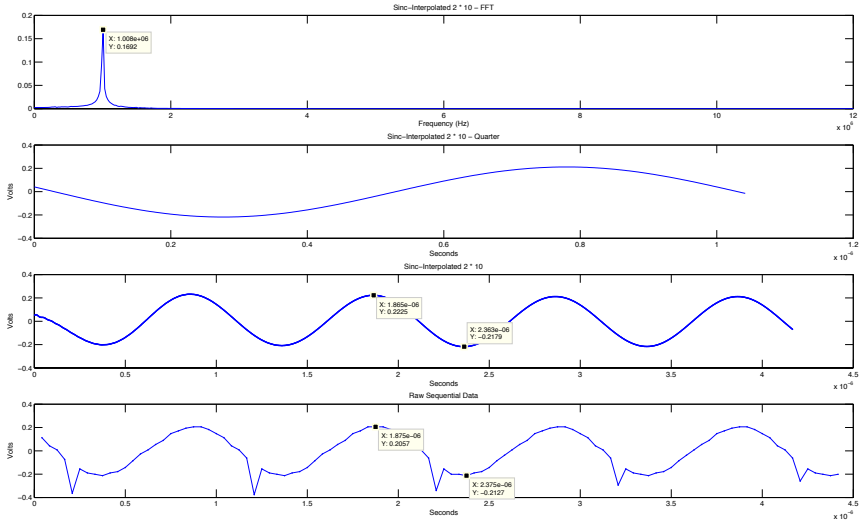


Figure 7-16. 1MHz Filtered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 11.12dB. Sinc-Interpolated SNR = 65.59dB. Blips are from trigger noise.

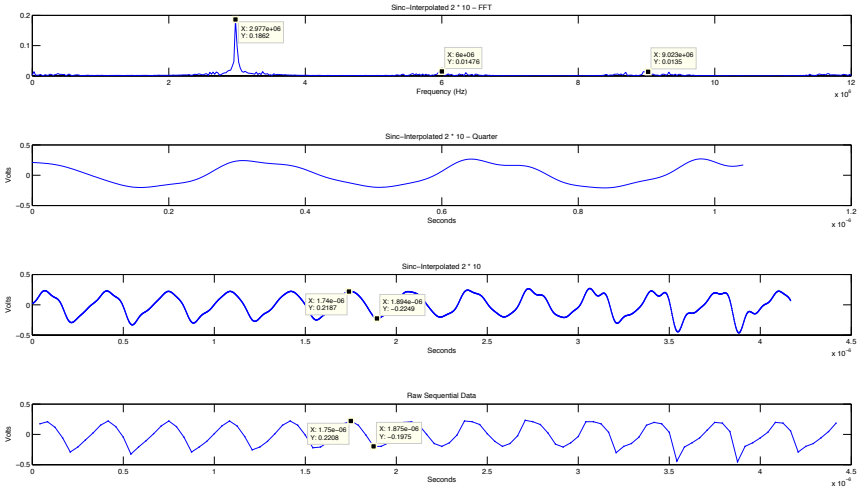


Figure 7-17. 3MHz Unfiltered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 3.55dB. Sinc-Interpolated SNR = 24.07dB.

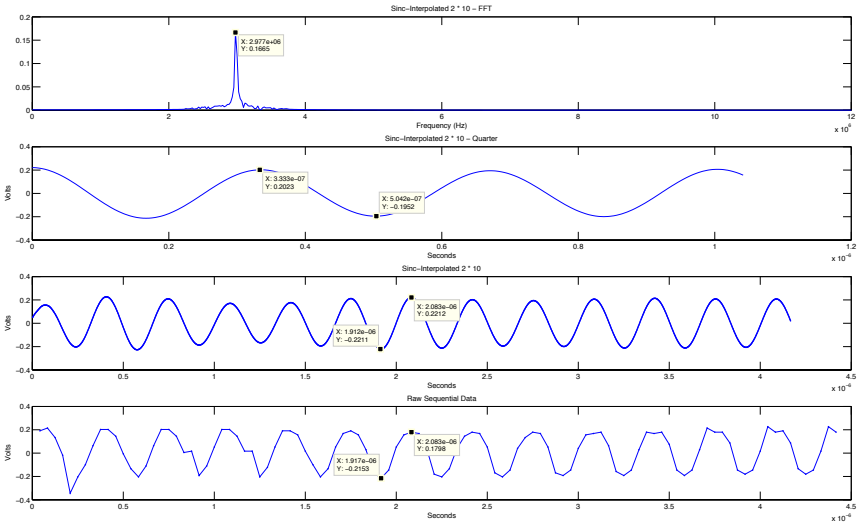


Figure 7-18. 3MHz Filtered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 8.11dB. Sinc-Interpolated SNR = 72.88dB.

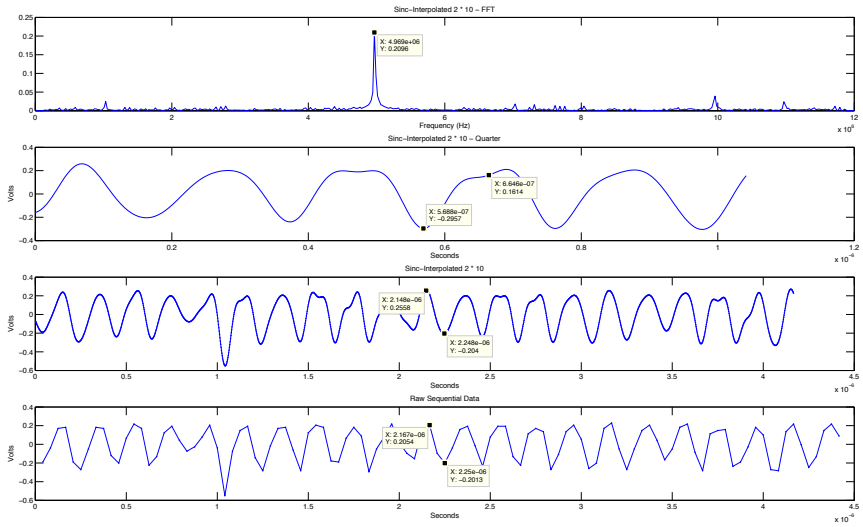


Figure 7-19. 5MHz-unfiltered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -0.55dB. Sinc-Interpolated SNR = 9.9dB.

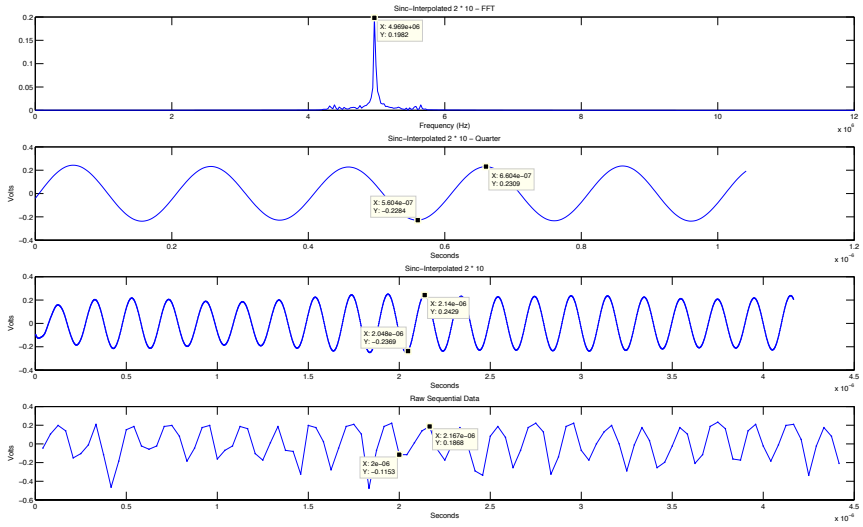


Figure 7-20. 5MHz Filtered sine 400mVpp reconstructed and FFT. Pre-Interpolated Data SNR = 1.94dB. Sinc-Interpolated SNR = 66.02dB.

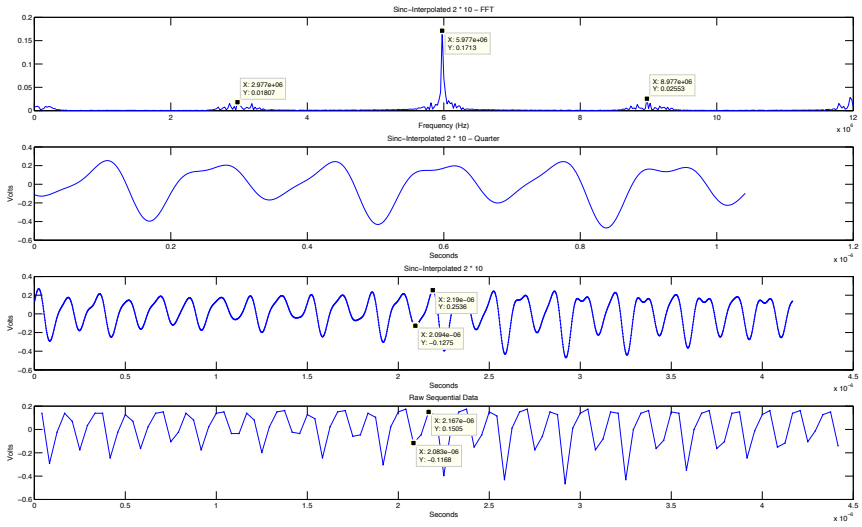


Figure 7-21. 6MHz unfiltered sine 300mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -4.08dB. Sinc-Interpolated SNR = 7.08dB.

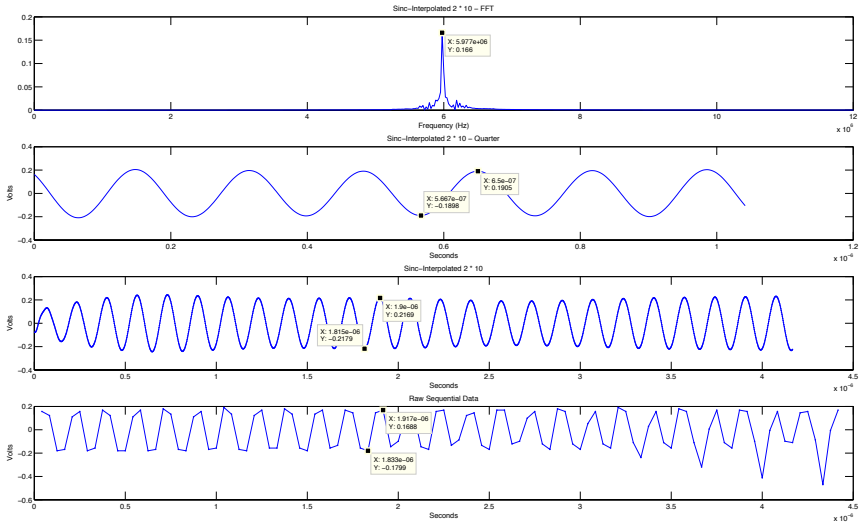


Figure 7-22. 6MHz Filtered sine 300mVpp reconstructed and FFT. Pre-Interpolated Data SNR =1.64dB. Sinc-Interpolated SNR = 87.16dB.

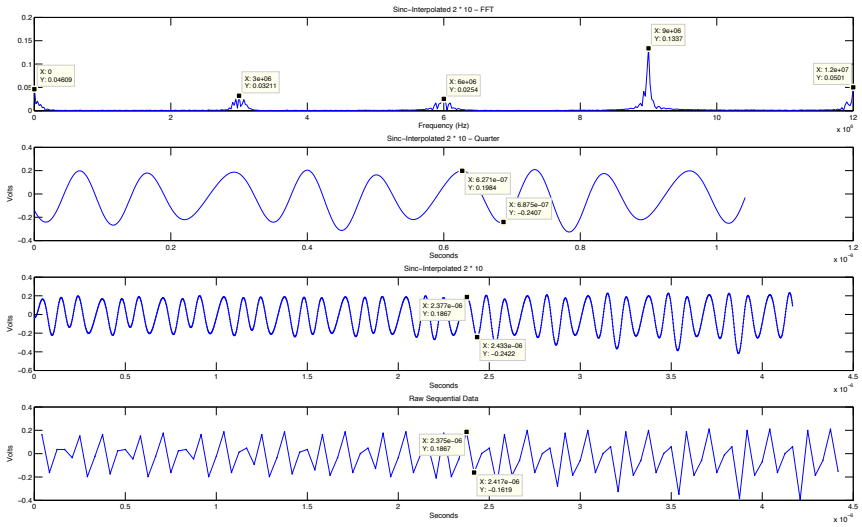


Figure 7-23. 9MHz unfiltered sine 240mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -0.30dB. Sinc-Interpolated SNR = 6.67dB.

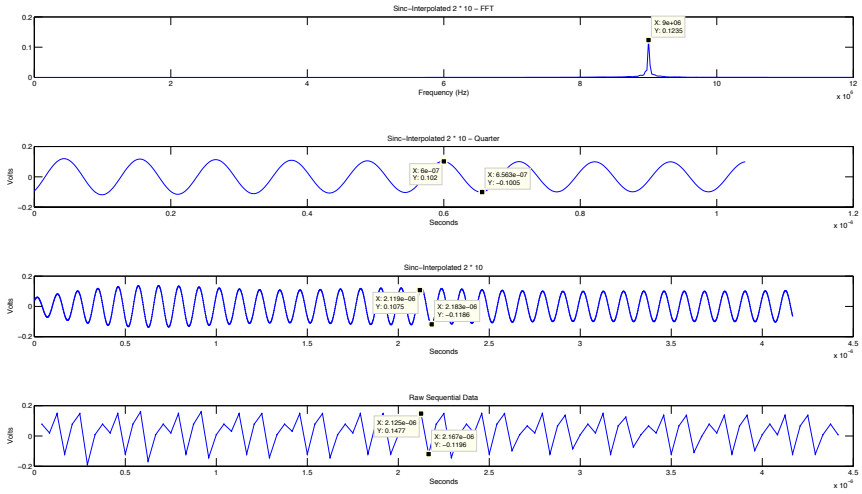


Figure 7-24. 9MHz Filtered sine 240mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -0.71dB. Sinc-Interpolated SNR = 78.92dB.

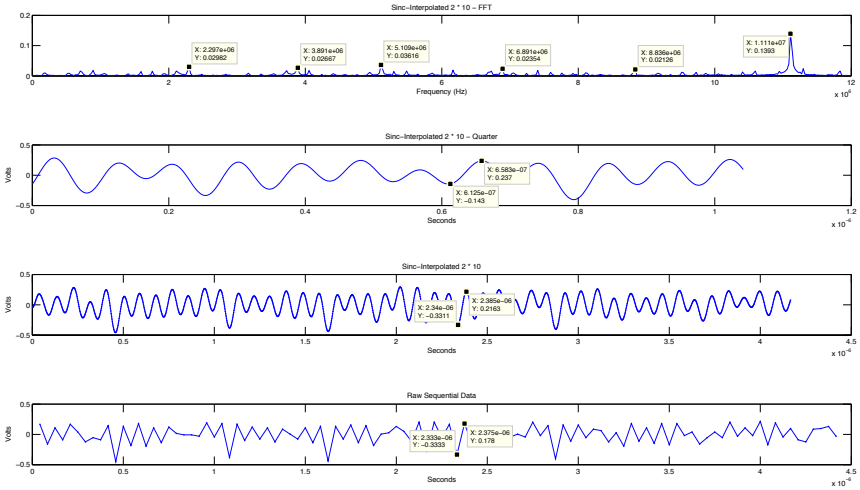


Figure 7-25. 11MHz unfiltered sine 240mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -2.09dB. Sinc-Interpolated SNR = 5.31dB.

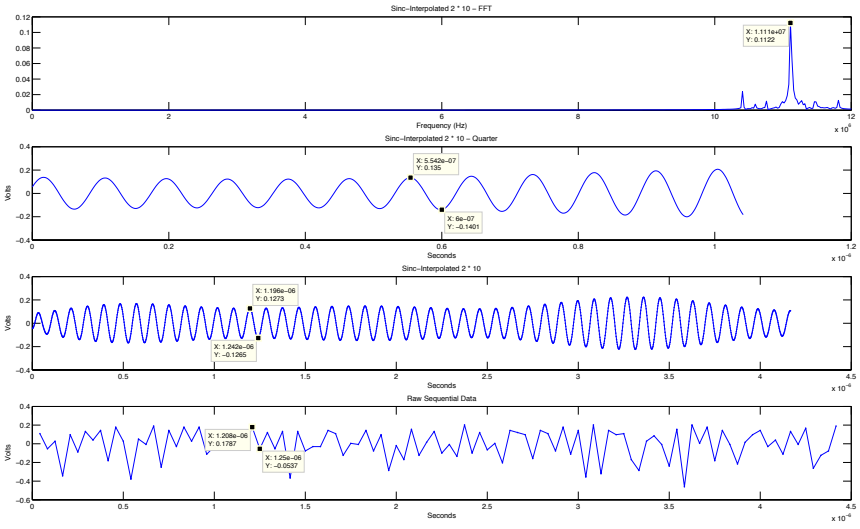


Figure 7-26. 11MHz Filtered sine 240mVpp reconstructed and FFT. Pre-Interpolated Data SNR = -0.934dB. Sinc-Interpolated SNR = 59.14dB.

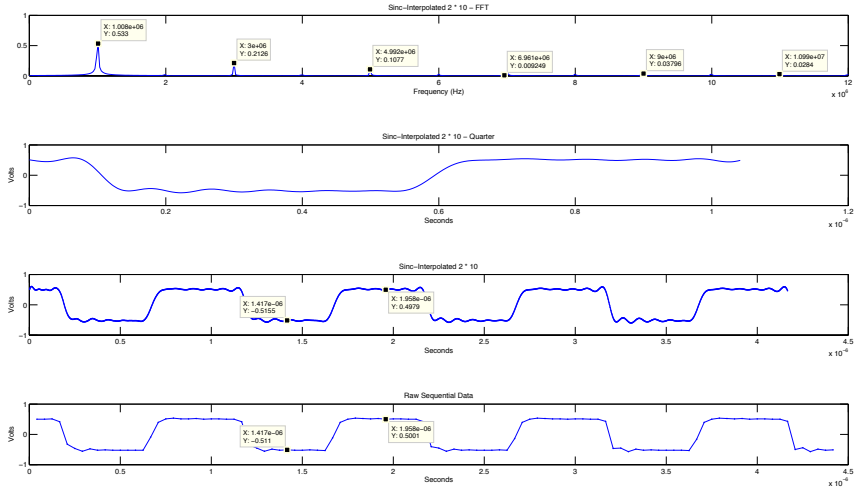


Figure 7-27. 1MHz Square. 1Vpp.

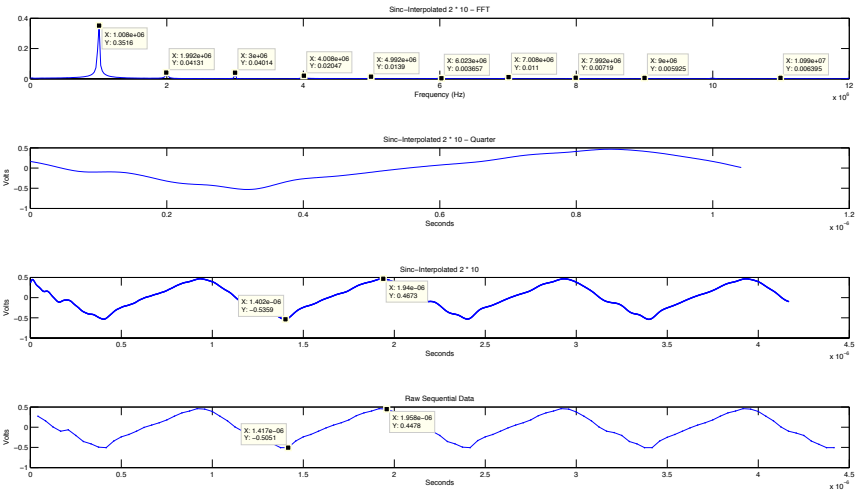


Figure 7-28. 1MHz Triangle. 1Vpp.

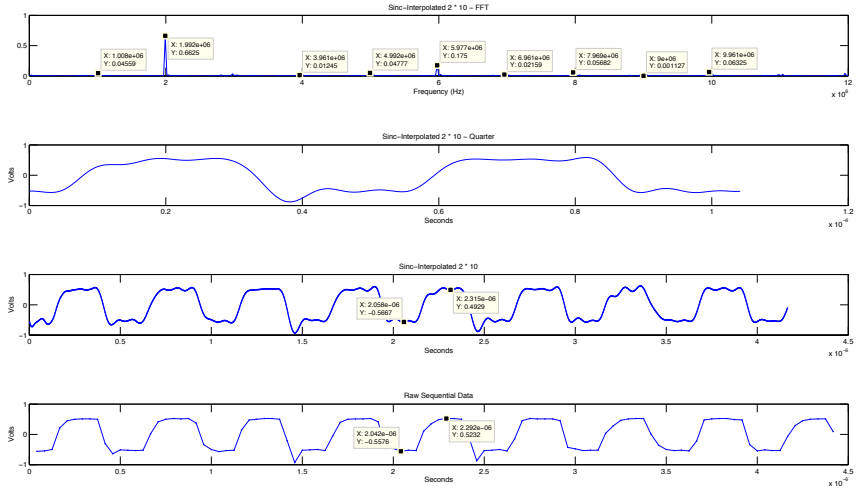


Figure 7-29. 2MHz Square. 1Vpp.

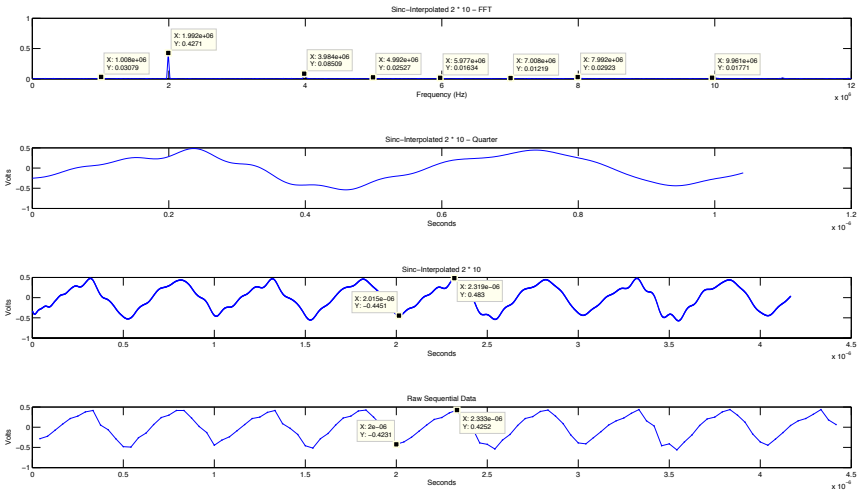


Figure 7-30. 2MHz Triangle. 1Vpp.

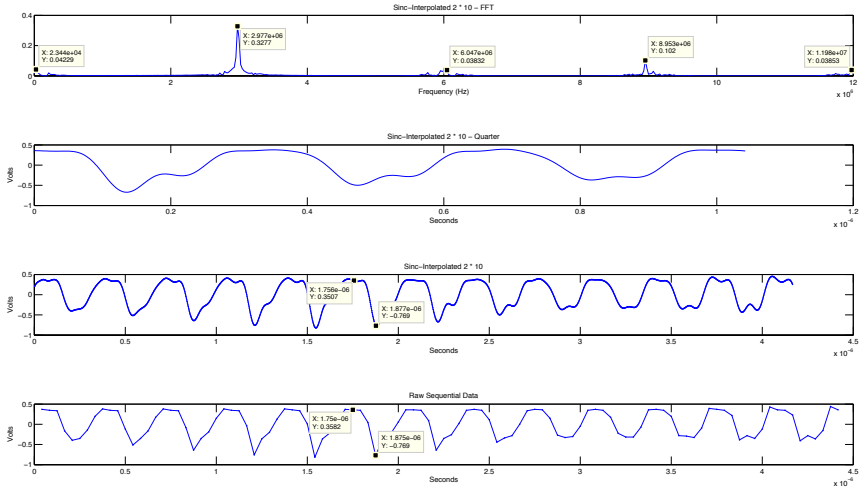


Figure 7-31. 3MHz Square. 660mVpp.

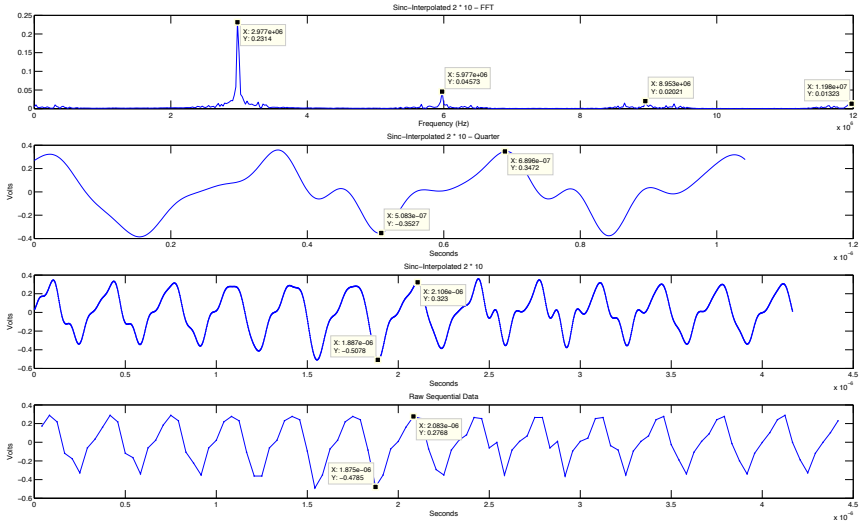


Figure 7-32. 3MHz Triangle. 660mVpp.

7.1.2.2 Issues

The signal becomes muddled without additional sinc\brick wall filtering after 3-4MHz. As seen in the 6MHz to 11MHz figures, there is a slight issue with high frequency signals aliasing into lower bands (as our anti-aliasing filter does not have a perfect cutoff at 12MHz). Additionally, due to the cyclic nature of sequential sampling, frequencies may unintentionally up modulated into higher bands. Moreover, due to our slew rate, at times our sine wave may appear more triangular or square, producing Fourier coefficients in higher, but passable frequencies.

Lastly, the trigger circuit magnetically couples with the signal line, inducing noise as seen in the middle waveform in Fig 7-31. With better layout techniques, this magnetic noise can be reduced. A simple way to reduce the effect of the spike is to increase the signal amplitude if possible. As the amplitude increases, the spike appears overall smaller ratio-metrically. The spike is also about 50ns in duration (20MHz), and therefore somewhat attenuated by the input filter.

Figure 7-33. 1MHz Sine 800mVpp. Middle signal (purple) goes to ADC. Top signal (pink) is trigger. Upon trigger edges, noise is induced on signal to ADC (middle purple) and original signal from sig gen (bottom, yellow).

8 Future Considerations

8.1 Signal Generator

A signal generator would be an excellent add-on. Outlined is a proposed signal generator system.

The signal generator could run perpetually but pause when the user inputs the waveform and the frequency. As seen in Fig 8-1, external GPIO pins select the output waveform (sine, square, triangle) and frequency. Should a pin change, an interrupt is generated on the falling edge and microcontroller outputs the corresponding waveform. If more than one pin is selected, the signal generator ignores the multiple selections and outputs the previous waveform.

The waveform images are stored in a lookup table with the time step stored as a variable. The timer triggers on the time step, essentially controlling output frequency. An ADC with a variable input voltage controls the time step size of the output waveform (Fig 8-2). This potentiometer can be a resistive pot or a digital pot. The user may choose whichever he or she wishes; however a digital pot will need an external communication bus and another subroutine. Note, this interface could also be programmed in software - a physical interface need not exist.

An extended output driver with variable gain must be attached to the on-chip DAC (Fig 8-3). The design is a non-inverting op amp gain with a potentiometer to control gain. We use an op-amp to have the capability of to manipulate DC gain.

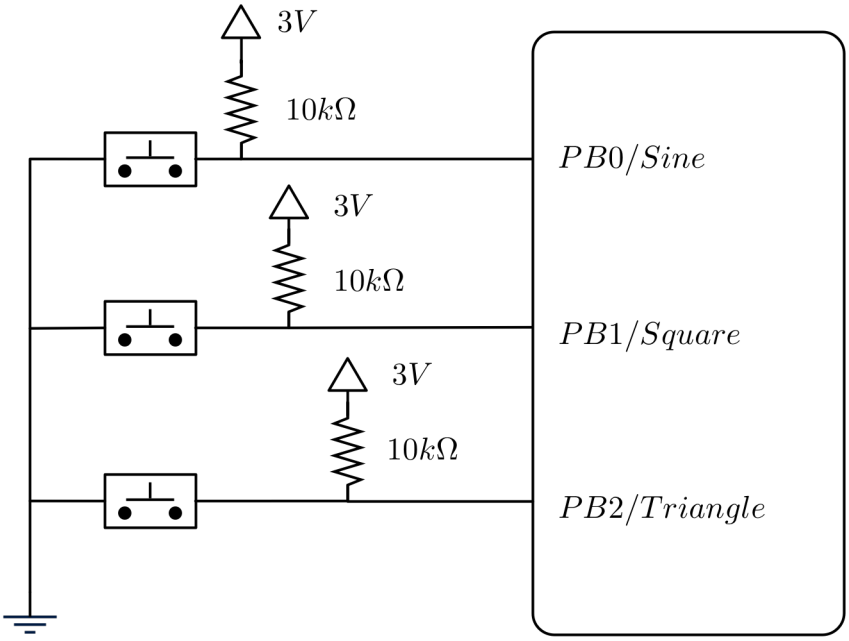


Figure 8-1. Output waveform selected by three buttons. The pull-up resistors for each GPIO sets the default “off” state as high.

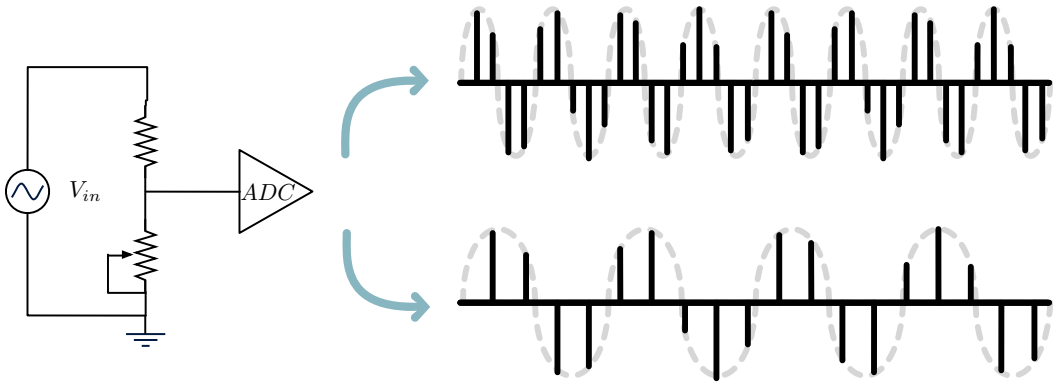
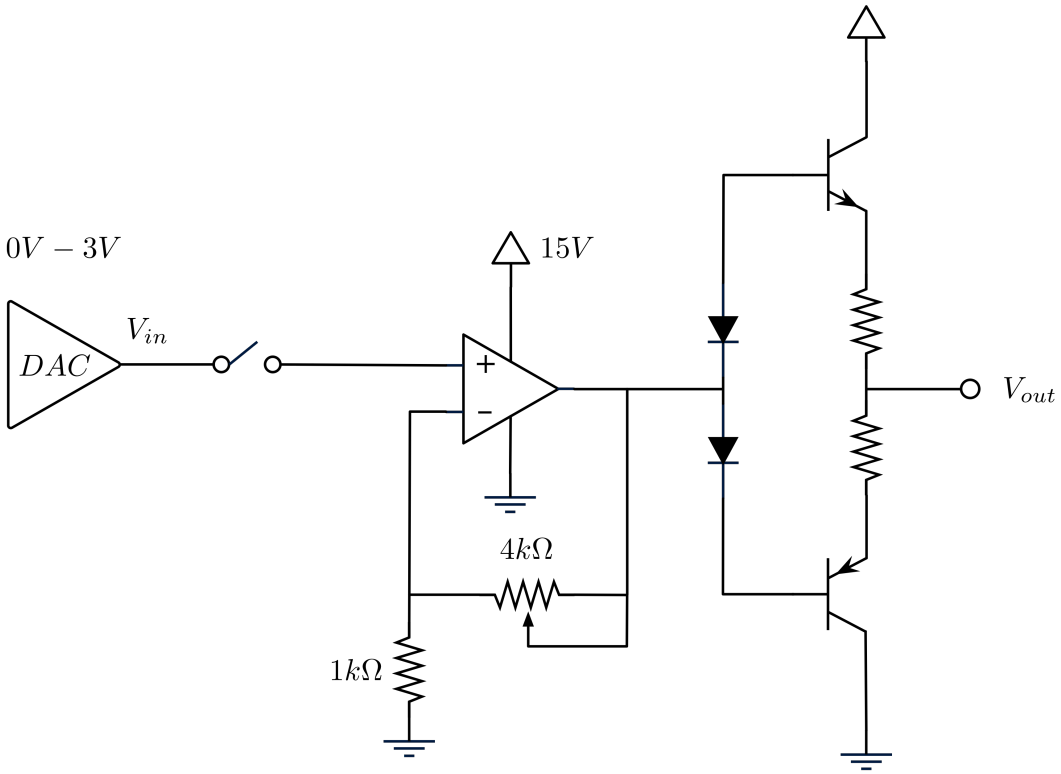


Figure 8-2. ADC voltage controls time step value.



$$V_{out} = \left(1 + \frac{R_{pot}}{R_{bot}}\right) V_{in}$$

$$V_{out} = \left(1 + \frac{R_{pot}}{1k\Omega}\right) V_{in}$$

Figure 8-3. Output driver for signal generator.

8.2 Over-Voltage

Within the STM32F3, there is an optional analog watchdog interrupt. This interrupt indicates that the ADC is experiencing voltage larger than the maximum input rating. A rough feedback loop could be implemented to decrease the input signal until the analog watchdog stops.

8.3 GUI/Automated Software

A practical GUI must also be developed for the users to manually manipulate voltage and time on their screens. This GUI must exist for all common OS platforms. In addition, refreshing the image on screen along with performing the sinc-interpolations and filters must be carefully timed so as to possibly update every 60Hz. A method of virtually connecting one's scope waveform online would be useful, for then a TA could perhaps remote login to a user's laptop and aid in the debugging process.

9 Conclusion

In conclusion, we have created firmware with which a 3MSPS ADC samples sequentially at 24MSPS. MATLAB scripts have been written to decipher the data for real-time sampling and sequential sampling. A Python USB interface exists. Input hardware presents some noise issues, particularly for bandwidth, slew-rate, and trigger-edge coupled noise. However, with better layout and noise-prevention techniques, these could be alleviated. Upon these initial steps, an affordable open-source oscilloscope can be further developed and optimized for mass production.

10 Firmware Appendix

10.1 Firmware for STM32F3 Discovery Development Board

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/dma.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/timer.h>

#include <stdlib.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/usb/usbd.h>
#include <libopencm3/usb/cdc.h>

#define LBLUE GPIOE, GPIO8
#define LRED GPIOE, GPIO9
#define LORANGE GPIOE, GPIO10
#define LGREEN GPIOE, GPIO11
#define LBLUE2 GPIOE, GPIO12
#define LRED2 GPIOE, GPIO13
#define LORANGE2 GPIOE, GPIO14
#define LGREEN2 GPIOE, GPIO15

#define LD4 GPIOE, GPIO8
#define LD3 GPIOE, GPIO9
#define LD5 GPIOE, GPIO10
#define LD7 GPIOE, GPIO11
#define LD9 GPIOE, GPIO12
#define LD10 GPIOE, GPIO13
#define LD8 GPIOE, GPIO14
#define LD6 GPIOE, GPIO15

#define   ADC_COMMON_REGISTERS_BASE_34
    (ADC3_BASE+0x300)
#define ADC34_CCR MMIO32
(ADC_COMMON_REGISTERS_BASE_34 + 0x8)


static volatile uint8_t usb_ready_to_send = 0;
static usbd_device *usb_device;
```

```
#define BUFF          (75)
#define ADC_SAMPLES   (62*BUFF) //8-BIT ADC samples
#define ADC_BYTES     (ADC_SAMPLES*1)

#define GELS          (8)   //4, 8
#define TOT           (16) //3MSPS use 16

static int sample_chunk_per_frame =0;
static int ans = 0;
static uint16_t adc_state = 0;
static uint16_t leds = 0;
static uint8_t trig[62] = {0};
static uint8_t window = GELS;
//static uint32_t old = 0x8000;

// number of phase offsets to run through
static uint8_t  num_frames = GELS; // 1 to 24, each one
is 1/48MHz or 240 if we sample at 100kHz for each
static uint8_t  frame = 0;
// number of samples to take for each frame
static uint16_t num_samples_per_frame = ADC_SAMPLES;

static uint8_t adc_samples[ADC_SAMPLES * GELS];
static uint8_t zeros[ADC_SAMPLES] = {0};


// INTERRUPTS
void tim3_isr(void) {
    if (timer_get_flag(TIM3, TIM_SR_CC1IF)) {
        /* Clear compare interrupt flag. */
        timer_clear_flag(TIM3, TIM_SR_CC1IF);

    }
}

void dma1_channel1_isr(void) {
    frame = frame + 1;
    if ( dma_get_interrupt_flag(DMA1, 1, DMA_TCIF) != 0 )
{
        TIM3_CR1 &= ~TIM_CR1_CEN;


        while ((ADC1_CR & ADC_CR_ADSTART) != 0){
        }
        ADC1_CR |= ADC_CR_ADSTP;
        while ( (ADC1_CR & ADC_CR_ADSTP) != 0){
        }
        ADC1_CR |= ADC_CR_ADDIS;
        while ((ADC1_CR & ADC_CR_ADEN) != 0){
```

```
        }

        dma_clear_interrupt_flags(DMA1, 1, DMA_TCIF);

        if (frame < window) {


            DMA1_CCR1 &= ~DMA_CCR_EN; //needs to be off
to change DMA1_CNDTR1
            DMA1_CMAR1 = (uint32_t)
&(adc_samples[frame*num_samples_per_frame]);
            DMA1_CNDTR1 = num_samples_per_frame;
            DMA1_CCR1 |= DMA_CCR_EN; //turn back on
because then it won't work, duh.

            ADC1_CR |= ADC_CR_ADEN; //enable ADC DMA
            while ((ADC1_ISR & ADC_ISR_ADRDY) == 0){}
//wait for it to turn on
            ADC1_ISR = ADC_ISR_ADRDY;

            ADC1_CR |= ADC_CR_ADSTART;

            TIM3_CCR1 = TIM3_CCR1 + TOT/GELS;  //unless
equals num_frames or smaller
            TIM3_SMCR |= TIM_SMCR_SMS_TM; //put timer in
trigger mode again
            TIM3_CR1 |= TIM_CR1_OPM;

            //get trigger
            ADC3_CR |= ADC_CR_ADSTART;

            while ((ADC3_ISR & ADC_ISR_EOS) == 0){

            }
            ADC3_ISR |= ADC_ISR_EOC;
            ADC3_ISR |= ADC_ISR_EOS;
            trig[frame] = ADC3_DR;

        }

        else{
            TIM3_CCR1 = 0x0001; //unless equals
num_frames or smaller
            TIM3_SMCR &= ~TIM_SMCR_SMS_TM;
            TIM3_SMCR |= TIM_SMCR_SMS_OFF;

            ADC3_CR |= ADC_CR_ADSTART;

            while ((ADC3_ISR & ADC_ISR_EOS) == 0){
            }
```

```
            ADC3_ISR |= ADC_ISR_EOC;
            ADC3_ISR |= ADC_ISR_EOS;
            trig[frame] = ADC3_DR;

            frame = 0;
            sample_chunk_per_frame = 0;
            //usbd_ep_write_packet(usb_device,0x82,
(uint8_t *) &(adc_samples[0]), 62);

            usbd_ep_write_packet(usb_device,0x82, trig,
62);


        }
    }
}

void adc1_2_isr(void) {

    if ( adc_eoc(ADC1) != 0 ) {

        adc_state++;

        /*
         if (adc_state == (62*BUFF+1)){

         while ((ADC1_CR & ADC_CR_ADSTART) != 0){
         }
         ADC1_CR |= ADC_CR_ADSTP;
         while ( (ADC1_CR & ADC_CR_ADSTP) != 0){
         }
         ADC1_CR |= ADC_CR_ADDIS;
         while ((ADC1_CR & ADC_CR_ADEN) != 0){
         }
         adc_state = 0;

         gpio_port_write(GPIOE, 0xAA00);
         }
         */

#if 0 //use this to avoid DMA use ADC interrupt as
pseudo-DMA
        if (adc_state == (ADC_SAMPLES)-1) {
            leds = leds+0x0100;
            adc_samples[adc_state] = ADC1_DR;
            adc_state = 0;
            sample_chunk_per_frame = 0;
            usbd_ep_write_packet(usb_device,0x82,
(uint8_t *) &(adc_samples[0]), 62);
            ADC1_CR |= ADC_CR_ADDIS; //CORRECT WAY TO
```

```
PAUSE ADC
        }

        else {
            adc_samples[adc_state] = ADC1_DR;
            adc_state++;
        }
#endif

    }

}



// SETUP ROUTINES
static void timer_setup(void) {
    rcc_periph_clock_enable(RCC_TIM3);

    TIM3_CR1 = TIM_CR1_CKD_CK_INT | !TIM_CR1_ARPE |
TIM_CR1_DIR_UP  | !TIM_CR1_URS | !TIM_CR1_UDIS |
!TIM_CR1_CEN; // one shot mode


    TIM3_CR2 = (!TIM_CR2_TI1S) |
TIM_CR2_MMS_COMPARE_OC1REF | (!TIM_CR2_CCDS);
    TIM3_SMCR = (!TIM_SMCR_ECE) | TIM_SMCR_ETP |
TIM_SMCR_ETPS_OFF | TIM_SMCR_ETF_OFF | !TIM_SMCR_MSM |
TIM_SMCR_TS_ETRF | (0x0 << 3); // trigger mode
    TIM3_SMCR  &= ~TIM_SMCR_ETP;

    //TIM3_CR1 |= TIM_CR1_OPM;
    //TIM3_SMCR |= TIM_SMCR_SMS_TM;


    /* TIM3_SMCR Bit 3
     OCCS: OCREF clear selection
     This bit is used to select the OCREF clear source
     0: OCREF_CLR_INT is connected to the OCREF_CLR input
     1: OCREF_CLR_INT is connected to ETRF
     */
    //nvic_enable_irq(NVIC_TIM3_IRQ); //TIMER ISR
    //TIM3_DIER = TIM_DIER_CC1IE; //TIMER ISR
    TIM3_CCMR1 = TIM_CCMR1_OC1M_PWM2;
    TIM3_CCER |= TIM_CCER_CC1E; //show PWM on PB4
    TIM3_CNT = 0;
    TIM3_PSC = 0; //no prescale
    TIM3_ARR = 0xFFFF;
    TIM3_CCR1 = 0x0001;
    //OCCS flag not included in libopen
```

```
}

static void dma_setup(void) {
    rcc_periph_clock_enable(RCC_DMA1);

    DMA1_CCR1 = DMA_CCR_PL_VERY_HIGH | DMA_CCR_MSIZE_8BIT
| DMA_CCR_PSIZE_8BIT |
    DMA_CCR_MINC | DMA_CCR_TCIE;

    DMA1_CNDTR1 = num_samples_per_frame;
    DMA1_CPAR1 = (uint32_t) &(ADC1_DR);
    DMA1_CMAR1 = (uint32_t) &(adc_samples[0]);

    nvic_enable_irq(NVIC_DMA1_CHANNEL1_IRQ);

    DMA1_CCR1 |= DMA_CCR_EN;
}

static void adc_setup(void) {
    //ADC
    rcc_periph_clock_enable(RCC_ADC12);
    rcc_periph_clock_enable(RCC_ADC34);
    rcc_periph_clock_enable(RCC_GPIOA);
    rcc_periph_clock_enable(RCC_GPIOB);
    rcc_periph_clock_enable(RCC_GPIOF);
    //ADC

    gpio_mode_setup(GPIOA, GPIO_MODE_ANALOG,
GPIO_PUPD_NONE, GPIO1); //pa1
    gpio_mode_setup(GPIOA, GPIO_MODE_ANALOG,
GPIO_PUPD_NONE, GPIO2); //pa2
    gpio_mode_setup(GPIOF, GPIO_MODE_ANALOG,
GPIO_PUPD_NONE, GPIO4); //f4
    gpio_mode_setup(GPIOA, GPIO_MODE_ANALOG,
GPIO_PUPD_NONE, GPIO6); //pa6
    gpio_mode_setup(GPIOB, GPIO_MODE_ANALOG,
GPIO_PUPD_NONE, GPIO1); //PB1

    if ((ADC1_CR & ADC_CR_ADEN) != 0 ){ //basically is
ADC ready yet. wait until ADDIS is done
        ADC1_CR |= ADC_CR_ADDIS;
        while (ADC1_CR & ADC_CR_ADDIS){}
    }

    //ADC3----------------------------------------------
----------------------------------

    if ((ADC3_CR & ADC_CR_ADEN) != 0 ){ //basically is
ADC ready yet. wait until ADDIS is done
```

```
        ADC3_CR |= ADC_CR_ADDIS;
        while (ADC3_CR & ADC_CR_ADDIS){}
    }

    ADC3_CFGR = ADC_CFGR_RES_8_BIT; //SINGLE CONVERSION
MODE
    ADC3_CFGR &= ~ADC_CFGR_CONT;

    //ADC1----------------------------------------------
----------------------------------

    ADC1_CFGR = ADC_CFGR_CONT | ADC_CFGR_DMAEN |
ADC_CFGR_EXTEN_RISING_EDGE |ADC_CFGR_EXTSEL_EVENT_4 |
ADC_CFGR_RES_8_BIT; //CONTINOUS MODE with DMA
    ADC1_CFGR &= ~ADC_CFGR_DMACFG;

    ADC1_SMPR1 = (ADC_SMPR1_SMP_7DOT5CYC) << 9;
    //ADC_SMPR1_SMP_1DOT5CYC = 4.8MSPS
    //ADC_SMPR1_SMP_2DOT5CYC = 4.36MSPS
    //ADC_SMPR1_SMP_4DOT5CYC = 3.7MSPS
    //ADC_SMPR1_SMP_7DOT5CYC = 3.0MSPS, gets a bit
unstable if higher.
    //ADC_SMPR1_SMP_19DOT5CYC = 1.7MSPS
    //ADC_SMPR1_SMP_61DOT5CYC = 685.7kSPS
    //ADC_SMPR1_SMP_181DOT5CYC = 252.6kSPS
    //ADC_SMPR1_SMP_601DOT5CYC = 78.7kSPS
    //Example:
    //With FADC_CLK = 72 MHz and a sampling time of 1.5
ADC clock cycles:
    //Tconv = (1.5 + 12.5) ADC clock cycles = 14 ADC
clock cycles = 0.194 us (for fast channels)


    ADC1_SQR1 = ( ( 3 ) << ADC_SQR1_SQ1_LSB ); //PA2
    ADC_CCR = ADC_CCR_CKMODE_DIV1 | ADC_CCR_VREFEN;
    //ADC1_IER = ADC_IER_EOCIE; //only on if you want to
see one conversion at a time

    // start voltage reg
    ADC1_CR = ADC_CR_ADVREGEN_INTERMEDIATE;
    ADC1_CR = ADC_CR_ADVREGEN_ENABLE;
    //calibration

    ADC1_CR |= ADC_CR_ADCAL; //single ended
    while ((ADC1_CR & ADC_CR_ADCAL) != 0) {}

    ADC1_CR |= ADC_CR_ADCALDIF; //diff
    ADC1_CR |= ADC_CR_ADCAL;
    while ((ADC1_CR & ADC_CR_ADCAL) != 0) {}
```

```
    //ADC3-----------------------------------------------
----------------------------------

    ADC3_SMPR1 = (ADC_SMPR1_SMP_7DOT5CYC) << 1;
    //ADC3_IER = ADC_IER_EOCIE | ADC_IER_EOSIE;

    ADC3_SQR1 = ( ( 1 ) << ADC_SQR1_SQ1_LSB ); //Pb1
    //ADC3_SQR1 |= ADC_SQR1_L_1_CONVERSION;

    ADC34_CCR = ADC_CCR_CKMODE_DIV1 | ADC_CCR_VREFEN;

    ADC3_CR = ADC_CR_ADVREGEN_INTERMEDIATE;
    ADC3_CR = ADC_CR_ADVREGEN_ENABLE;

    ADC3_CR |= ADC_CR_ADCAL; //single ended
    while ((ADC3_CR & ADC_CR_ADCAL) != 0) {
        //gpio_port_write(GPIOE, 0x5500);
    }

    //ADC1-----------------------------------------------
----------------------------------
    // power on ADC1
    ADC1_CR |= ADC_CR_ADEN;
    //nvic_enable_irq(NVIC_ADC1_2_IRQ); //only on if you
want to see one conversion at a time

    //ADC3-----------------------------------------------
----------------------------------
    // power on ADC3

    ADC3_CR |= ADC_CR_ADEN;


    /* Wait for ADC1 and ADC3 starting up. ------------
----------------------------------------*/
    while ((ADC1_ISR & ADC_ISR_ADRDY) & (ADC3_ISR &
ADC_ISR_ADRDY)  == 0){
        //gpio_port_write(GPIOE, 0x9900);
    }

    ADC1_ISR = ADC_ISR_ADRDY;

    ADC3_ISR = ADC_ISR_ADRDY;

}

static void gpio_setup(void) {
    rcc_periph_clock_enable(RCC_GPIOE);
    rcc_periph_clock_enable(RCC_GPIOD); //timer
```

```
    rcc_periph_clock_enable(RCC_GPIOB);

    gpio_set_af(GPIOD, GPIO_AF2, GPIO2); //timer
    gpio_mode_setup(GPIOD, GPIO_MODE_AF, GPIO_PUPD_NONE,
                    GPIO2); //timer

    gpio_mode_setup(GPIOE, GPIO_MODE_OUTPUT,
GPIO_PUPD_NONE,
                    GPIO8 | GPIO9 | GPIO10 | GPIO11 |
GPIO12 | GPIO13 |
                    GPIO14 | GPIO15);

    gpio_set_af(GPIOB, GPIO_AF2, GPIO4); //PWM on PB4
    gpio_mode_setup(GPIOB, GPIO_MODE_AF,
GPIO_PUPD_PULLUP,
                    GPIO4);
}



// USB CALLBACKS
static void cdcacm_data_rx_cb(usbd_device *usbd_dev,
uint8_t ep) {
    (void)ep;
    (void)usbd_dev;

    char buf[64];
    ans = usbd_ep_read_packet(usbd_dev, 0x01, buf, 64);

    gpio_port_write(GPIOE, buf[0]<<8);

    if (buf[0] == 0x6E){ //ans == "n"
        DMA1_CCR1 &= ~DMA_CCR_EN;
        DMA1_CNDTR1 = num_samples_per_frame * GELS;
        window = 1;
        DMA1_CCR1 |= DMA_CCR_EN;

        TIM3_SMCR &= ~TIM_SMCR_SMS_TM; //turn off trigger
        TIM3_SMCR |= TIM_SMCR_SMS_OFF;
        TIM3_CR1 |= TIM_CR1_CEN;


    }

    else if (buf[0] == 0x73){ //ans = "s"
        DMA1_CCR1 &= ~DMA_CCR_EN;
        DMA1_CNDTR1 = num_samples_per_frame;
        window = GELS;
        DMA1_CCR1 |= DMA_CCR_EN;
        //gpio_port_write(GPIOE, 0x5500);
```

```
    }

    else{
        //gpio_port_write(GPIOE, 0xFF00);
    }


    //TIM3_CR1 |= TIM_CR1_CEN; //enable so that only usb
sends can gather data
    TIM3_SMCR &= ~TIM_SMCR_SMS_OFF;
    TIM3_SMCR |= TIM_SMCR_SMS_TM;
    TIM3_CR1 |= TIM_CR1_OPM;


}

static void cdcacm_data_tx_cb(usbd_device *usbd_dev,
uint8_t ep) {
    (void)ep;
    (void)usbd_dev;
    TIM3_SMCR &= ~TIM_SMCR_SMS_TM;

    usb_ready_to_send = 1;
    sample_chunk_per_frame = sample_chunk_per_frame + 1;

    if (sample_chunk_per_frame == (num_frames*BUFF+1)){
//num_frames*BUFF+1, so 8*75+1


        sample_chunk_per_frame = 0;

        DMA1_CCR1 &= ~DMA_CCR_EN; //needs to be off to
change DMA1_CNDTR1
        DMA1_CMAR1 = (uint32_t) &(adc_samples[0]);
        DMA1_CNDTR1 = num_samples_per_frame;
        DMA1_CCR1 |= DMA_CCR_EN; //turn back on
        TIM3_CCR1 = 0x0001;


        ADC1_CR |= ADC_CR_ADEN;

        while ((ADC1_ISR & ADC_ISR_ADRDY) == 0){}
        ADC1_ISR = ADC_ISR_ADRDY;

        ADC1_CR |= ADC_CR_ADSTART;

    }

    else {
        usbd_ep_write_packet(usb_device,0x82, (uint8_t *)
```

```
&(adc_samples[62*(sample_chunk_per_frame-1)]), 62);

    }


}

//SCARY USB STUFF

static const struct usb_device_descriptor dev = {
    .bLength = USB_DT_DEVICE_SIZE,
    .bDescriptorType = USB_DT_DEVICE,
    .bcdUSB = 0x0200,
    .bDeviceClass = USB_CLASS_CDC,
    .bDeviceSubClass = 0,
    .bDeviceProtocol = 0,
    .bMaxPacketSize0 = 64,
    .idVendor = 0x0483,
    .idProduct = 0x5740,
    .bcdDevice = 0x0200,
    .iManufacturer = 1,
    .iProduct = 2,
    .iSerialNumber = 3,
    .bNumConfigurations = 1,
};

/*
 * This notification endpoint isn't implemented.
According to CDC spec its
 * optional, but its absence causes a NULL pointer
dereference in Linux
 * cdc_acm driver.
 */
static const struct usb_endpoint_descriptor comm_endp[] =
{{
    .bLength = USB_DT_ENDPOINT_SIZE,
    .bDescriptorType = USB_DT_ENDPOINT,
    .bEndpointAddress = 0x83,
    .bmAttributes = USB_ENDPOINT_ATTR_INTERRUPT,
    .wMaxPacketSize = 16,
    .bInterval = 255,
}};

static const struct usb_endpoint_descriptor data_endp[] =
{{
    .bLength = USB_DT_ENDPOINT_SIZE,
    .bDescriptorType = USB_DT_ENDPOINT,
    .bEndpointAddress = 0x01,
    .bmAttributes = USB_ENDPOINT_ATTR_BULK,
    .wMaxPacketSize = 64,
```

```
    .bInterval = 1,
}, {
    .bLength = USB_DT_ENDPOINT_SIZE,
    .bDescriptorType = USB_DT_ENDPOINT,
    .bEndpointAddress = 0x82,
    .bmAttributes = USB_ENDPOINT_ATTR_BULK,
    .wMaxPacketSize = 64,
    .bInterval = 1,
}};

static const struct {
    struct usb_cdc_header_descriptor header;
    struct usb_cdc_call_management_descriptor call_mgmt;
    struct usb_cdc_acm_descriptor acm;
    struct usb_cdc_union_descriptor cdc_union;
} __attribute__((packed)) cdcacm_functional_descriptors =
{
    .header = {
      .bFunctionLength = sizeof(struct
usb_cdc_header_descriptor),
      .bDescriptorType = CS_INTERFACE,
      .bDescriptorSubtype = USB_CDC_TYPE_HEADER,
      .bcdCDC = 0x0110,
    },
    .call_mgmt = {
      .bFunctionLength =
        sizeof(struct
usb_cdc_call_management_descriptor),
      .bDescriptorType = CS_INTERFACE,
      .bDescriptorSubtype = USB_CDC_TYPE_CALL_MANAGEMENT,
      .bmCapabilities = 0,
      .bDataInterface = 1,
    },
    .acm = {
      .bFunctionLength = sizeof(struct
usb_cdc_acm_descriptor),
      .bDescriptorType = CS_INTERFACE,
      .bDescriptorSubtype = USB_CDC_TYPE_ACM,
      .bmCapabilities = 0,
    },
    .cdc_union = {
      .bFunctionLength = sizeof(struct
usb_cdc_union_descriptor),
      .bDescriptorType = CS_INTERFACE,
      .bDescriptorSubtype = USB_CDC_TYPE_UNION,
      .bControlInterface = 0,
      .bSubordinateInterface0 = 1,
    },
};
```

```
static const struct usb_interface_descriptor comm_iface[]
= {{
    .bLength = USB_DT_INTERFACE_SIZE,
    .bDescriptorType = USB_DT_INTERFACE,
    .bInterfaceNumber = 0,
    .bAlternateSetting = 0,
    .bNumEndpoints = 1,
    .bInterfaceClass = USB_CLASS_CDC,
    .bInterfaceSubClass = USB_CDC_SUBCLASS_ACM,
    .bInterfaceProtocol = USB_CDC_PROTOCOL_AT,
    .iInterface = 0,

    .endpoint = comm_endp,

    .extra = &cdcacm_functional_descriptors,
    .extralen = sizeof(cdcacm_functional_descriptors),
}};

static const struct usb_interface_descriptor data_iface[]
= {{
    .bLength = USB_DT_INTERFACE_SIZE,
    .bDescriptorType = USB_DT_INTERFACE,
    .bInterfaceNumber = 1,
    .bAlternateSetting = 0,
    .bNumEndpoints = 2,
    .bInterfaceClass = USB_CLASS_DATA,
    .bInterfaceSubClass = 0,
    .bInterfaceProtocol = 0,
    .iInterface = 0,

    .endpoint = data_endp,
}};

static const struct usb_interface ifaces[] = {{
    .num_altsetting = 1,
    .altsetting = comm_iface,
}, {
    .num_altsetting = 1,
    .altsetting = data_iface,
}};

static const struct usb_config_descriptor config = {
    .bLength = USB_DT_CONFIGURATION_SIZE,
    .bDescriptorType = USB_DT_CONFIGURATION,
    .wTotalLength = 0,
    .bNumInterfaces = 2,
    .bConfigurationValue = 1,
    .iConfiguration = 0,
    .bmAttributes = 0x80,
    .bMaxPower = 0x32,
```

```
    .interface = ifaces,
};

static const char *usb_strings[] = {
    "Black Sphere Technologies",
    "CDC-ACM Demo",
    "DEMO",
};

/* Buffer to be used for control requests. */
uint8_t usbd_control_buffer[128];

static int cdcacm_control_request(usbd_device *usbd_dev,
struct usb_setup_data *req, uint8_t **buf,
                                  uint16_t *len, void
(**complete)(usbd_device *usbd_dev, struct usb_setup_data
*req))
{
    (void)complete;
    (void)buf;
    (void)usbd_dev;

    switch (req->bRequest) {
        case USB_CDC_REQ_SET_CONTROL_LINE_STATE: {
            /*
             * This Linux cdc_acm driver requires this to
be implemented
             * even though it's optional in the CDC spec,
and we don't
             * advertise it in the ACM functional
descriptor.
             */
            char local_buf[10];
            struct usb_cdc_notification *notif = (void
*)local_buf;

            /* We echo signals back to host as
notification. */
            notif->bmRequestType = 0xA1;
            notif->bNotification =
USB_CDC_NOTIFY_SERIAL_STATE;
            notif->wValue = 0;
            notif->wIndex = 0;
            notif->wLength = 2;
            local_buf[8] = req->wValue & 3;
            local_buf[9] = 0;
            // usbd_ep_write_packet(0x83, buf, 10);
            return 1;
     }
```

```
        case USB_CDC_REQ_SET_LINE_CODING:
            if (*len < sizeof(struct
usb_cdc_line_coding))
                return 0;
            return 1;
    }
    return 0;
}

static void cdcacm_set_config(usbd_device *usbd_dev,
uint16_t wValue) {
    (void)wValue;
    (void)usbd_dev;

    usbd_ep_setup(usbd_dev, 0x01, USB_ENDPOINT_ATTR_BULK,
64, cdcacm_data_rx_cb);
    usbd_ep_setup(usbd_dev, 0x82, USB_ENDPOINT_ATTR_BULK,
64, cdcacm_data_tx_cb);
    usbd_ep_setup(usbd_dev, 0x83,
USB_ENDPOINT_ATTR_INTERRUPT, 16, NULL);

    usbd_register_control_callback(
                                    usbd_dev,
                                    USB_REQ_TYPE_CLASS |
USB_REQ_TYPE_INTERFACE,
                                    USB_REQ_TYPE_TYPE |
USB_REQ_TYPE_RECIPIENT,

cdcacm_control_request);
}

static void usb_setup(void) {
    /* Enable clocks for GPIO port A (for GPIO_USART2_TX)
and USART2. */
    rcc_usb_prescale_1();
    rcc_periph_clock_enable(RCC_USB);
    rcc_periph_clock_enable(RCC_GPIOA);

    /* Setup GPIO pin GPIO_USART2_TX/GPIO9 on GPIO port A
for transmit. */
    gpio_mode_setup(GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE,
GPIO11 | GPIO12);
    gpio_set_af(GPIOA, GPIO_AF14, GPIO11| GPIO12);
}




int main(void) {
    rcc_clock_setup_hsi(&hsi_8mhz[CLOCK_48MHZ]);
```

```
    gpio_setup();
    adc_setup();
    dma_setup();
    timer_setup();


    usb_setup();

    usb_device = usbd_init(&stm32f103_usb_driver, &dev,
&config, usb_strings,
                           3, usbd_control_buffer,
sizeof(usbd_control_buffer));
    usbd_register_set_config_callback(usb_device,
cdcacm_set_config);



    gpio_port_write(GPIOE, 0x1100);
    ADC1_CR |= ADC_CR_ADSTART;

    while (1){
     usbd_poll(usb_device);


    }

}
```

11 Software Appendix

11.1 Python USB interface

```
#!/usr/local/opt/python/bin/python2.7

import serial, sys
import math
import numpy
import matplotlib.pyplot as plt
import csv
import matplotlib.patches as mpatches


gels = 8;
frame_size = (gels*62*75)+62;

bytes_per_sample = 1;

def tc8(val):
    out = 0;
    if (val & 0x80):
        out = (~val & 0x7F) + 1;
        out = -1 * out;
    else:
        out = val;
    return out;

def csv_read(a):
    x = []
    x_final = []
    x = open(a+".csv","U")
    x_reader = csv.reader(x, delimiter=' ',
quotechar='|')
    for row in x_reader:
        x_final.append(float(row[0]))

    return x_final



if __name__ == "__main__":
    frames = [];

    usbfriend = serial.Serial(sys.argv[1]);
    usbfriend.write(sys.argv[2])
```

```
    #usbfriend.write('n');
    data = usbfriend.read(frame_size * bytes_per_sample);

    for frame_idx in xrange(gels):
        #data = usbfriend.read(frame_size *
bytes_per_sample); #Can be updated for sequence of frames
if desired
        f = open('log.txt', 'w');
        f.write(data);
        f.close();

        frame = [];


        for byte_idx in xrange(len(data)):
            val = ord(data[byte_idx]);
            frame.append(val);


        frames.append(frame);

    usbfriend.close();

    for x in range(len(frames)):
        with open(str(x)+'data.csv', 'wb') as csvfile:
            spamwriter = csv.writer(csvfile, delimiter='
', quotechar='|', quoting=csv.QUOTE_MINIMAL)
            for y in range(len(frames[x])):
                spamwriter.writerow([frames[x][y]])

    #print frames

    plt.xlabel("Samples");
    plt.ylabel("Signal");
    plt.plot(frames[0]);
    #plt.plot(frames[0][:100]);
    plt.show();
```

11.2 MATLAB Signal Reconstruction

11.2.1 Sinc Filter Program

Written by Christopher Hummersone. Access original:

http://www.mathworks.com/matlabcentral/fileexchange/42956-sinc-filter

There are two files, conv_fft.m and sinc_filter.m

11.2.1.1 Conv_fft.m

```
function c = conv_fft(a,b,shape)
% Convolve two vectors using FFT multiplication
%
%   c = conv_fft(a,b)
%   c = conv_fft(a,b,shape)
%
%   Convolution using traditional overlapping methods can
be
%   slow for very long signals. A more efficient method
is
%   to multiply the FFTs of the signals and take the
inverse
%   FFT of the result. However, this comes at a cost:
%   FFT-based convolution is subject to floating-point
%   round-off errors, and requires more memory [1].
%
%   c = conv_fft(a,b) convolves vectors a and b. The
%   resulting vector is length: length(a)+length(b)-1.
%
%   c = conv_fft(a,b,shape) returns a subsection of the
%   convolution with size specified by shape:
%     'full'  - (default) returns the full convolution,
%     'same'  - returns the central part of the
convolution
%               that is the same size as a.
%     'valid' - returns only those parts of the
convolution
%               that are computed without the zero-padded
%               edges. length(c) is L-2*(min(P,Q)-1)
where
%               P = numel(a), Q = numel(b), L = P+Q-1.
%
%   Based on code written by Steve Eddins, 2009.
%
%   References
%
```

```
%   [1] http://blogs.mathworks.com/steve/2009/11/03/
%       the-conv-function-and-implementation-tradeoffs/
%
%   See also CONV.

% !---
%
=============================================================
=
% Last changed:     $Date: 2012-10-28 13:03:43 +0000
(Sun, 28 Oct 2012) $
% Last committed:   $Revision: 214 $
% Last changed by:  $Author: ch0022 $
%
=============================================================
=
% !---

assert(isvector(a) & isvector(b),'a and b must be
vectors')

if nargin<3
    shape = 'full';
end
assert(ischar(shape),'Unknown shape parameter')

P = numel(a);
Q = numel(b);
L = P + Q - 1;
K = 2^nextpow2(L);

% do the convolution
afft = fft(a, K);
bfft = fft(b, K);
c = ifft(afft(:).*bfft(:));

% specify index range for shape
switch lower(shape)
    case 'full'
        range = [1, L];
    case 'same'
        range = [floor(length(b)/2)+1, L-
ceil(length(b)/2)+1];
    case 'valid'
        range = [min(P,Q), L-min(P,Q)+1];
    otherwise
        error(['shape ''' shape ''' is invalid. The
options are ''full'' (default), ''same'', or
''valid''.'])
end
```

```
% crop to shape
c = c(range(1):range(2));

% restore orientation
if shape(1) == 'f'
    if length(a) > length(b)
        if size(a,1) == 1 %row vector
            c = c.';
        end
    else
        if size(b,1) == 1 %row vector
            c = c.';
        end
    end
else
    if size(a,1) == 1 %row vector
        c = c.';
    end
end

% [EOF]
```

11.2.1.2 Sinc_filter.m

```
function y = sinc_filter(x,Wn,N,dim)
% Apply a near-ideal low-pass or band-pass brickwall
filter
%
%   y = sinc_filter(x,Wn)
%   y = sinc_filter(x,Wn,N)
%   y = sinc_filter(x,Wn,N,dim)
%   y = sinc_filter(x,Wn,[],dim)
%
%   y = sinc_filter(x,Wn) applies a near-ideal low-pass
or
%   band-pass brickwall filter to the array x, operating
%   along the first non-singleton dimension (e.g. down
the
%   columns of a matrix). The cutoff
frequency/frequencies
%   are specified in Wn. If Wn is a scalar, then Wn
%   specifies the low-pass cutoff frequency. If Wn is a
%   two-element vector, then Wn specifies the band-pass
%   interval. Wn must be 0.0 < Wn < 1.0, with 1.0
%   corresponding to half the sample rate.
%
%   The filtering is performed by FFT-based convolution
of x
%   with the sinc kernel.
%
%   y = sinc_filter(x,Wn,N) allows the filter length to
be
%   specified. The default value is N=1025. The filter
%   length is doubled in the band-pass case. In either
case,
%   if N is even, the final filter length will be N+1.
%
%   y = sinc_filter(x,Wn,N,dim) applies the specified
filter
%   along the dimension dim.
%
%   y = sinc_filter(x,Wn,[],dim) applies the specified
%   filter along the dimension dim using the default
filter
%   length.
%
%   See also CONV_FFT.

% !---
%
============================================================
```

```
=
% Last changed:     $Date: 2013-12-20 11:44:30 +0000
(Fri, 20 Dec 2013) $
% Last committed:   $Revision: 265 $
% Last changed by:  $Author: ch0022 $
%
=============================================================
=
% !---

%% test input

assert(nargin>=2,'Not enough input arguments')
assert((numel(Wn)==1 || numel(Wn)==2) & isnumeric(Wn),'Wn
must be a scalar or two-element vector.')
assert(isnumeric(x),'x must be a numeric array.')
assert(all(Wn<=1) & all(Wn>=0),'Wn must be 0.0 < Wn <
1.0, with 1.0 corresponding to half the sample rate.')
dims = size(x);
if nargin<4
    dim = [];
else
    assert(isint(dim) & numel(dim)==1,'dim must be an
integer')
    assert(dim<=length(dims),'dim must be less than or
equal to the number of dimensions in x')
end
if nargin<3
    N = [];
elseif ~isempty(N)
    assert(isscalar(N) & isint(N),'N must be an integer
scalar.')
end

%% assign defaults

if isempty(N)
    N = 1025;
end
if isempty(dim)
    dim = find(dims>1,1,'first');
end

%% reshape input to matrix with requested dim made as
first dimension

% reshape data so function works down columns
order = mod(dim-1:dim+length(dims)-2,length(dims))+1;
x = permute(x,order);
dims_shift = dims(order);
```

```
x = reshape(x,dims_shift(1),numel(x)/dims_shift(1));
y = zeros(size(x));

%% create filter kernel

if numel(Wn)==1 % low-pass
    n = -floor(N/2):floor(N/2); % create time base
    B = sinc_kernel(Wn,n); % make kernel
else % band-pass
    n = -N:N; % create time base
    B = sinc_kernel(Wn(2),n)-sinc_kernel(Wn(1),n); % make
kernel
end

%% apply filter

for N = 1:size(y,2)
    y(:,N) = conv_fft(x(:,N),B,'same');
end

%% reshape out to match input

y = reshape(y,dims_shift);
y = ipermute(y,order);

% end of sinc_filter()

% ------------------------------------------------------
% Local functions:
% ------------------------------------------------------

% ------------------------------------------------------
% sinc_kernel: Make sinc kernel
% ------------------------------------------------------
function k = sinc_kernel(Wn,n)

k = sinc(Wn*n).*Wn;

% ------------------------------------------------------
% isint: Test whether x is integer value (not type)
% ------------------------------------------------------
function y = isint(x)

y = x==round(x);


% [EOF]
```

11.2.2 Sinc Interpolation Program

Written by Ted Pavlic [11].

```
function y = sinc_interp(x,s,u,N)
    % Interpolates x sampled sampled at "s" instants
    % Output y is sampled at "u" instants ("u" for
"upsampled")
    % Optionally, uses the Nth sampling window where N=0
is DC
    % (so non-baseband signals have N = 1,2,3,...)

    if nargin < 4
        N = 0;
    end

    % Find the period of the undersampled signal
    T = s(2)-s(1);

    % When generating this matrix, remember that "s" and
"u" are
    % passed as ROW vectors and "y" is expected to also
be a ROW
    % vector. If everything were column vectors, we'd do.
    %
    % sincM = repmat( u, 1, length(s) ) - repmat( s',
length(u), 1 );
    %
    % So that the matrix would be longer than it is wide.
    % Here, we generate the transpose of that matrix.
    sincM = repmat( u, length(s), 1 ) - repmat( s', 1,
length(u) );

    % Equivalent to column vector math:
    % y = sinc( sincM'(N+1)/T )*x';
    y = x*( (N+1)*sinc( sincM*(N+1)/T ) - N*sinc(
sincM*N/T ) );

end
```

11.2.3 Real-Time (Normal) Mode

For use in "normal" mode. Takes ADC data from USB and simply reconstructs without sequential sampling. Up-interpolates signal by factor of two. Does not require trigger voltage.

```
addpath('/Users/jinstone/Desktop/stmsetup/thesis/sw/sinc_
filter/');
addpath('/Users/jinstone/Desktop/stmsetup/thesis/sw/sinc_
interp/');

data0 = csvread('0data.csv');
%data0 = data0 - 128; %remove 1.5V offset
trigger_set = data0(1:62);
disp(max(data0)-min(data0));
data0 = data0(63:end);
ave = mean(data0)
%ave = 136; %1.6V/3.0V * 256
data0 = data0 - ave;
data0 = data0.*(2.975/256);

trigger = mean(trigger_set(2:9).*(2.975/256));
trigger = trigger - ave*(2.975/256);

disp('SNR of Raw Data');
disp(snr(data0));

frac = 3;

figure('Name','Normal - No Seq');
subplot(3,1,1);
plot((1/3E6):(1/3E6):length(data0)*(1/3E6)/(frac*100),dat
a0(1:length(data0)/(frac*100)),'.-
',(1/3E6):(1/3E6):length(data0)*(1/3E6)/(frac*100),trigge
r);
title('Raw Data from ADC')
xlabel('Seconds')
ylabel('Volts')

seqraw = data0';

endpt = 2000;
split = 10;
samp = 3E6;

xl =
length(sinc_interp(seqraw(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(2*endpt-1)/(split*samp)));
```

```
out = zeros(xl,1);
out = sinc_interp(seqraw(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(2*endpt-1)/(split*samp));

subplot(3,1,2);
plot((1/30E6):(1/30E6):length(out)*(1/30E6)/frac,out(1:le
ngth(out)/frac),'.-');
title('Interpolated by 10')
xlabel('Seconds')
ylabel('Volts')

samp1 = 30E6;

NFFT = 2^nextpow2(length(out));
Y = fft(out,NFFT)/length(out);
%plot(fft(out));
f = samp1/2*linspace(0,1,NFFT/2+1);
f = f(1:end/10);
%Plot single-sided amplitude spectrum.
new = 2*abs(Y(1:NFFT/2+1));
new = new(1:end/10);
[val, idx] = max(new);
szStr=['SNR of Interpolated Data at ',
num2str(f(idx)/1E6),' MHz'];
disp(szStr);
disp(snr(out));

subplot(3,1,3);
plot(f,new);
%semilogx(f,new);
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```

11.2.4 Sequential Sampling Mode

For use in sequential mode. Requires trigger voltage that produces trigger pulse.

```
addpath('/Users/jinstone/Desktop/stmsetup/thesis/sw/sinc_
filter/');
addpath('/Users/jinstone/Desktop/stmsetup/thesis/sw/sinc_
interp/');

data0 = csvread('0data.csv');
%data0 = data0 - 128; %remove 1.5V offset
trigger_set = data0(1:62);
mean(trigger_set(2:8));
data0 = data0(63:end);
ave = mean(data0)
%ave = 136; %1.6V/3.0V * 256
data0 = data0 - ave;
data0 = data0.*(2.975/256);
trigger = mean(trigger_set(2:8).*(2.975/256));
trigger = trigger - ave*(2.975/256);

disp('SNR of Raw Data');
disp(snr(data0));
%plot(data0);
%figure('Name', 'Display Results')
%subplot(2,1,1);
%pwelch(data0(6000:8000));
%subplot(2,1,2);
%plot(data0(6000:8000));
tot = 16; %7*3;
gels = 8;
hc = (tot)/(2*gels);
buff = 75;
p = zeros(62*buff,gels);
preseq = zeros((62*buff - hc*(gels-1)), gels);



for n = 0:(gels-1)

    p(:,(n+1)) = data0((1+n*62*buff):62*buff*(1+n));
    x = p(hc*(gels-n-1) + 1:(end -hc*(n)),(n+1));
    preseq(:,(n+1)) = x;

end
```

```
seqa = [];
seqb = [];
seqraw = [];

%just P data not chopped
for ii = 1:length(p);

    for i = 1:gels
        seqraw(end+1) = p(ii,i);

    end


end

endpt = 2000;
split = 2;
samp = 24E6;



xl =
length(sinc_interp(seqraw(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(2*endpt-1)/(split*samp)));
out = zeros(xl,1);
out = sinc_interp(seqraw(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(2*endpt-1)/(split*samp));
seqb2 = out;
samp1 = 48E6;

%plot(out);


NFFT = 2^nextpow2(length(out));
Y = fft(out,NFFT)/length(out);
%plot(fft(out));
f = samp1/2*linspace(0,1,NFFT/2+1);
f = f(1:end/2);
%Plot single-sided amplitude spectrum.
new = 2*abs(Y(1:NFFT/2+1));
new = new(1:end/2);

figure('Name','seqb');
subplot(2,1,1);
plot(f,new);
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```

```
[val, idx] = max(new);

max_frq = (f(idx))/1E6; %max freq
frq = max_frq -.75; %.75 = 12 (MHz)/(gels*2), gels = 8
start = frq/24 ;
frac = 0.125/2 ;

if (0.5 - start) <= frac
    sinc_end = 0.5;

elseif ( start < 0)
    start = 0;
    sinc_end = start + frac;
else
    sinc_end = start + frac;
end

%seqb = sinc_filter(out, [start sinc_end]); %goes from 0
to 1
seqb = seqb2; %no sinc-ing


NFFT = 2^nextpow2(length(seqb));
Y = fft(seqb,NFFT)/length(seqb);
f = samp1/2*linspace(0,1,NFFT/2+1);
subplot(2,1,2);
%plot(fft(seqb));
%Plot single-sided amplitude spectrum.
subplot(2,1,2);
plot(f,2*abs(Y(1:NFFT/2+1)))
xlim([0 12E6]);
title('Single-Sided Amplitude Spectrum of y(t) (after
Sinc Block)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')



%pwelch(seqb);
%endpt = 3000;
split = 10;

samp = 48E6;

xl = length(sinc_interp(seqb(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(endpt-1)/(split*samp)));
out = zeros(xl,1);
out = sinc_interp(seqb(1:endpt),0:1/(samp):(endpt-
1)/(samp),0:1/(split*samp):(endpt-1)/(split*samp));
sampl = 1*48E6;
```

```
len = 400;
offset = 0; %4000;

szStr=['SNR of Interpolated Data at ',
num2str(f(idx)/1E6),' MHz'];
disp(szStr);
disp(snr(out));

figure('Name', 'Raw Data')

 for n = 1:gels %n = 1:gels+1
    if n ~= gels+1

        %subplot(gels+2,1,n);
        subplot(gels,1,n);

plot((1/24E6):(1/24E6):len*(1/24E6),p(1:len,n),(1/24E6):(
1/24E6):len*(1/24E6),trigger);
        title(strcat('Gel', int2str(n),' - No Filter'));
        xlabel('Seconds')
        ylabel('Volts')
        ylim([-.5 .5])

%     else
%         subplot(gels+2,1,n);
%
plot((1/480E6):(1/480E6):length(out)*(1/480E6),out,(1/480
E6):(1/480E6):length(out)*(1/480E6),trigger);
%         title('Sinc-Interpolated 2 * 10 w/Trigger')
%         ylim([-1.5 1.5])
%
%         subplot(gels+2,1,n+1);
%
plot((1/24E6):(1/24E6):len*(1/24E6),seqraw(1:400),(1/24E6
):(1/24E6):len*(1/24E6),trigger);
%         title('Raw Sequential Data')
%         ylim([-1.5 1.5])
%

    end

end

figure('Name', 'Display Results')
subplot(4,1,1);


NFFT = 2^nextpow2(length(out));
Y = fft(seqb,NFFT)/length(out);
```

```
plot(fft(out));
f = sampl/2*linspace(0,1,NFFT/2+1);
% Plot single-sided amplitude spectrum.
plot(f,2*abs(Y(1:NFFT/2+1)))
title('Sinc-Interpolated 2 * 10 - FFT');
xlabel('Frequency (Hz)')
xlim([0 12E6]);

%pwelch(out(1:end));
subplot(4,1,2);
%r =
length(out(length(out)/2:length(out)/2+length(out)/4));
%plot(1:r,out(length(out)/2:length(out)/2+length(out)/4),
1:r,data0(1));
plot(0:(1/480E6):(1/480E6)*length(out)/4,out(length(out)/
2:length(out)/2+length(out)/4))
title('Sinc-Interpolated 2 * 10 - Quarter');
%ylim([-1 1])
xlabel('Seconds')
ylabel('Volts')

subplot(4,1,3);
plot((1/480E6):(1/480E6):(1/480E6)*length(out),out,'.-');
title('Sinc-Interpolated 2 * 10');
%ylim([-1 1])
xlabel('Seconds')
ylabel('Volts')

subplot(4,1,4);
plot(1/24E6:(1/24E6):length(seqraw)*(1/24E6)/350,seqraw(1
+1:length(seqraw)/350+1),'.-
',1/24E6:(1/24E6):length(seqraw)*(1/24E6)/350,trigger,'re
d')
title('Raw Sequential Data')
%ylim([-1 1])
xlabel('Seconds')
ylabel('Volts')
```

Bibliography

[1] Agilient, Application Note 5989-8794EN, 2013.

[2] Tektronix, Sampling Oscilloscope Techniques, 1989.

[3] John Mulvey, *Sampling Oscilloscope Circuits*, 1st ed., Tektronix, Ed. Beaverton, OR, USA: Tektronix, 1970.

[4] Robert Sugarman, "Sampling Oscilloscope," Grant US2951181 A, Aug 30, 1960.

[5] Analog Devices, AD8031/AD8032, 2014, Rev G.

[6] Analog Devices, AD823, 2011, Rev E.

[7] Kent Lundberg, "Internal and External Op-Amp Compensation: A Control-Centric Tutorial ," in *American Control Conference*, vol. 6, Boston, 2004 , pp. 5197 - 5211.

[8] M Miyahara, Y Asada, D Paik, and A Matsuzawa, "A low-noise self-calibrating dynamic comparator for high-speed ADCs," in *Solid-State Circuits Conference*, Fukuoka , 2008, pp. 269 - 272.

[9] STMicroelectronics, RM0316 Reference manual , May 2014.

[10] William Siebert, *Circuits, Signals, and Systems*. Cambridge, MA, USA: The MIT Press, 1986.

[11] Ted Pavlic. (2008, June) Phase Portrait. [Online]. http://phaseportrait.blogspot.com/2008/06/sinc-interpolation-in-matlab.html

[12] Endolith. (2011, October) GitHub. [Online]. https://gist.github.com/endolith/1297227#file-sinc_interp-py

[13] G. L. Williams, "Sub-Nyquist Sampling and Moire-Like Waveform Distortions.," National Aeronautics and Space Administration (NASA), Cleveland, OH, Technical report N20020027355, 2000.

[14] Tektronix. (2001, January) Tektronix. [Online]. http://www.tek.com/document/application-note/real-time-versus-equivalent-time-sampling