Electrical Engineering Department

California Polytechnic State University

Senior Project Report

Iridium ADC

June 8, 2018

Bill Blakely

Contents

1 Abstract.

Today's modern digital audio systems are pushing performance boundaries. Most high quality audio digital to analog converters are capable of 24 bit, 96 kHz conversion. Professional audio interfaces frequently use 24 or 32 bit 192 kHz sample rates. These systems are able to achieve dynamic ranges greater than 100 dB, and distortion on the magnitude of single parts per million. This project proposes a high resolution analog digital converter (ADC) to measure and characterize these types of devices.

The proposed system will be capable of 24 bit conversion and an extended frequency range compared to the typical 20 Hz to 20 kHz audio band. This allows for a theoretical dynamic range of over 140 dB, ensuring the systems noise performance is limited by other factors. The extended frequency response offers several benefits:

1. Measurement of distortion products that occur on fundamental frequencies above 10kHz. Even though these products are not necessarily audible, they should be observed if present.
2. Observation of out-of-band oscillations, interference, or other noise. Various issues in a device could result in unintentional self oscillation which will likely be at a greater frequency than conventional audio measurement equipment operating at a maximum of 192kHz can observe.
3. Measurement of non-audio signals. Many common signals operate in the tens or hundreds of kHz range. Switch mode power supplies operate in frequencies of 100kHz and up[1]. Industrial, scientific, and medical equipment may use ultrasonic equipment, and vibration or resonant analysis may produce signals that go well beyond 20 kHz[2].

The digitized data will be transferred over USB, and a custom program will display the data in time and frequency domain simultaneously. This interface also enables control over the sampling and data processing parameters.

2 Background.

High quality audio is a field full of interesting challenges. Audio reproduction involves maintaining a flat response across a minimum three decades of bandwidth. Additionally, human hearing has an incredibly broad dynamic range, with the ability to discern quiet whispers and roaring jet engines. The signal to noise ratio of most audio systems must exceed that of our hearing. It is not uncommon for consumer devices to claim SNR of over 110dB[3], a range from 1 V down to about 70 nV. When designing and building these devices very sensitive, stable, and transparent measurement equipment must be used.

The Iridium ADC fits these needs and more by supplying a high quality converter with a bandwidth greatly in excess of what is normally required for audio and a dynamic range of over 130 dB. This allows an engineer to perform tests that they can not do with ordinary systems and allows the Iridium ADC to be used for measurement in other engineering and scientific fields.

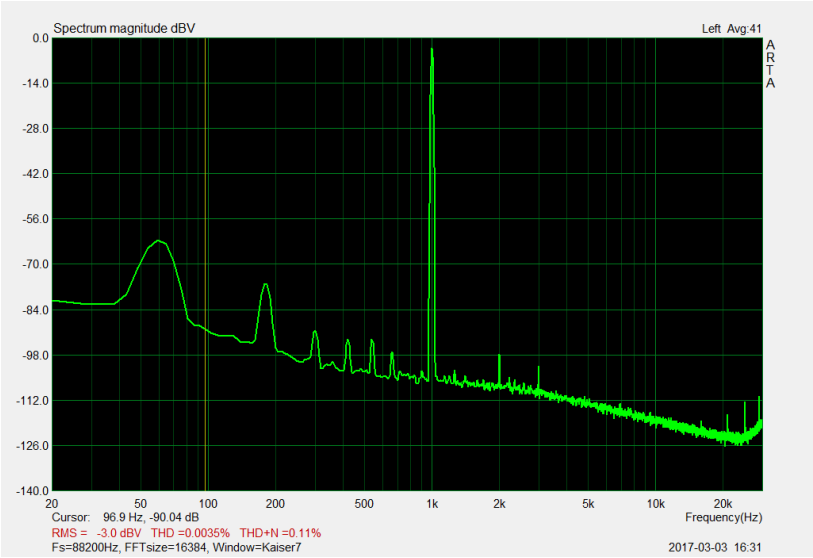


Figure 1: A measurement of total harmonic distortion at 1 kHz using ARTA and frequency domain methods.

Many audio quality measurements are performed in the frequency domain, and the first step in that analysis is to convert the analog signal to a digital one. Metrics such as frequency response, signal-to-noise ratio, and, as seen in **figure 1**, harmonic distortion are often easier to perform in the frequency domain. However, to accurately characterize an analog system using these digital tools, an accurate and precise ADC must be used.

What makes an ADC precise and accurate? ADC's have several key qualities. Bit depth determines the number of signal magnitude levels that are available. Since the lowest noise floor that can be achieved is approximately one bit of conversion, this also sets the theoretical dynamic range. A 24 bit converter has a theoretical dynamic range of $20\log(2^{24}) = 144dB$. The sample rate determines the maximum frequency that can be accurately measured.

There are several types of analog to digital conversion, but in this frequency range, delta sigma converters dominate. These currently have an upper frequency limit of a few MHz but offer very good linearity and exceptional noise performance. In general, ADC's have two types of nonlinearities, integral non-linearity (INL), and differential non-linearity (DNL) as seen in **figure 2**[4].

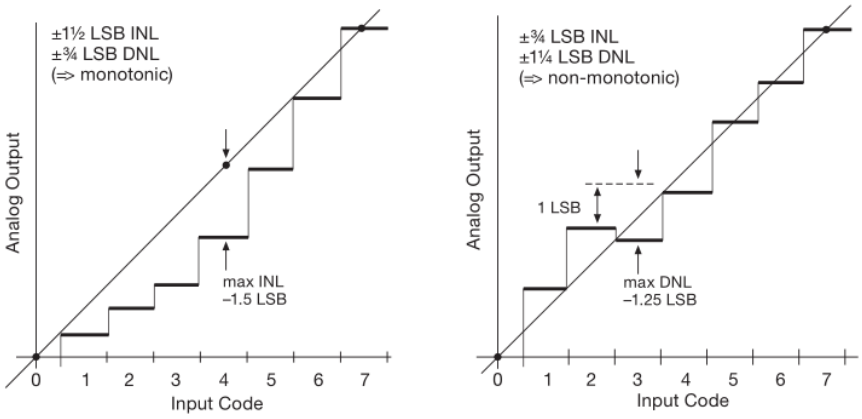


Figure 2: Diagram of INL (left) and DNL (right), from *The Art of electronics, 3rd* by Horowitz and Hill.

INL determines the maximum deviation from a straight line over all conversion levels while DNL determines the maximum step change from each level to adjacent levels. Delta sigma converters are typically guaranteed monotonic for less than 1 bit DNL. Delta-sigma converters also feature very low INL, and these two properties make them an ideal choice for a system that must have excellent linearity.

3 Overview of the Need.

3.1 Product Description.

The core of the Iridium ADC is a high dynamic range, low distortion data acquisition system. A stable reference voltage makes it suitable for measurement of absolute (instead of relative) signal levels. It features a bandwidth much larger than most audio measurement solutions. The resulting hardware is a compact unit only 3.3" by 2.8". Three external connections are required: a power input, a USB cable, and a connection to the signal that is being measured.

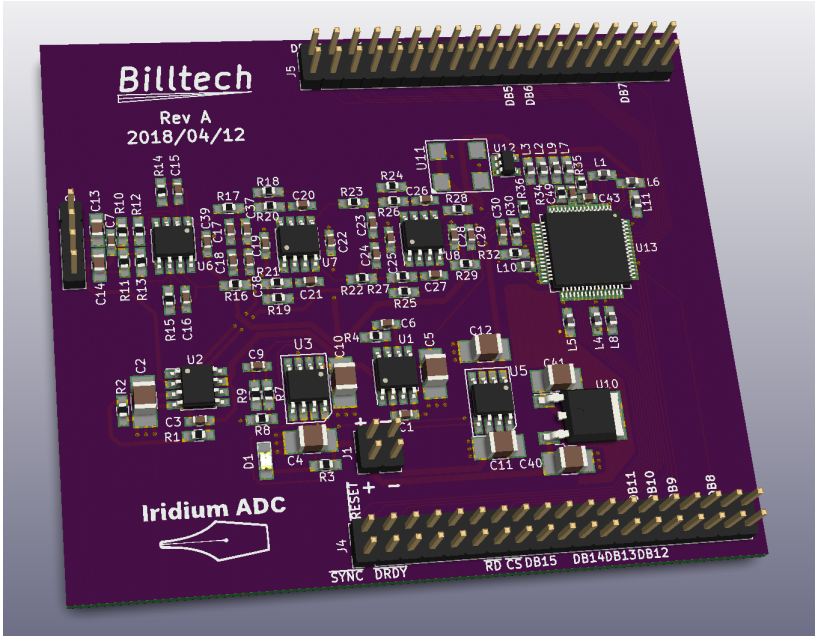


Figure 3: 3d Model of the assembled PCB.

The companion software, seen in **figure 4** displays the signal in the time and frequency domain. Through the GUI, the user can control the sampling rate, number of samples, number of averages, and windowing function. Simultaneous display of time and frequency domain allows for an immediate investigation of any unexpected artifacts. The plots allow for interactive zooming, data collection, annotation, and data export. All of this makes it a powerful test and measurement system that fills a space not satisfied by most existing equipment.

3.2 Product Technology/Market Research.

For audio analysis the current market leader is Audio Precision. Their audio measurement systems are the industry standard for professional audio companies but carry a heavy price tag, reaching into the thousands or tens of thousands of dollars.

Common lab equipment for electrical engineering is not suitable for many audio measurements either. Oscilloscopes sacrifice sensitivity for bandwidth, with most oscilloscopes on the market offering 8 bit converters. This causes a correspondingly large noise floor as the theoretical limit for 8 bit systems is only 50 dB of dynamic range. Even the new wave of "high resolution" oscilloscopes, such as the LeCroy HRO series, only offer 12 bit conversion[5]. This would be unable to characterize even the aging CD standard of 16 bit audio. Additionally, oscilloscopes are first and foremost time domain systems and are not designed for measurements like distortion that are ideally suited to the frequency domain.

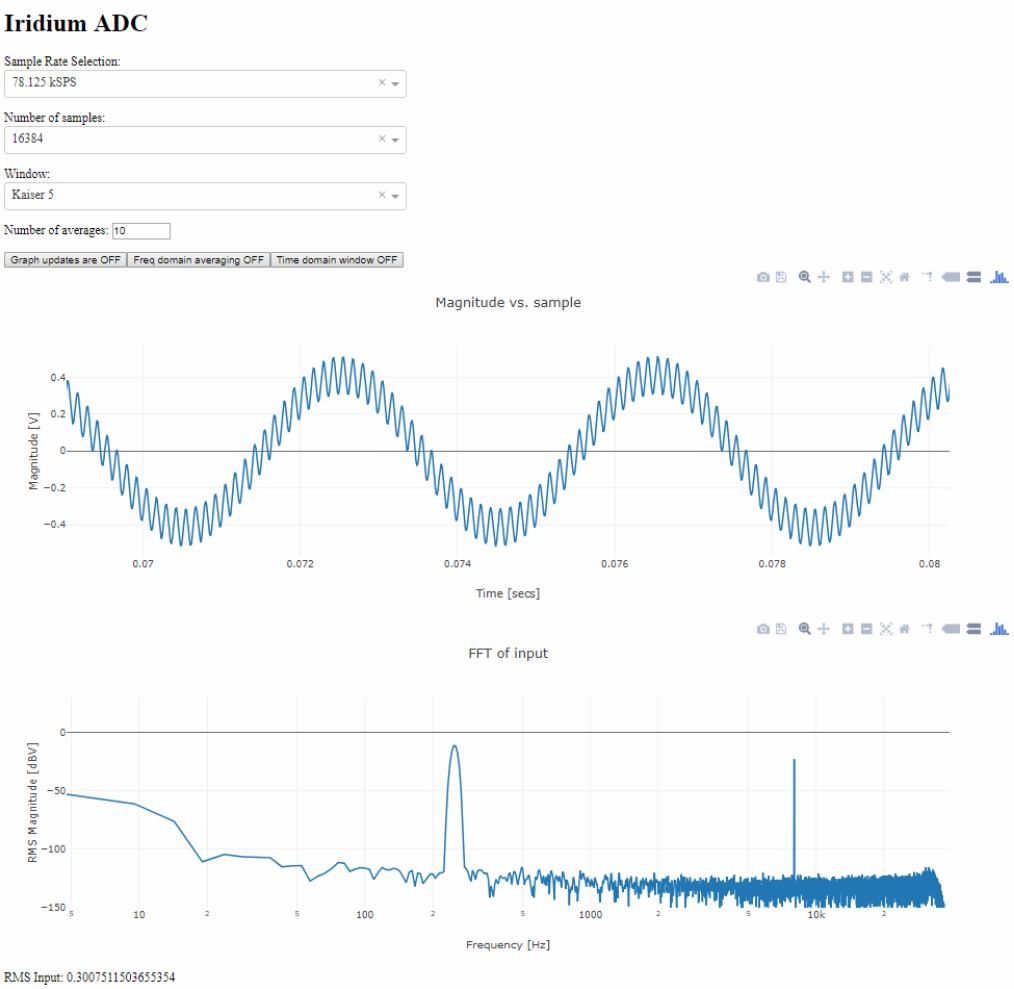


Figure 4: An example of the GUI during use, here demonstrating the ability to clearly distinguish a dual sine wave input as two distinct peaks in the frequency domain.

Many scientific pursuits also require characterizing high dynamic range broadband signals. Mechanical resonance, vibration modes, biological signaling, and ultrasonic inspection are all fields that could benefit from having a measurement device such as this. The wide range of signal magnitude and frequency that can be measured without adjustment of any front end filter or amplifier could make the Iridium ADC useful for scientific measurement, production line QA, or signal integrity verification.

4 Customer Archetype.

The Iridium ADC is a tool that allows engineers, product developers, and researchers to obtain and process the data that they need to. The user of this device may fit into several categories.

Specialty audio companies. The market for specialty audio companies has been expanding. Increasingly, independent companies are becoming the choice for home audio enthusiasts. Guitar effect pedals have seen an explosion in the number of companies developing and selling pedals that take on all sorts of mixtures of digital and analog processing. A resurgence in popularity of high end headphones has spurred a market devoted to headphone amplifiers to buffer the mobile audio devices we all have in our phones.

Many of these companies have product lines developed by just one or two engineers on a tight budget. Some probably develop their product entirely by ear because there are no currently available solutions for audio measurement under the cost of several thousand dollars. By providing a much larger bandwidth than typical audio analysis devices, the Iridium ADC is able to investigate the characteristics of switch mode power supply noise, out-of-band distortion products, EMI pickup, self oscillation, and other signals that would not be shown on most of the low cost options for audio analysis. By making the tools available, the Iridium ADC provides these engineers the tools to close the development feedback path on these devices, reducing development time, improving the performance, and producing accurate technical specifications.

Scientific Research The Iridium ADC can function as a multi-use data acquisition system. There are many signals similar to audio present in natural phenomenon or that are useful in scientific or industrial research. The ability to measure signals of a few micro volts and of a several volts simultaneously can open up the ability of a researcher to collect data and perform the cleanup and filtering in the digital domain. The bandwidth allows for research into both ultrasonic and subsonic bands. Accurately and confidently measure signals that could not be captured on most traditional lab equipment.

4.1 Pains and Gains.

One of the major problems with current solutions is the cost. Audio Precision's introductory audio analyzer, the APx515, starts at over $6000[6]. Additionally, this unit has a bandwidth of only 90 kHz. By offering our unit at a lower price point, there is a great potential to increase the number of people who can perform this testing. That means better designs and faster development.

The features of the Iridium ADC also can help during product development. The wide bandwidth means fewer problems will be missed and the high dynamic range means huge swings in signal amplitude can be measured without changing the test setup. There is far less setup and configuration when using this system, allowing an engineer to do more work with less hassle.

4.2 Competitors.

Audio Precision: The market leader in audio measurement, their devices lead the pack in performance and types of measurement. They offer analyzers from the 2 channel APx515 to the modular multichannel APx586 with 16 input channels. Their flagship offers a residual THD+N of –117dB +1.0 uV. That's a maximum of 0.00024% over a bandwidth of 1 MHz! Audio Precision is almost always likely to dominate the competition on performance and testing standards, but they also carry a hefty price tag.

Prism Sound: Makers of the dScope audio analyzer, this offers the 90 kHz bandwidth with THD+N as low as 0.00040% [7]. An article from 2008 claimed they were slashing prices by 24% to $9,850[8]. This appears to offer similar testing capability to some of the Audio Precision line but is starting to show its age technologically: it advertises automation using ActiveX, a technology now being shunned even by its creator, Microsoft.

Avermetrics: A newer company, Avermetrics was founded in 2011[9]. The AverLAB is a relatively low cost, portable desktop audio analyzer released in August 2017. It is available on Amazon for a cost of $3000, has a bandwidth from 10 Hz to 88 kHz and a THD+N of 0.00032%. It features a unique ethernet interface instead of USB and a compact, fanless design[10]. The AverLAB is the nearest competitor to the Iridium ADC. The AverLAB contains more features, but the Iridium ADC is targeting expanded performance.

5 Requirements and specifications.

The products requirements and targeted specifications are estimated in this section.

5.1 Marketing and Engineering Requirements.

Marketing requirements in **table 1** describe what is considered necessary to the project. The engineering specifications attempt to set defined guidelines for the project.

Importance	Marketing Requirement	Engineering Requirement	Justification
High	Low noise	Noise floor < −110dB @ 1kHz with 50 ohm input termination.	Low self noise preserves as much of the available dynamic range as possible. This enables measurement of signal than span from a few micro volts to several volts.
High	High bandwidth	+/- 1 dB from 10Hz to 130 kHz.	Bandwidth allows for measurement of signal and products that are beyond the scope of other signal analyzers. These signals include ultrasonics and switch mode power supply noise.
High	Low Distortion	THD < 0.001%	Attempting to measure device that have very low distortion requires a measurement unit with even lower distortion.
High	Stable, known reference	4.096 +/- 0.05%, 3 ppm temp co, 40 ppm long term stability	This allows for confidently transforming the relative measurement of decibel-full scale into an absolute voltage level that will not change significantly with environment or time. This is a large part of what separates this from the available consumer level devices.
High	Balanced inputs	CMRR > 60 dB.	Essential for eliminating noise pickup from sources with very low output levels. Professional audio systems are nearly always differential.
Medium	Common/generic power supply	DC power supply, 5mm barrel connector.	Simplifies the setup and operation of the device and does not tie up a lab supply which may be required for the device under test.
Medium	GUI	Display of data and selection of parameters.	Most of the characterization happens in the GUI, control from this enables easy measurement setup. Data could just be logged to a file and analyzed in another program.

Table 1: Marketing and engineering requirements.

5.2 Block Diagram and Specifications.

The Level 0 block diagram of the physical unit is seen in **figure 5**. The signal input is specified as a differential input, although it may be used with unbalanced signals as well. The USB connection serves dual purpose as a way to get the data out and to receive control signals.

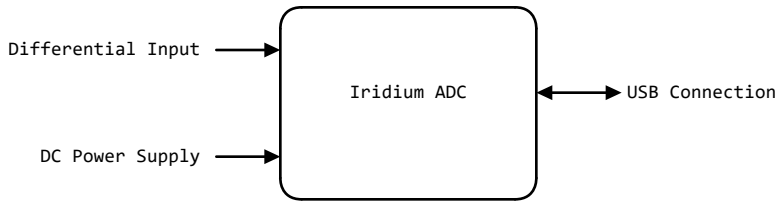


Figure 5: Level 0 Block Diagram showing external connections.

The level 1 block diagram in **figure 6** demonstrates the internal architecture of the device. Each block is summarized in **table 2**.

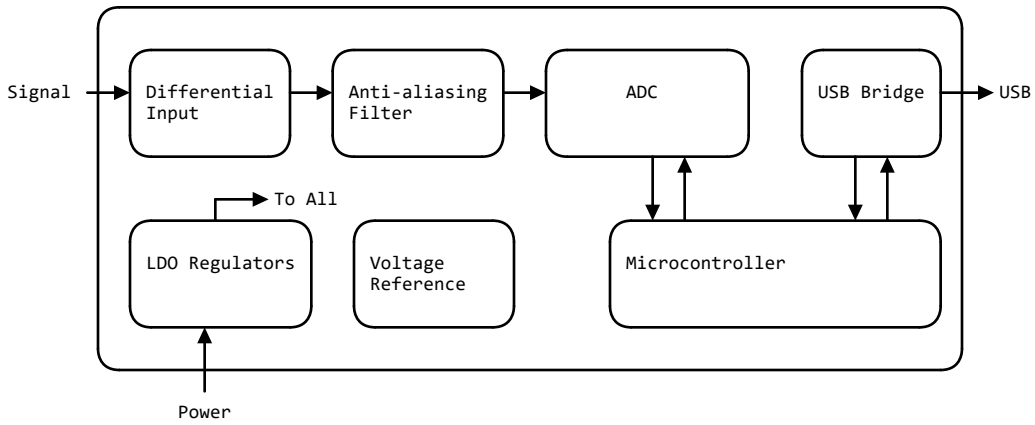


Figure 6: Level 1 Block Diagram

Block	Description
Differential Input	Maximum 10 V differential. Extremely low noise and distortion.
Anti Aliasing Filter	Cut-off frequency will be determined by bandwidth of final ADC.
ADC	>20 bit to use all of the potential dynamic range. Delta-Sigma converters offer the best linearity and noise performance in this frequency range.
Controller	Responsible for setting up the digital devices and preparing data for USB transfer. Must be fast enough to handle stream of data from ADC.
USB Bridge	Communication with host computer, 24 bit samples at 100's of kSPS result in a lot of data which requires a fairly high bandwidth to transmit.
Voltage Reference	Low Drift, Low Temp co. reference.
LDO Regulators	Must generate low noise 12 V supply for input stage as well as 5V and 2.5V sources for the ADC.

Table 2: Level 1 block diagram with major internal functional blocks.

6 Development and Design.

An overview of the major design choices and implementation. Full design files can be found in the Schematic, PCB, and Program listings in the appendices at the end of the report.

6.1 ADC Selection.

The design of the system can be divided into several parts, although the three major components are the analog design, the embedded system, and the user application. Each of these sections is dependent on the ADC which is being used, making that part the first choice to be made. The market for high resolution, high bandwidth ADC's is relatively small. A summary of the parts that were considered for the project is present in **Table 3**

Part No.	Manufacturer	Maxi Output Data Rate	Maximum BW (−3dB)	Bit Depth	Price (qty. 1)	Notes
AK5578	AKM Semiconductor	768 kSPS	144.5 kHz	32	$19.51	Audio grade device, offers up to 8 channels of conversion. I2S serial output.
ADS1675	Texas Instruments	4 MSPS	1.90 MHz	24	$36.21	Single channel, CMOS or LVDS data output, LVDS required at 2 highest data output rates.
AD7760	Analog Devices	2.5 MSPS	1.06 MHz	24	$57.27	Single channel, 16 bit parallel data bus.

Table 3: A selection of high speed delta sigma converters.

The AK5578 is a tempting choice. It offers lowest cost, and drastically lower cost per channel. It can be set up to provide lower noise by paralleling the input channels and outputting the average of all channels. When all 8 channels are mixed down to a single output, the signal to noise ratio increases from 121 dB to 130 dB. The maximum output data rate should be more than enough to meet the target bandwidth but this is where the chip fails. The –3 dB bandwidth is much lower than the expected 1/2 maximum sample rate. Looking at the frequency response chart, the part displays a very slow roll off that limits the bandwidth and does not offer true antialiasing at the output data rate's Nyquist limit. For the intended use this would not be a problem, there is ample bandwidth to create a true antialiasing filter outside of the 20 kHz audio band limit. For our broad band measurement device, this chip will not work.

The next choice would be the ADS1675, which offers the largest bandwidth out of these options. However, this chip suffers from a similar problem as the AK5578. The digital filter that converts the low bit depth, modulator rate data stream to the high bit depth output data rate is not true antialiasing. The filters in the ADS1675 are set with their –6dB point at the output data rate Nyquist limit. This once again means that there must be an analog anti aliasing filter to reduce aliasing to acceptable levels. The problem is even worse when multiple data rates will be supported. A different anti alias filter is required at each output data rate.

The AD7760 is the most expensive of the group, but it at does offer true antialiasing in the digital filter. This means that the analog front end only needs to provide antialiasing at the modulator data rate, in this case 20 MHz. With this setup, the analog filtering is simplified and the operation is more versatile, with the ability to support multiple bandwidth selections without compromising performance. After choosing the AD7760, the rest of the design can proceed.

6.2 ADC Implementation.

6.2.1 Clock.

The input clock can be selected to achieve different sample rates, but for maximum bandwidth, the highest speed clock of 40 MHz is picked. Clock performance has a direct relationship to ADC performance. Any jitter in the clock will show up as noise in the resulting signal. Because delta sigma converters over sample the input, the effects of jitter are reduced. From the AD7760 data sheet we find the following equation:

$$t_{j(rms)} = \frac{\sqrt{OSR}}{2 \times \pi \times f_{IN} \times 10^{\frac{SNR\,(dB)}{20}}}$$

where:
OSR = oversampling ratio = f_{ICLK}/ODR.
f_{IN} = maximum input frequency.
$SNR(dB)$ = target SNR.

Figure 7: Clock RMS jitter requirements vary with OSR, input frequency, and desired SNR.

Using the equation in **figure 7** with the minimum over sampling rate (OSR) of 8, the maximum frequency of 1 MHz, and desired SNR of 110 dB, a maximum jitter of 1.42 ps is determined. A CTS components model CB3 clock oscillator module is picked with a maximum rms jitter of 1 ps. It requires no external parts to operate. Following the recommendation in the data sheet, the clock signal is buffered using a On Semi NC7SZ08M5 AND gate. The resulting schematic design can be seen in **figure 8**.

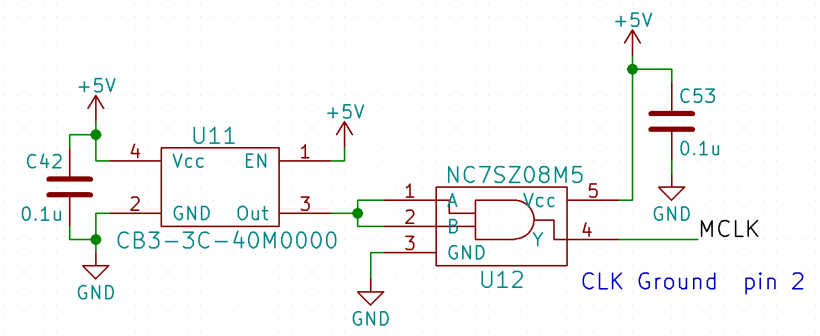


Figure 8: Implementation of 40 MHz clock with buffer.

6.2.2 Reference Voltage.

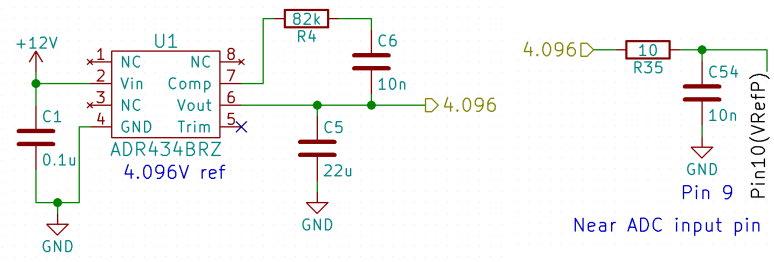


Figure 9: Reference voltage and with filtering

Another input directly affecting the sampling quality is the reference voltage. For maximum input range, the maximum reference voltage of 4.096 V is used. The recommended part is Analog Devices' ADR434. Low temperature coefficient of 1 ppm ensures there is little to no drift during use and after initial calibration. The long term stability 40 ppm during the first 1000 hours, reducing the need recalibrate. The signal is filtered with a large, low ESR ceramic capacitor as well as a compensation network near the reference IC. An additional filter placed near the ADC further reduces any noise that may be picked up on board.

6.2.3 Input Driver.

The AD7760 includes a fully differential amplifier intended to drive the input of the stage. This built in amplifier automatically sets the common mode level to the correct half reference level. The recommended circuit is followed for this, but the capacitor values are increased to set the filter cutoff frequency at about 2 MHz. The maximum differential signal at the input to this stage is 5 V.

6.2.4 Digital Interface.

Five control signals and a 16 bit parallel data bus is used for communication to the microcontroller. These are powered by the 2.5 V power source. They are tolerant to 3.3 V (and the microcontroller reads 2.5 V signals fine) so no level shifter is required. Each data read consists of two reads of the 16 bit data bus, making the maximum frequency of reads equal to twice the maximum output data rate (2.5 MHz), or 5 MHz. This is fairly low (in signal integrity terms) so controlled impedances, matched length traces or other high frequency considerations are largely unimportant here. Care is taken to ensure that the digital signals do not couple into the analog signal path, and this is largely achieved by physical separation of digital lines from the sensitive analog lines.

6.2.5 Power Supply Filtering.

There is a lot going on inside of the AD7760, and everything needs power. Two supply voltages are required, 5 V and 2.5 V. The extensive decoupling detailed in the datasheet is followed closely. Each power pin sees a ferrite bead and 0.1 uF coupling capacitor for best isolation of power supply noise. The coupling caps are kept as close to the pins as possible to minimize the effects of stray inductance and ensure best transient performance.

The 2.5 V supplies only the digital circuitry and digital communication drivers. The quality of this supply is not critical, but it does supply transient pulses of current (as many as 16 pins could be drawing switching current simultaneously), which must be kept out of the other section. This supply is tapped off the 15 VDC power input and regulated down to 2.5 V. This results in a lot of heat, so the NCV1117 in DPAK package is used for good heat transfer capabilities. It is connected to a copper pour on the PCB to help dissipate heat. A switching regulator would be a better choice here to operate more efficiently and is recommended for the next design cycle.

The 5 V regulator powers both the sensitive input amplifier, the internal reference voltage buffer, the external 40 MHz oscillator, and the delta sigma modulator circuit operating at 20 MHz. The 5 V supply is critical to overall system performance. The low noise, high performance, Analog Devices ADP7104 with a fixed 5V output is picked for this supply. This part is suggested for noise sensitive applications including ADCs, precision amplifiers, and high frequency

oscillators. The ferrite beads on each pin help to isolate the high frequency pulses of the clock and modulator from the noise sensitive analog circuitry. Low ESR ceramic capacitors with X5R and X7R dielectrics round out the filtering here and no performance issues from this supply are detectable in the final product.

6.2.6 Input Stage.

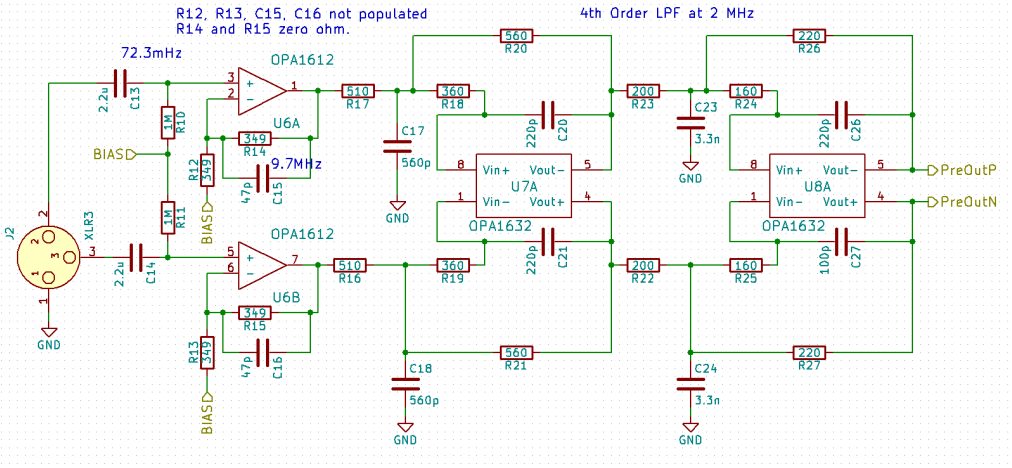


Figure 10: The input stage provides signal buffering, filtering, and scaling.

The input stage in **figure 10** is responsible for scaling the signal appropriately and providing the necessary antialiasing filtering. Handling of balanced or single ended signals is desired, and the ADC must accept a balanced signal at the other end, so a decision to maintain a balanced signal path throughout is made.

The main section of the circuit consists of two of Texas Instrument's OPA1632 fully differential op amps. These were selected for their 1.3 nV/sqrt(Hz) noise power density and extremely low harmonic distortion of 0.000022%. Fully differential op amps lend themselves to designs based around an inverting op amp. Texas Instruments Webench tool is used to design a fourth order Butterworth low pass filter at 2 MHz using a multiple feedback topology. This design is adapted to the balanced signal path. Low value resistors are used to keep noise low through these stages. Gain in these stages is about 0.6 so that the input signal range can be larger than the ADC input.

A buffer is placed before the OPA1632 stages because of their low input impedance. Two sections of an op amp are set up as non-inverting buffer amps to provide a high input impedance. The input is coupled through 2.2 uF capacitors and input impedance is set by 1 Mohm resistors, creating a high pass filter at 0.072 Hz. The stages are setup so that they can be used as gain stages if required, although in the current iteration they are only used as unity gain buffers. The OPA1612 is used for its excellent performance. At 1.1 nV/sqrt(Hz) and 0.000015% distortion this manages to exceed the performance of the OPA1632. Power for the input amplifiers come from a 12 VDC single supply. It uses the same ADP7104 series regulator as the 5V rail but is an adjustable version setup to produce a 12 V output. The bias voltage used in the first stage is buffered by a Texas Instruments OPA207 low noise op amp to provide stable DC performance without introducing low frequency drift into the circuit.

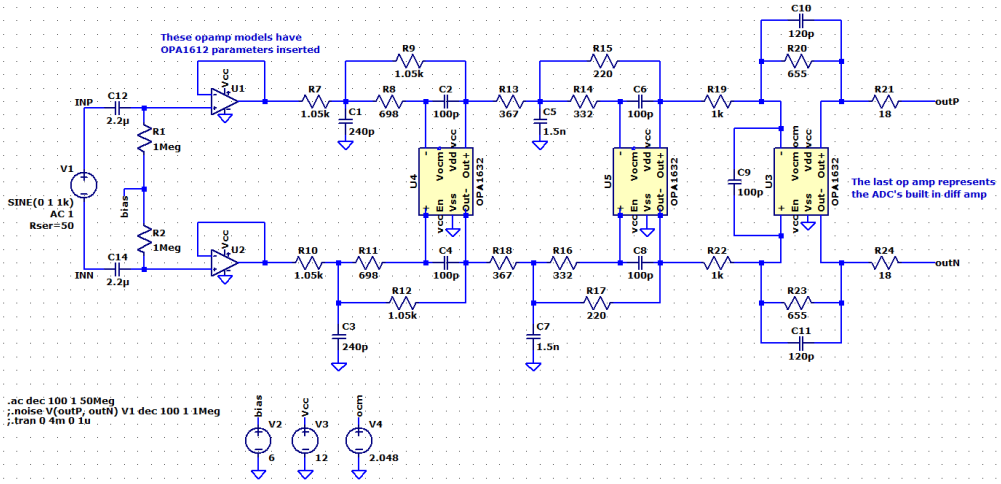


Figure 11: Schematic for LTSpice simulation of noise and ac response.

These stages were simulated in LTSpice and checked for frequency response, noise performance, and stability. Spice models for the OPA1632 are available from Texas Instruments and were used in the simulation. The simulation schematic can be seen in **figure 11**, while the .ac and .noise analyses are performed resulting in the plots in **figure 12** and **figure 13** respectively.

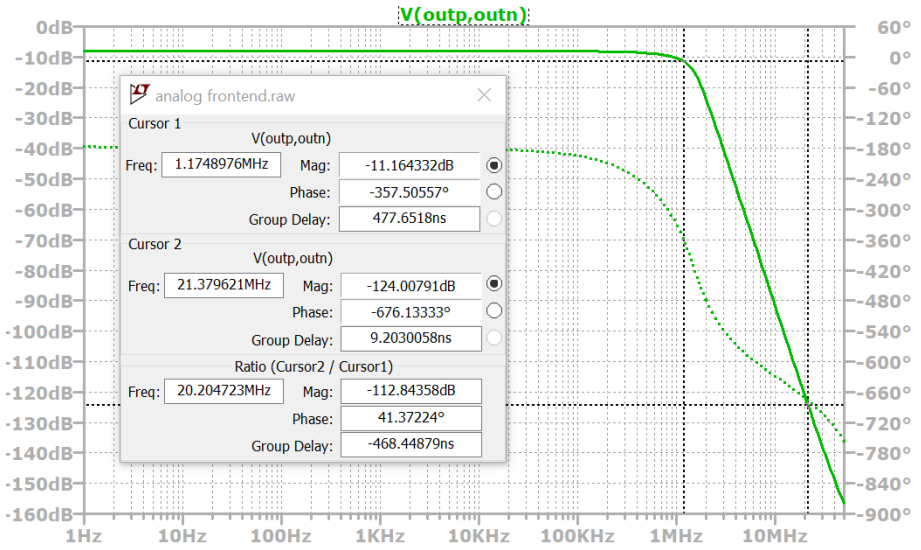


Figure 12: Simulated frequency response of circuit with cursors showing the −3 dB at 1.1 MHz and the magnitude at the 20 MHz aliasing equal to −124 dB.

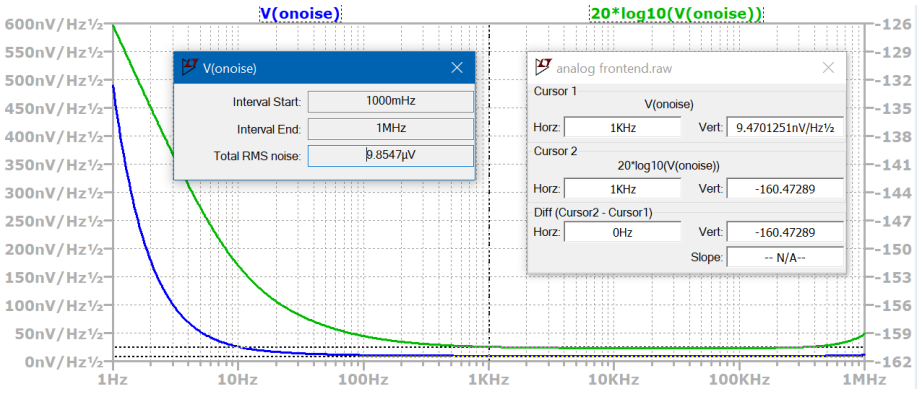


Figure 13: Noise analysis showing the noise voltage density in linear scale (blue line, left axis) and decibel scale (green line, right axis). Also shown is the RMS noise over this bandwidth. The simulated noise is incredibly low, indicating that other sources are likely to dominate the system.

To analyze stability of the filter circuits, the method from an Analog Device video "LTspice: Stability of Op Amp Circuits"[11] was adapted to the fully differential op amp. In this method, the feedback path is broken at the inverting input and a 0 VDC AC 1 source is inserted. This allows the op amp to operate at the correct DC bias without restricting the ratio of the feedback to the input, and it is this ratio which equals the open loop gain of the circuit.

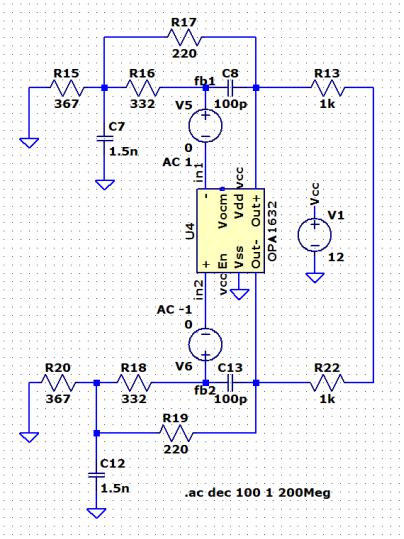


Figure 14: One method of obtaining open loop response in LTSpice.

The simulation schematic is seen in **figure 14**. To adapt to the fully differential op amp, both feedback paths must be broken and sources of AC 1 and AC –1 are inserted. After plotting the ratio of fb1 to in1 (fb2 to in2 is equivalent), the phase margin is determined.

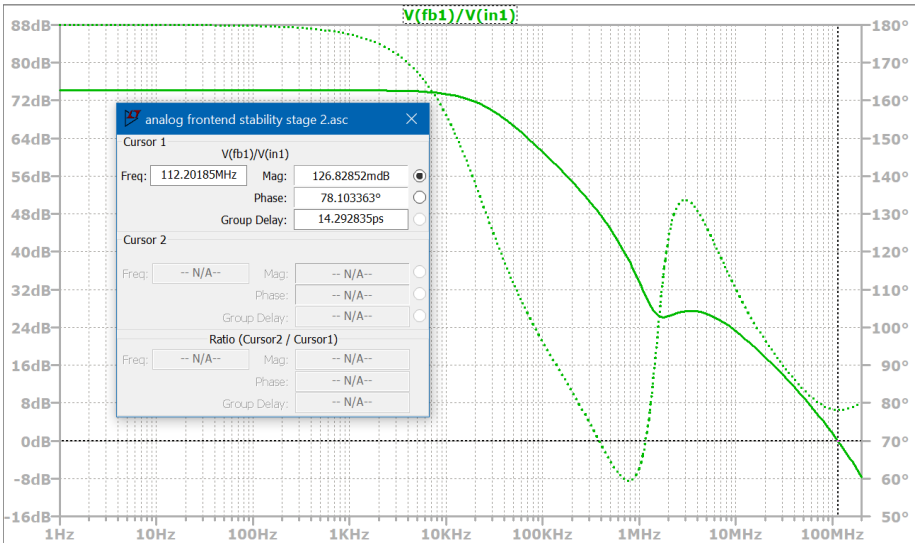


Figure 15: The phase margin of the filter is shown to be around 78°.

The phase margin is found to be adequate at about 78°. Only the second filter stage is shown because this stage, having less than unity gain, is the most likely to be unstable.

6.3 Microcontroller.

The STM Microelectronics Nucleo F446RE development board is used as the digital interface for this project. The board features an ARM cortex M4 32 bit processor and runs at a high clock rate of 180 MHz. As well as this speed it is also selected for:

- Large pin count (50 GPIO pins) for compatibility with parallel data bus.
- Support for mbed RTOS to speed development time.
- Can be USB powered.
- Low cost (approx. $15)

Additionally, the STM32F446RE processor on this board features support for USB 2.0 high speed when used with an external transceiver. Support for this mode of data transfer is an eventual goal of the project making this a good choice for future development.

6.4 PCB Design.

The PCB is designed around the form factor of the Nucleo development board. It is approximately the same dimensions and features two 2×19 pin header connectors so that it can mount directly on top of the Nucleo board. To keep costs low, the board layout uses only two layers. Some of the key design features of the PCB:

- Connectors and off board wires use through hole connections, and everything else is surface mount. This allows the bottom layer to be almost entirely ground plane.
- To keep power supply bypass caps close to the ADC, some parts are mounted to the bottom side but this is generally avoided.
- The differential signal path is kept nearly symmetrical to maintain a high CMRR.

6.5 Embedded System.

The embedded system is built on the mbed RTOS system. This allows a higher level abstraction of the underlying hardware speeding up development at the cost of additional overhead in the execution. While this project may eventually need to move away from mbed for performance reasons, it enabled the short development time available for this phase of the project.

Control flow is fairly simple:

1. At power on or after receiving reset, the initial state is written to the ADC including the sample rate.
2. A signal is received from the host computer to begin sampling. The ADC's data ready pin triggers an interrupt to read the data off the parallel bus.
3. After the preprogrammed number of samples is taken, the controller begins sending the data back to the host computer.
4. The microcontroller powers down the ADC and waits for a new control signal.

The embedded system code is written in C++ and consists of four C++ source files and their respective header files. They are summarized as follows:

- main.cpp: Runs the main program loop and catches the data ready flag to begin data transfer.
- adc.cpp: Responsible for control and interaction with the ADC, including reading and writing control registers, and collecting the data during sampling.
- communications.cpp: Handles control signals and data transmission with the host computer.
- pins.cpp: Pin assignment, setup, and functions for reading and writing data words.

Currently communication to the host computer happens over UART serial connection. The Nucleo uses a virtual serial port over USB, and this allows for a baud rate of 1.5 Mbaud. Even at this rate, the transmission of data back to the host is the primary bottleneck in the overall update rate of the system.

6.6 Host Application.

The host application runs on a PC and is responsible for the processing and display of the data received from the ADC. The host application is written in python and consists of two files.

- adc_comms.py contains a class for the interaction with the ADC. It sends and receives data over the UART connection to the microcontroller. This class also contains functions that decode the data received and convert it to a correctly scaled, floating point representation.
- pc_data.py serves the graphical user interface and performs the final data processing. This application displays the time and frequency domain signals simultaneously and provides control over the sample rate, number of samples, and FFT window functions.

7 Calibration and Testing.

The product will be tested by analysis of its sampled data output. Using a known signal as input, the dynamic range, CMRR, frequency response, THD and more can be measured. Acceptable performance will match what is set out in the Engineering requirements of **table 1**.

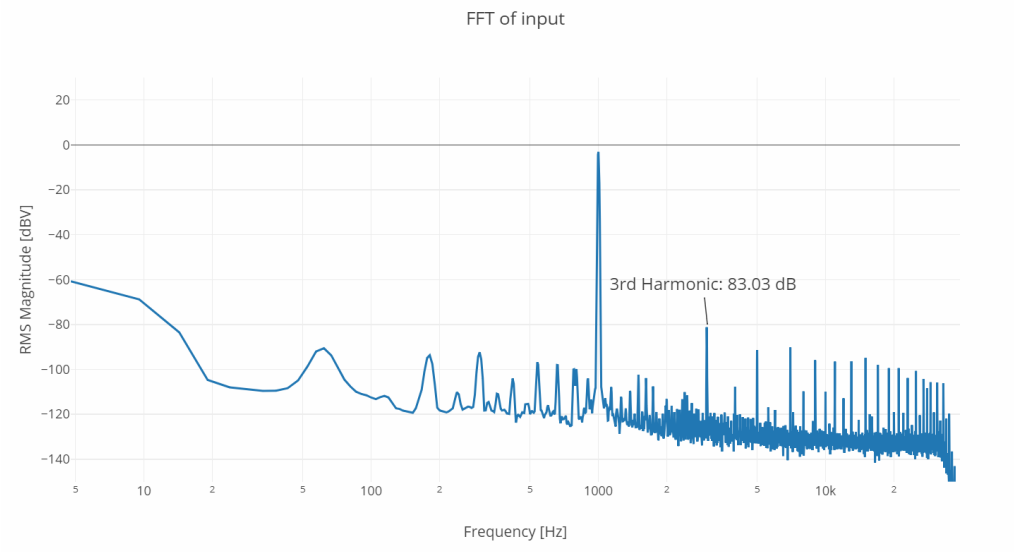


Figure 16: The 3rd harmonic alone counts for 0.01% distortion in this measurement of a Rigol DS1062Z Function generator.

Part of the reason for beginning this project was the lack of test equipment that could perform these low frequency, high dynamic range measurements. Oscilloscopes cover the frequency range but do not have the dynamic range to perform sensitive distortion measurements. Spectrum analyzers often do have the dynamic range, but do not cover the lowest part of the frequency range. Signal generators pose another problem, as seen in **figure 16**. Keysight's line of Trueform function generators have a 14 or 16 bit converter, limiting the noise floor to well above the level of this project. They also claim lower harmonic distortion, but the published specification of 0.03% is over 100x more than what we wish to achieve.[12]

So, using the lab equipment available, a source can not be characterized fully in order to compare performance across systems. And even if it could, the available signal sources have so much distortion and noise that measuring the system performance will be impossible.

The second reason for this project is to keep pace with the ever increasing performance of audio equipment. After looking at the performance of a Focusrite Scarlett 6i6 USB audio interface, it is determined that this offers very good performance that will allow some of the performance aspects to at least set some boundaries on performance characteristics. For each measurement, the Focusrite will be connected with one output into the Iridium ADC and the other back into the Focusrite interface. The performance can then be compared by examining our systems output with the Focusrite. The data from the Focusrite will be taken using ARTA spectrum analyzer mode. Using this setup allows us to estimate the performance of the system's distortion levels including THD and IMD.

Many additional characteristics may be made without the need of a low distortion source and may be made with other equipment as stated.

7.1 Calibration.

Before any measurements can be made, the systems magnitude must be calibrated. Because of the uncertainty of the exact reference voltage and circuit gain, a scale factor is found by sending a known signal into the input and reading the resulting signal in the Iridium interface. The input signal is measured using an Agilent 34661A multimeter to create a histogram of the signal and find a very accurate average level. This is compared against the RMS readout in the GUI which is calculated using the time domain sampled data. Because many measurements use a –3 dB input signal. The system is calibrated near this level. The Focusrite interface is used as signal generator for this test to produce a balanced signal input.

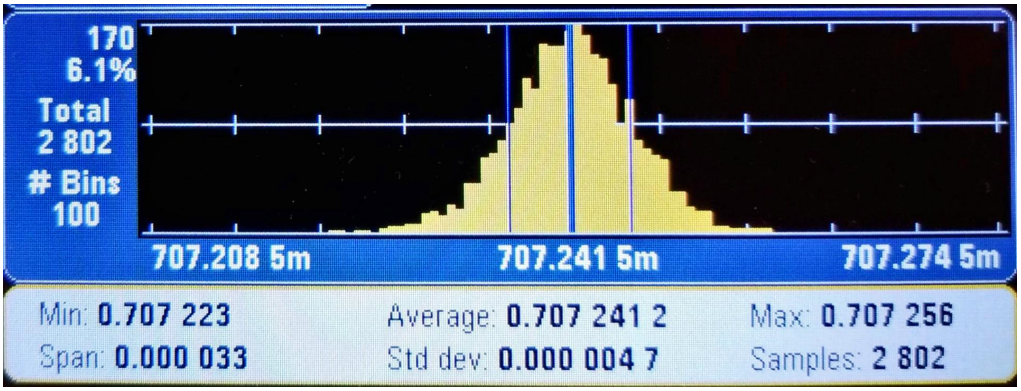


Figure: [histogram]: Measured input signal with average value of 0.7072412 V.

Input Signal	System Measurement	Scale Factor
0.7072412 V	0.0837775	8.44190 V

Table 4: Data and results of calculating scale factor

This value is inserted into the code to scale the data correctly.

7.2 Frequency Response and bandwidth.

For frequency response measurements, a Rigol DS1062Z function generator is used to create a white noise source. The response is viewed in the frequency domain with no window, maximum number of samples, and frequency domain averaging on with 100 samples averaged. The resulting plots for each sample rate are shown in **figure 17**. The minimum frequency domain measurable frequency and the estimated –3dB maximum frequency are listed in **table 5**.

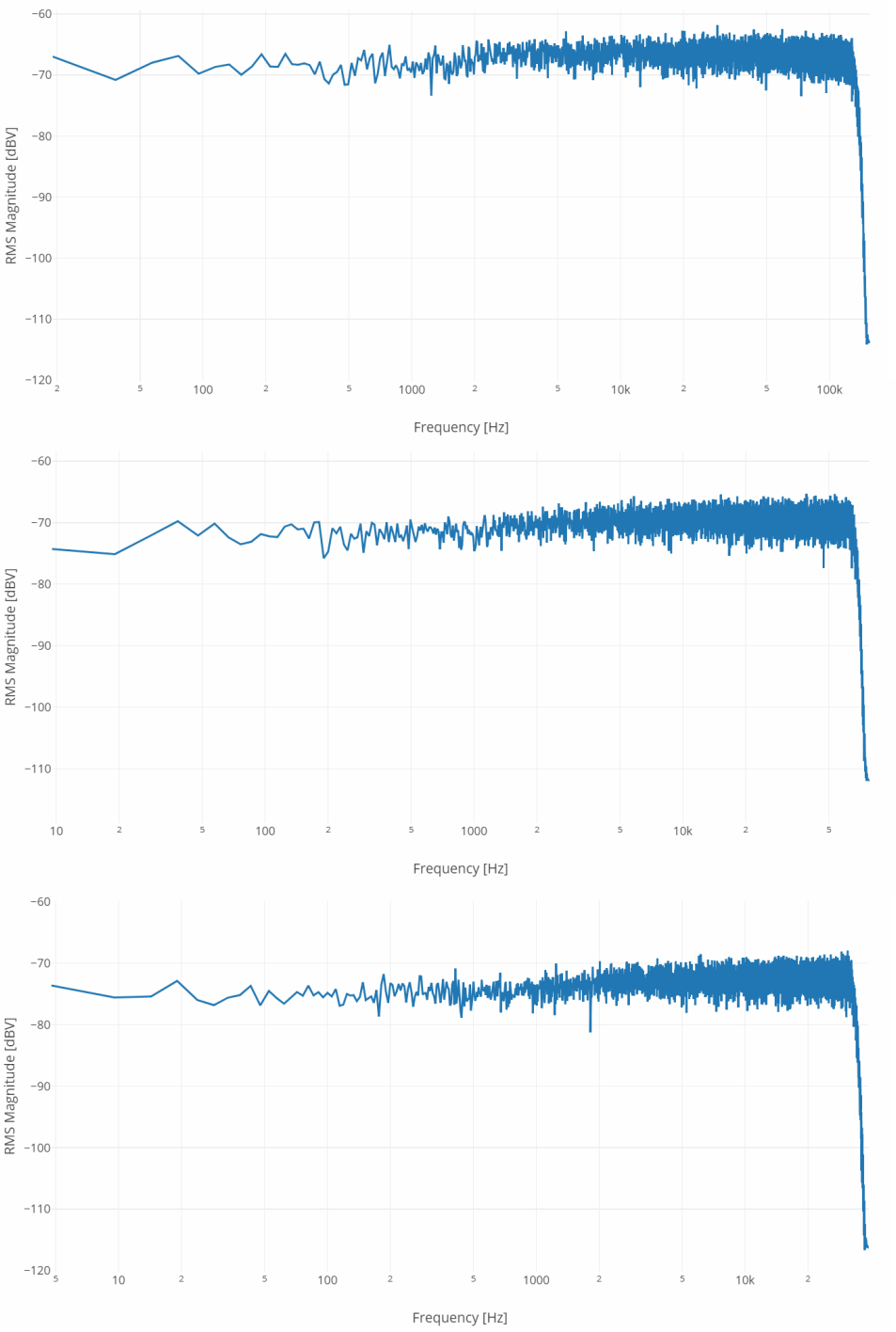


Figure 17: From top to bottom, frequency response plots of highest, middle, and lowest sample rates.

Sample Rate [kSPS]	Min Freq [Hz]	Max Freq [kHz]
312.5	19.07	133
156.25	9.54	68
78.125	4.77	33

Table 5: Summary of frequency response measurements.

7.3 Noise.

To measure noise, the input should be terminated with a resistor similar to the expected source impedance. After connecting the resistor, the RMS signal level and the voltage noise density can be measured in the time and frequency domain respectively. The RMS noise is measured at maximum bandwidth, and no window is used in the frequency domain.

Resistor	RMS noise voltage	Voltage noise density
47.0 ohms	24.03 uV	−127.70 dB at 1 kHz

Table 6: Noise measurements in time domain and frequency domain.

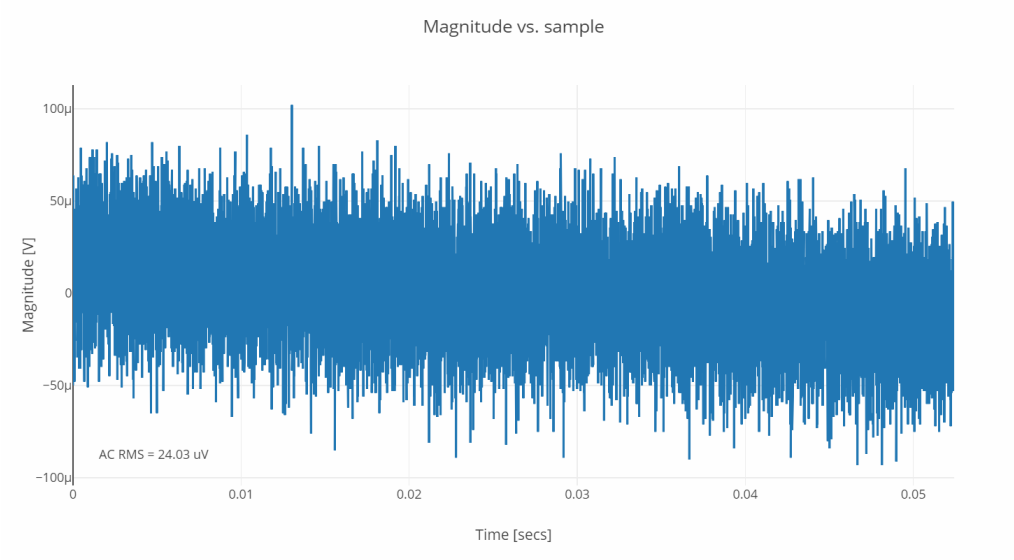


Figure 18: Noise in the time domain under system's maximum bandwidth terminated with 50 ohm differential impedance.

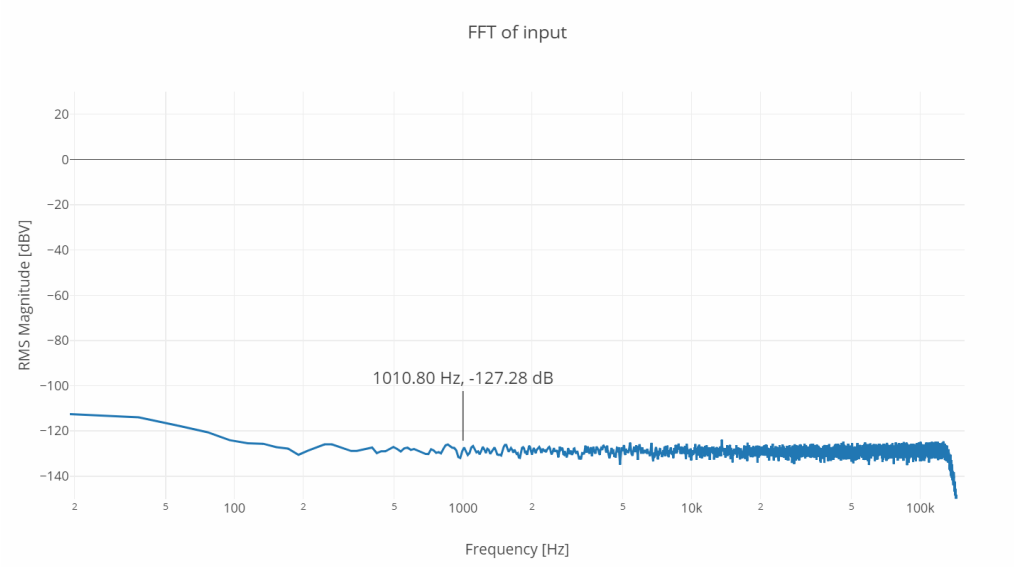


Figure 19: Noise in the frequency domain. No window used, frequency domain averaging on.

7.4 CMRR.

Common mode rejection ratio is determined by sending a known signal into both of the differential inputs and measuring the signal that makes it through the system. The ratio of these is computed and converted to dB:

$$CMRR = 20\log\left(\frac{measured}{input}\right)$$

A setup similar to the calibration system is used here, but the Rigol DS1062Z is used as signal source for its extended bandwidth. CMRR can vary with frequency, so it is checked for several different frequencies. The measured level is taken from the RMS display in the GUI while the input level is measured with the Agilent 34661A multimeter.

Frequency	Input level [V]	Measured level [uV]	CMRR [dB]
100 Hz	0.707362	72.52977	79.78
1 kHz	0.707215	21.17327	90.48
100 kHz	0.706289	287.713	67.80

Table 7: Common mode rejection ratio at several frequencies.

7.5 Dynamic Range.

The dynamic range of a system is the ratio of maximum signal input relative to the minimum signal, expressed in decibels. Typically, this is compared to the noise floor. In the time domain the dynamic range is given by the maximum input level over the RMS noise level. However in frequency domain, this becomes the maximum signal level over the noise density. Since noise density is always lower than the RMS noise, dynamic range is higher in the frequency domain than the time domain.

The maximum signal input is estimated by measuring the peak amplitude for a clipped signal input and dividing by $\sqrt{2}$. This gives the theoretical maximum RMS value of the signal input. This is then divided by the RMS noise for time domain dynamic range and by the voltage noise density for frequency domain dynamic range found in **table 6**. The final dynamic range is expressed in decibels.

Maximum Signal	Est Max RMS	DR in time domain	DR in frequency domain
6.01 V	4.25 V	105.0 dB	140.3 dB

Table 8: Data and results of determining dynamic range. The lower limits used for calculation is the same as in **table 6**.

7.6 Distortion.

Total harmonic distortion is most often specified for a 1 kHz −3dB signal level. Distortion can change with frequency and input level, so for a better understanding of the system, measurements are made at −3 and +6dB, and at 1 and 10 kHz.

The Focusrite is used as signal generator here and the THD display in ARTA is used to establish a boundary on distortion performance. It is important to note that the ARTA measurement will be including distortion in both the output and input of the Focusrite interface, while the signal feeding the Iridium ADC will be the distortion of the Focusrite's output and the distortion of the Iridium system itself. This may introduce some error into the measurement and is the reason this section can only set bounds on the distortion level and not precisely state the distortion itself.

Total harmonic distortion is calculated by finding the ratio of power in the upper harmonics to the power in the main harmonic. The square root of this is taken, and the value is usually expressed as a percentage.

$$THD = 100 \cdot \sqrt{\frac{\sum_{n=2}^{\inf} H_n^2}{H_1^2}}$$

Where H_n refers to the harmonic of the signal input.

To calculate distortion, a quick python script is used. The fundamental is measured from the frequency domain plot and input to the script, and the measurable harmonics are input as a list of values. All measurements can be input in dB and are converted to linear units for calculation.

```
'''Calculate harmonic distortion as a percent
using dB magnitude values from spectrum analyzer.'''

# fundamental level in dB.
```

```
fundamental = -3.2096

# list of higher harmonic levels in dB.
harmonics = [-121.58, -124.35, -124.55, -123.9, -127.23]

harmonic_power = sum([10**(peak/10) for peak in harmonics])
fundamental_power = 10**(fundamental/10)

distortion = (harmonic_power / fundamental_power)**0.5

# Print distortion as a percent..
print(f'Distortion: {distortion*100:.4}%')
```

Listing 1: An example of the python script for calculating THD.

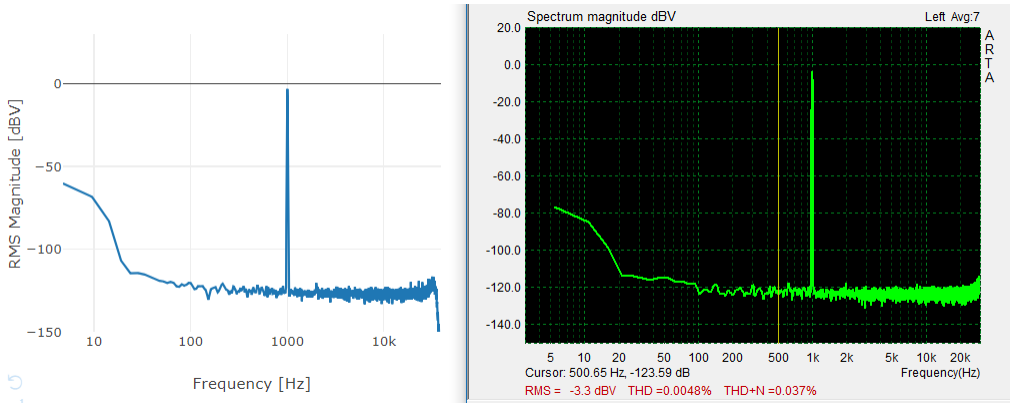


Figure 20: Test signal 1kHz, −3dB Iridium, left and Arta, right.

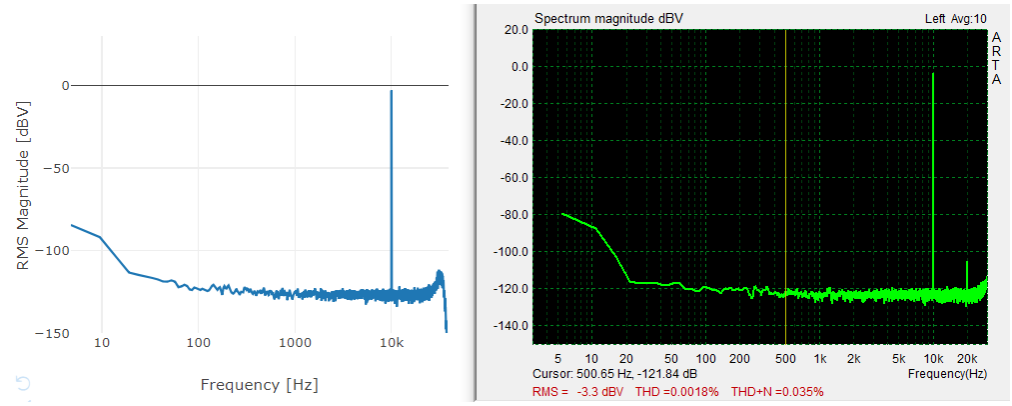


Figure 21: Test signal 10kHz, −3dB Iridium, left and Arta, right.

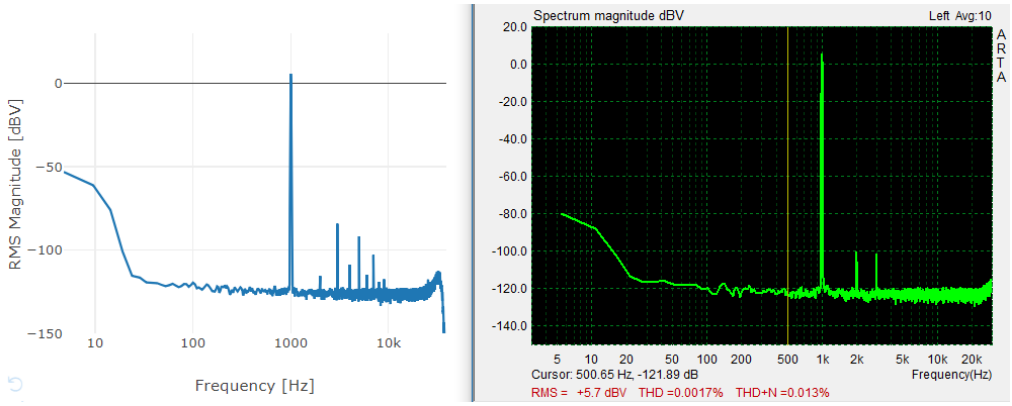


Figure 22: Test signal 1kHz, +6dB Iridium, left and Arta, right.

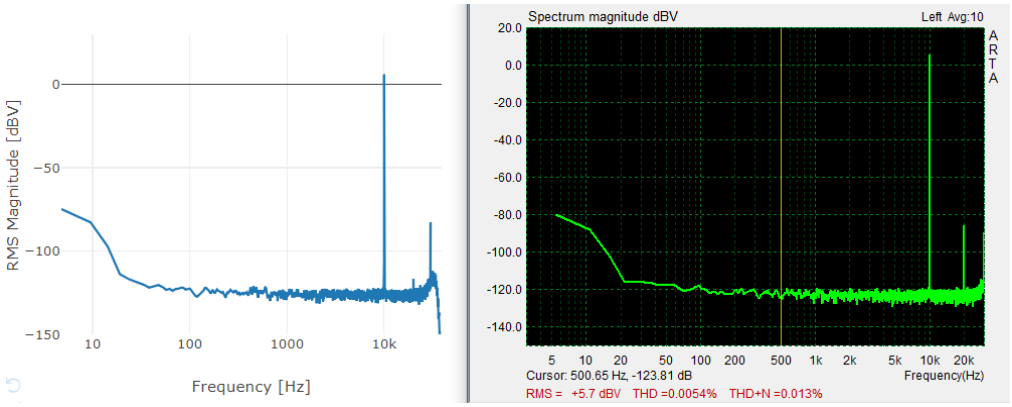


Figure 23: Test signal 10kHz, +6dB Iridium, left and Arta, right.

Frequency	input magnitude	Measured THD [%]	Loopback THD [%]
1 kHz	−3 dB	0.0002051	0.0048
10 kHz	−3 dB	0.0003415	0.0018
1 kHz	+6 dB	0.003334	0.0017
10 kHz	+6 dB	0.003509	0.0054

Table 9: Results of distortion test, Measured THD is the distortion calculated from Iridium ADC measurements, Loopback THD is the distortion of the Focusrite interface in loopback connection as measured by ARTA.

In most cases the distortion measured by Iridium ADC is lower than with the Focusrite interface. Even better equipment is required to characterize the system to a high degree of accuracy. The one exception is at 1 kHz with a +6 dB which displays noticeably different distortion, indicating that further investigation into the cause of this may be required.

8 Conclusion.

The Iridium ADC project is, overall, a success. Most of the performance specifications are met or exceeded. The distortion and noise performance in particular exceeds the target goal by a significant margin. There is, however, much that can be improved upon in a new revision.

The update rate is currently around 2 seconds for the maximum sample rate. This is a limitation of transferring the data using UART over a USB virtual serial port. This transfer method was easy to set up in both the embedded system and the host application but is very limited in speed. It is also not very robust; there is not error detection and there may be potential issues with buffer overflow or underflow that could result in bad data. This is the most pressing issue currently because it is a barrier to several of the other improvements that could be made.

Increasing the number of samples would have several benefits, including better frequency resolution and lower minimum frequencies. There are two limitations to this currently. The first is the aforementioned update rate. The transfer time scales approximately equally with the number of samples, so attempting to take four times the number of samples would result in a painfully slow 8 second update rate. The second limitation is system memory. 16384 is the maximum number of samples because it is the largest power of two that fits in memory. Use of external memory could alleviate this issue.

Currently only a small portion of the ADC sample rate capability is utilized. Two barriers are in the way of increased sample rates. Currently the ADC signals a sample is ready by pulling a pin low. This triggers and interrupt in the microcontroller that reads the data. There is around 1 us of latency between the pin pulling low and the data being read. A poll based approach reduces this lag considerably, but the pulse does not last long enough and is sometimes samples are missed entirely. External circuitry to extend the time the pin stays low could enable the poll based approach and allow a doubling of the sample rate. Additional speed could be achieved if all 16 data bus pins are connected on a single port. This may be possible with the current Nucleo F446RE development board, but it will require some modification of the Nucleo board to free up all of Port A. This reduces a read cycle from 4 port reads to two, saving several hundred nanoseconds.

A PCB redesign will be necessary. Several problems were encountered with traces near the ADC. Major problems occurred due to traces being run under components. Cycles of heating and cooling and mechanical abrasion during assembly caused solder mask to wear off and allowed shorts to develop. These traces can be rerouted easily if a switch to a four layer board is made.

9 References.

[1] Keeping, S. "Design Trade-offs when Selecting a High-Frequency Switching Regulator" Digikey article library. [**https://www.digikey.com/en/articles/techzone/2015/feb/design-trade-offs-when-selecting-a-high-frequency-switching-regulator]**

[2] "High Frequency Vibration Analysis" Emerson Reliability Solutions [**https://www.emerson.com/documents/automation/high-frequency-vibration-analysis-en-39104.pdf]**

[3] William, T. "Fundamentals of Audio Test." Audio Precision. [**https://www.ap.com/download/fundamentals-of-audio-test-2/]**

[4] Horowitz, P. & Hill, W. *Art of Electronics*., 3rd Ed. New York: Cambridge University Press, 2016.

[6] DellaSala, G. "New Audio Precision Two-Channel APx515 Audio Analyzer." *Audioholics*. June 14, 2010. [**http://www.audioholics.com/news/audio-precision-apx515]**

[7] Prism Sound. "dScope product page." [**http://www.prismsound.com/test_measure/products_subs/dscope/dscope_spec.php]**

[8] Test Measurement World Staff. "Prism Sound slashes price of audio analyzer." EDN Network. December 29, 2008. [**https://www.edn.com/design/test-and-measurement/4378395/Prism-Sound-slashes-price-of-audio-analyzer]**

[9] audioXpress staff, "Avermetrics Makes High-Performance Audio Testing Affordable with New AverLAB Audio Analyzer." audioXpress. June 12, 2017. [**http://www.audioxpress.com/news/avermetrics-makes-high-performance-audio-testing-affordable-with-new-averlab-audio-analyzer]**

[10] Avermetrics. "Averlab Product Page." **https://www.avermetrics.com/products/averlab/**

[11] LTspice: Stability of Op Amp Circuits. Analog Devices. [**http://www.analog.com/en/education/education-library/videos/5579254320001.html]**

[12] Keysight Technologies. "Trueform waveform generators data sheet." [**https://literature.cdn.keysight.com/litweb/pdf/5992-2572EN.pdf?id=2937598]**

10 Appendices

10.1 Appendix A: Analysis of Senior Project.

Project Title: Iridium ADC

Student's Name: Bill Blakely

10.1.1 Summary of Functional Requirements.

This project is intended to be used for scientific and engineering measurement of highly dynamic signals. It features a differential input that is sampled with a precision ADC. The digital data is then sent to a computer over a USB connection where it is processed and displayed by a companion program. The PC based program allows the user to view the signal in time and frequency domain simultaneously. It also provides control over the sample rate, sample length, windowing function, and frequency domain magnitude averaging.

10.1.2 Primary Constraints.

This device is intended to be a very high performance device in regard to noise and distortion. Extreme care must be taken to preserve as much of the linearity and dynamic range as possible. Digitally, high-resolution, high-bandwidth signals result in a large number of bits. The digital controller and USB bridge must be specified and correctly implemented to handle the amount of data that is expected to be pushed through the channel. The user mode program must be intuitive and easy to use.

10.1.3 Economics.

10.1.3.1 Impacts.

Human Capital: This device is intended to fit into the relatively low volume market of professional test equipment. This creates the potential for jobs in development and manufacturing, as well as in support. The device seeks to increase the level of frequency domain analysis done in the space of low frequency signals, and therefore increases the knowledge and capability of engineers and researchers all around the world.

Financial Capital: Initial development is to be self-funded. Moving beyond the prototyping stage will require seeking out partners, investors, and venture capitalists willing to fund the project.

Manufactured or Real Capital: Serious development of this project into a company will require investment to acquire a suitable suite of test and measurement equipment. It will also require that the manufacturing facilities and office space are acquired.

Natural Capital: The device has similar environmental impact to much other lab equipment such as bench multimeters. Effort to reduce the natural capital can be made such as use of lead-free parts and sustainable energy used in office and manufacturing facilities. The display-less design and power management help to reduce the power consumption of the device. The best way to prevent wasted natural capital in this device is to create a robust, dependable product that will not need to be replaced due to performance limitations or breakage.

10.1.3.2 Cost and Benefits.

During prototype development stage, by far the largest associated cost is time. Large amounts of time must be put into the research of the techniques and methods of high performance analog design, parts selection, code development, careful PCB layout, and more. The only significant capital expense is in the parts and PCB manufacture, no additional special equipment is required for development of the prototype.

The total BOM cost comes to 175.34

A condensed version of the BOM is presented in **table 10**.

Part	Approximate Cost ($)
ADC	57
Microcontroller dev board	15
Precision op amps	20
Power Supply and reference	20
Connectors	12
Passives (est. avg. cost)	38
PCB	15

Table 10: Condensed BOM showing the cost breakdown of the project.

10.1.4 Manufacturing.

Initial estimated volume is expected to be low. The value to customers is high and this will justify a retail price of around 5x the BOM cost. Using the lower cost estimate from **table 10** this will result in a retail price of around $900. With a direct sales model and estimated first year volume of 100 units, this is a yearly revenue of $72,500.

10.1.5 Environmental.

Environmentally friendly products start in the design phase. This device is designed to be durable and should not need to be replaced for a long time. Additional environmental impact reduction will be had by seeking out environmentally friendly manufacturers. Reputable PCB vendors that are careful with the hazardous chemicals used in production. Ideally, we would use a manufacturing plant that makes use of sustainable energy sources to power their facilities.

10.1.6 Manufacturability.

Units have several things that make manufacturing challenging. Each unit is expected to be tested and calibrated individually to assure high accuracy, precision, and performance. Aside from this, there is little that is especially challenging about the device. It reduces to PCB assemblies in a box, which is something that manufacturers have gotten very good at doing. While a custom enclosure may be desired, it is not critical to the function, so that the only actually unique part of the unit is the PCB assembly.

10.1.7 Sustainability.

The product is designed to have a long lifetime. The anticipated support issues if produced are repairs and recalibration. The core performance of the hardware currently exceeds the performance of the software, so it is possible that units can be software upgradeable to unlock additional performance. This can extend the product lifetime therefore saving natural resources.

10.1.8 Ethical.

There is an ethical obligation to design and manufacture a device that does not incorporate planned obsolescence. It must also be designed for safe operation, especially the power supply. Perhaps most critically for this device is to publish the performance specifications as honestly and completely as possible.

10.1.9 Health and Safety.

The device poses little direct health risk. There are no dangerous voltages present or moving mechanical parts. The indirect health dangers are byproducts of manufacturing. Use of lead free solder and environmentally friendly manufacturing processes are important to minimize the long term impact of any electronic device.

10.1.10 Social and Political.

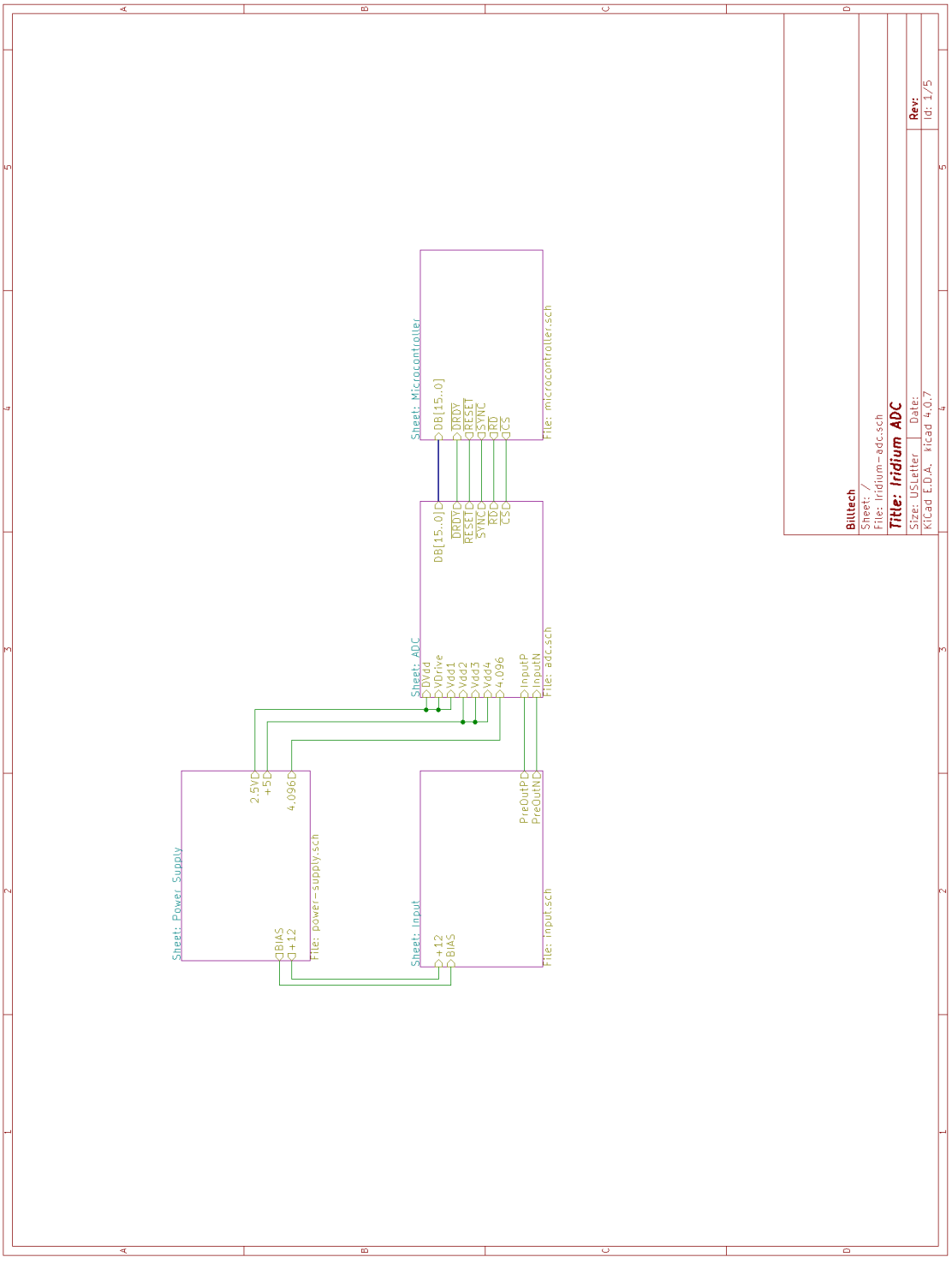
The audio market has a long history of rejecting measurement and characterization. To some extent, this is changing. Most purchasing is done online and because devices can not be heard as beforehand, people have turned to test data to inform their purchasing decisions. Currently small audio companies do not have a cost effective way to accurately

characterize their devices. This has the potential to influence the way that audio companies publish their performance characterization tests.

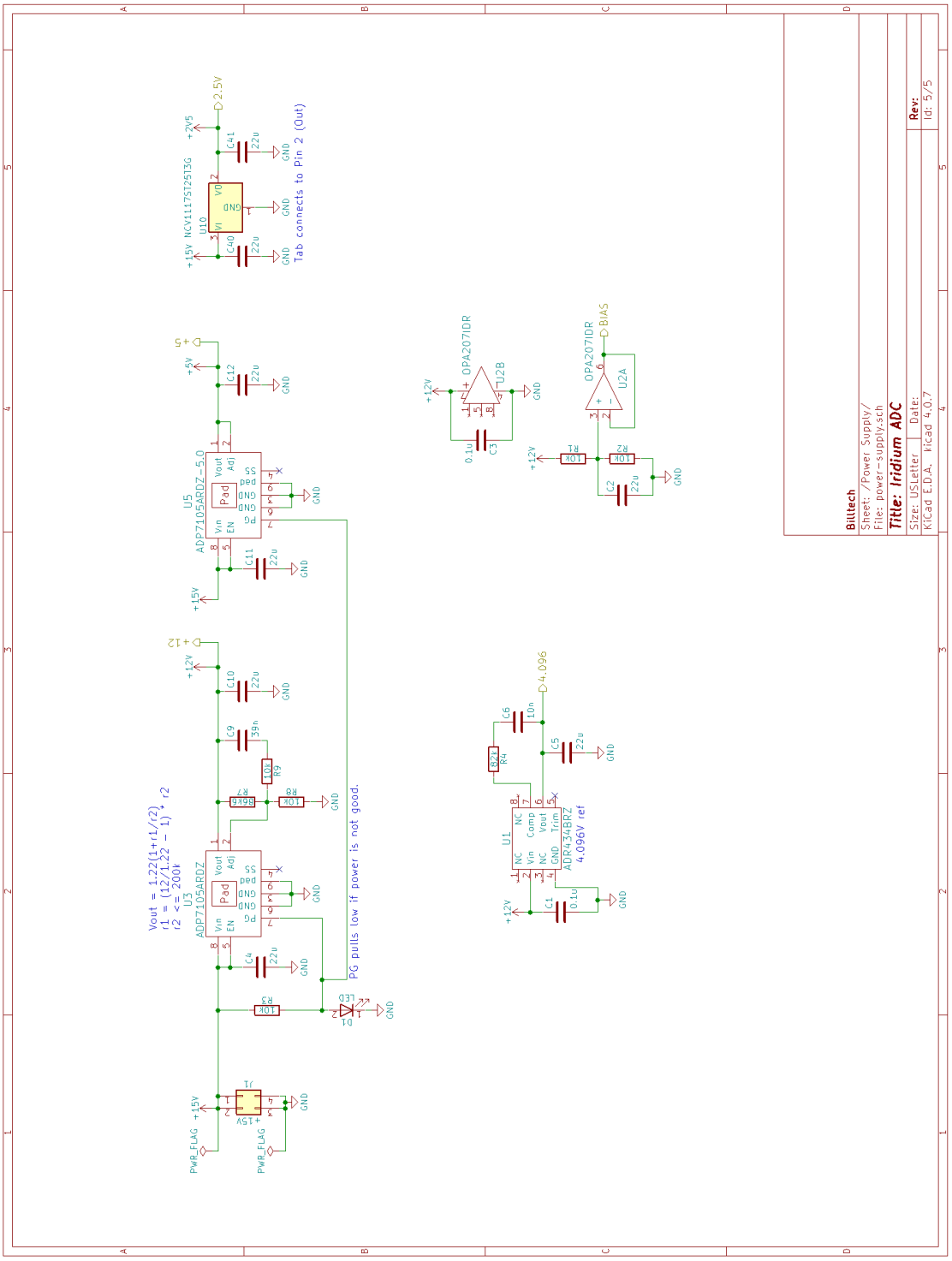
10.1.11 Development.

Development of this project utilized a diverse skillset. The project mixed high performance analog, embedded system design, and a PC application together into one project. This required filter design, noise analysis, schematic capture, PCB layout, C++ and python development, as well as an understanding of the FFT and frequency domain analysis. This tied together many separate interests into one comprehensive package. The steepest learning curve was in the PCB layout. The ADC package, a 64 pin QFP, 0.5 mm pin pitch, and exposed paddle proved to be quite a challenge to layout and assemble. In a new design, moving to a four layer PCB, not routing any traces underneath the package, and using wider traces and spacing would be highly recommended.

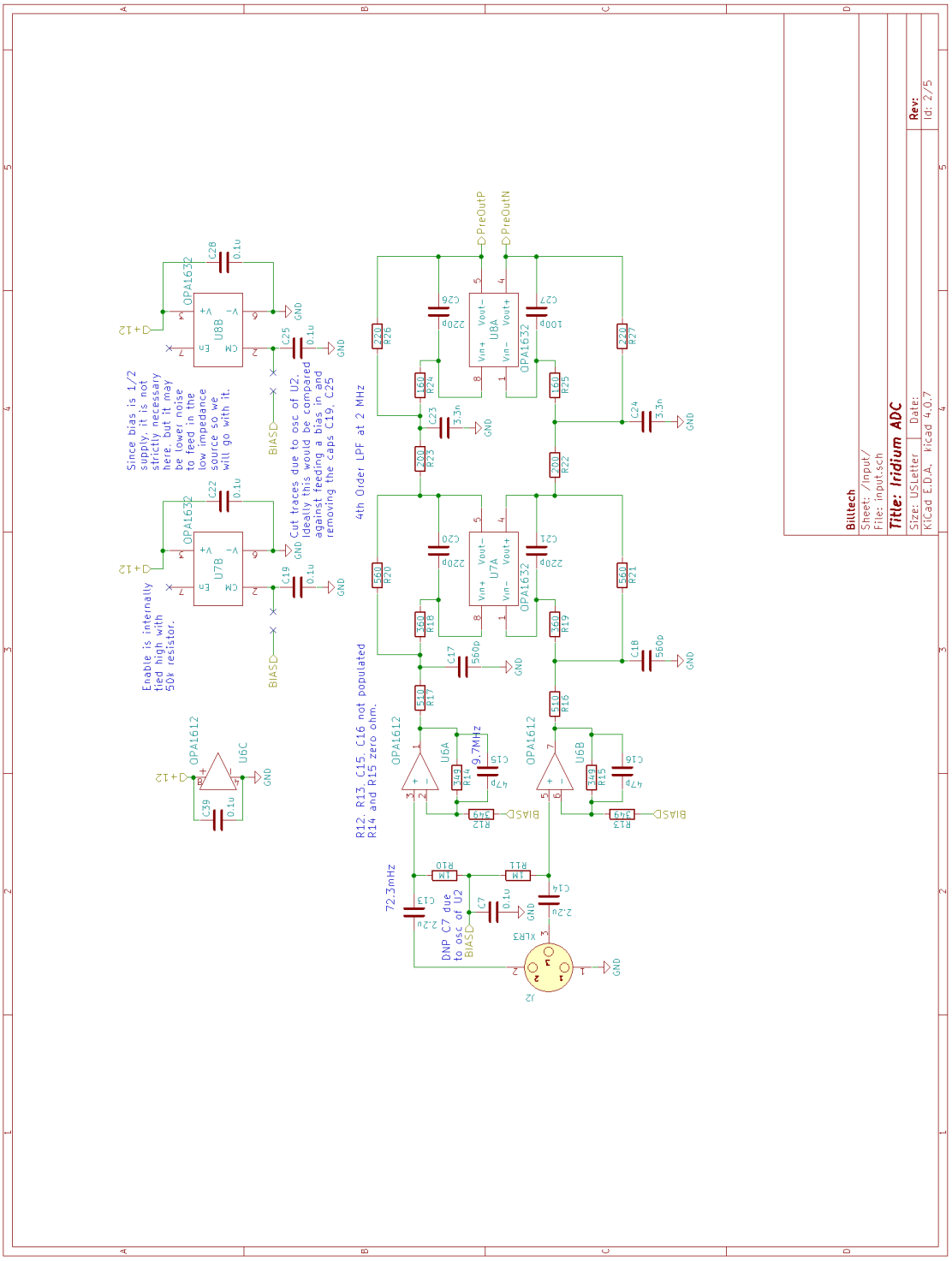
10.2 Appendix B: Schematics



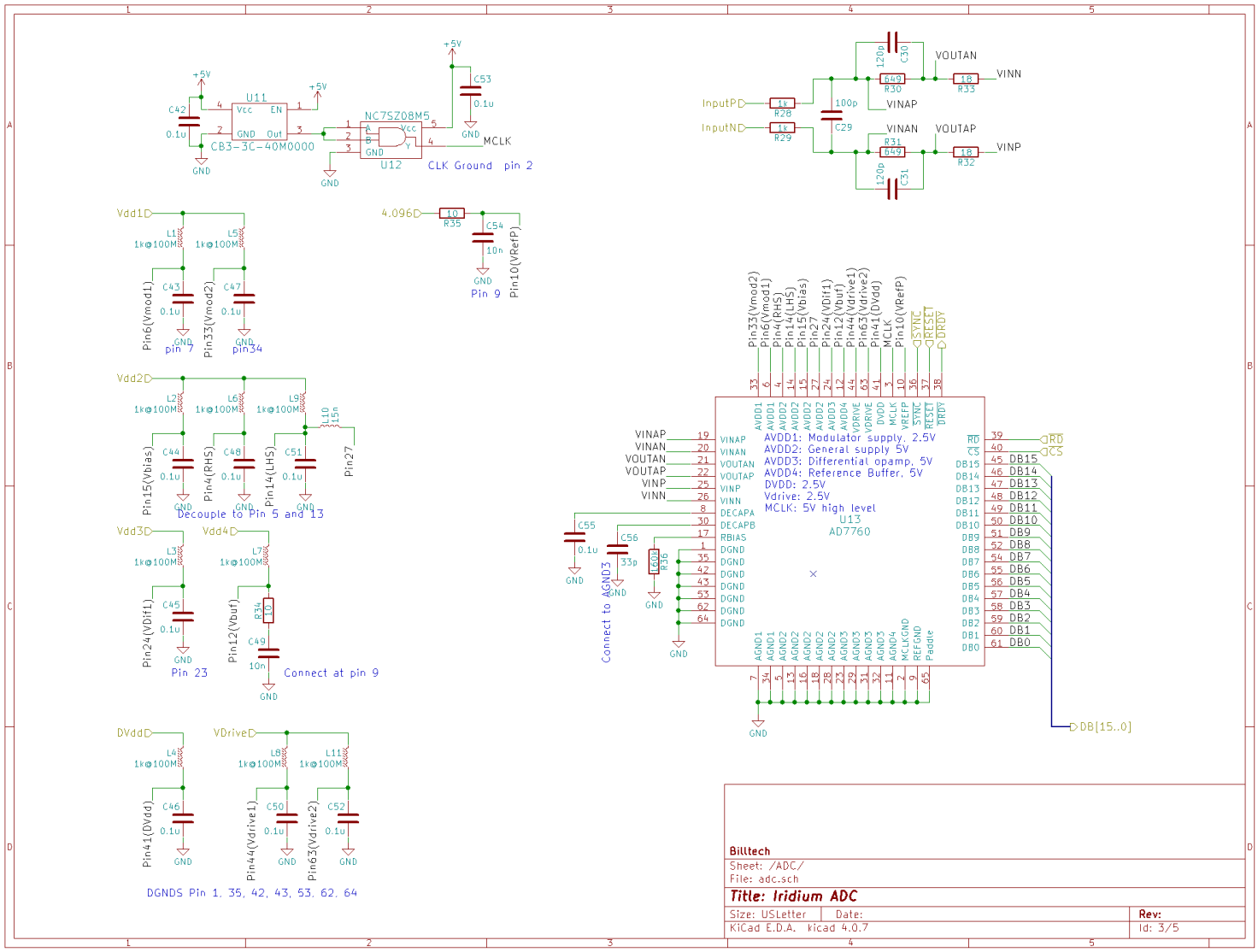
Outer Hierarchy

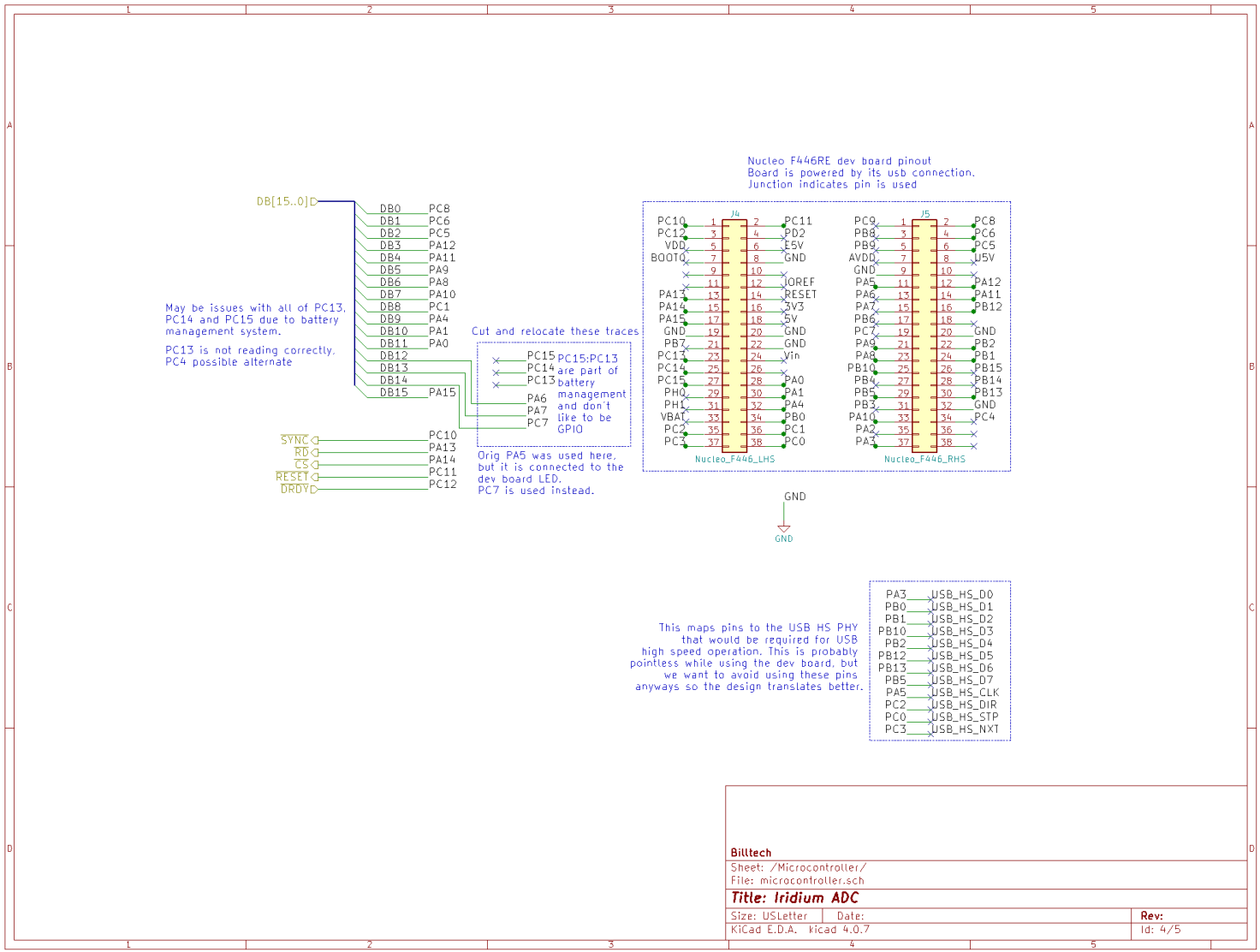


Power supply

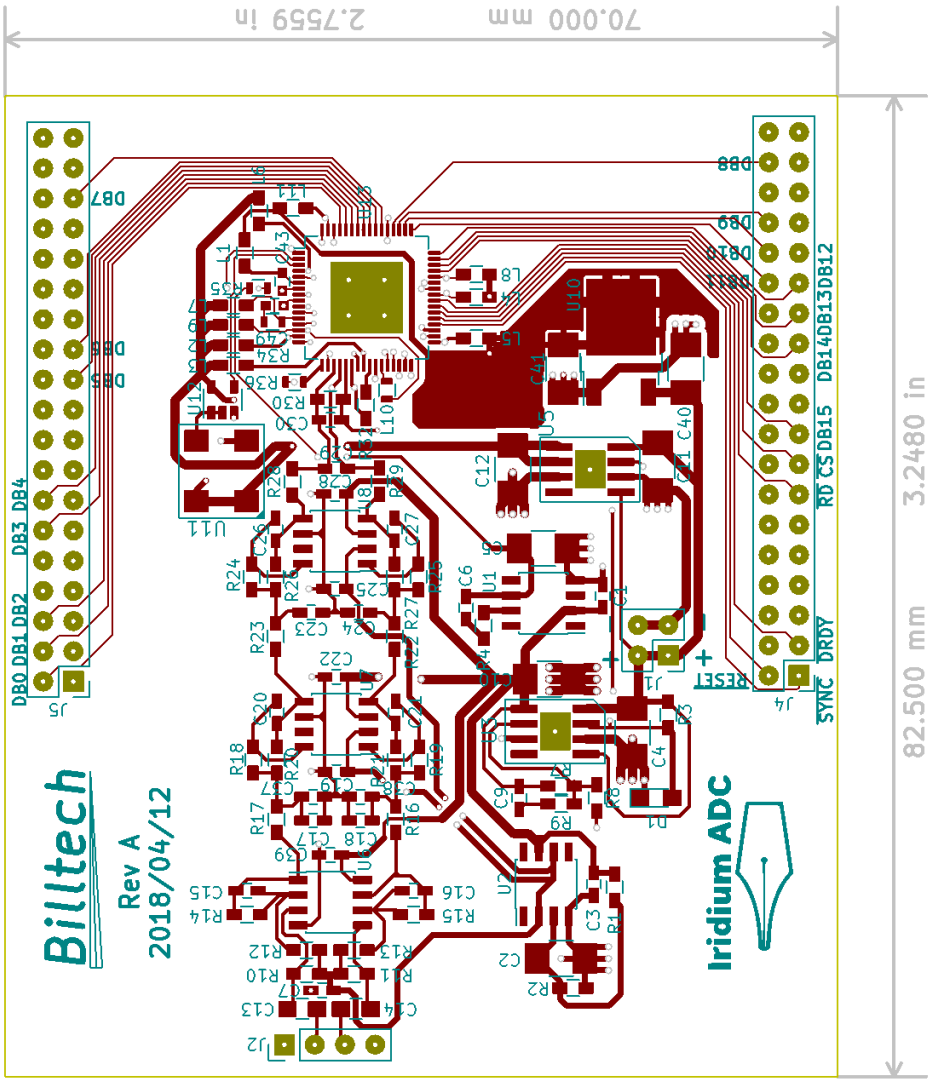

Billtech
Sheet: /input/
File: input.sch
Title: Iridium ADC
Size: USLetter
Date:
Rev:
KiCad E.D.A. KiCad 4.0.7
Id: 2/5
PreOutP
PreOutN
4th Order LPF at 2 MHz
R12, R13, C15, C16 not populated
R14 and R15 zero ohm.
72.3mHz
DNP C7 due
to osc of U2
BIAS
GND
+1.2
OPA1632
OPA1612
9.7MHz
Since bias is 1/2
supply, it is not
strictly necessary
here. but it may
be lower noise
to feed in the
low impedance
source so we
will go with it.
Cut traces due to osc of U2.
Ideally this would be compared
against feeding a bias in and
removing the caps C19, C25
Enable is internally
tied high with
50k resistor.

Input stage

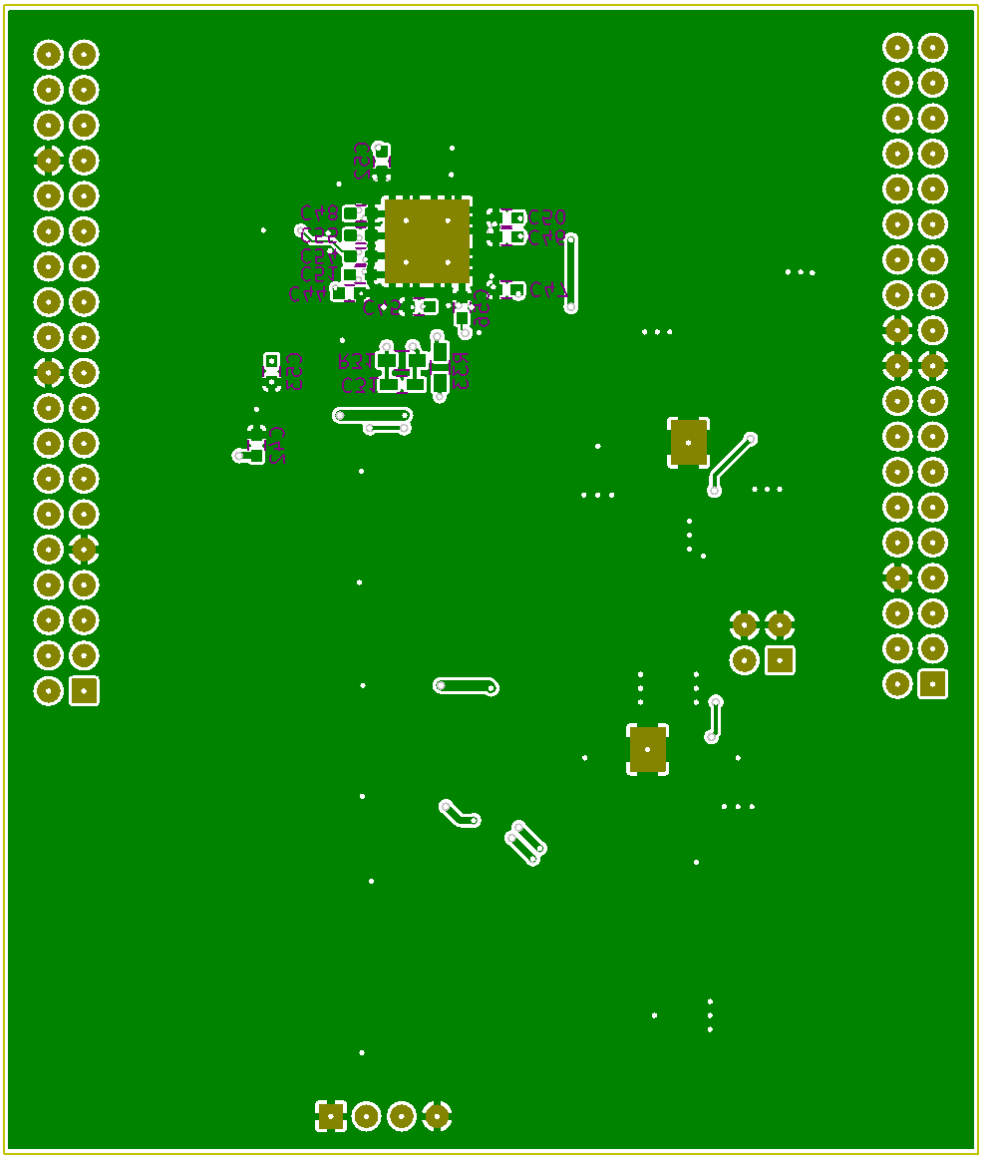

ADC
U11
CB3-3C-40M0000
NC7SZ08M5
U12
CLK Ground pin 2
MCLK
Pin 9
Pin10(VRefP)
4.096
Vdd1
Vdd2
Vdd3
Vdd4
DVdd
VDrive
Pin6(Vmod1)
Pin33(Vmod2)
Pin15(Vbias)
Pin4(RHS)
Pin14(LHS)
Pin27
Pin24(VDif1)
Pin12(Vbuf)
Pin44(Vdrive1)
Pin63(Vdrive2)
Pin41(DVdd)
Decouple to Pin 5 and 13
Pin 23
Connect at pin 9
DGNDS Pin 1, 35, 42, 43, 53, 62, 64
InputP
InputN
VOUTAN
VINN
VINAP
VINAN
VOUTAP
VINP
U13
AD7760
AVDD1: Modulator supply, 2.5V
AVDD2: General supply 5V
AVDD3: Differential opamp, 5V
AVDD4: Reference Buffer, 5V
DVDD: 2.5V
Vdrive: 2.5V
MCLK: 5V high level
Connect to AGND3
DB[15..0]
Billtech
Sheet: /ADC/
File: adc.sch
Title: Iridium ADC
Size: USLetter
Date:
Rev:
KiCad E.D.A. kicad 4.0.7
Id: 3/5


Microcontroller interface
DB[15..0]
DB0 PC8
DB1 PC6
DB2 PC5
DB3 PA12
DB4 PA11
DB5 PA9
DB6 PA8
DB7 PA10
DB8 PC1
DB9 PA4
DB10 PA1
DB11 PA0
DB12
DB13
DB14
DB15 PA15
May be issues with all of PC13, PC14 and PC15 due to battery management system.
PC13 is not reading correctly. PC4 possible alternate
Cut and relocate these traces
PC15
PC14
PC13
PA6
PA7
PC7
PC15:PC13 are part of battery management and don't like to be GPIO
Orig PA5 was used here, but it is connected to the dev board LED. PC7 is used instead.
SYNC PC10
RD PA13
CS PA14
RESET PC11
DRDY PC12
Nucleo F446RE dev board pinout
Board is powered by its usb connection.
Junction indicates pin is used
J4
Nucleo_F446_LHS
J5
Nucleo_F446_RHS
GND
This maps pins to the USB HS PHY that would be required for USB high speed operation. This is probably pointless while using the dev board, but we want to avoid using these pins anyways so the design translates better.
PA3 USB_HS_D0
PB0 USB_HS_D1
PB1 USB_HS_D2
PB10 USB_HS_D3
PB2 USB_HS_D4
PB12 USB_HS_D5
PB13 USB_HS_D6
PB5 USB_HS_D7
PA5 USB_HS_CLK
PC2 USB_HS_DIR
PC0 USB_HS_STP
PC3 USB_HS_NXT
Billtech
Sheet: /Microcontroller/
File: microcontroller.sch
Title: Iridium ADC
Size: USLetter
Date:
Rev:
KiCad E.D.A. kicad 4.0.7
Id: 4/5

10.3 Appendix C: PCB.



PCB front side: copper and silkscreen.



PCB back side: copper and silkscreen.

10.4 Appendix D: Bill of Materials.

Qty	Line Item	Manufacturer	Schematic Reference	Description	Unit Price	Line Total
20	GRM188R71E104MA01D	Murata	C1 C3 C7 C19 C22 C25 C28 C39 C42 C43 C44 C45 C46 C47 C48 C50 C51 C52 C53 C55	0.1u 10% 0603 X7R 100V		
10	GRM32ER61E226KE15L	Murata	C2 C4 C5 C10 C11 C12 C40 C41	22u 1210 X5R	1.566	15.66
3	GRM1885C1H103JA01D	Murata	C6 C49 C54	10n 5% C0G 0603 50V	0.24	0.72
1	GRM188R71H393KA61D	Murata	C9	39n 10% 0603 X7R 50V	0.19	0.19
2	GRM21BR71A225KA01L	Murata	C13 C14	2.2u 0805 X7R	0.28	0.56
2	GRM1885C1H470JA01D	Murata	C15 C16	47p 5% 0603 C0G 50V	0.11	0.22
2	GQM1885C1H221*	Murata	C17 C18	220p 2% C0G 0603 50V	0.2	0.4
5	GQM1885C1H101JB01D	Murata	C20 C21 C26 C27 C29	100P 5% C0G 0603 50V	0.82	4.1
2	GRM1885C1H152*	Murata	C23 C24	1.5n 5% C0G 0603 50V	0.29	0.58
2	GRM1885C1H121JA01D	Murata	C30 C31	120P 5% C0G 0603 50V	0.15	0.3
2	GQM1885C1H200GB01D	Murata	C37 C38	20p 2% C0G 0603 50V	0.92	1.84
1	GQM1885C1H330GB01D	Murata	C56	33p 0603 C0G 2%	0.92	0.92
1	LTST-C170KFKT	Lite-On	D1	LED orange 0805	0.31	0.31
1	AC3FDZ	Amphenol	J2	XLR Input Panel Mount Female	3.5	3.5
1	AC3MM	Amphenol	J2B	XLR Conn Cable mount Male	3.45	3.45
2	S7122-ND	Sullins	J4 J5	header 2×19 female 0.1" pitch	2.26	4.52
10	74279266	Würth Elektronik	L1 L2 L3 L4 L5 L6 L7 L8 L9 L11	Ferrite Bead almost 0603 package	0.17	1.70
1	LQP03TN15NJ02D	Murata	L10	Inductor 15nH 0603	0.1	0.1
5	RR08P10.0KDCT-ND	Susumu	R1 R2 R3 R8 R9	10k 0.5% 0603 Thin Film ORDERED 100K BY MISTAKE	0.11	0.55
1	RR08P82.0KDCT-ND	Susumu	R4	82kR 0.5% 0603 Thin Film	0.11	0.11
1	311-2390-1-ND	Yageo	R7	86k6R 0.5% 0603	0.12	0.12
2	A102234CT-ND	TE Connectivity	R10 R11	1MR 0.1% 0603	0.44	0.88
4	P348DBCT-ND	Panasonic	R12 R13 R14 R15	348R 0.1% 0603	0.35	1.4
4	YAG4645CT-ND	Yageo	R16 R17 R20 R21	1.05kR 0.1% 0603	0.33	1.32
2	RR08P698DCT-ND	Susumu	R18 R19	698R 0.5% 0603	0.11	0.22
2	RR08P365DCT-ND	Susumu	R22 R23	365R 0.5% 0603	0.11	0.22
2	RR08P332DCT-ND	Susumu	R24 R25	332R 0.5% 0603	0.11	0.22
2	RR08P220DCT-ND	Susumu	R26 R27	220R 0.5% 0603 Thin Film	0.11	0.22
2	RR08P1.0KDCT-ND	Susumu	R28 R29	1kR 0.5% 0603 Thin Film	0.11	0.22
2	311-649DCT-ND	Yageo	R30 R31	649R 0.5% 0603 Thin Film	0.12	0.24
2	RR08Q18DCT-ND	Susumu	R32 R33	18R 0.5% 0603 Thin Film	0.11	0.22
2	RNCP0603FTD10R0CT-ND	Stackpole Electronics	R34 R35	10R 1% 0603 Thin Film	0.1	0.2
1	RR08P160KDCT-ND	Susumu	R36	160kR 0.5% 0603 ORDERED 16K BY MISTAKE	0.11	0.11

Continued from previous page

Qty	Line Item	Manufacturer	Schematic Reference	Description	Unit Price	Line Total
1	ADR434BRZ	Analog Devices	U1	Reference 4.096 V	9.81	9.81
1	296-47934-ND	Texas Instruments	U2	Op amp: Low-Power High-Precision Low-Noise Rail-to-Rail Output	2.34	2.34
1	ADP7104ARDZ-R7	Analog Devices	U3	regulator low noise adjustable	4.97	4.97
1	ADP7104ARDZ-5.0-R7	Analog Devices	U5	Regulator low noise fixed 5v	4.97	4.97
1	OPA1612AID	Texas Instruments	U6	Opamp 1.1nV/rtHz 0.15 ppm distortion	6.91	6.91
2	OPA1632DR	Texas Instruments	U7 U8	Differential opamp amp with 1.3nV 0.4 pA /rt(HZ) noise < 3ppm THD < 1ppm IMD	5.36	10.72
1	NCV1117DT25RKG	ON Semiconductor	U10	2.5V Lin Reg DPAK	0.84	0.84
1	CB3-3C-40M0000	CTS Components	U11	40MHz osc 5V out SMD 1ps RMS jitter	1.52	1.52
1	NC7SZ08M5	ON Semiconductor	U12	AND gate for clock buffer	0.62	0.62
1	AD7760BSVZ	Analog Devices	U13	24 bit 2.5 MSPS delta sigma Parallel output	57.27	57.27
1	NUCLEO-F446RE	STMicroelectronics		STM32 Nucleo Dev Board F446RE	14.9	14.9
1	Iridium PCB	BillTech		OSH Park 3 @ $14.97 per board	14.97	14.97

10.5 Appendix E: Program Listing.

10.5.1 Embedded.

10.5.1.1 main.h

```
#ifndef MAIN_H
#define MAIN_H

#include "mbed.h"
#include "adc.h"
#include "communications.h"

int main();

#endif
```

10.5.1.2 main.cpp

```
#include "main.h"

int main()
{
    usb_serial.baud(1500000);

    // attach the serial interrupt. These signals are handled in
    // communications.cpp by the control signals function.
    usb_serial.attach(&control_signals);
    while(1)
    {

        if(data_ready == 1)
        {
            // Send the data array.
            data_tx();
            // Reset the flag
            data_ready = 0;
        }
    }
    printf("ERROR: main loop ended somehow.\n");
}
```

10.5.1.3 communications.h

```
#ifndef COMMUNICATIONS_H
#define COMMUNICATIONS_H

#include <stdint.h>
#include <string.h>
#include "mbed.h"

#include "adc.h"
#include "pins.h"

extern Serial usb_serial;

void control_signals();

void data_tx();

#endif
```

10.5.1.4 communications.cpp

```
#include "communications.h"

Serial usb_serial(USBTX, USBRX);

volatile int number_of_samples_to_TX = 0;

void control_signals()
{
    switch (usb_serial.getc())
    {
        case 'P':
        case 'p':
        {
            // prepare the adc.
            adc.setup();
            break;
        }
        case 'R':
        case 'r':
        {
            // Check status register
            // printf("%d samples, %dx decimation\n", NUMBER_OF_SAMPLES, 1<<decimation_rate); adc.read_statu
#include "mbed.h"
#include "pins.h"
#include "communications.h"

#define MCLK_FREQ 40000000

#define HIGH 1
#define LOW 0


#define DEFAULT_DECIMATION_RATE 0x5
// Decimation rate table:
// | Binary | Rate |   BW   | Output Data Rate |
// |--------|------|--------|------------------|
// | 0b000  |  1x  |   1M   |       2.5 MSPS   |
// | 0b001  |  2x  |  500k  |      1.25 MSPS   |
// | 0b010  |  4x  |  250k  |       625 kSPS   |
// | 0b011  |  8x  |  125k  |     312.5 kSPS   |
```

```
// | 0b100  | 16x  | 62.5k  |    156.25 kSPS   |
// | 0b101  | 32x  | 31.25k |    78.125 kSPS   |

// 16384 is the maximum power of two that can fit in memory. We are always sampling the highest number of sa
// only transmit a subset because the transmission takes much much longer than the sampling.
#define NUMBER_OF_SAMPLES 16384


class ADC_Class
{
public:
    ADC_Class();
    void static setup();
    void static power_up();
    void static power_down();
    void static start_sampling();
    void static stop_sampling();
    void static change_decimation_rate(int multiplier);
    uint16_t static read_status_register(bool print_to_console);
    uint16_t static read_offset_register(bool print_to_console);
    uint16_t static read_gain_register(bool print_to_console);
    uint16_t static read_overrange_register(bool print_to_console);
    void static collect_samples();

// private:
    uint16_t static control_reg_1_state;
    uint16_t static control_reg_2_state;

    uint16_t static read_adc_reg(uint8_t offset);
    uint32_t static read_data_word();
    void static write_control_register(uint16_t control_register, uint16_t value);
    void static wait_4_MCLK_cycles();

};

extern ADC_Class adc;
extern volatile int data_ready;
extern volatile uint32_t sample_array[];
#endif
```

10.5.1.5 adc.cpp

```
#include "adc.h"

uint16_t ADC_Class::control_reg_1_state = 0x0000;
uint16_t ADC_Class::control_reg_2_state = 0x0000;

ADC_Class adc;

volatile uint32_t busy_wait_variable;
volatile uint32_t sample_array[NUMBER_OF_SAMPLES];
volatile int data_ready = 0;

ADC_Class::ADC_Class()
{
    dataBus.input();
    notSync = HIGH;
    notReset = HIGH;
    notChipSelect = HIGH;
    notRead = HIGH;

    // reset to start in a defined state.
    notReset = LOW;
```

```
    wait_4_MCLK_cycles();
    notReset = HIGH;
    wait_4_MCLK_cycles();
    power_down();
}

void ADC_Class::setup()
{

    // configure pins for write operations.
    // dataBus is bi directional.
    dataBus.output();

    // Control Register 2
    // // BIT | NAME  | DESCRIPTION
    // // ----|------ |-------------
    // //  5  | ~CDIV | Clock Divider Bit. This sets the divide ratio of the MCLK signal to produce the inte
    // //  3  | PD    | Power Down. Setting this bit powers down the AD7760, reducing the power consumption
    // //  2  | LPWR  | Low Power. If this bit is set, the AD7760 is operating in a low power mode. The powe
    // //  1  | 1     | Write 1 to this bit.
    // //  0  | D1PD  | Differential Amplifier Power Down. Setting this bit powers down the on-chip differer

    // // Set lowpower mode for improved noise performance.
    control_reg_2_state = (0x0000 | 1 << 2 | 1 << 1);
    write_control_register(0x0002, control_reg_2_state);

   // Control Register 1:
    // | BIT    | NAME      | DESCRIPTION
    // | 15     | DL_FILT   | Download Filter. Before downloading a user-defined filter, this bit must be se
    // | 14     | RD_OVR    | 2 Read Overrange. If this bit has been set, the next read operation outputs th
    // | 13     | RD_GAIN   | 2 Read Gain. If this bit has been set, the next read operation outputs the cor
    // | 12     | RD_OFF    | 2 Read Offset. If this bit has been set, the next read operation outputs the c
    // | 11     | RD_STAT   | 2 Read Status. If this bit has been set, the next read operation outputs the c
    // | 10     | 0         | 0 must be written to this bit.
    // | 9      | SYNC      | Synchronize. Setting this bit initiates an internal synchronization routine. S
    // | 8 to 5 | FLEN[3:0] | Filter Length Bits. These bits must be set when the DL_FILT bit is set before
    // | 4      | ~BYP_F3   | Bypass Filter 3. If this bit is 0, Filter 3 (programmable FIR) is bypassed.
    // | 3      | ~BYP_F1   | Bypass Filter 1. If this bit is 0, Filter 1 is bypassed. This should only occu
    // | 2 to 0 | DEC[2:0]  | Decimation Rate. These bits set the decimation rate of Filter 2. All 0s implie
    //
    // Set for 2.5MHZ output data rate (enable Filt 1 & 3, No Decimation).
    // In testing a lower rate may be required, and 0b11 == 3 should be included
    // to enable 8x decimation, lowering data rate to 312.5 kHz.
    control_reg_1_state = (0x0000 | 1<<4 | 1<<3 | DEFAULT_DECIMATION_RATE);
    write_control_register(0x0001, control_reg_1_state);

    // These might be redundant, but this will
    // guarantee a defined state for all pins.
    dataBus.input();
    notRead = HIGH;
    notChipSelect = HIGH;
    notSync = HIGH;
    notReset = HIGH;

    power_down();
}

void ADC_Class::power_up()
{
    write_control_register(0x0002, (control_reg_2_state & ~(1 << 3)));
    // worst case filter latency is about 350 us.
    wait_ms(100);
}
```

```
void ADC_Class::power_down()
{
    write_control_register(0x0002, (control_reg_2_state | (1 << 3)));
}

void ADC_Class::start_sampling()
{
    power_up();
    notDataReady.rise(&collect_samples);
    notDataReady.enable_irq();
}

void ADC_Class::stop_sampling()
{
    notDataReady.disable_irq();
    // Save power, and power down when not being used.
    power_down();
}


void ADC_Class::change_decimation_rate(int multiplier)
{
    power_up();
    control_reg_1_state = ((control_reg_1_state & 0xFFF8) | multiplier);
    write_control_register(0x0001, control_reg_1_state);
    power_down();
}

uint16_t ADC_Class::read_status_register(bool print_to_console)
{
    uint16_t status_reg = 0;

    power_up();
    status_reg = read_adc_reg(11);
    power_down();
    if (print_to_console)
    {
        // Pretty print the received status register to stdout.
        printf("Status Register\n");
        printf("| Part No. | Die No. | Low Pwr | Overrange | Download O.K. | User Filt O.K. | User Filt EN |
        printf("|     %d    |    %d    |    %d    |     %d     |       %d       |       %d        |      %d
                (status_reg & 0xC000) >> 14, //Part No.
                (status_reg & 0x3800) >> 11, // Die No.
                (status_reg & 0x0200) >> 9,  // Low Pwr
                (status_reg & 0x0100) >> 8,  // Overrange
                (status_reg & 0x0080) >> 7,  // Download OK
                (status_reg & 0x0040) >> 6,  // User Filt OK
                (status_reg & 0x0020) >> 5,  // User Filt EN
               !(status_reg & 0x0010) >> 4,  // Byp Filt 3
               !(status_reg & 0x0008) >> 3,  // Byp Filt 1
                1 << (status_reg & 0x0007)); // Dec 2:0, Use left shift to do exponentiation.
    }

    return status_reg;
}

uint16_t ADC_Class::read_offset_register(bool print_to_console)
{
    uint16_t status_reg = 0;

    power_up();
    status_reg = read_adc_reg(12);
    power_down();
    if (print_to_console)
```

```
    {
        printf("%04x\n", status_reg);
    }
    return status_reg;
}

uint16_t ADC_Class::read_gain_register(bool print_to_console)
{
    uint16_t status_reg = 0;

    power_up();
    status_reg = read_adc_reg(13);
    power_down();
    if (print_to_console)
    {
        printf("%04x\n", status_reg);
    }
    return status_reg;
}

uint16_t ADC_Class::read_overrange_register(bool print_to_console)
{
    uint16_t status_reg = 0;

    power_up();
    status_reg = read_adc_reg(14);
    power_down();
    if (print_to_console)
    {
        printf("%04x\n", status_reg);
    }
    return status_reg;
}

void ADC_Class::collect_samples()
{
    // called on interrupt from notDataReady pin
    static int32_t sample_index = 0;

    if (sample_index < NUMBER_OF_SAMPLES)
    {
        // this should all happen in under 250 ns for maximum data rate.
        sample_array[sample_index] = ADC_Class::read_data_word();
        sample_index++;
    }
    else
    {
        stop_sampling();
        sample_index = 0;
        data_ready = 1;
    }
}

uint16_t ADC_Class::read_adc_reg(uint8_t offset)
{
    uint16_t adc_reg;

    // Set signals up for writing to adc.
    notRead = HIGH;
    notChipSelect = HIGH;
    dataBus.output();

    // write to Control Register 1.
```

```
    // | BIT    | NAME      | DESCRIPTION
    // | 15     | DL_FILT   | Download Filter. Before downloading a user-defined filter, this bit must be se
    // | 14     | RD_OVR    | 2 Read Overrange. If this bit has been set, the next read operation outputs th
    // | 13     | RD_GAIN   | 2 Read Gain. If this bit has been set, the next read operation outputs the cor
    // | 12     | RD_OFF    | 2 Read Offset. If this bit has been set, the next read operation outputs the c
    // | 11     | RD_STAT   | 2 Read Status. If this bit has been set, the next read operation outputs the c
    // | 10     | 0         | 0 must be written to this bit.
    // | 9      | SYNC      | Synchronize. Setting this bit initiates an internal synchronization routine. S
    // | 8 to 5 | FLEN[3:0] | Filter Length Bits. These bits must be set when the DL_FILT bit is set before
    // | 4      | ~BYP_F3   | Bypass Filter 3. If this bit is 0, Filter 3 (programmable FIR) is bypassed.
    // | 3      | ~BYP_F1   | Bypass Filter 1. If this bit is 0, Filter 1 is bypassed. This should only occu
    // | 2 to 0 | DEC[2:0]  | Decimation Rate. These bits set the decimation rate of Filter 2. All 0s implie
    //

    write_control_register(0x0001, (control_reg_1_state | (1 << offset)));

    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    notChipSelect = HIGH;

    // Reset dataBus to read data, this takes long enough
    // that a explicit pause isn't necessary.
    dataBus.input();
    // The dataBus should now have the status on it.
    notRead = LOW;
    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    // Grab the raw input
    adc_reg = dataBus.read();
    notChipSelect = HIGH;
    notRead = HIGH;

    // put the bit from the raw input into the correct order.
    adc_reg = dataBus.detangle(adc_reg);

    return adc_reg;
}

uint32_t ADC_Class::read_data_word()
{
    static uint16_t MSB_16;
    static uint16_t LSB_16;

    notRead = LOW;
    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    MSB_16 = dataBus.read();
    notChipSelect = HIGH;
    notRead = HIGH;

    wait_4_MCLK_cycles();

    notRead = LOW;
    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    LSB_16 = dataBus.read();
    notChipSelect = HIGH;
    notRead = HIGH;

    // concatenate and return the 32 bit data word.
    return ((MSB_16 << 16) | LSB_16);
}

void ADC_Class::write_control_register(uint16_t control_register, uint16_t value)
```

```
{
    dataBus.output();

    dataBus.write(control_register);
    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    notChipSelect = HIGH;
    wait_4_MCLK_cycles();

    dataBus.write(value);
    notChipSelect = LOW;
    wait_4_MCLK_cycles();
    notChipSelect = HIGH;
    wait_4_MCLK_cycles();

    // Pull the lines down before switching to input
    dataBus.write (0x0000);

    dataBus.input();
}

void ADC_Class::wait_4_MCLK_cycles()
{
    // Calling this function results in a delay of about 100 ns
    // or around 4 cycles of the ADC MCLK at 40 MHz.
    // Without this a high-low-high cycle take about 30 ns ~= 1 MCLK cycle.
    busy_wait_variable = 0;
    busy_wait_variable ++;
}
```

10.5.1.6 pins.h

```
#ifndef PINDEFS_H
#define PINDEFS_H

#include <stdint.h>
#include "mbed.h"

//                  LSB0    1     2     3      4      5     6     7      8     9    10    11    12     13
// #define DATAPINS PC_8, PC_6, PC_5, PA_12, PA_11, PA_9, PA_8, PA_10, PC_1, PA_4, PA_1, PA_0, PA_6, PA_7, P
#define PORT_A_MASK (                 1<<12 |1<<11 |1<<9 |1<<8 |1<<10       |1<<4 |1<<1 |1<<0 |1<<6 |1<<7
#define PORT_C_MASK (1<<8|1<<6 |1<<5                                 |1<<1                            |1

#define HIGH 1
#define LOW 0

class DataBusClass
{
private:
    uint16_t port_a;
    uint16_t port_c;
public:
    DataBusClass();
    void input();
    void output();
    void mode(PinMode mode);
    uint16_t read();
    uint16_t detangle(uint16_t raw_input);
    void write(uint16_t word);
};

extern DataBusClass dataBus;

extern InterruptIn notDataReady;
extern DigitalOut notSync;
extern DigitalOut notRead;
extern DigitalOut notChipSelect;
extern DigitalOut notReset;

// extern DigitalOut led;

#endif
```

10.5.1.7 pins.cpp

```
#include "pins.h"

DataBusClass dataBus;

InterruptIn notDataReady(PC_12);
DigitalOut notSync(PC_10);
DigitalOut notRead(PA_13);
DigitalOut notChipSelect(PA_14);
DigitalOut notReset(PC_11);

// DigitalOut led(LED1);

PortInOut dataBusA(PortA, PORT_A_MASK);
PortInOut dataBusC(PortC, PORT_C_MASK);

// DigitalOut LED()
```

```
DataBusClass::DataBusClass()
{
    port_a = 0;
    port_c = 0;
}

void DataBusClass::input()
{
    dataBusA.input();
    dataBusC.input();
}

void DataBusClass::output()
{
    dataBusA.output();
    dataBusC.output();
}

void DataBusClass::mode(PinMode mode)
{
    dataBusA.mode(mode);
    dataBusC.mode(mode);
}

uint16_t DataBusClass::read()
{
    // efficiently pack the bits into a 16 bit container.
    // bit order is even more messed up than before of course,
    // but that detanglement can be handled on the pc side.
    uint16_t received;

    // leave port a value where they are.
    received = dataBusA.read();

    // Fill in space between port a values (pa02, pa03, pa 13, and pa14 are not used) with the port c values
    port_c = dataBusC.read();
    received |= (port_c & (1 << 1)) << 1;      // pc01 to bit 2
    received |= (port_c & (1 << 8)) >> 5;      // pc08 to bit 3
    received |= (port_c & (1 << 5));           // pc05 to bit 5
    received |= (port_c & ((1 << 6) | (1 << 7))) << 7;      // pc06 to bit 13 and  pc07 to bit 14
    return received;
}

uint16_t DataBusClass::detangle(uint16_t raw_input)
{
    uint16_t decoded_word = 0x0000;

    decoded_word |= (raw_input & (1 << 0))  << (-(0 - 11));   // pa00 to bit 11
    decoded_word |= (raw_input & (1 << 1))  << (-(1 - 10));   // pa01 to bit 10

    decoded_word |= (raw_input & (1 << 2))  << (-(2 - 8));    // pc01 in pos 2 to bit 8
    decoded_word |= (raw_input & (1 << 3))  >> (3 - 0);       // pc08 in pos 3 to bit 0

    decoded_word |= (raw_input & (1 << 4))  << (-(4 - 9));    // pa04 to bit 9

    decoded_word |= (raw_input & (1 << 5))  >> (5 - 2);        // pc05 in pos 5 to bit 2

    decoded_word |= (raw_input & (1 << 6))  << (-(6 - 12));   // pa06 to bit 12
    decoded_word |= (raw_input & (1 << 7))  << (-(7 - 13));   // pa07 to bit 13
    decoded_word |= (raw_input & (1 << 8))  >> (8 - 6);       // pa08 to bit 6
    decoded_word |= (raw_input & (1 << 9))  >> (9 - 5);       // pa09 to bit 5
    decoded_word |= (raw_input & (1 << 10)) >> (10 - 7);      // pa10 to bit 7
    decoded_word |= (raw_input & (1 << 11)) >> (11 - 4);      // pa11 to bit 4
```

```
    decoded_word |= (raw_input & (1 << 12)) >> (12 - 3);      // pa12 to bit 3

    decoded_word |= (raw_input & (1 << 13)) >> (13 - 12);     // pc06 in pos 13  to bit 1
    decoded_word |= (raw_input & (1 << 14)) >> (14 - 14); // pc07 in pos 14 to bit 14

    decoded_word |= (raw_input & (1 << 15)) >> (15 - 15);     // pa15 to bit 15

    return decoded_word;
}

void DataBusClass::write(uint16_t word)
{
    port_a = 0x0000;
    port_c = 0x0000;

    // port,   bit select, shift by (pin num - bit pos)
    port_c |= (word & (1 << 0)) << (8 - 0);       // pc08
    port_c |= (word & (1 << 1)) << (6 - 1);       // pc06
    port_c |= (word & (1 << 2)) << (5 - 2);       // pc05

    port_a |= (word & (1 << 3)) << (12 - 3);      // pa12
    port_a |= (word & (1 << 4)) << (11 - 4);      // pa11
    port_a |= (word & (1 << 5)) << (9 - 5);       // pa09
    port_a |= (word & (1 << 6)) << (8 - 6);       // pa08
    port_a |= (word & (1 << 7)) << (10 - 7);      // pa10

    port_c |= (word & (1 << 8)) >> (-(1 - 8));    // pc01

    port_a |= (word & (1 << 9)) >> (-(4 - 9));    // pa04
    port_a |= (word & (1 << 10)) >> (-(1 - 10));  // pa01
    port_a |= (word & (1 << 11)) >> (-(0 - 11));  // pa00
    port_a |= (word & (1 << 12)) >> (-(6 - 12));  // pa06
    port_a |= (word & (1 << 13)) >> (-(7 - 13));  // pa07

    port_c |= (word & (1 << 14)) >> (-(7 - 14));  // pc07

    port_a |= (word & (1 << 15)) >> (-(15 - 15)); // pa15

    dataBusA.write(port_a);
    dataBusC.write(port_c);
}
```

10.5.2 Host application.

10.5.2.1 pc_data.py

```
import logging
import serial
import io
import dash
import dash_core_components as dcc
import dash_html_components as html
import dash.dependencies as dd
import dash.exceptions as de
from adc_comms import SerialComms

# deques are used for thread safe sharing of data
from collections import deque
import numpy as np
from time import sleep

# Initialization
adc = SerialComms()
```

```
# Initialize various arrays, global arrays are used for preserving data between
# function invocations.
magnitude = [0]
sample_index = [0]
freq_mag = [1e-24]
freq_mag_history = deque(maxlen=1)
freq_axis = [0]
rms_mag = 0
# Define the refresh interval, determined empirically. it would be better if
# this could retrigger on the acquisition loop callback completion, but I can
# only get this to function off a timer so here we are.
update_period_ms = {'1024': 900,
                    '4096': 2000,
                    '8192': 2000,
                    '16384': 2000 }

# This array constructs the window functions used.
# It interacts with the window-functions id in the dash app.
window_map = {}

# Create the dash app object
app = dash.Dash()

# This is the GUI layout.
app.layout = html.Div(id='Body',
    children=
    [
        # Page/Project title
        html.H1(id='title',
            children="Iridium ADC"
        ),

        # Insert a blank line
        html.P(''),

        # Controls to change sampling properties
        html.Div(id='sampling-properties-container',
            style={'width': '40%'},
            children=
            [
                html.P(''),
                html.Label('Sample Rate Selection:'),
                dcc.Dropdown(id='sample-rate',
                    options=
                    [
                        # 'label' is the sample rate, 'value' corresponds to the
                        # corresponding decimation rate register setting.

                        # higher rates are not supported yet
                        # {'label':'2.5 MSPS', 'value': '2500000'},
                        # {'label':'1.25 MSPS', 'value': '1250000'},
                        # {'label':'625 kSPS', 'value': '625000'},
                        {'label':'312.5 kSPS', 'value': '312500'},
                        {'label':'156.25 kSPS', 'value': '156250'},
                        {'label':'78.125 kSPS', 'value': '78125'},
                    ],
                    value='78125',
                ),

                html.P(''),
                html.Label('Number of samples:'),
                dcc.Dropdown(id='number-of-samples',
                    options=
```

```
            [
                {'label': '1024', 'value': '1024'},
                {'label': '4096', 'value': '4096'},
                {'label': '8192', 'value': '8192'},
                {'label': '16384', 'value': '16384'},
            ],
            value='16384',
        ),

        html.P(''),
        html.Label('Window:'),
        dcc.Dropdown(id='window-functions',
            options=
            [
                {'label':'Rect', 'value':'rect'},
                {'label':'Blackman', 'value':'blackman'},
                {'label':'Kaiser 5', 'value':'kaiser5'},
                {'label':'Kaiser 7', 'value':'kaiser7'},
            ],
            value='kaiser5',
        ),

        html.P(''),
        html.Label('Number of averages: '),
        dcc.Input(id='number-of-averages',
            placeholder='Enter a number',
            type='number',
            value=10,
            pattern='[0-9]+',
            min=2,
            max=1000
        ),

        html.P(''),

        html.Div(id='control-buttons',
            style={'column':3},
            children=
            [
                # Button to turn graph updating on or off.
                html.Button(id='graph-update-button',
                    n_clicks=0,
                    accessKey='p'
                ),


                # Button to turn frequency domain magnitude averaging on or off.
                html.Button(id='freq-average-button',
                    n_clicks=0,
                ),

                # Button to toggle time domain data in raw form or windowed form.
                html.Button(id='show-windowed-button',
                    n_clicks=0,
                ),
            ]),
    ],
),

html.Div(id='graph-div',
    children=
    [
        # The time domain graph object.
        dcc.Graph(id='time-domain-graph'),
```

```
                # The frequency domain graph object.
                dcc.Graph(id='freq-domain-graph'),
            ]
        ),

        html.Div(id='statistics-display', children='Can you see me?'),

        # Interval object triggers the data acquisition
        dcc.Interval(id='update-timer',
            interval=4000,
            n_intervals=0
        ),

        # Accumulated status indicates if the sample contained overrange events.
        # This is returned from the acquisition loop and triggers the update of
        # the graph objects.
        html.Div(id='accumulated-status',
            style={'display': 'none'},
            children=''
        ),

        html.Div(id='placeholder',
            style={'display': 'none'}
        ),
    ])


@app.callback(
    dd.Output('placeholder', 'children'),
    [dd.Input('sample-rate', 'value'),
     dd.Input('number-of-samples', 'value'),
     dd.Input('number-of-averages', 'value')])
def update_properties(sample_rate, number_of_samples, number_of_averages):
    '''Send a control signal to the MCU to change the sample rate or number
    of samples'''
    global window_map, freq_axis, freq_mag_history

    if int(sample_rate) != adc.sample_rate or int(number_of_samples) != adc.number_of_samples:
        # Requires board reset so only do if necessary.
        status_reg_f = '\n'.join(adc.modify_sample_properties(sample_rate, number_of_samples))
        # rebuild the window map
        window_map = {'rect': [1] * adc.number_of_samples,
                      'blackman': np.blackman(adc.number_of_samples),
                      'kaiser5': np.kaiser(adc.number_of_samples, 5 * np.pi),
                      'kaiser7': np.kaiser(adc.number_of_samples, 7 * np.pi),}
        # Get, format, and log the status register.
        logging.info(status_reg_f)

    # set the size of the history deque, and in the process clear it..
    freq_mag_history = deque(maxlen=number_of_averages)

    # Create a new frequency axis.
    freq_axis = np.fft.rfftfreq(n=adc.number_of_samples, d=1/adc.sample_rate)

    return ''


@app.callback(
    dd.Output('update-timer', 'interval'),
    [dd.Input('graph-update-button', 'n_clicks'),
     dd.Input('number-of-samples', 'value')])
def toggle_graph_update(n_clicks, number_of_samples):
    '''Prevent the graphs from updating by setting a really long interval time.'''
```

```
    if (n_clicks % 2 == 0):
        return update_period_ms[number_of_samples]
    else:
        # Setting disable is not working so instead just set a ridiculously long
        # interval. In this case about 317 years.
        return 10e12

@app.callback(
    dd.Output('graph-update-button', 'children'),
    [dd.Input('graph-update-button', 'n_clicks')])
def update_graph_button(n_clicks):
    '''Update the text on the graph update button'''
    if (n_clicks % 2 == 0):
        return 'Graph updates are ON'
    else:
        return 'Graph updates are OFF'


@app.callback(
    dd.Output('freq-average-button', 'children'),
    [dd.Input('freq-average-button', 'n_clicks')])
def freq_average_button(n_clicks):
    '''Update text on button, the actual effect of the button is handled in
    the frequency domain graph callback.'''
    if (n_clicks % 2 == 0):
        return 'Freq domain averaging OFF'
    else:
        return 'Freq domain averaging ON'


@app.callback(
    dd.Output('show-windowed-button', 'children'),
    [dd.Input('show-windowed-button', 'n_clicks')])
def show_windowed_button(n_clicks):
    '''Update text on button, the actual effect is handled in the time domain
    update callback'''
    if (n_clicks % 2 == 0):
        return 'Time domain window OFF'
    else:
        return 'Time domain window ON'


@app.callback(
    dd.Output('accumulated-status', 'children'),
    [dd.Input('update-timer', 'n_intervals')])
def acquire_data(n_intervals):
    '''This function triggers the adc to take a measurement and then transfer the
    data back to us. The accumulated status is updated, triggering the update
    of both graphs.'''
    if adc.stop_loops or adc.loop_running or n_intervals == 0:
        # Prevent starting a serial connection when something is already active,
        # including another threaded instance of this function.
        raise de.PreventUpdate
    adc.loop_running = True
    logging.info("Starting acq loop")
    adc.acquisition_loop()
    adc.decode_loop()
    adc.loop_running = False
    return (f'accumulated status: {adc.accumulated_status}')


@app.callback(
    dd.Output('time-domain-graph', 'figure'),
    [dd.Input('accumulated-status', 'children'),
```

```
     dd.Input('show-windowed-button', 'n_clicks'),
     dd.Input('window-functions', 'value')],
    [dd.State('time-domain-graph', 'relayoutData')])
def time_domain_update(status, show_windowed_clicks, window, relayoutData):
    '''Process the acquired data and display in time domain.'''
    global sample_index, rms_mag
    try:
        magnitude = adc.decoded_data_queue[0]
        rms_mag = np.sqrt(np.mean((magnitude - np.mean(magnitude))**2))
        magnitude -= np.mean(magnitude)
        # print('magnitude: {}'.format(magnitude))
        sample_index = 1 / adc.sample_rate * np.linspace(0, adc.number_of_samples,
                num=adc.number_of_samples,
                endpoint=False)
    except IndexError:
        # no data in queue, exit the function without doing anythin
        return

    show_windowed = (show_windowed_clicks % 2 != 0)

    if show_windowed:
        magnitude = magnitude * window_map[window]

    # This is how to access the new axis limits if the user changes them. They
    # start as None so some logic will be required. Will also need to figure out
    # how resetting axis limits works with this.
    # print(relayoutData)
    # print(relayoutData['xaxis.range[0]'])

    return {'data': [{'x': sample_index, 'y': magnitude,
                      # 'type': 'line',
                      'name': 'time domain',
                      # 'mode': 'lines+markers',
                     }],
            'layout': {'title': 'Magnitude vs. sample',
                       'xaxis': {'title': 'Time [secs]',
                                 'range': [0, adc.number_of_samples/adc.sample_rate]
                                },
                       'yaxis': {'title': 'Magnitude [V]',
                                 # 'range': [-2, 2]
                                }
                      }
           }

@app.callback(dd.Output('freq-domain-graph', 'figure'),
              [dd.Input('accumulated-status', 'children'),
               dd.Input('window-functions', 'value')],
              [dd.State('freq-average-button','n_clicks'),
               dd.State('sample-rate', 'value'),
               dd.State('number-of-samples', 'value'),
               dd.State('number-of-averages', 'value')])
def freq_domain_update(status,
                       window,
                       average_clicks,
                       sample_rate,
                       number_of_samples,
                       number_of_averages):
    '''Apply a window and take the fft of the input signal to display the
    information in the frequency domain'''
    global freq_axis, freq_mag, freq_mag_history

    try:
        magnitude = adc.decoded_data_queue[0]
        freq_mag = (magnitude - np.mean(magnitude)) * window_map[window]
```

```
    except IndexError:
        # Indicates no data in the queue, exit function without doing something.
        return
    except ValueError:
        # The magnitude array is not the same length as the window. This is a
        # problem.
        logging.error(f'Input buffer size {len(magnitude)} expected {adc.number_of_samples}')
        raise de.PreventUpdate

    # Compute the frequency magnitude in dBRMS
    freq_mag = np.abs(np.fft.rfft(a=freq_mag, n=adc.number_of_samples)
        * np.sqrt(2) / adc.number_of_samples
        / sum(window_map[window]) * len(window_map[window]))
    # Store every value and calculate the average.
    freq_mag_history.append(freq_mag)
    freq_avg_mag = np.mean(freq_mag_history, axis=0)

    # convert to decibels and decide if we are using the averaged values or not:
    average_on = (average_clicks % 2 != 0)
    if average_on:
        freq_mag_dB = 20 * np.log10(freq_avg_mag)
    else:
        freq_mag_dB = 20 * np.log10(freq_mag)

    return {'data': [{'x': freq_axis, 'y': freq_mag_dB,
                      # 'type': 'line',
                      'name': 'freq domain',
                      'mode': 'lines'}],
            'layout': {'title': 'FFT of input',
                       'xaxis': {'title': 'Frequency [Hz]',
                                 'type': 'log',
                                 # 'range': np.log10([10, adc.sample_rate/2])
                                 'range': 'auto',
                                 },
                       'yaxis': {'title': 'RMS Magnitude [dBV]',
                                 'range': [-150, 30]
                                 }
                      }
           }


@app.callback(
    dd.Output('statistics-display', 'children'),
    [dd.Input('accumulated-status', 'children')])
def statistics_display(fft_plot):
    global rms_mag
    return(f'AC RMS Input: {rms_mag}')

if __name__ == '__main__':
    dash_log = logging.getLogger('werkzeug')
    dash_log.setLevel(logging.ERROR)
    app.run_server(debug=False)
```

10.5.2.2 adc_comms.py

```
import logging
import serial
import io
from time import sleep

# deques are used for sharing of data
from collections import deque
```

```
# Use logger for, um, logging...
logging.basicConfig(level=logging.DEBUG)

# Couple of helpful macros used in conversion of strings to ints.
HEX = 16
BIN = 2

# SCALE_FACTOR is multiplied by the normalized signal magnitude, where the full
# scale range is -1 to 1
SCALE_FACTOR = 8.44190

data_buffer = []

class SerialComms():
    ser = serial.Serial()
    ser.baudrate = 1500000
    ser.bytesize = serial.EIGHTBITS
    ser.parity = serial.PARITY_NONE
    ser.stopbits = serial.STOPBITS_ONE
    ser.timeout = 0.5
    ser.port = 'COM5'

    number_of_samples = 0
    sample_rate = 0

    loop_running = False
    stop_loops = False
    accumulated_status = '0b00000000'

    def __init__(self):
        self.input_data_queue = deque(maxlen=1)
        self.decoded_data_queue = deque(maxlen=1)
        self.ser.open()
        self.sio = io.TextIOWrapper(io.BufferedRWPair(self.ser, self.ser), encoding='utf-8')
        self.reset()

    def status(self):
        '''Get the status register
        Calling this will also pick out the decimation rate and update the
        sample rate.'''

        # Consume anything in the input buffer.
        self.sio.read()
        # Write control signal
        self.write('R')
        logging.debug('Message sent: read status register')

        status_table = [self.readline() for kk in range(4)]
        return status_table

    def start_sampling(self):
        '''Get the adc to start sampling.'''
        self.write('S')
        logging.debug('Message sent: Start sampling')

    def write(self, signal):
        '''Send a control signal'''
        self.sio.write(signal)
        self.sio.flush()
        sleep(0.1)

    def readline(self):
        '''Read a line from the buffer.'''
        return self.sio.readline().strip()
```

```
def reset(self):
    self.ser.send_break(0.1)
    logging.debug('Message sent: Break command ')
    self.ser.reset_input_buffer()
    sleep(0.1)

def setup(self):
    self.write('P')
    logging.debug('Message sent: Setup Signal')


def acquisition_loop(self):
    self.start_sampling()

    data_buffer = []
    '''Loop that waits for start, collects all the samples and stores result.'''
    for nn in range(int(self.number_of_samples)*6):
        # find the start of the data
        if self.stop_loops:
            logging.info(f'Stop loop encountered')
            return
        cur_line = self.readline()
        if 'start' in cur_line:
            logging.info(f'start line found: "{cur_line}"')
            cur_line = self.readline()
            logging.debug(f'first received message: "{cur_line}"')
            for kk in range(int(self.number_of_samples)*2):
                if ('stop' in cur_line):
                    break
                if self.stop_loops:
                    logging.info(f'Stop loop encountered')
                    return
                try:
                    data_buffer.append(int(cur_line, HEX))
                except ValueError:
                    logging.error(f'Non Hex message received: "{cur_line}"')
                    # signal not received correctly. retransmit the
                    # same signal
                # logging.info(f'Current size of data buffer {len(data_buffer)}')
                #  Write the signal
                cur_line = self.readline()
                # logging.debug(f'message received: "{cur_line}"')

            logging.info(f'End line found: "{cur_line}"')

            self.input_data_queue.append(data_buffer)
            return
    logging.error('"start" was never found.')
    return

def decode_loop(self):
    untangled_buffer = []
    try:
        tangled_buffer = self.input_data_queue.pop()
        self.accumulated_status = '0b0'

        for tangled_sample in tangled_buffer:
            # Convert to 32 bit binary LSB at index 0 for easiest match
            # to the tangled bit order.
            T = format(tangled_sample, '032b')

            #         Bit: |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
            # Pos in Word: |15|14| 7| 6| 0| 1| 4| 2|10| 8| 9|11|12| 5|13| 3|
```

```
            #  Pos in Str: | 0| 1| 8| 9|15|14|11|13| 5| 7| 6| 4| 3|10| 2|12|

            # Reorder Bits:         7       6       5       4       3       2       1       0
            untangled_string = (  T[0]  + T[1]  + T[8]  + T[9]  + T[15] + T[14] + T[11] + T[13]
                                + T[5]  + T[7]  + T[6]  + T[4]  + T[3]  + T[10] + T[2]  + T[12]
                                + T[16] + T[17] + T[24] + T[25] + T[31] + T[30] + T[27] + T[29]
                                + T[21] + T[23] + T[22] + T[20] + T[19] + T[26] + T[18] + T[28] )
            if untangled_string[-6:]  != '010000':
                pass
                logging.debug(f'Impossible status, Untangled String: {untangled_string}')
            # logging.debug(f'Untangled String: {untangled_string}')
            self.accumulated_status = format((int(self.accumulated_status, BIN)
                                              | int(untangled_string[-8:], BIN)), '#010b')
            if untangled_string[0] == '1':
                # indicates negative in twos complement
                untangled_buffer.append((int(untangled_string[0:24], BIN)
                                         - (1<<24)) / (2**23-1) * SCALE_FACTOR)
            else:
                untangled_buffer.append((int(untangled_string[1:24], BIN))
                                        / (2**23 - 1) * SCALE_FACTOR)

        self.decoded_data_queue.append(untangled_buffer)

    except IndexError:
        # indicates input_data_queue is empty, can happen if stop_loops was
        # set true.
        pass

def modify_sample_properties(self, sample_rate, number_of_samples):

    self.stop_loops = True

    sample_rate_to_decimation_dict = {'2500000': '0',
                                      '1250000': '1',
                                      '625000': '2',
                                      '312500': '3',
                                      '156250': '4',
                                      '78125': '5',
                                      '': '0'}

    number_of_samples_to_MCU_code = {'1024': '0',
                                     '4096': '1',
                                     '8192': '2',
                                     '16384': '3'}

    decimation_rate = sample_rate_to_decimation_dict[sample_rate]
    sample_code = number_of_samples_to_MCU_code[number_of_samples]

    self.reset()
    self.setup()

    # write the decimation rate
    self.write('D')
    self.write(decimation_rate)
    # write the sample_rate
    self.write('L')
    self.write(sample_code)

    logging.debug(f'Message sent: Decimation "D{decimation_rate}, Sampling "L{sample_code}')

    # Update the internal parameters.
    self.sample_rate = int(sample_rate)
    self.number_of_samples = int(number_of_samples)
```

```
        status_reg = self.status()

        self.stop_loops = False
        logging.debug("stop loop set to false")

        return status_reg

if __name__ == '__main__':
    '''if this file is run on its own it will perform some tests including
    timing how long a transfer takes '''

    import timeit
    TEST_ITERATIONS = 10

    ser_test = SerialComms()

    ser_test.input_data_queue = deque(maxlen=TEST_ITERATIONS)
    ser_test.decoded_data_queue = deque(maxlen=TEST_ITERATIONS)

    ser_test.modify_sample_properties('156250', '1024')

    print('\n'.join(ser_test.status()))

    ser_test.write('v')
    logging.info(f'Overrange: 0x{ser_test.readline().upper()} default is 0xCCCC')
    ser_test.write('f')
    logging.info(f'Offset: 0x{ser_test.readline().upper()} default is 0x0000')
    ser_test.write('g')
    logging.info(f'Gain: 0x{ser_test.readline().upper()} default is 0xA000')


    logging.info('Acquisition Loop time: {}'.format(
        timeit.timeit(ser_test.acquisition_loop, number=TEST_ITERATIONS) / TEST_ITERATIONS))
    logging.info('Decode Loop time: {}'.format(
        timeit.timeit(ser_test.decode_loop, number=TEST_ITERATIONS) / TEST_ITERATIONS))
    logging.info(f'length of decoded buffers:\n\t{[len(x) for x in list(ser_test.decoded_data_queue)]}')
```