DRAGONFLY SYSTEMS

Custom Signal Acquisition Prototype

Project Specifications

Dragonfly Systems

Rev. #	Revision Description	Date	Revised By
1	Initial Draft	2022/19/04	Shaheer R.
2			

Table of Contents

Summary

A major product of Dragonfly Systems is a proprietary signal acquirer. This system can be subdivided into 4 stages:

1. **Reception:** analog signals are received via a hardware interface.
2. **Conversion:** the analog signal is converted into a digital format using an analog to digital converter (ADC).
3. **Processing and Transmission:** the signal is processed and transmitted elsewhere using a microcontroller.
4. **Storage and Playback:** a dedicated node (such as a desktop computer) stores the processed signal for further analysis.

The existing signal acquisition product physically separated each of the stages as shown in the signal flow in Figure 1:

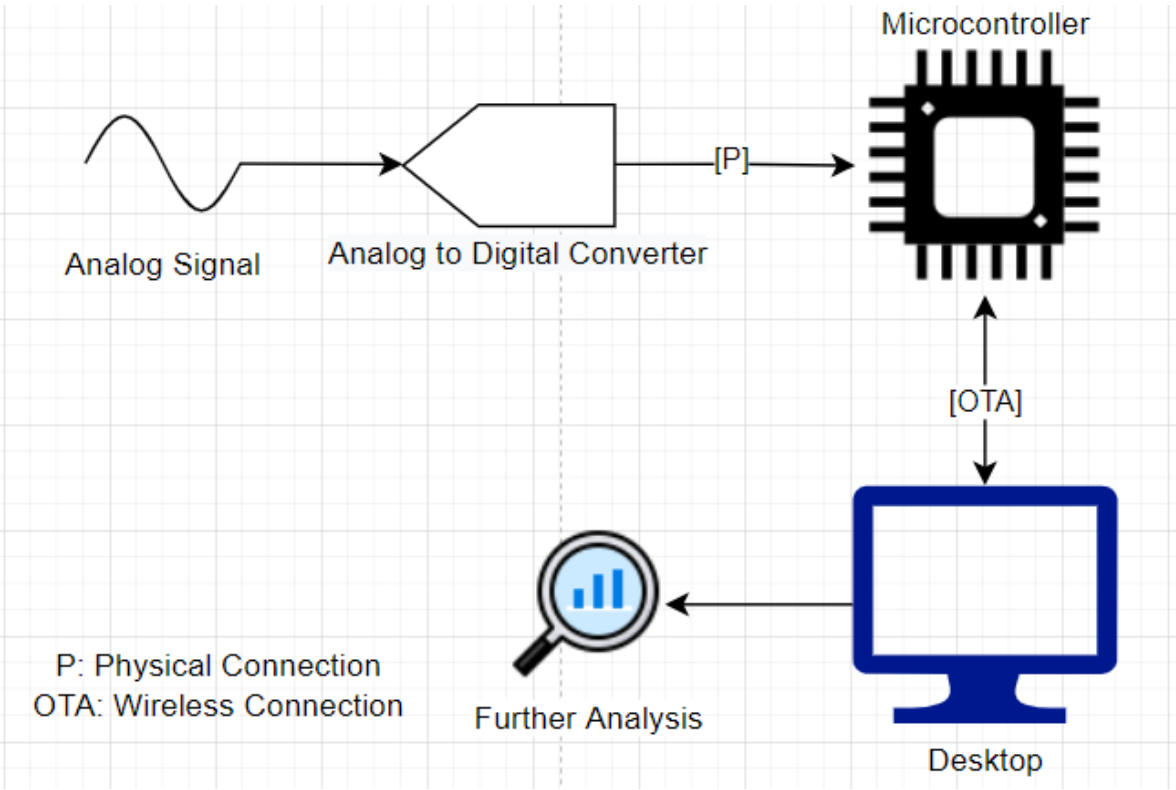


Figure 1: Existing Signal Flow

The purpose of this project is to improve the product by simplifying the signal flow by consolidating stages 1 - 3 into a singular physical package by using an ADC and

operational amplifiers (op-amp) built into the microcontroller. The improved signal flow in the new product is shown in Figure 2:

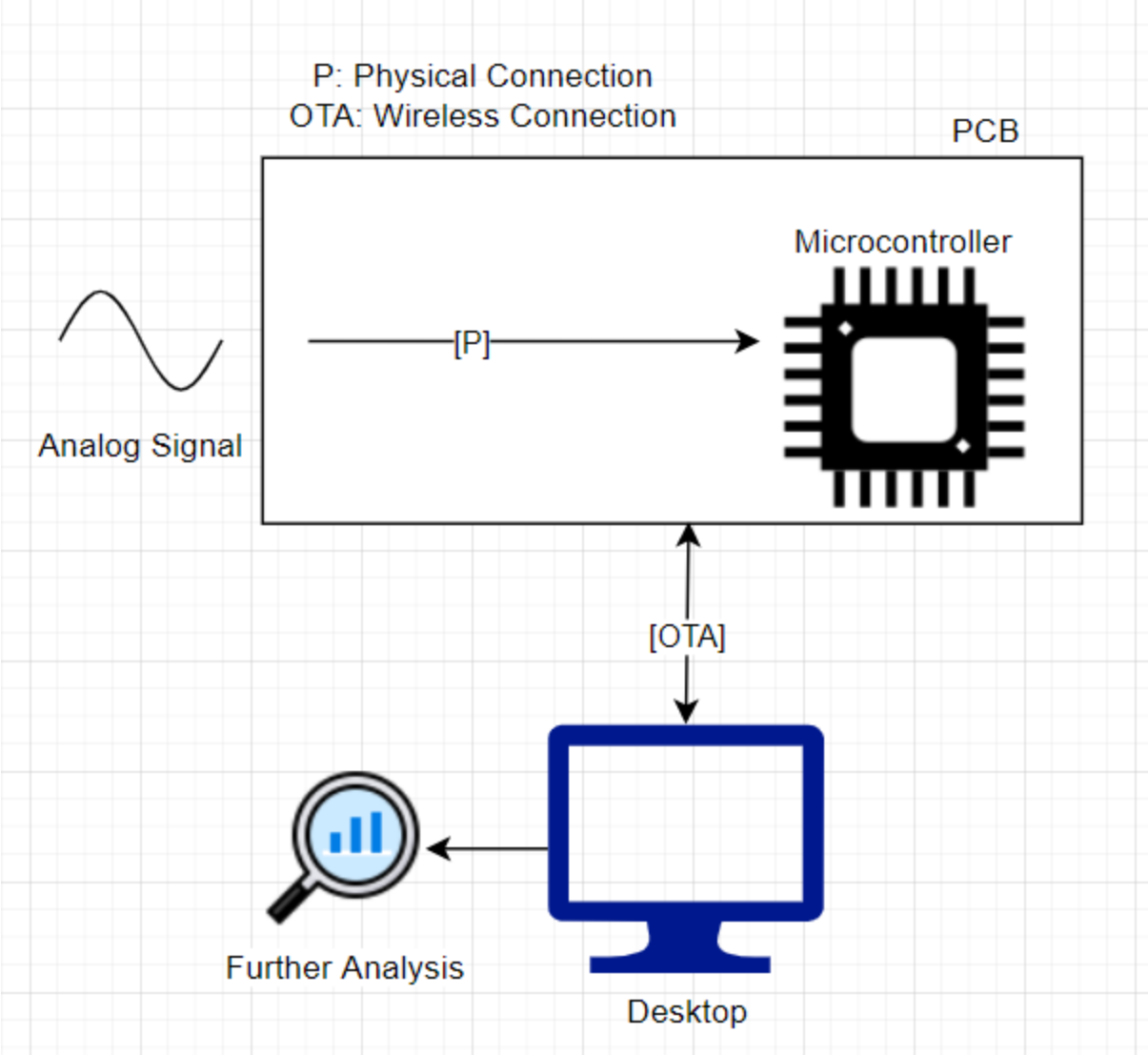


Figure 2: Proposed Signal Flow

A key requirement of the signal acquisition product is to ensure that the signal integrity is preserved throughout the stages. The majority of the signal losses occur during stage 2, due to limitations on the resolution of the ADC and the sampling frequency (the rate at which the ADC performs its conversion). Therefore, the quality of the ADC will be the determining factor as to whether the proposed product improvement is feasible. Since the ADC quality differs in each microcontroller, any microcontroller series that may be used in the signal acquisition product will need to be tested to verify its quality is sufficient.

At the time of this report, a complete testing pipeline for all four stages has been implemented and is functional for the EFR32FG14 Gecko microcontroller by SIlicon Labs, and a similar implementation is in progress for the STM32H7 microcontroller by STMicroelectronics. The majority of the remaining work is in Stage 2 and the operational amplifiers.

Stage 2

Analog to Digital Converters

Mapping Analog Inputs to a Digital Range

Computer logic is only capable of processing binary instructions, 1 or 0. Since these instructions are relayed as an electrical signal, a certain voltage threshold must be met for the microcontroller to register the signal as a 1, otherwise it will be interpreted as a 0. Once the threshold is reached, from the perspective of the microcontroller there is no interpretable difference between an input value at the threshold and an input value orders of magnitude above the threshold, both will register as a 1. For the purposes of computer logic, this arrangement is sufficient, however, for voltage measurement, this introduces error as any readings above the threshold are tied to 1.

For more accurate measurement, an **analog to digital converter** (ADC) implements many different threshold levels, and reports a different value to the microcontroller for each threshold. For example, an ADC with 1 bit resolution has 1 threshold level, where it reports 1 if the input voltage exceeds 1 V, and 0 otherwise. An ADC with 2 bit resolution has 4 threshold levels where it reports 1 after 0.25 V, 2 after 0.5 V, 3 after 0.75 V, and 1 after 1 V. Increasing the bit resolution of the ADC, decreases the increments between each threshold level, and the ADC will report increasingly more accurate values to the microcontroller. Most ADCs have a bit resolution of 12 where the ADC will report a value between 0 - 4095 as $2^{12} = 4096$, where the threshold level that reports 4095 can be selected via software, but is typically reached at 3.3 V (operating microcontroller logic).

Oversampling

While the ADC is physically constrained to 12 bit resolution, through a software trick known as **oversampling**, a higher artificial resolution can be reached, which for the EFR32FG14 is 16 bits. Normally, the ADC immediately relays the sample to the microcontroller after a conversion is complete, however, when oversampling the ADC

relays the sum of several sequential samples which when averaged results in a more precise value. While oversampling artificially increases the resolution, the ADC takes longer to complete a single sample, which may significantly impact sampling quality.

Sampling Frequency

While improving the quality of individual samples is helpful, there are other important factors to consider when capturing continuous signals. When observing a signal or even a simple sinusoidal wave, the input voltage varies with time. To replicate the input signal in the final stage of the system, each input value would need to be recorded before it changes. According to the Nyquist sampling theorem, to perfectly replicate the input signal, the **sampling frequency** must be greater than twice the input signal frequency. For this particular application, the desired sampling frequency is 16 kHz.

To ensure the desired sampling frequency is met, a timer in the microcontroller with the same frequency is used to trigger the ADC. However, the ADC must finish sampling before the next timer trigger, otherwise it will miss a sample and the frequency will be lower than desired. For the EFR32FG14 the LETIMER peripheral was used to trigger the ADC. Testing with and without oversampling at 32 kHz demonstrated that the oversampling outpaced the timer and the resulting waveform was more accurate at 12 bit resolution. The signal is even further distorted at 64x oversampling. At a 16 kHz sampling frequency, differential input (- to VSS), and 1 non-inverting op-amp with a gain of 1.33, the following plots were obtained.

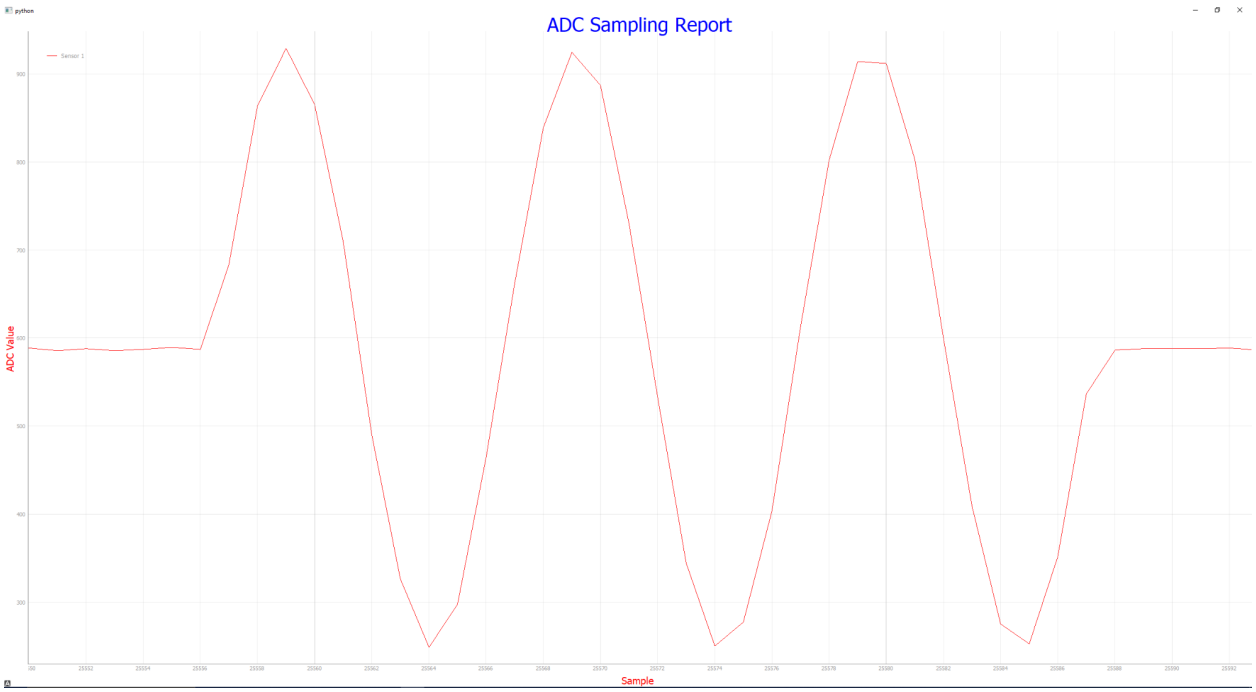


Figure 3: Input 1kHz sine wave, volume 20, 12 Bit. Data

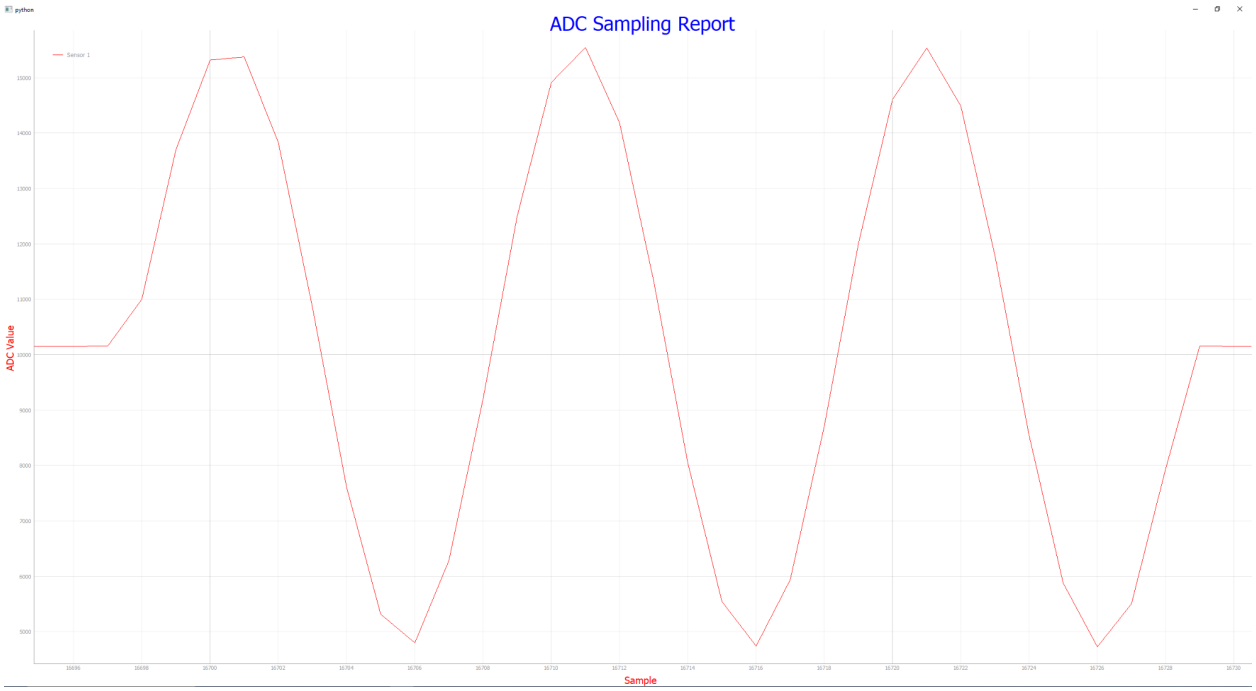


Figure 4: Input 1kHz sine wave, volume 20, 16x OVS. Data

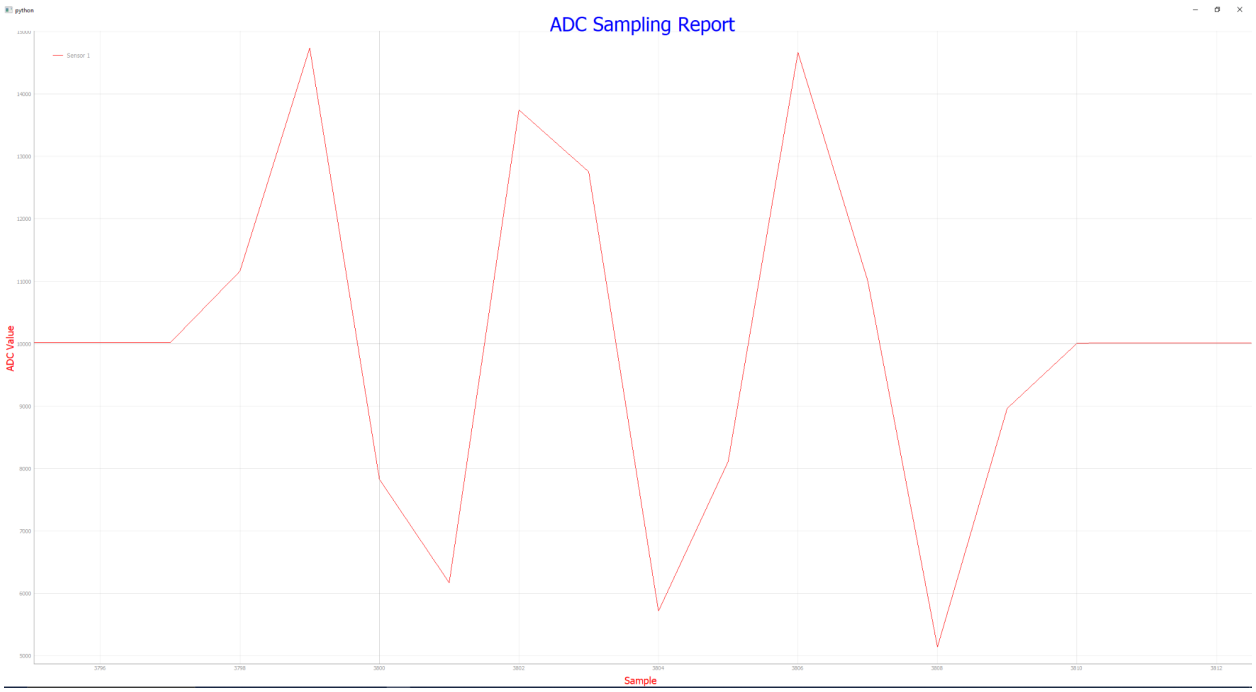


Figure 5: Input 1kHz sine wave, volume 20, 64x OVS. Data

Since no difference was apparent at 16 kHz between 12 bit and 16x OVS resolution, the experiment was repeated at 32 kHz.

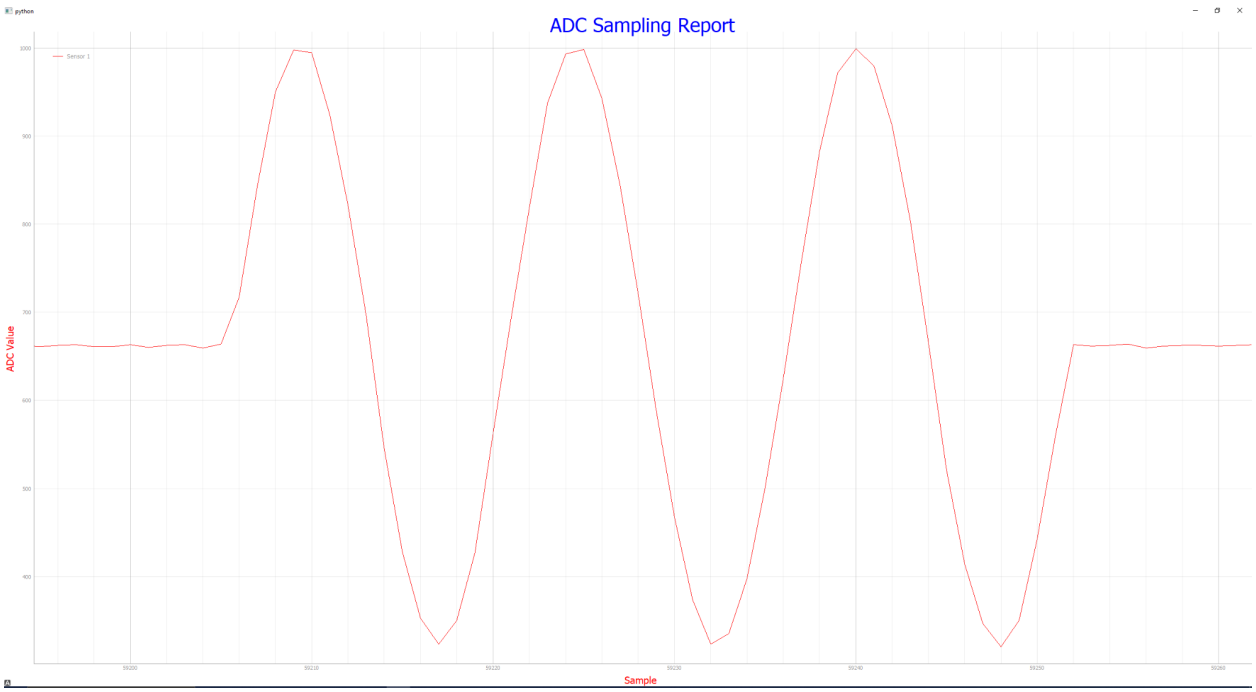


Figure 6: Input 1kHz sine wave, volume 20, 12 bit, 32 kHz. Data

Figure 7: Input 1kHz sine wave, volume 20, 16x OVS, 32 kHz. Data

Another parameter that can be adjusted with the ADC is the **acquisition time**. This dictates the time the ADC takes to perform part of its capture. WIth oversampling active, the minimum acquisition time yielded the most accurate waveform at 32 kHz.

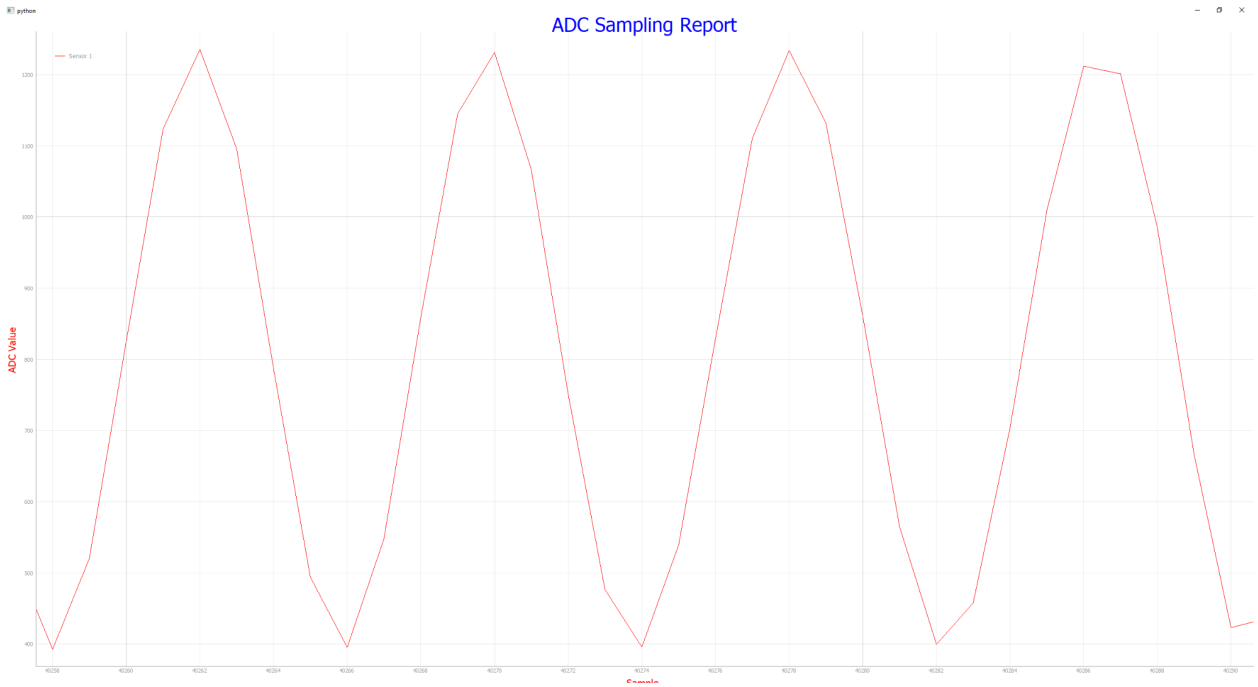


Figure 8: Input 1kHz sine wave, volume 30, 16x OVS, minimum acquisition time

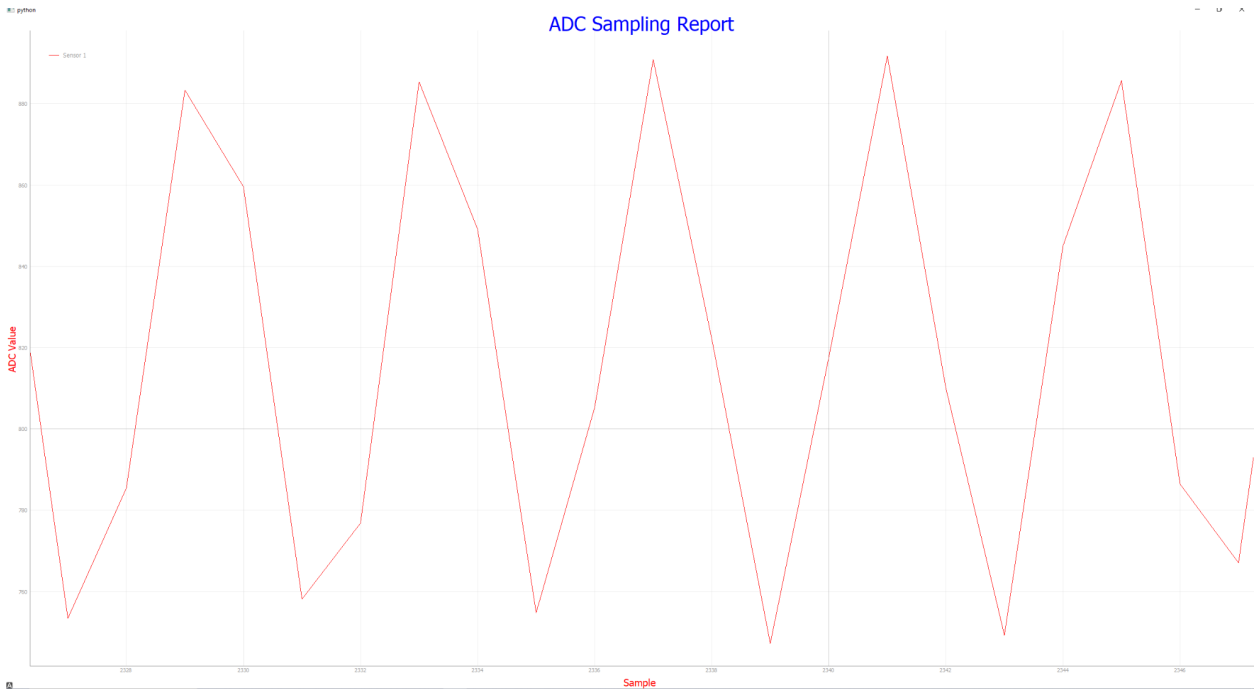


Figure 9: Input 1kHz sine wave, volume 30, 16 OVS. maximum acquisition time

Another parameter that impacts the ADC sampling rate is **warm up time**. Between each sample, parts of the ADC are deactivated and reactivated. To speed up the sample time, this procedure can be skipped by keeping all parts of the ADC active or warm at the cost of increased power consumption, as shown in the following Figure.

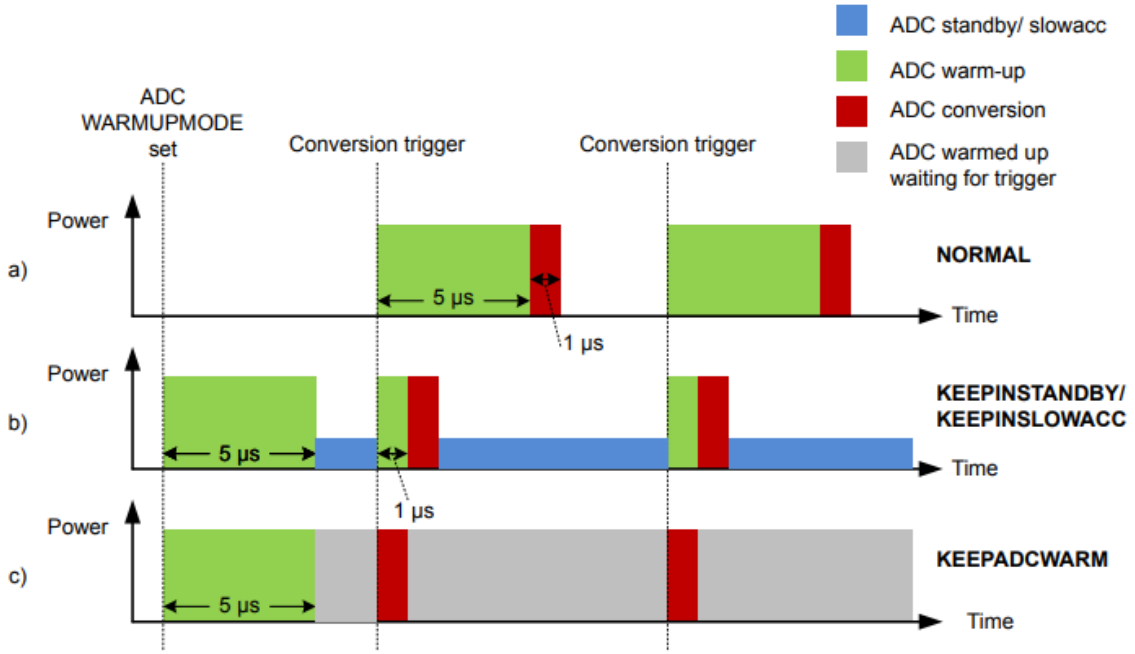


Figure 10: ADC Power Consumption. Source

However, the preliminary testing in Figure 11 showed no demonstrable variation to Figure 7 so this was not explored further.

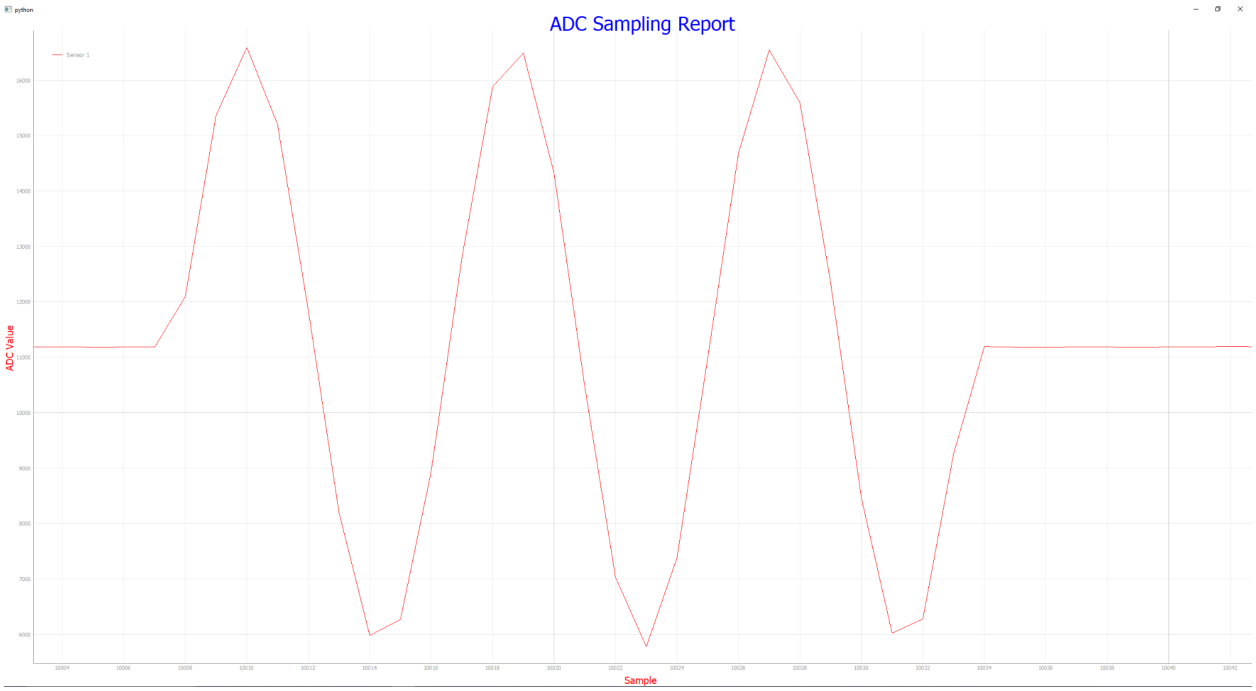


Figure 11: Input 1kHz sine wave, volume 20, 16x OVS, always warm.

The ADC also has its own clock to perform its functions. Increasing the clock frequency increases the speed at which the ADC performs its functions at the cost of increased power consumption. Testing demonstrated that increasing the ADC clock frequency past 7 MHz resulted in no samples being taken despite there being several software options for higher clock values. One possibility is that the power consumption required is higher than what is available for the ADC in EM2 which is the default power range assigned to the ADC.

Operational Amplifiers

An **operational amplifier** (op-amp) is a circuit tool used to modify input signals by scaling, offsetting, and/or filtering them. The 2 basic types of op amps and their scaling equations are shown in the following figure:

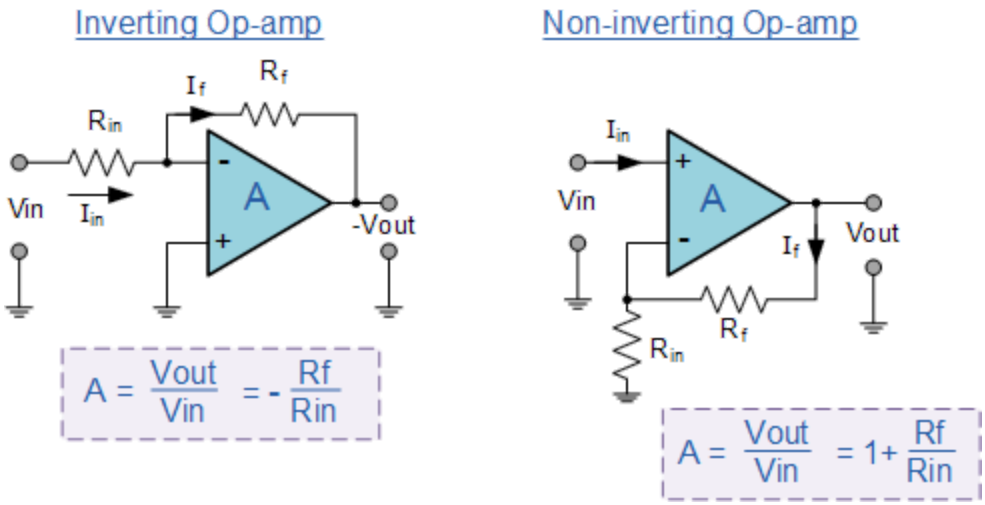


Figure 12: Inverting and Non-Inverting Op-Amp Configuration and Scale Equations. Source

The scaling ability of op-amps makes them highly desirable for ADC sampling, as very low voltage signals are now perceptible by the ADC. To minimize input noise, the lowest possible voltage should be selected for the input signal which would then be routed to the highest gain op-amp. For the purpose of ADC sampling, the non-inverting op-amp is preferred. Multiple op-amps can be cascaded together to further increase the gain as shown in the following Figure:

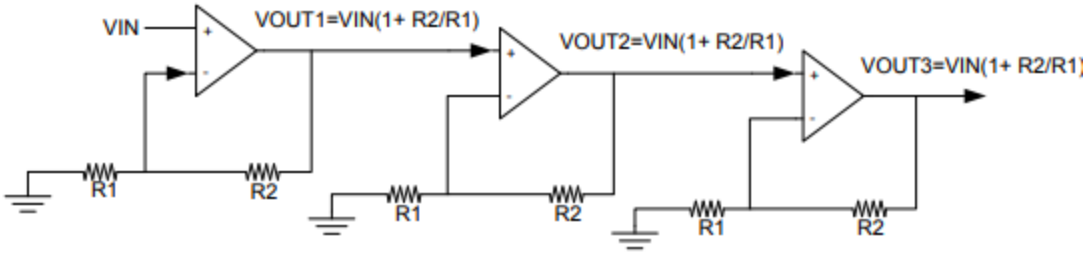


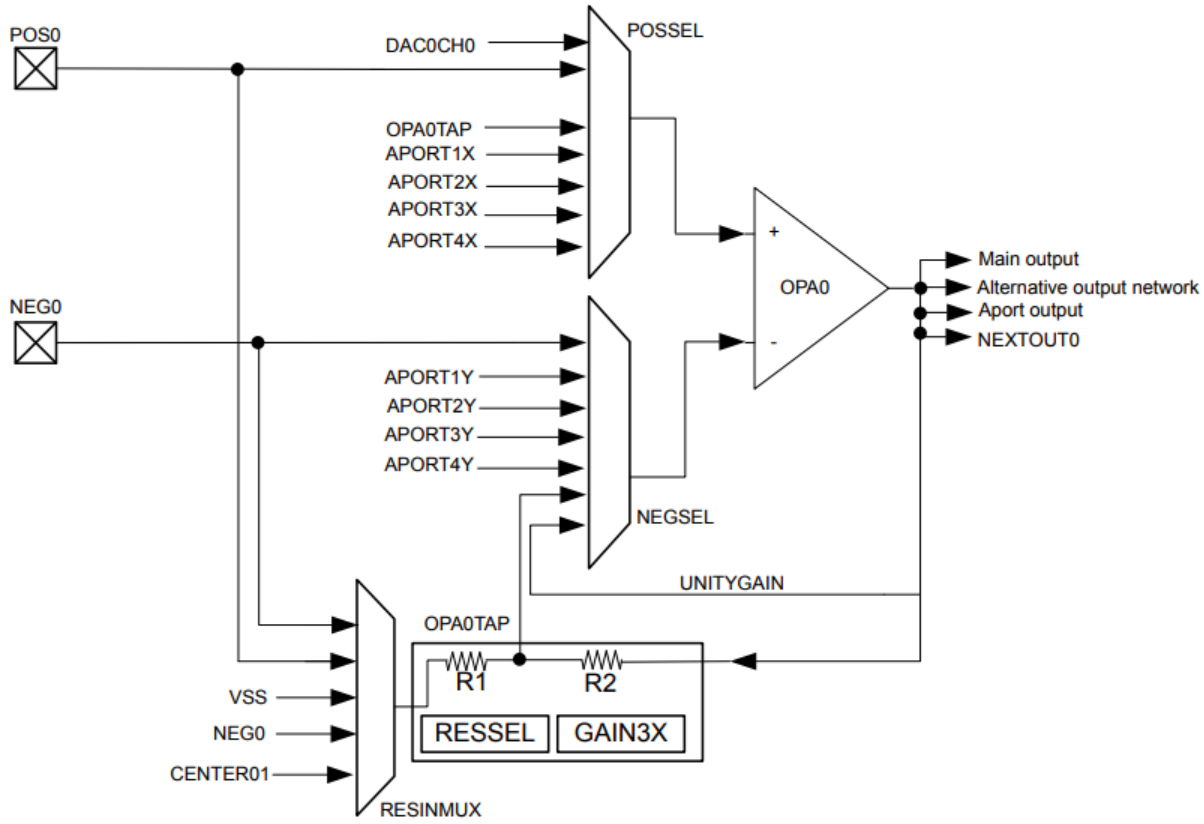
Figure 13: Cascaded Non-Inverting Op-Amps. Source

While the individual values of the resistors cannot be programmed, the resistor ladder ratio can be selected from a preexisting list in software. The maximum gain that can be achieved by one non-inverting op-amp is 16 where R2 = 15R1. In a cascaded format with 2 non-inverting op-amps, the highest possible total gain factor is $16^2 = 256$.

Testing and Issues

The op-amp configuration for the EFR32FG14 is shown in the following figure. For the non-inverting op-amp POSSEL is simply routed to the input signal and NEGSEL should be

routed to the resistor ladder. The input to the resistor ladder should be set to ground, which should be VSS, but if configured correctly could theoretically also be NEG0 (NEGPAD). Testing with a single non-inverting op-amp on both OPA0 and OPA1 was successful with the above settings where RESINMUX = VSS. An unpredictable result was obtained when RESINMUX was set to NEGPAD, this is likely because NEGPAD was not tied to GND in software.



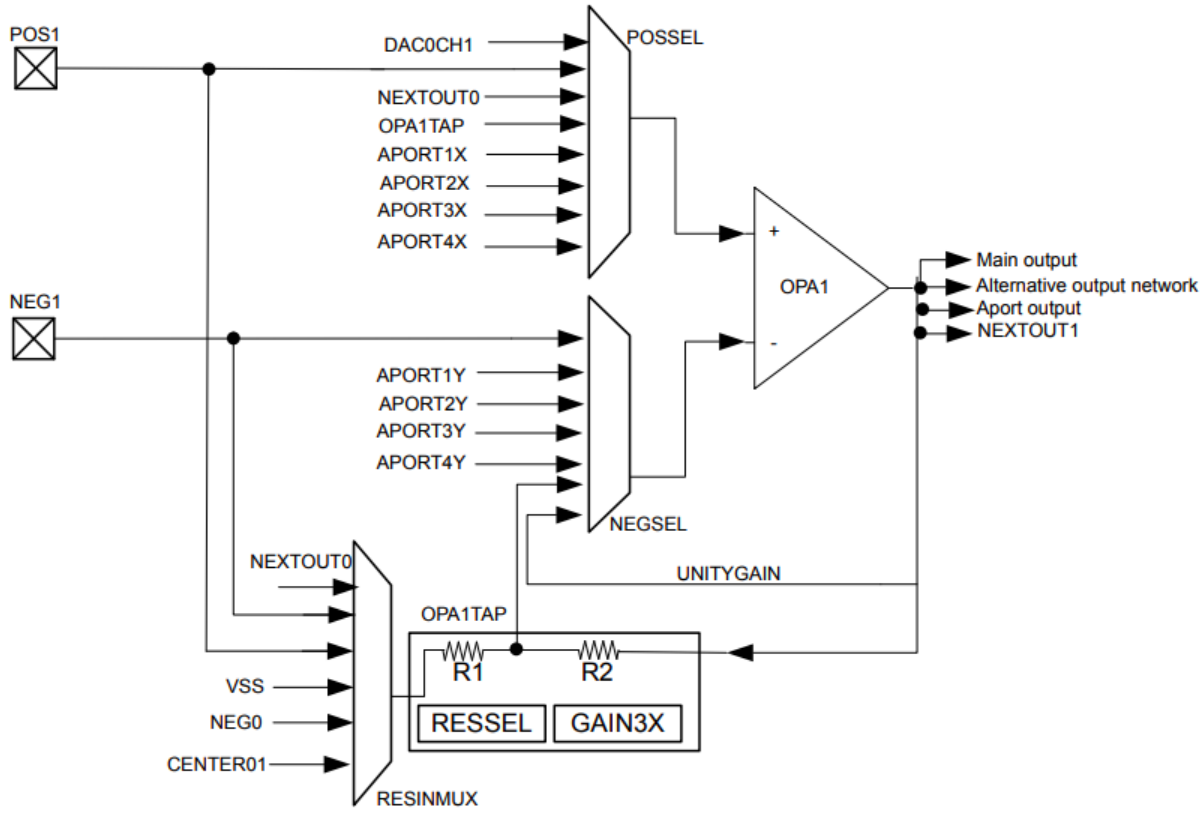
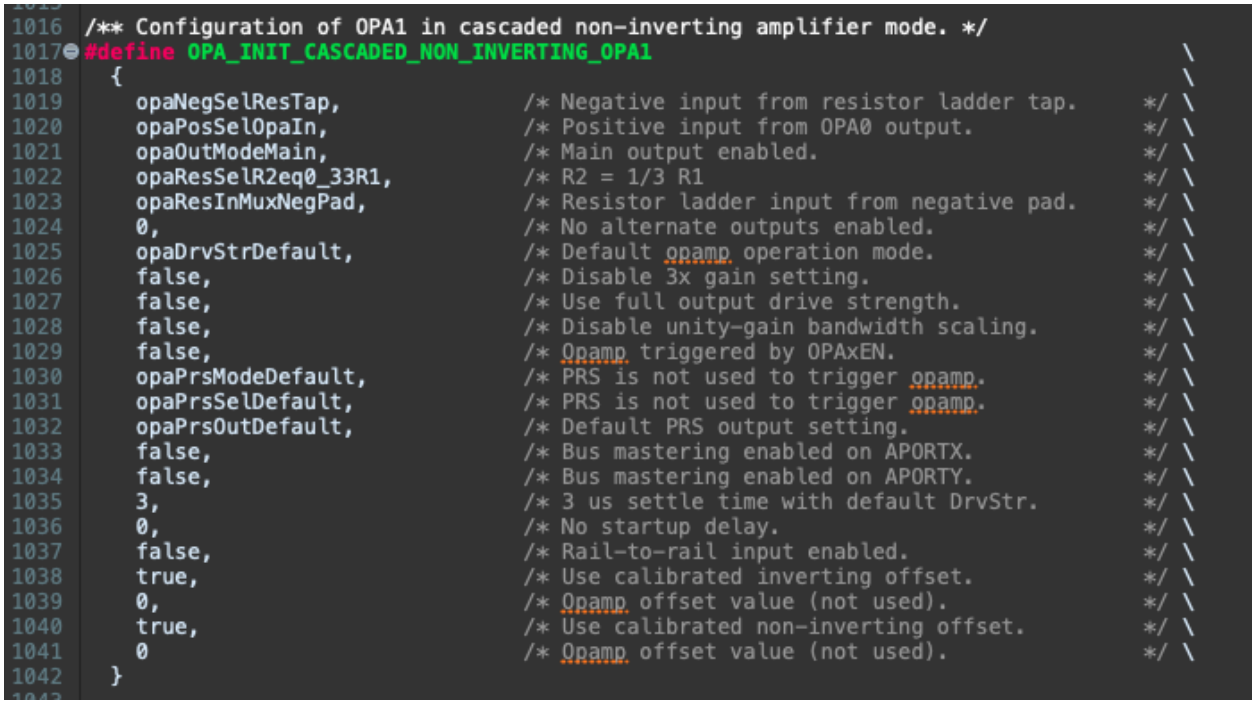


Figure 14: EFR32G14 Op-Amp Configuration. Datasheet

The EFR32FG14 only supports cascading for 2 op-amps, however, testing with these failed and unexpected results were obtained. For the default configuration (same as the single non-inverting op amp), the test input voltage to the ADC was pulled to 3.3 V despite the input signal being set much lower. For all other configurations, the input signal was received by the ADC, however, no gain factor was applied. Since 3.3 V are only drawn from the input test signal when both op-amps are set to VSS, it is possible that a short is created somewhere in that configuration. It could also be possible that the output from OPA0 is not correctly configured to route to OPA1.

The following Figure shows the default op-amp configuration settings for the second op-amp in the cascaded format:



```
/** Configuration of OPA1 in cascaded non-inverting amplifier mode. */
#define OPA_INIT_CASCADED_NON_INVERTING_OPA1                                  \
  {                                                                           \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelOpaIn,               /* Positive input from OPA0 output.             */ \
    opaOutModeMain,               /* Main output enabled.                         */ \
    opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                                  */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    0,                            /* No alternate outputs enabled.                */ \
    opaDrvStrDefault,             /* Default opamp operation mode.                */ \
    false,                        /* Disable 3x gain setting.                     */ \
    false,                        /* Use full output drive strength.              */ \
    false,                        /* Disable unity-gain bandwidth scaling.        */ \
    false,                        /* Opamp triggered by OPAxEN.                   */ \
    opaPrsModeDefault,            /* PRS is not used to trigger opamp.            */ \
    opaPrsSelDefault,             /* PRS is not used to trigger opamp.            */ \
    opaPrsOutDefault,             /* Default PRS output setting.                  */ \
    false,                        /* Bus mastering enabled on APORTX.             */ \
    false,                        /* Bus mastering enabled on APORTY.             */ \
    3,                            /* 3 us settle time with default DrvStr.        */ \
    0,                            /* No startup delay.                            */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use calibrated inverting offset.             */ \
    0,                            /* Opamp offset value (not used).               */ \
    true,                         /* Use calibrated non-inverting offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Figure 15: Default Cascaded Non-Inverting Op-Amp Settings for OPA1

Remaining Topics to Explore

Introducing a DC offset to the op-amps would be beneficial as there would be no chance of attenuation at 0V provided the peak to peak voltage is within the op-amp and ADC limits. Furthermore, for analysis if the stored signal file is processed in Audacity, with a DC offset the program should register the values below the equilibrium as negatives which should improve analysis quality. Currently, Audacity interprets the signals as positive only. There seem to be registers that can set the offset value internally, otherwise it should be possible to route a second input into the op-amp to control the offset.

The op-amps could potentially be used as filters against unwanted frequency ranges introduced by noise and other factors.

More testing should be done to resolve the cascaded op-amp issue. This could be a consequence of a damaged board so the test should be repeated on another board.

Stage 3

Direct Memory Access

Direct Memory Access (DMA) is a peripheral that can handle data transfers between memory locations without direct CPU intervention. For this application, the DMA is simply used to forward ADC samples to an array which will be transmitted to stage 4. A DMA was used to ensure that all of the ADC samples will be transmitted to stage 4 without slowing down any of the other peripheral functions.

After the ADC finishes a sample it triggers an interrupt that calls upon the DMA to transfer the data. The DMA will transfer a set number of samples at once to fill one half of the array ADC_BUFFER with the other half of the buffer's data being transmitted to stage 4. Once the DMA transfer is complete, an **interrupt** is raised and the DMA will switch to the other half of the buffer and trigger the application layer code to start a transmission to stage 4 if it is idle while assigning it the other half of the buffer. This DMA implementation of switching destination addresses is known as **ping ponging**. This implementation does not restrict the DMA functions and allows it to dictate when and from where the **application layer code** can transmit data, if the application layer code is still transmitting the DMA will still switch buffers but will not retrigger the application layer code or change the source address. This logic was implemented to prevent a **race condition** between the DMA and application layer code, where the application attempts to transmit data at the same time the DMA overwrites it, resulting in missing data. The overall logic for stage 3 is shown in the following diagram.

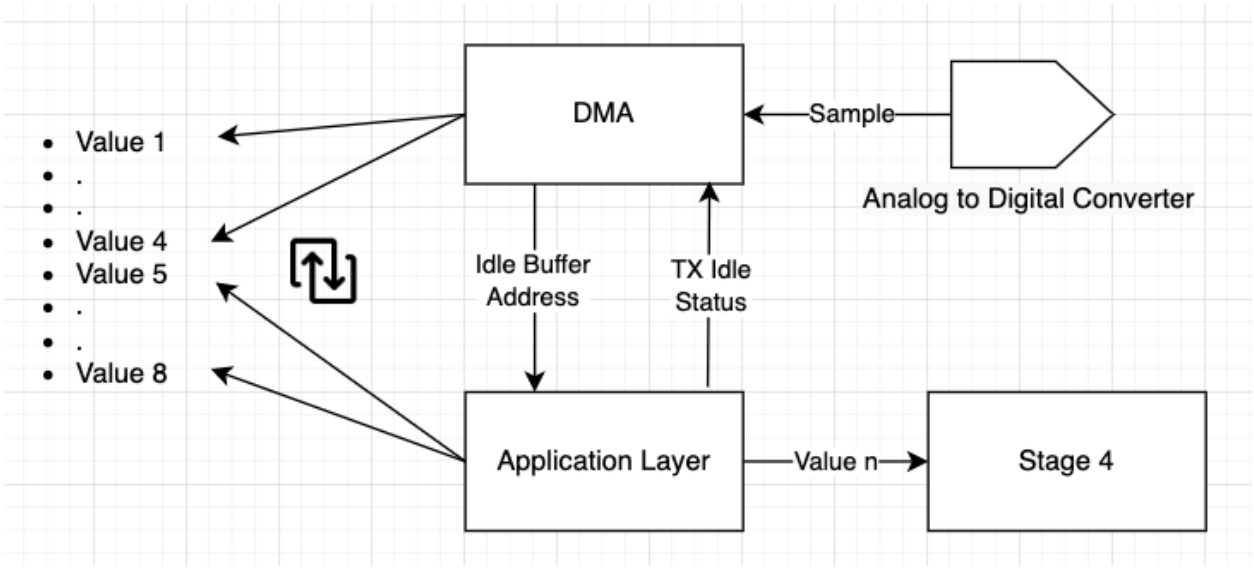


Figure 16: Stage 3 Software Structure

As a result of the buffer sharing mechanism implemented, an even array size must be chosen. Testing with stage 3 shows that the final implementation of the application layer code is able to transmit all of the data without any missing packages.

Universal Synchronous Asynchronous Serial Transmitter

While the final implementation will feature wireless communication between the PCB and stage 4, for prototyping purposes, a physical **UART** connection was used instead. UART is a communication protocol that can be interpreted by the serial ports on any desktop. At minimum, a UART frame bundles 1 byte of data with a start and stop bit which tells the receiving end when to start and stop receiving data.

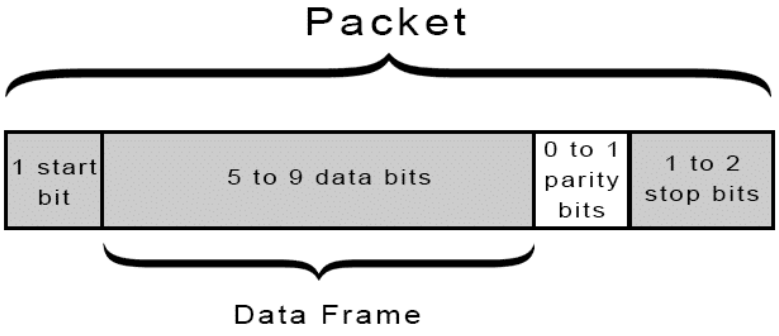


Figure 17: Example UART Packet

Once the application layer code receives the source buffer from the DMA, it will loop through the buffer and transmit the data using a UART. Since the ADC resolution is 12 bits, 2 UART frames need to be sent for each sample. Luckily, for the EFR32FG14 a built-in function exists that will perform the bit manipulations necessary to unpack and transmit each byte separately. Originally these bit manipulations were performed in the application layer code. Stage 4 will have to process the separated bytes to correctly interpret the data. Since the application layer code loops through the buffer and requests to transmit each sample, it is possible that these requests occur faster than the rate of transmission. However, a built in transmission buffer exists that will store the data before transmitting so no samples are lost.

One key parameter that needs to match between the receiving and transmitting end is the **baud rate**, or the rate at which bits are transmitted over UART. This also needs to be set at a sufficiently high speed that the application layer code can transmit before the DMA completes its transfer. The **endianness** (bit order) for the UART frame is set to little, and the number of stop bits is set to 1 on both ends as well.

Testing with stage 3 was a success. For testing purposes the digital value reported by the ADC was directly transmitted, however, the application layer code may be fast enough to do the conversion to volts and then transmit as well.

Stage 4

Python

Reception and Storage

To receive data over UART from stage 3, stage 4 uses this [MPSSE to Serial converter] and a Python script to read from the serial port. pySerial was used to read 2 bytes at a time, so no bit manipulations were necessary to read the 12 bit data from stage 3. Only the yellow (TDI) wire and black (GND) need to be wired in the converter cable.

Some common issues that caused bytes to flip during reading were:

- Not waiting for at least 2 bytes to enter the buffer
- Baud rate mismatch or not being fast enough
- Endianness mismatch
- Printing to console significantly slowed down the script, causing bytes to be missed

After receiving the data, Python writes it to a file. If the data is to be used for graphical interpretation a counter is used to count the number of samples and the data is written as a string into a CSV file with the counter. If the data is to be used for analysis, then the raw data is written directly into a text file where it will be processed by Audacity.

Note that rerunning the Python script will erase all previous data contained in the storage file.

Graphing

Originally, Matplotlib was used to graph the results stored in the file, however, it was extremely slow to process and zoom in on any meaningful chunks of data. PyQt was used instead and proved to be significantly faster. The script is largely self explanatory, however, it is recommended to lock zoom on the x-axis only, so that the y-axis values remain constrained at the upper and lower bounds of the screen for maximum visibility.

When interpreting graphical results, it was also useful to open the text file so the values of individual points can be read and compared. When comparing individual points it is possible to track byte flipping by checking if values increment by $2^8 = 256$.

Further work can be done to plot in the time or frequency domain instead of per sample if desired.

Audacity

Audacity is a waveform editor that can be used to analyze the signal sampled by the signal acquisition product. Select raw data in the import screen and select the text file with the signal data. In the parameter selection screen select 16 bit PCM (2 bytes), match the frequency to the sample frequency, endianness to the UART endianness (little), and import all of the data. The signal can then be analyzed using the built in software tools. The example import configuration is shown in the following figure.

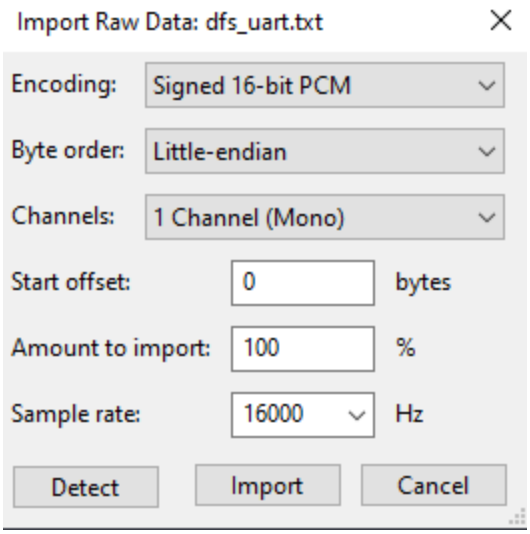


Figure 18 Audacity Import Settings for Raw Data

An issue that was discovered was a mismatch between sample frequency and actual playback frequency. During some analysis, the import frequency matched the sample frequency but the signal was at a much higher frequency than the input waveform. By decreasing the import frequency, the original pace of the waveform was correctly observed. More testing needs to be done with this, flagging a GPIO at every sample and using a logic analyzer to confirm the frequency could verify the sample frequency. Note that this issue was found when oversampling was implemented, which is no longer being used.

For evaluation purposes, Audacity was used to generate various waveforms as an input signal to the signal acquisition product. This was done by generating a tone, selecting a waveform and frequency, and then connecting a split headphone cable into the signal acquisition product.

Test Plans

For this project, inputs were controlled using a programmable test input and waveforms from a split headphone cable. Pin P2 and a GND pin were soldered onto the board as the inputs for this project. More pinout information for the evaluation board can be found here.

The debugging tool in the Simplicity Studio IDE was an invaluable tool for developing the firmware for the EFR32FG14. In particular setting breakpoints, pausing execution, and

watching the values of variables change was helpful. It may be beneficial to set the compiler optimization flag to -Og to optimize for debugging.

Project Documents

The Python scripts can be accessed here: https://github.com/ShaheerRana/dfs_python
The firmware can be accessed here: https://github.com/ShaheerRana/dfs under the dfs_signal_acquisition_prototype project

Diagrams:
DFS Signal Acquisition Prototype Stage 3 Structure
DFS Signal Acquisition Signal Flow

References

People to consult:

Name	**What they worked on**
Shaheer Rana	All work covered in the first revision of this document.
Kathryn Adeney	Supervisor for this project. Firmware for similar products.
Patrick Maheral	Firmware for similar products. Advised on this project.
Brandon Death	Will transition this code to the main code base.
Joe Hyland	Custom hardware and PCB for this project.

Include links to any references used