Contents

LAMDA, the Lightweight Automatic Multichannel Data Acquisition System

Liam Hays

November 25, 2022

Chapter 1

Objectives and Constraints

The system is intended to sample each channel at a minimum of 100 kilosamples per second (ksps). Analog to digital conversion (ADC) resolution of 16 bits per sample is desired, but the chosen hardware only supports 12 bits (9 real) per sample resolution.

The system must sample either four or six channels, to support the following data:

- Record to SD card 2 (or optionally 4) radiometer signals
- Optionally record one microphone channel
- Sample one pulse-per-second (PPS) signal from a GPS receiver

The PPS channel does not need to be saved, only the relationship between the start of the recorded signal and the rising edge of the PPS.

The system should not allow recording, or even run detection, unless it is *armed* by an input digital signal. On assert the system starts to monitor a single channel for a rising edge, and upon detecting that event caches as many samples as the memory size allows. Once that cache is full, the system stops capturing samples and writes cache to the micro secure digital (µSD) card. The system then reinitializes itself and, if still armed, will begin detection again.

1.0.1 Performance Versus Objectives

As Table 1.1 shows, as the number of channels increases, both the number of samples per channel and the total recording time decreases. This table was generated with the script from Section 2.7.

Sampled chan.	Recorded chan.	Samples/chan.	Record t (ms)
3	2	111111	288
5	4	66666	240
6	5	55555	230

Table 1.1: Maximum samples and recording time on each channel for channel counts specified in objectives.

Chapter 2

Implementation

An Adafruit Feather STM32F405 development board and a PPS signal from a global positioning system (GPS) receiver comprise a system known as lightweight automatic multichannel data acquisition (LAMDA). A short name was chosen so that the projects data files on the µSD card (whose names start with the project name) would fit within the 8.3 filename format constraint of the µSD filesystem library.

On startup, LAMDA configures the STM32F405 chip and its internal peripherals, then enters a waiting state where it waits for the *arming pin* to go high. The arming pin is a single digital input pin read by a state machine in the code on the chip. When the pin goes high, the state is changed and direct memory access (DMA) begins, from the internal ADC peripheral to memory.

LAMDA transfers one *block* at a time from the ADC to memory. Once LAMDA is armed, the Feather initiates circular DMA from the ADC peripheral into a small buffer in the STM32F405's internal static random access memory (SRAM). This DMA transfers only 1024 bytes per channel into the small buffer before looping around and reading new data from each channel. The DMA peripheral on the Feather generates interrupts when a transfer is half complete and complete. A block is defined to be the amount of data transferred to the small buffer at the triggering of either of these interrupts. Therefore, the small buffer contains two blocks. The callback for these interrupts changes the global state of a state machine, instructing it to process a particular ADC channel through a OnePole (see Section 2.8) filter and look for the PPS rising edge. LAMDA counts the number of samples since the last PPS rising edge to provide an accurate timebase for the recorded data, see Figure 2.1.

Once the OnePole filter finds a sufficiently large rise on one channel in a block, recording is enabled. Instead of processing new data through detectors, the state machine copies blocks from the small buffer to a large buffer in memory when either DMA interrupts trigger. When the large buffer is full, DMA is stopped and the data in the large buffer is written to a new file on the µSD card. LAMDA then returns to the inactive state, where it looks for the arming pin to be enabled.

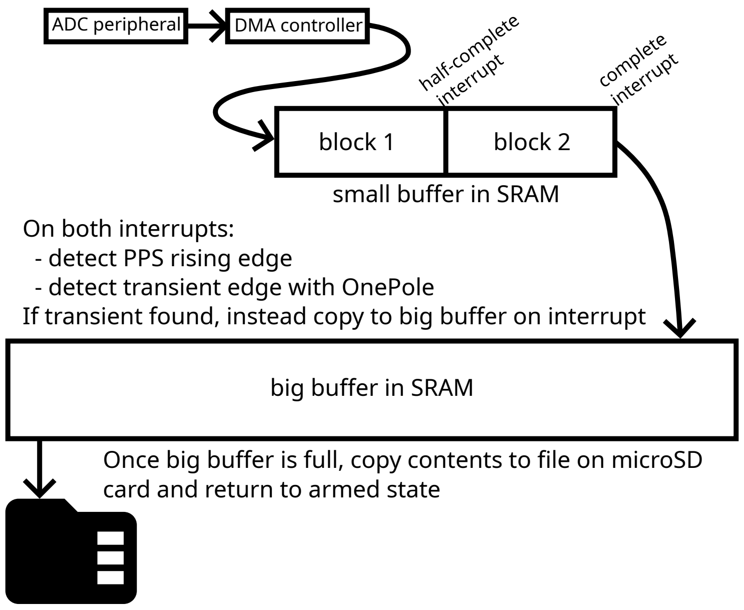

ADC peripheral
DMA controller
half-complete interrupt
complete interrupt
block 1
block 2
small buffer in SRAM
On both interrupts:
- detect PPS rising edge
- detect transient edge with OnePole
If transient found, instead copy to big buffer on interrupt
big buffer in SRAM
Once big buffer is full, copy contents to file on microSD card and return to armed state

Figure 2.1: LAMDA DMA process

The very first block read by the ADC contains useless data, because the voltages inside the ADC peripheral (likely consisting of series capacitors that must be charged) have not settled yet. This data often triggers both the PPS rising edge detector and the OnePole filter, which could cause recording to start too early. Therefore, LAMDA keeps track of whether or not the DMA has just transferred the very first block of ADC samples to memory, and will not process the very first block.

2.1 Hardware Connections

The Feather is connected as shown in Figure 2.2.

The "oscilloscope outputs" shown in Figure 2.2 are used for testing and indicate the start of both ADC blocks and the time needed to process one block's data through the OnePole filter and the PPS detector. These signals are used to generate the graph in Figure 2.4.

The data from analog input connected to the PPS signal is not saved in the large buffer nor written to the µSD card. This enables more data on the other two channels to be recorded in the large buffer before needing to write to the µSD card.

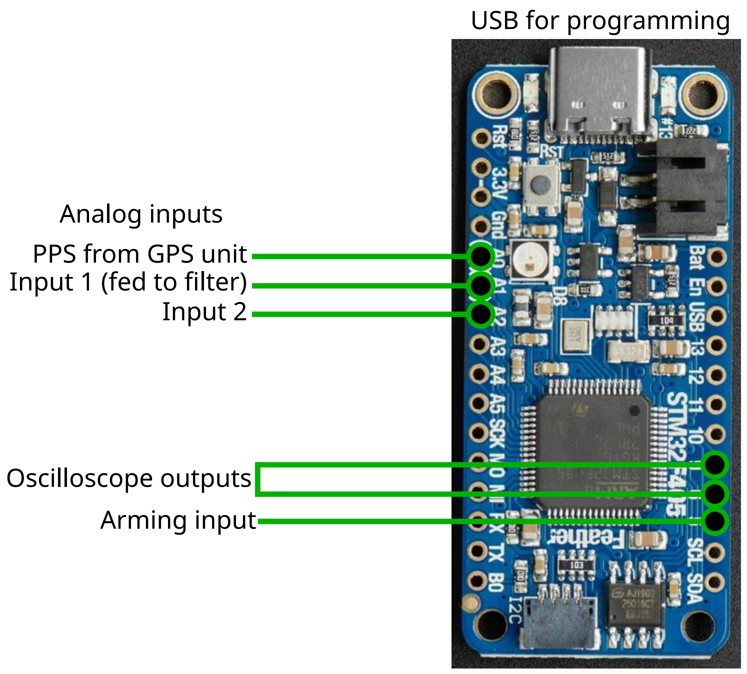


Figure 2.2: Feather F405 hardware connections

2.2 Test Circuit

Figure 2.3 shows how the Feather is connected physically. All power is provided through the type C universal serial bus (USB-C) port on the Feather. In a production-ready system, power would be supplied to the Feather via the black JST connector. The connector is designed for lithium ion polymer rechargeable batteries, and the power system on the Feather can recharge these batteries with power from USB-C.

If a lithium ion polymer battery is not available, regulated 5 V can be supplied to the pin labeled USB, or voltage from about 3.7 V to 4.2 V can be supplied to the pin labeled Bat. To disable the entire system, pull the En pin low, which will disable the onboard regulator and cause the system to draw negligible amounts of power.

2.3 Power Usage

The data in Table 2.1 was measured using a USB-C power meter, and both the meter and LAMDA were powered by an "always on" laptop USB-A port.

Recording lasts only about 150 ms, so the 235 mW used during that time can be considered a peak power draw. For all other times, 189 mW for the Feather alone is an accurate estimate of LAMDA's power usage.

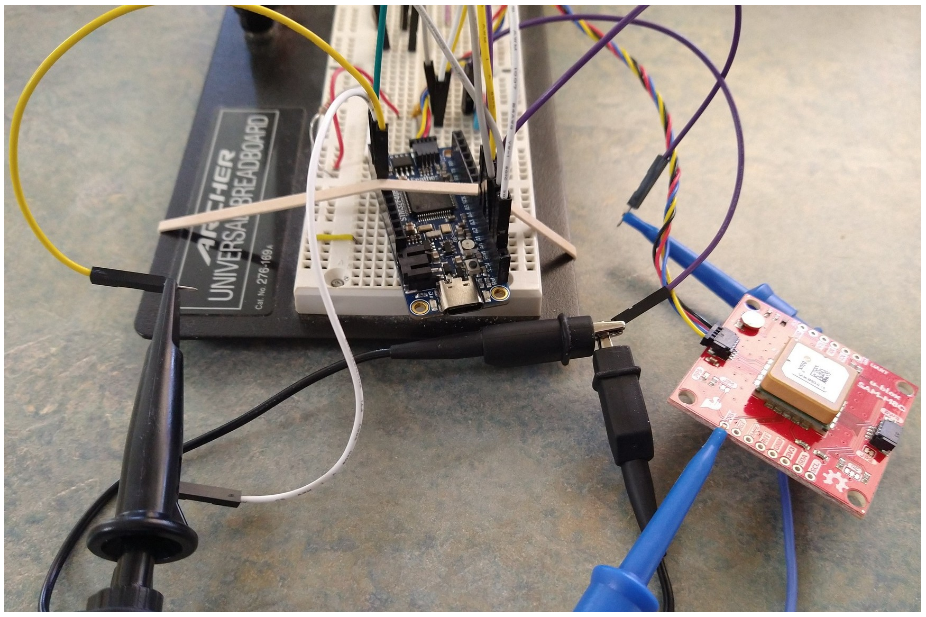


Figure 2.3: Adafruit Feather STM32F405 connected to a u-blox SAM-M8Q GPS module (providing a PPS signal). Analog channel 0 is connected to PPS, channel 1 is connected to a switch between 3.3 V and GND for testing the OnePole filter, and channel 2 is connected to a voltage divider.

State	Power (mW)
Dearmed	163
Armed, not recording	189
Armed, recording	235
Dearmed, powering GPS with fix	350

Table 2.1: Power usage of the entire Feather test circuit.

2.4 Output File Format

The files generated by LAMDA consist of a file header followed by the contents of the big buffer. The header consists of several fields, each described in Table 2.2.

Raw data	Bytes	Purpose
`LAMDA`	5	Project identification
`0xDEADBEEF`	4	Determine endianness of current system
Format version	1	File format version used, unsigned int
Number of channels	1	Number of ADC channels recorded in file, unsigned int
Bits per channel	1	Number of sampled bits of each channel
Samples since PPS	8	Number of samples since last PPS rising edge, `unsigned long`
Samples	-	Data, first sample, first channel, second channel, ...then second sample ...

Table 2.2: Individual components in LAMDA file header

2.5 Example Results

Figure 2.4 shows the contents of one LAMDA data file plotted against time.

A Python script parses the header in the file and uses it to configure the graph display.

2.6 Verification

Figure 2.4 has a visible PPS rising edge on channel 0 at about 64 ms. The Python script that generated the plot processes the samples since PPS field in the file header and converts it to seconds, displayed in the title of the graph. 64 ms + 0.934 s is equal to 0.998 s, which is sufficiently close to 1 s to show that the PPS detector is working. The 0.002 s of error comes from measuring the edge location on the graph with a mouse pointer.

2.7 Sample Rate Calculation

The number of sampled channels is not the same as the number recorded. The recorded number of channels is k_{chan}. The maximum recording length, at two bytes per sample, depends on the amount of memory. The Feather has 128 accessible kilobytes of random access memory (RAM), so the maximum number of samples possible assuming no other memory usage is 64 ks. The per-channel

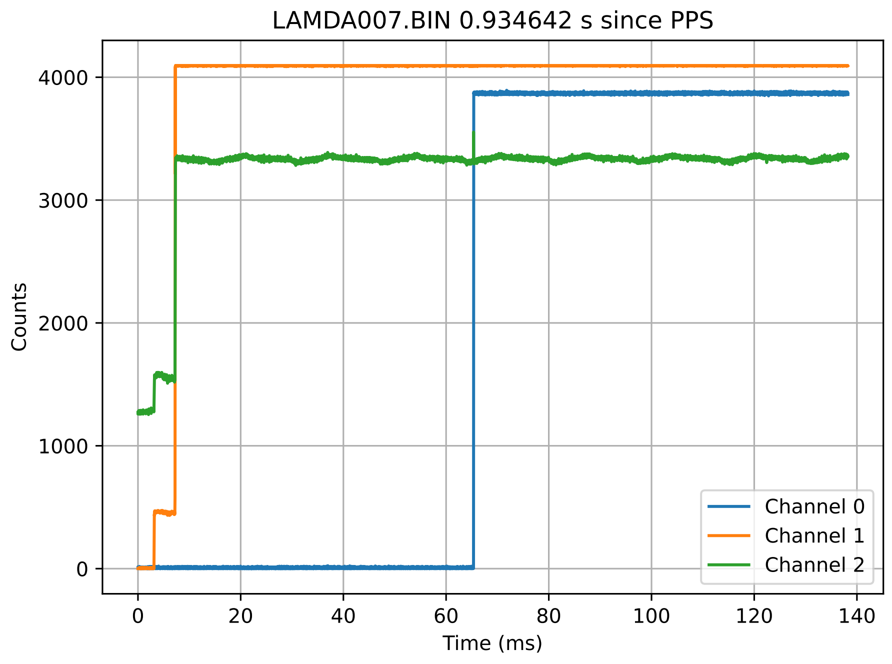


Figure 2.4: A plot of three ADC channels in LAMDA. Note that Channel 0 contains a PPS rising edge—this channel was enabled for testing to verify that the edge detector was working. This channel would not be recorded in real use.

Table 2.3: Sample rates of recorded signals. The recording time t is an upper bound for $k_{\text{chan}} = 2$ recorded channels if all 128 kB were allocated to cache.

a_{scale}	n_{chan}	f_s	t ms (max)
2	3	444 444.4	72.0
4	3	222 222.2	144
6	3	148 148.1	216
8	3	111 111.1	288

maximum samples is 64 ks/k_{chan}. For a typical value of $k_{\text{chan}} = 2$ this is 24 ks. Table 2.3 summarizes the maximum potential recording duration as a function of clock divider.

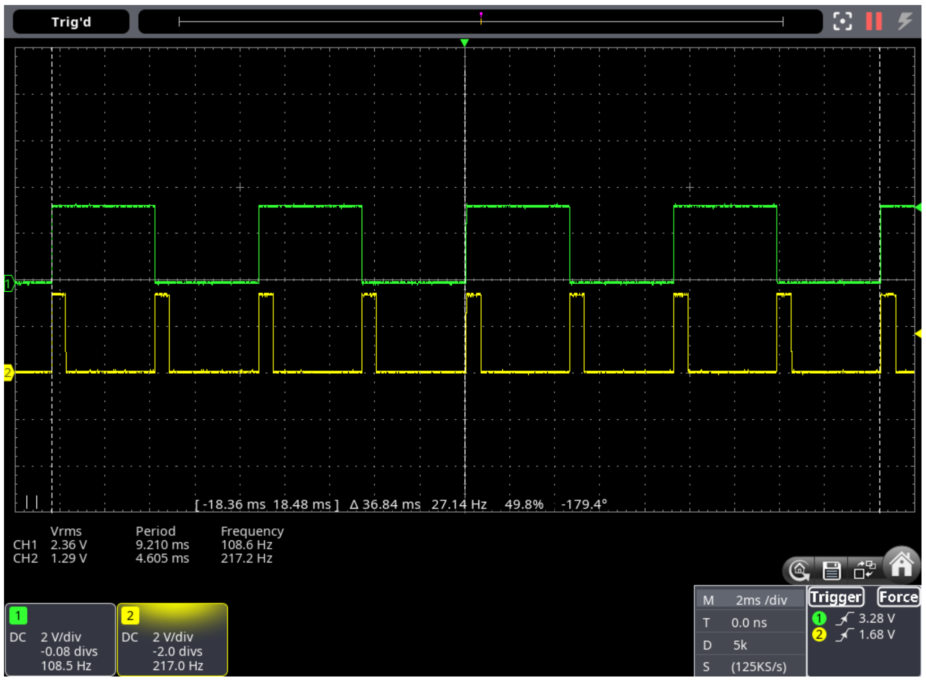
Calculation of the sample frequency is surprisingly opaque. The Feather has a system clock frequency of $f_{\text{sys}} = 168$ MHz. This signal passes through a divider with a value of four to produce the peripheral bus clock at 72 MHz. The ADC clock is divided by the ADC clock prescaler, which can be set to either 2, 4, 6, or 8, called a_{scale} here. It takes some clock cycles to acquire a sample–one cycle per bit plus a settling time of 15 cycles, for a total of 27 cycles. In one read operation, the ADC cycles through all the inputs to sample, or n_{chan} channels. All together, each channel is sampled at frequency f_s.

Table 2.3 was calculated with the Python script file sampleRate.py, listed here:

```python
APB2clk = 72e6
# may need different number of cycles if bits != 12
bits = 12
settleCycles = 15
conversionCycles = settleCycles + bits
numChan = 3

# Buffer length
numChanSaved = 2
maxBufferBytes = 128e3
bytesPerSample = 2

print("Clock Divider",
      "Channel Sample Rate",
      "Max Length (ms)")
for clkDiv in [2,4,6,8]:
    channelSampleRate = (
        APB2clk/clkDiv/conversionCycles/numChan)
    recordingDuration = (
        1e3 * (maxBufferBytes/
        (bytesPerSample * numChanSaved * channelSampleRate)))
    print(clkDiv, channelSampleRate, recordingDuration)
```



```
[-18.36 ms  18.48 ms ]  Δ 36.84 ms  27.14 Hz   49.8%   -179.4°

      Vrms       Period      Frequency
CH1   2.36 V     9.210 ms    108.6 Hz
CH2   1.29 V     4.605 ms    217.2 Hz

1                    2
DC  2 V/div          DC  2 V/div
    -0.08 divs           -2.0 divs
    108.5 Hz             217.0 Hz

M   2ms /div
T   0.0 ns
D   5k
S   (125KS/s)

1  3.28 V
2  1.68 V
```

Figure 2.5: LAMDA's oscilloscope outputs while armed and not recording. The green plot changes state at the start of a new block, so there are 8 blocks between the two dashed lines. The yellow line goes high at the start of the PPS and OnePole processing, and goes low when both detectors finish. The yellow line shows that both filters take a fraction of the time required to transfer the next block from the ADC.

Figure 2.5 shows 8 blocks (on the green line) transferred from the ADC to memory in 36.84 ms (displayed in white text at the bottom of the graph). A new block has been transferred to the small buffer by the DMA. Therefore, the periods of high and low on the green signal represent one block. Eight blocks in 36.84 ms gives 4.605 ms per block, and with 3 channels in use, a block is 1536 samples. 1536 samples/4.065 ms gives 333 550 samples per second total, and 333 550/3 = 111 183 samples per second per channel, accurate to 4 significant digits to the calculated value in Table 2.3.

In addition, the fact that the PPS detector is working and can be shown on the graph in the correct timebase is further evidence that the sample rate has been correctly calculated.

2.8 Single Pole Transient Detection

This section was written by Park Hays, please direct questions and comments to him. The signals measured on the ADC are all positive only. We model these as some signal of interest with a fast-rising and probably slow-falling signal added to some random noise. This filter attempts to identify the rising edge of a signal that is unlikely to have been generated by noise. We assume the relatively steady background can be treated as a constant signal. To compare the signal to a threshold, we first have to remove that background. This is accomplished with a discrete single-pole filter. This filter is implemented so that the background is

$$y[n] = \alpha y[n-1] + \beta x[n] = \alpha y[n-1] + (1-\alpha)x[n]. \tag{2.1}$$

The pole, or cutoff frequency, is

$$f_c = f_s \frac{\ln(\beta)}{-2\pi}, \tag{2.2}$$

where f_s is the sample rate.

The filter is implemented in single precision floating point, rather than fixed point logic, to avoid instability for low frequency cutoff designs.

A similar single-pole filter is applied to the residual signal ($r[n] = x[n] - y[n]$) which, unless a signal is occurring, is noise. Section 2.8.1 shows how a threshold h can be set. Detection occurs when $r[n] > h$. Since h is always positive, this threshold also requires $r[n]$ to be positive, which makes sense for a rising additive signal. The moving average of the magnitude residual is *rho*, and is calculated

$$\rho[n] = \alpha_{\text{MAV}} \rho[n-1] + \beta_{\text{MAV}} r[n]. \tag{2.3}$$

The separate filter parameters α_{MAV} and β_{MAV} might be different from the constant subtraction or the same–they are defined separately for flexibility.

2.8.1 Setting the Threshold

Assuming that the near-constant background has been removed by the filtering process previously described, the remaining signal upon which to perform detection will be normally distributed with zero average value and σ^2 variance. Call this signal $y[t]$, or when we're being cavalier about the discrete sampling of the signal $y(t)$. Ideally we would like to estimate σ by running a moving average on the σ^2; and then taking the square root of that averaged value. Unfortunately, the square root operation is relatively computationally costly. The solution in this software is to perform a moving average on the $|r[t]|$. Remember that $r \sim \mathcal{N}(0, \sigma^2)$ and so the average value of $|r[t]|$ is

$$\begin{aligned} \mathbb{E}(r) &= \int_0^\infty 2\frac{x}{\sqrt{2\pi}} \exp\left(\frac{-x^2}{\sigma^2}\right) \mathrm{d}x \\ &= \frac{\sigma}{\sqrt{2\pi}}. \end{aligned} \tag{2.4}$$

Fortunately, the result is trivially related to the underlying distribution parameter σ.

Our objective is to set the threshold as low as possible to maximize sensitivity while simultaneously keeping the probability of incorrectly recording a signal due to noise tolerable. To work out this problem we imagine that a *mission* is a relatively short period of time where a recording is wanted, this is the mission duration t_{mission}, which we will assume is 5 min for this discussion. We want the probability of starting a recording due to noise during a mission to very low, so assume $P_{fa\text{-mission}} \le 0.001$. The system is collecting data on the monitored channel at f_s samples per second. The first step is to determine the per-sample probability of false alarm that achieves the desired mission probability of false alarm. The mission probability of false alarm is the one minus the probability that the test threshold was not exceeded on the first sample *and* it was not exceeded on the second sample *and* so on for all samples taken during the mission. In other words,

$$P_{fa\text{-mission}} = 1 - \left(1 - P_{fa\text{-sample}}\right)^{f_s t_{\text{mission}}}. \tag{2.5}$$

We can assume σ is known since we estimate a term closely related to it. If we set a threshold h then the per-sample probability of exceeding that threshold simply due to noise is

$$P_{fa\text{-sample}} = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{h}{\sigma\sqrt{2}}\right)\right). \tag{2.6}$$

Equation (2.6) can be solved for h,

$$h = \operatorname{erfc}\left(2P_{fa\text{-sample}} - 1\right)\sigma\sqrt{2}. \tag{2.7}$$

Finally, to set a threshold based on some constant times a mean absolute value (which is what we actually estimate),

$$\begin{aligned} h_{\text{MAV}} &= \frac{h\sqrt{2\pi}}{\sigma} \\ &= \operatorname{erfc}\left(2P_{fa\text{-sample}} - 1\right)2\sqrt{\pi} \end{aligned} \tag{2.8}$$

Pleasingly, this does not actually rely on what the σ is! If we use a five minute mission operating at a sample rate of 116 666.7 Hz, and tolerate a mission probability of false alarm of $P_{fa\text{-mission}} = 0.001$ then we find a our *threshold multiplier to be h_{MAV}* $= 6.53$. This is the smallest value to consider.

Python code to accomplish this is in file calcThresh.py and listed here:

```
import numpy as np
from scipy.special import erf, erfc

# From Mathematica
# In[2]:= Integrate[(2 x/(sigma Sqrt[
#          2 \[Pi]])) Exp[-(x^2)/(sigma^2)],
#          {x, 0, \[Infinity]}]
#
# Out[2]= ConditionalExpression[
#            sigma/Sqrt[2 \[Pi]], Re[sigma^2] > 0]

Pfa_mission = 0.001
fs = 116666.7
missionDuration_min = 5

missionDuration = missionDuration_min * 60
missionDuration_samples = missionDuration * fs

# Pmission = 1 - (1-Pfa_sample)^numberSamples,
# solve for Pfa_sample
# (1-Pmission) = (1-Pfa_sample)^n
# log(1-Pmission)/n = log(1-Pfa_sample)
# 1 - exp( log(1-Pmission)/n) = Pfa_sample
Pfa_sample = 1 - np.exp(
    np.log(1-Pfa_mission)/missionDuration_samples)
print("Probability␣of␣a␣sample␣exceedance:␣", Pfa_sample)

# Pfa_sample = 0.5*(1 + erf(thresh/(sigma*sqrt(2))))
# thresh = erfc( 2*Pfa_sample - 1) * sigma * sqrt(2)

# Calculate the multipliers
threshMult = erfc(2*Pfa_sample -1)*2*np.sqrt(np.pi)
print("Minimum␣multiplicative␣threshold:␣", threshMult)
```

2.9 PPS Rising Edge Detection

The PPS detector determines if the numerical difference between two ADC samples exceeds a predefined threshold. The PPS rising edge should rise effectively instantaneously, so the difference between even two samples should be detectable. However, using unsigned integers means that there is a risk that a subtraction operation could underflow and create a false detection. Code like the following poses this risk:

```
if (sample - lastSample > PPS_RISE_THRESHOLD) {
  // PPS rising edge detect
  risingEdgeFound = true;
}
```

This snippet would be wrapped by a loop that iterates over one channel's samples in the newest block, and the variable `lastSample` is updated with the previous value of `sample` on each iteration. The subtraction done in the `if` statement is likely done as an unsigned subtraction if both variables are unsigned, as they are in LAMDA, so there is a possibility of underflow.

The solution is to adapt the `if` statement as follows:

```
if (sample > PPS_RISE_THRESHOLD &&
    lastSample < PPS_RISE_THRESHOLD) {
```

This accomplishes the same comparison with no risk of underflow.

Chapter 3

Design Considerations

There are three main design structures: 1) write data to the SD card as it is digitized, 2) store data in RAM until cued then fill RAM once and write out to µSD, or 3) continuously digitize but process for a transient and store a fixed length record upon detection. Each option offers benefits and challenges which table 3.1 summarizes.

The minimum data rate needed is about 100 ks/s.

The largest contiguous section of RAM on the STM32F405 is 128 kB. Assuming 100 kHz sample rate, two bytes per sample, and two channels, we get 400 kB/s permitting storage for about 1/3 of a second. This is actually sufficient if we can detect the event, but is not enough if it has to be externally cued. The rate calculation is straightforward, but fully detailed with K_{chan} channels, an overall sample rate r shared between the channels, b bytes per sample,

$$\begin{aligned} \text{data rate} &= K_{\text{chan}}\,[\text{channel}]\; b \left[\frac{\text{bytes}}{\text{sample}}\right] r \left[\frac{\text{sample}}{\text{second}}\right] && (3.1)\\ &= 2\,[\text{channel}]\; 2 \left[\frac{\text{bytes}}{\text{sample}}\right] 100\times 10^3 \left[\frac{\text{sample}}{\text{second}}\right] && (3.2)\\ &= 400\,\text{kB/s} = 391\,\text{KiB/s} && (3.3) \end{aligned}$$

Note that the assumption of two channels assumes the PPS signal is processed into a short list sample indices associated with the PPS rising edges.

The Portenta H7 has 1 MB of RAM, and so could theoretically support about 2.6 seconds of continuous recording of two channels. This is too short to make continuous or cued recording a possibility.

3.1 Adding More Channels

Adding more channels would allow for recording from more sensors, but less data per sensor. LAMDA will sample every channel but is still constrained by the

Table 3.1: Summary of major design options.

Option	Technical Advanges	Technical Disadvantages
1 Direct-to-μSD	Store very long sequences. Simple implementation.	Simultaneous transfers from the ADC and to μSD may strain bus bandwidth.
2 RAM once cued	Very simple implementation. No risk of bus contention due to simultaneous input and output.	Length of sequence limited by amount of RAM. Latency in cue delivery system is challenging.
3 Detect event in signal	Optimal length of storage in memory. Pre-segmented signals. Large number of records possible.	Development and test of detection algorithm is challenging.

size of the large buffer. Therefore, the recording time for one file will be shorter, as the large buffer will fill faster than it would with fewer channels.

The time needed to run the PPS detector and the OnePole filter is minimal compared to the time needed to acquire one block, so adding more channels will not interfere with processing. Adding more channels will actually reduce the number of samples that must be processed in one block, so the detectors would actually run faster.

3.2 Channel Crosstalk

One result of the internal design of the ADC is electrical crosstalk between channels. If one channel is driven higher than other channels, those other channels will read as artificially high values. The inverse is also true if one channel is pulled lower than the other channels.

One solution to this problem is to place the channels physically farther apart. The STM32F405 has more than enough discrete ADC channels to use channels on pins that are farther apart (for example, channels 0 and 4, with no channels connected between), so that the crosstalk is decreased. However, it may be necessary to actually sample the unused channels for this to have any effect, which would significantly decrease the sample rate.

Another possibility is that the test breadboard that the Feather board is connected to (see Section 2.2). This setup uses female header and jumper wires into a breadboard, making it a noisy and unreliable circuit. The crosstalk may be less significant if the Feather is soldered into a complete system.

3.3 Filename Limitations

The filesystem library used by LAMDA, known as FatFs, is limited to filenames in the 8.3 format, like `LAMDA000.BIN`. Therefore, LAMDA can record a maximum of 1000 files before it will enter an error state. 1000 files is sufficient for several minutes of recording, but if more files are needed, the code can be modified to support different filenames.

Chapter 4

Major Challenges

4.1 Portenta H7

I set up the Arduino Portenta H7 to capture 16-bit samples on 3 channels and DMA them to a buffer in memory. I achieved this by using STM32CubeIDE, the official STMicroelectronics development system, and using its graphical chip configuration tool to generate code for the ADC and DMA peripherals. I then attempted to use the ARM mbed OS (used by the Arduino framework on the Portenta H7), which provides a serial peripheral interface (SPI) µSD card interface to store the data in memory on an SD card. However, I proved this to be too slow, primarily because it uses the SPI mode of the secure digital standard (flash memory card) (SD) card, which is 1-bit and lower clock speed than the secure digital input/output interface (SDIO) mode. The official Arduino framework library for STM32 SD card access, known as `STM32SD`, does not support the Portenta H7. The only way to get usable SD card speeds from the board would be to use the hardware abstraction layer (HAL) with STM32CubeIDE.

Unfortunately, the Portenta H7 board cannot be easily programmed with STM32CubeIDE. Most STM32 chips, including the one on the Portenta H7, include a region of read-only memory (ROM) that contains a universal serial bus (USB) device firmware update (or upgrade) (DFU) bootloader, allowing arbitrary user code to be uploaded to the chip. While the Portenta H7 contains this ROM, it can only be enabled by inserting the board into a larger carrier board and manipulating a dual in-line package (DIP) switch on the carrier board, which are fragile and difficult to use. The bootloader pre-loaded on the Portenta H7 is designed only to work with ARM mbed OS compatible code, making the H7 a dead end.

4.2 Feather F405

The solution was to use the Adafruit Feather STM32F405. This board exposes numerous analog and digital pins on the chip, as well as the pin that enables the bootloader in ROM. I continued by using Arduino with some code generated by STM32CubeIDE.

Tests with the F405 show SD write speeds of about 6 MB/s, which is in theory fast enough for continuous data streaming to the SD card. However, in practice, the system cannot write this quickly. In testing the write of the ADC data, the function call to write data to the SD card does not always complete before another block is transferred to memory. Once this happens, the memory card (or the internal file buffer, or even the SDIO peripheral onboard the STM32F405) appears to crash and can never write more data. Even when the write runs without exceeding the time limit, the data on the card is often incomplete or nonexistent.

I then decided to stop using Arduino, as I felt it was adding too much bloat and complexity to the code, and switched to pure HAL code in STM32CubeIDE. I thought that enabling DMA on the SDIO peripheral would help make the SD card write faster, but it instead caused the card to become unusable. I gave up on SD DMA after struggling for multiple weeks with no success.

This is when the direction of the project changed, from being a continuous recorder to a detection-triggered recorder. With the OnePole filter, LAMDA detects significant variation on a particular analog channel and records as much data as will fit in its RAM. This data is then written to the SD card, creating small gaps in data where the card is being accessed.

Despite what I was able to do with the Feather, it comes at the cost of the STM32F405 having only a 12-bit ADC, not 16-bit like on the Portenta H7. It also has much less RAM than the Portenta, so the total recording time is shorter than the Feather is capable of. I would like to reimplement LAMDA on a board like the Portenta H7, to get the benefits of better, newer hardware.

Glossary

μSD micro secure digital 1, 3, 15, 16, 18

ADC analog to digital conversion 1, 3, 4, 7–11, 13, 16, 18, 19

DFU device firmware update (or upgrade) 18

DIP dual in-line package 18

DMA direct memory access 3, 4, 10, 18, 19

GPS global positioning system 1, 3

HAL hardware abstraction layer 18, 19

JST Japan Solderless Terminal, a company that makes removable electric connectors 5

LAMDA lightweight automatic multichannel data acquisition 3–5, 7, 8, 10, 14, 15, 17, 19

PPS Class of signals typical in the global positioning system where one pulse is emitted on the second . 1, 3, 4, 6–8, 10, 13, 15, 16

RAM random access memory 7, 15, 16, 19

ROM read-only memory 18, 19

SD secure digital standard (flash memory card) 1, 15, 18, 19

SDIO secure digital input/output interface 18, 19

SPI serial peripheral interface 18

SRAM static random access memory 3

USB universal serial bus 18

USB-C type C universal serial bus 5