WHITEPAPER

Getting the Best of all Worlds with Zephyr RTOS:

Bare Metal Hardware Access, Direct IRQ Handling and Digital Signal Processing for Cloud Connected Acoustic Machine Monitoring

Eli Hughes, Principal at Wavenumber LLC.

AUGUST 2023

Contents

In this article, we will explore an acoustic machine monitoring application built using the Nordic NRF9160 and the Zephyr real-time operating system. We'll demonstrate how to get close to the "metal" and optimize low-level peripheral access, signal processing, and efficient data packing for transmission to a cloud backend, all while leveraging Zephyr high-level abstractions that help with rapid application development and overall portability, and lifecycle management with its components. However, there are times when one must get "to the metal" to achieve the highest performance metrics such as power consumption. This article will show an acoustic machine monitoring application that uses Zephyr with the Nordic NRF9160 cellular System-In-Package (SIP) combined with high-performance vibration and temperature sensors.

While Zephyr is a very rich embedded development framework, it does not mean that you have to give up any of the advantages of "bare metal" for high performance applications. In fact, you can get the best of all worlds while having a development framework that scales across projects, and this is what this article will be showing you. More specifically, we will cover how to:

- Access peripherals directly for minimal overhead
- Have direct access to IRQ handlers to weave peripherals together in ways not supported by the Zephyr driver model.
- Use the Zephyr build system to bring in DSP libraries to implement acoustics & vibration processing algorithms such as a power spectrum computation.
- Apply direct peripheral access techniques to achieve the lowest sleep currents for industrial IoT / battery powered applications.

Battery Powered Acoustic Machine Condition Monitoring

One of the best methods of implementing machine health on critical, high value equipment is to analyze the vibration of the machine in real-time. Using high bandwidth accelerometers, it is possible to ascertain the health and possible future failures of a critical asset over time. Combining frequency domain analysis with typical sensing modalities, such as temperature, one can perform complicated sensor fusion and data reduction at the point of measurement. Data fusion and reduction at the "extreme edge" reduces the amount of data needed to be transmitted to a backend improving both transmission costs and battery life. Cloud based machine learning and trending algorithms can have access to just the right amount of data to balance complexity in the sensing node and the backend processing.

Cellular System in Package (SIP) Technologies, such as the Nordic nRF9160, offer a high level of integration enabling densely packaged sensor solutions. In addition to enabling point connectivity, the nRF offers sufficient processing power to handle complex sensor fusion and data reduction tasks.

Prior to Nordic nRF9160 and open source RTOS solutions such as Zephyr, cellular based industrial IOT development could be consuming and difficult. A typical method for connecting sensors is to use an intermediate wireless network. It is common to network sensors via a local wireless connection. Raw sensor data is collected into a "bridge". The bridge may have more processing capabilities and a high-level operating system such as Linux to deal with intricacies of modern, secure connectivity. With Zephyr and the nRF9160, it is possible to do all precision measurements and data processing in-situ and securely connect to a backend without the need for an intermediate bridge. This point of measurement connectivity approach can greatly simplify deployment of sensors for predictive maintenance and condition based maintenance applications.

Furthermore, the nRF9160 can be battery powered using ruggedized chemistries such as lithium thionyl chloride (Li-SoCl2) for years of operation in the most demanding environments. An example of a point of measurement industrial IoT sensor is the "MachineMailbox" shown in figure 1.

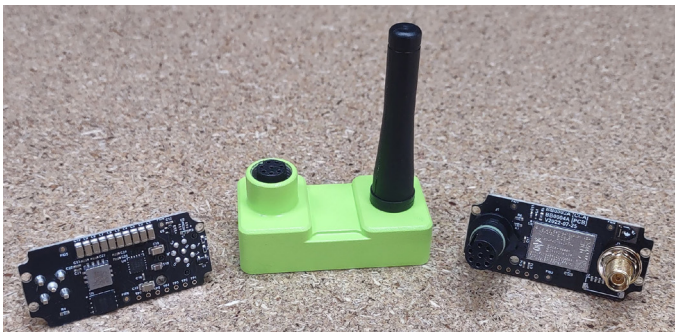

FIGURE 1. THE MACHINEMAILBOX : AN ACOUSTIC MACHINE MONITORING SENSOR USING THE NORDIC NRF9160 AND ZEPHYR.

The MachineBox combines the nRF9160, an Analog Devices ADXL355 3-Axis MEMs accelerometer and an ADT7420 temperature sensor.

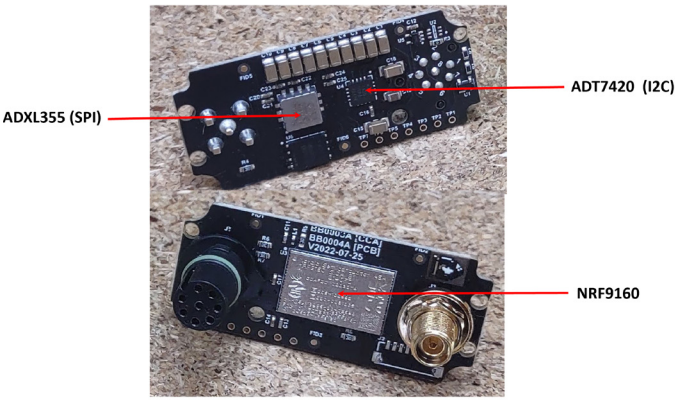

FIGURE 2. NRF9160 BASED ACOUSTIC MACHINE CONDITION MONITORING SENSOR USING THE ADXL355 AND THE ADT7420

Two important considerations when selecting an accelerometer for machine condition monitoring is **bandwidth** and **noise spectral density**. The majority of MEMS accelerometers used for orientation and motion control are optimized for low frequency / DC operation but exhibit significant spectral noise in the regions of interest for structural vibration analysis. The ADXL355 operates in the sweet spot of structural acoustics with a 2KHz usable bandwidth and 22ug/√Hz noise spectral density.

https://www.analog.com/en/technical-articles/mems-accelerometers-for-condition-monitoring.html

Noise spectral density is an important and often overlooked metric when selecting an acoustic sensor. Often the magnitude of vibration of interest is extremely small. Acoustic analysis for machine health and trending is performed in the frequency domain. A power spectral density computation is often the best choice for understanding energy in a vibrating system. The low noise spectral density properties of the ADXL355 combined with processing gain of a Fourier transform based algorithm means one can resolve the smallest of characteristics in a vibration system.

MachineMailbox Vibration Sensor Interface

For the MachineMailbox design, the ADXL355 was connected to the nRF9160 via its SPI interface. The ADXL355 implements a sigma-delta ADC inside of the device package. Streaming vibration data is converted and placed in a FIFO which can be read via SPI transactions.

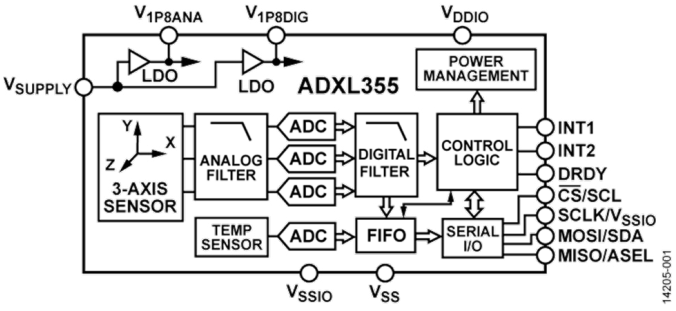

FIGURE 3. THE ADXL355 FUNCTIONAL INTERFACE
[SOURCE : HTTPS://WWW.ANALOG.COM/MEDIA/EN/TECHNICAL-DOCUMENTATION/DATA-SHEETS/ADXL354_355.PDF]

An important point to consider is that once the data stream is started, measurements are pumped into the FIFO continuously. The FIFO is 96 words deep, so it is important to read out data fast enough so there are no discontinuities in the data. The MachineMailbox used the ADXL355 with a 4KHz output data rate (ODR). New sample data is ready every 250uS.

It would be possible to continuously poll the FIFO over the SPI bus. However, this approach is inefficient in terms of CPU activity and energy usage. The DRDY line can be used to efficiently trigger read operations. Acquiring a large capture buffer, say 16384 samples, can be done completely in the background with minimum CPU intervention. With 4KHz ODR, SPI transactions need to occur within the 250uS sample period.

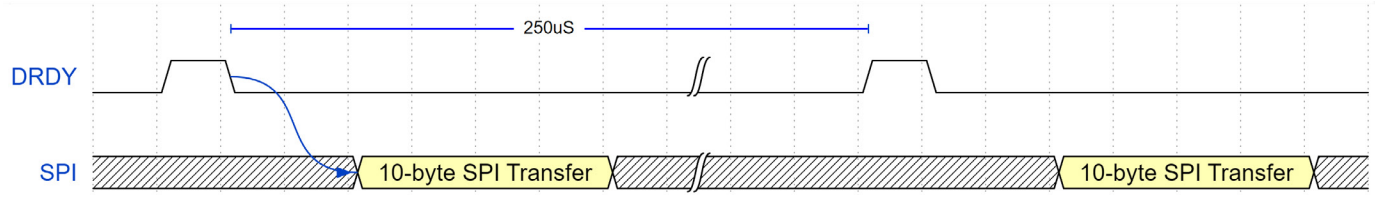

FIGURE 4. ADXL355 DRDY –> SPI DATA READ

With the nRF9160, we can use a combination of GPIO interrupts and SPI DMA to achieve data capture in a background operation while the CPU spends most of its time sleeping. This approach also frees up time for the code to be accessing the ADT7420 temperature sensor. The ADT7420 is accessed via an I2C connection. In the AD7420 “Normal” operating mode, temperature measurements are ready every 240mS. Reads of the ADT7420 can be overlapped with the SPI transactions to ensure the CPU is minimally involved and can be put to sleep when idle. In the MachineMailbox firmware, the ADT7420 is read at a nominal 4Hz sample rate while the ADXL is read at a 4KHz sample rate. Both operations are done with minimal CPU interaction to maximize sleep time.

Direct Peripheral Access Approach

Zephyr has a common driver API for the most common access patterns to typical microcontroller I/O (UART, SPI, I2C, etc.). The APIs enable rapid development and portability across different SOCs. There are use cases however that are not covered by the existing driver models. Microcontroller peripherals can often be interlinked in novel ways to achieve system requirements such as latency or power consumption. In many cases I choose a microcontroller because a peripheral set is unique to the device. It is not uncommon to require specific features that are not handled in generic device models as they are specific to a SOC or device package. Zephyr allows direct access hardware for these use cases so users can get the benefits of the common driver models as well as being able to make use of SOC specific features. This flexibility can simplify development as SOC specific features can be used while still being able to get benefits from other Zephyr services. Developers can choose which features best benefit their project requirements. In the case of the MachineMailbox design, being able to choreograph peripherals directly translated to improved energy consumption. This improvement enabled for a smaller, lower cost battery and improved mechanical packaging. I was simultaneously able to make use of the high-level Zephyr services such as the networking stack and device settings API to be able to focus on the differentiating features of the product.

The SPI (called SPIM) and I2C (called TWIM) peripherals in the NRF9160 are simple to use via direct register access. They have been designed to handle the most common access patterns. The SPIM module has “EasyDMA” functionality. EasyDMA enables simple configuration of a single DMA transfer to/from a SPI peripheral. I have found that the number of lines of code to directly program Nordic EasyDMA peripherals is *less* than using a high level driver API.

For this application, we need to trigger a 10-byte SPI operation whenever the ADXL355 flags new data is ready.

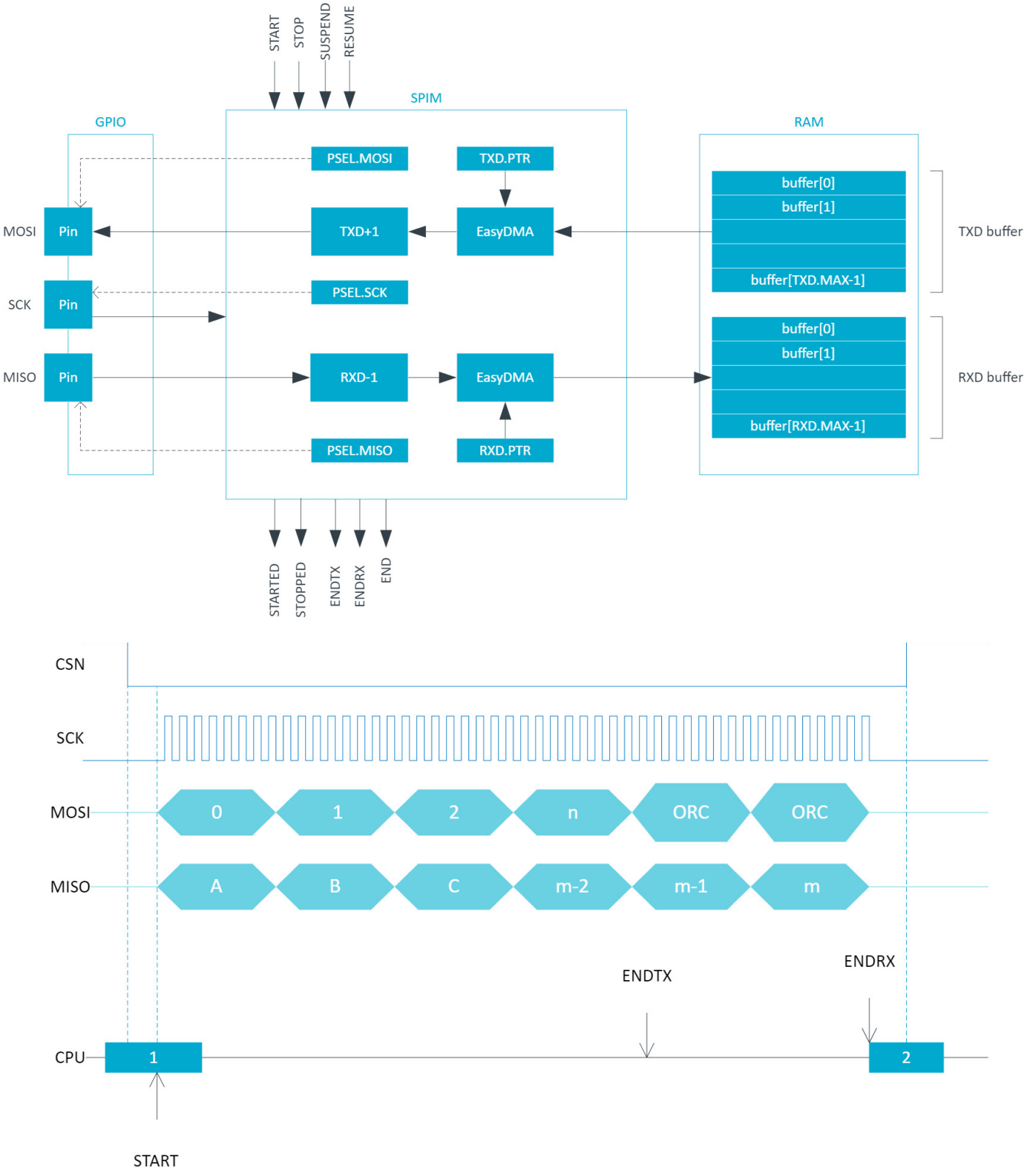

FIGURE 5. NRF9160 SPIM W/ EASYDMA [SOURCE : NRF9160 PRODUCTION SPECIFICATION : HTTPS://INFOCENTER.NORDICSEMI.COM/]

The ADXL355 use case requires smaller transactions to be triggered from an external pin. We can use the NRF GPIOTE module to trigger an interrupt on the rising edge of DRDY. The SPI transmit buffer is fixed in length. It contains control/address bytes that do not change between samples simplifying IRQ handling and DMA restart.

Note: The NRF9160 does have a specialized Distributed Programmable peripheral Interface (DPPI) mechanism that can link events between peripherals to reduce interrupt handling. However, I specifically did not use it as I needed to perform some ancillary operations when new sample data is ready and wanted to show a direct IRQ example in Zephyr.

In the MachineMailbox application, the capture buffer for vibration data is 16384 samples in length. The process for capturing a vibration generally followed these steps:

1. Receive GPIOTE IRQ triggered by the rising edge of ADXL355 DRDY.
2. Format / store data from the last SPI DMA transfer, except on the 1st interrupt
3. Assert the ADXL355 Chip Select
4. Start the next SPIM EasyDMA transfer if the capture buffer is not full.
5. Use SPIM ENDRX IRQ to de-assert CS and flag that capture is complete if the capture buffer is full.

Graphically we can show this interaction as:

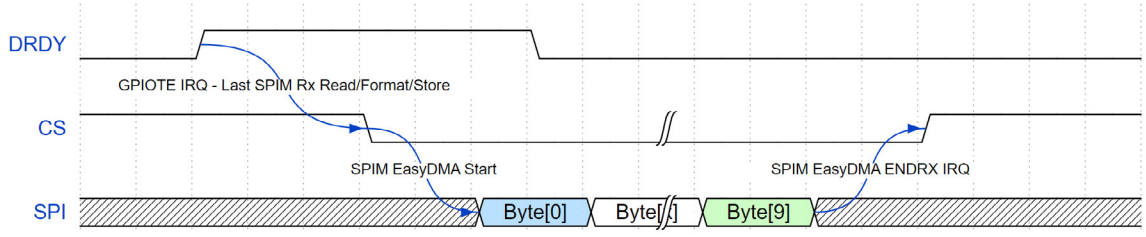

FIGURE 6. GPIOTE, SPIM DMA AND ADXL355 DRDY INTERACTION

In the bare metal context, stitching this behavior together is straightforward. Zephyr allows users to directly attach custom IRQ handlers with little added latency.

Step 1: Enable Zero Latency IRQs

In the Zephyr application prj.conf, zero latency IRQs are enabled with a single **Kconfig** setting

```
CONFIG_ZERO_LATENCY_IRQS=y
```

Step 2: Mark your interrupt handler as a Direct IRQ

In the MachineMailbox application, I used direct IRQ handling for the SPIM2 and GPIOTE peripherals.

```
ISR_DIRECT_DECLARE(DRDY_Callback)
{
    /*
        Insert ADXL355 DRDY ISR code here.
    */
}

ISR_DIRECT_DECLARE(SPIM2_Callback)
{
    /*
        Insert SPIM2 ISR code here.
    */
}
```

The **ISR_DIRECT_DELCARE** macro will add platform specific tasks before and after your routine so it can "plug in". The nRF9160 uses a Cortex M, so you can see what this macro does in the file

`arch/include/arm/aarch32/irq.h`

I always inspect these low-level macros to know **exactly** what is going on.

```
#define ARCH_ISR_DIRECT_DECLARE(name) \
    static inline int name##_body(void); \
    __attribute__ ((interrupt ("IRQ"))) void name(void) \
    { \
        int check_reschedule; \
        ISR_DIRECT_HEADER(); \
        check_reschedule = name##_body(); \
        ISR_DIRECT_FOOTER(check_reschedule); \
    } \
    static inline int name##_body(void)

#if defined(CONFIG_DYNAMIC_DIRECT_INTERRUPTS)
```

You can poke around in the platform **irq.h** to inspect the macros wrapped around your IRQ handler. In the case of the Cortex-M, the wrapper is thin and resolves to very little additional code. It is possible to 100% examine what is added to understand the implications on real-time performance.

Step 3: Link your IRQ handler to the Vector Table

The IRQ_DIRECT_CONNECT is used to attach your specific callback function to an IRQ index. This macro will implement any platform specific behaviors to get your function pointer in the IRQ vector table.

```
IRQ_DIRECT_CONNECT(UARTE2_SPIM2_SPIS2_TWIM2_TWIS2_IRQn, 0, SPIM2_Callback, IRQ_ZERO_LATENCY);
```

The 1st argument is the platform specific IRQ number. I pulled this from **`NRF9160.h`** in **`modules\hal\nordic\nrfx\mdk`**

It is also possible to pull the IRQ index from the device tree. As an example, I connected the GPIOTE IRQ handler using the IRQ index from the device tree.

```
#define GPIOTE_NODE DT_INST(0, nordic_nrf_gpiote)

IRQ_DIRECT_CONNECT(DT_IRQN(GPIOTE_NODE), 0, DRDY_Callback, IRQ_ZERO_LATENCY);
```

You can find more information about direct connected IRQs in the Zephyr documentation:

https://docs.zephyrproject.org/apidoc/latest/group__isr__apis.html

If you want more examples of how to use direct IRQs, you can read through the source code of various peripheral drivers for a specific microcontroller.

Once I put all the IRQ & DMA logic together, I capture the real-world timing with a logic analyzer.

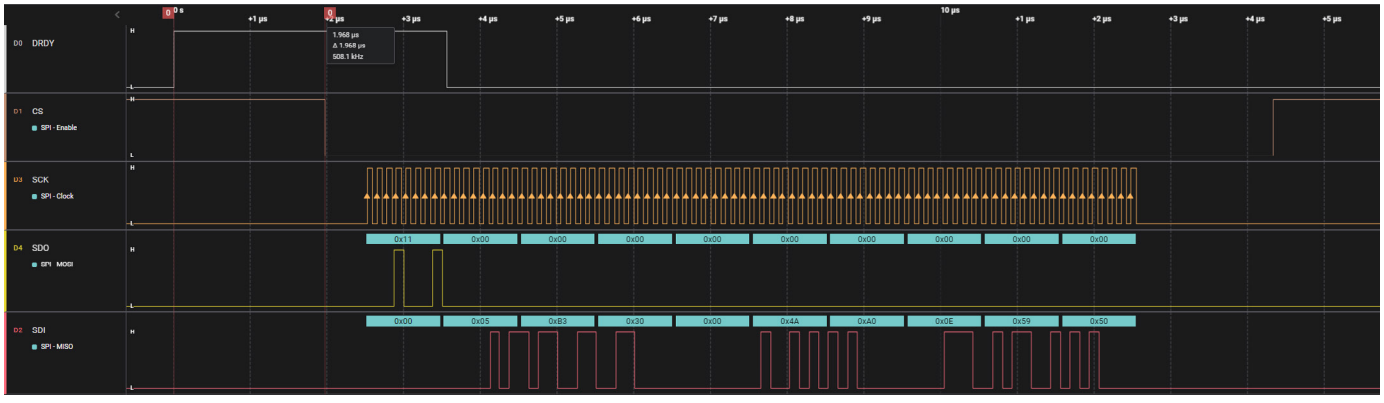

FIGURE 7. DIRECT IRQ TIMING MEASUREMENTS USING A SALEAE LOGIC PRO 8

With the 1st implementation, I measured an approximate 2uS latency between assertion of DRDY and the assertion of CS. The IRQ handler had to perform some housekeeping and buffer management. When I consider the 64-MHz nRF9160 clock rate, flash execution and the assembly code generated by the compiler (from the direct IRQ wrapper and my C code) this latency could be accounted for. With time critical IRQ code, it is a good idea to inspect what the compiler generates. I like to peek at the time critical code sections to see that everything is in order.

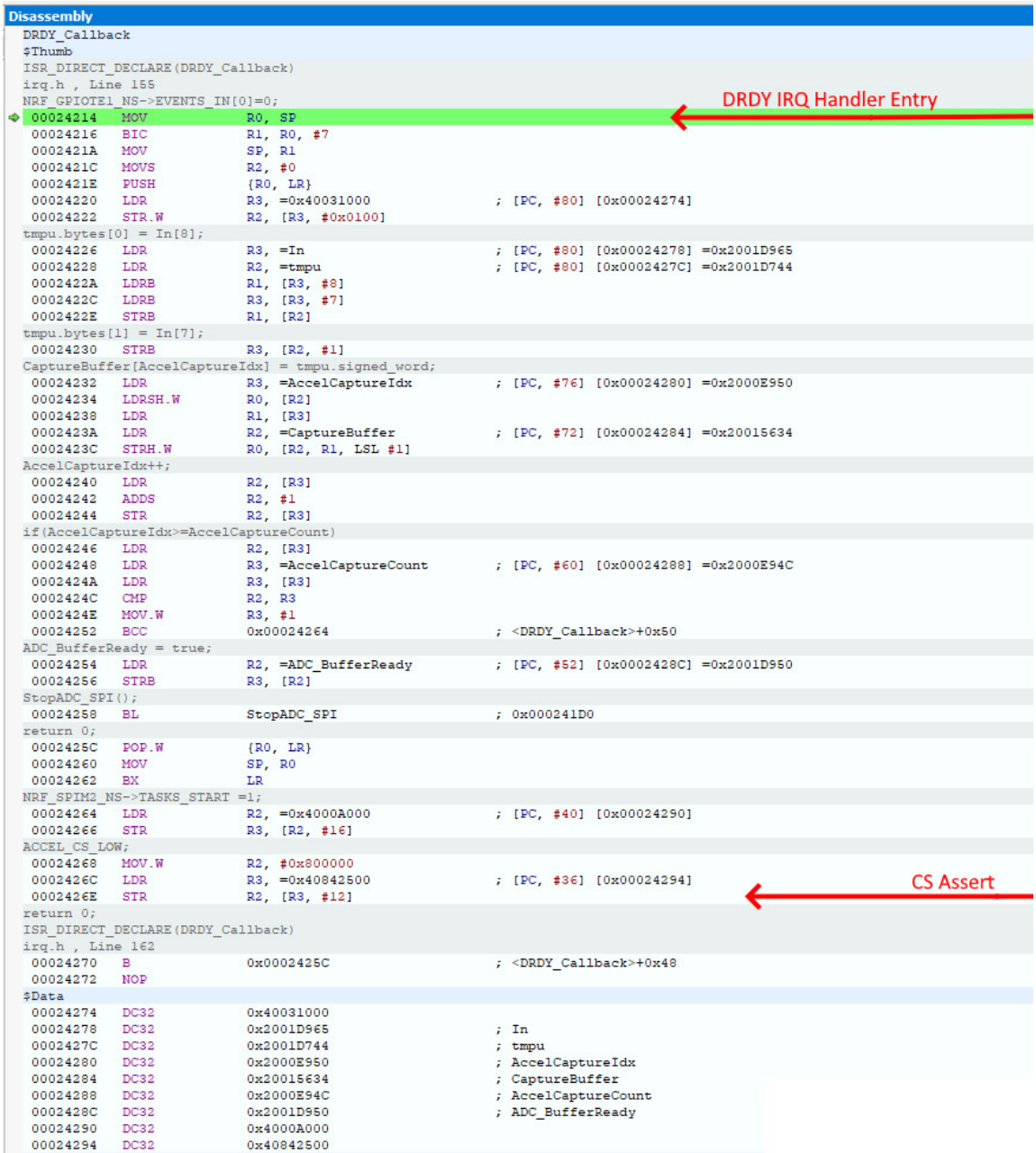

FIGURE 8. DIRECT IRQ ASSEMBLY CODE

The 1st implementation was well within the “good enough” margin to meet my real-time deadlines. Zooming out, I could see that this implementation left quite a bit of idle time for the CPU to remain inactive to conserve power.

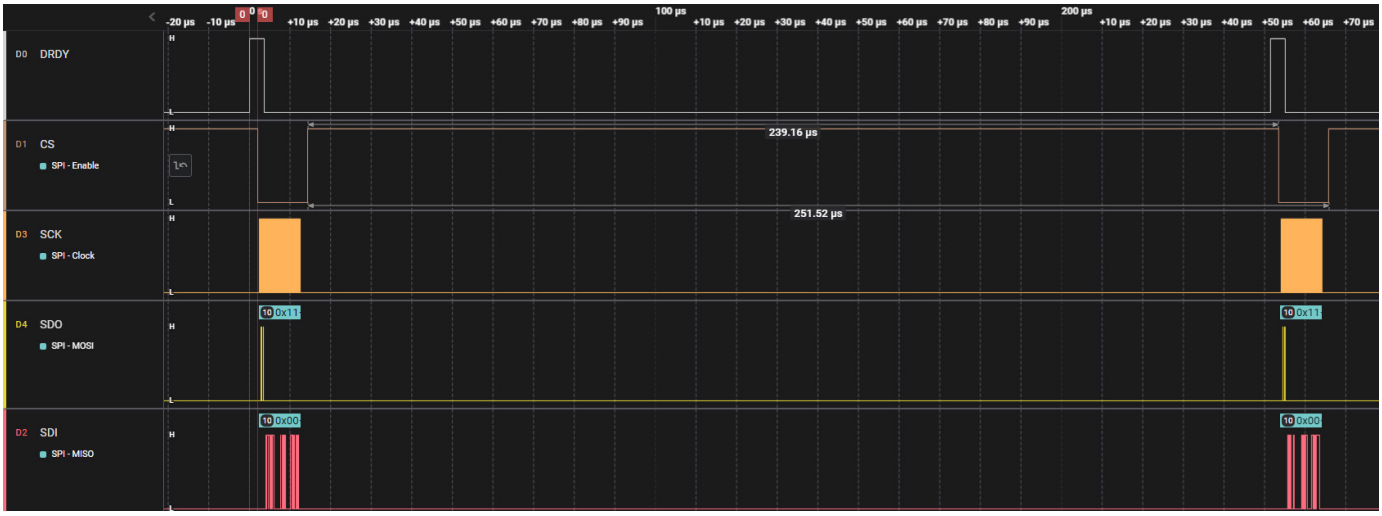

FIGURE 9. REMAINING CPU IDLE TIME IN BETWEEN VIBRATION DATA SAMPLES

While the SPIM/GPIOTE code was running in background IRQ handlers to capture a four second vibration data buffer, I could perform bare metal access to the I2C peripheral) to acquire temperature sensor data.

Accessing peripheral registers directly is as simple as including the NRF9160 headers:

```
#include "nrf9160_bitfields.h"
#include "nrf9160.h"
```

You can then access device registers the same as any other bare metal application. For example, setting or clearing an IO pin is as simple as:

```
#define ACCEL_CS_HIGH     NRF_P0_NS->OUTSET = (1<<ACCEL_CS_PIN)
#define ACCEL_CS_LOW      NRF_P0_NS->OUTCLR = (1<<ACCEL_CS_PIN)
```

This approach makes IO access both fast and simple. The trade-off of this approach vs the Zephyr device driver model is application portability. However, given the specialized nature of this application and the nRF9160 SIP, this was an easy trade to make.

The MachineMailbox DSP and Data Processing Approach

Many acoustic machine monitoring applications boil down to analyzing data recorded from rotating machinery and connecting physical structures. The measured time signal consists of cyclical behaviors combined with a relatively stationary background noise. In many applications, the cyclical behavior you are looking for can be small in magnitude in relation to other stochastic elements in the signal. The background "noise" is typically stationary which is a fancy way to say the statistics of the noise don't change significantly over the timescale you are looking at.

Frequency domain processing is a highly efficient method for separating the different components in complex acoustic data. The workhouse of frequency domain analysis in acoustic machine monitoring is a power spectrum estimation.

https://en.wikipedia.org/wiki/Spectral_density

The simplest explanation of the power spectrum output is a set of **magnitude squared values** (V^2, g^2) the signal normalized to some spectral width (commonly 1Hz). A power spectrum can give one an estimate of where the energy of signal if located in the frequency domain. This is helpful when analyzing signals which are known to have cyclical characteristics in the presence of other random processes. The power spectrum is mathematically defined for infinitely long, continuous signals. However, we exist in the real world of finite sample data. A common approach to estimating a power spectrum of discrete data is via Welch's method.

Welch's Method

Welch's method [297] (also called the *periodogram method*) for estimating power spectra is carried out by dividing the time signal into successive blocks, forming the periodogram for each block, and averaging.

Denote the m th windowed, zero-padded frame from the signal x by

$$x_m(n) \triangleq w(n)x(n+mR), \quad n = 0, 1, \ldots, M-1, \; m = 0, 1, \ldots, K-1, \tag{7.26}$$

where R is defined as the window *hop size*, and let K denote the number of available frames. Then the periodogram of the m th block is given by

$$P_{x_m,M}(\omega_k) = \frac{1}{M}\left|\mathrm{FFT}_{N,k}(x_m)\right|^2 \triangleq \frac{1}{M}\left|\sum_{n=0}^{N-1} x_m(n)e^{-j2\pi nk/N}\right|^2$$

as before, and the Welch estimate of the power spectral density is given by

$$\hat{S}_x^W(\omega_k) \triangleq \frac{1}{K}\sum_{m=0}^{K-1} P_{x_m,M}(\omega_k). \tag{7.27}$$

In other words, it's just an average of periodograms across time. When $w(n)$ is the rectangular window, the periodograms are formed from non-overlapping successive blocks of data. For other window types, the analysis frames typically overlap, as discussed further in §6.13 below.

FIGURE 10. WELCH'S METHOD FOR COMPUTING POWER SPECTRA. [SOURCE HTTPS://CCRMA.STANFORD.EDU/~JOS/SASP/WELCH_S_METHOD.HTML]

DSP code can look obtuse from the purely mathematical expression. However, it is relatively simple to show the algorithm graphically.

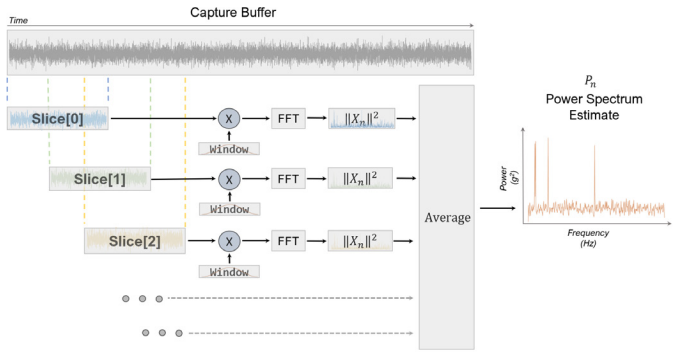

FIGURE 11. GRAPHICAL REPRESENTATION OF AN EMBEDDED IMPLEMENTATION OF WELCH'S METHOD W/ CMSIS DSP ON THE NRF9160

Starting with a large capture buffer, you take overlapping slices of data. Each slice is multiplied by a "window" function. This product fed to an FFT. The magnitude squared value of the FFT result is computed and then Averaged. The result is a set of data that represents the power in a range frequency bin. It is common to normalize the power in each bin to a 1Hz bin width.

Writing embedded code to estimate a power spectrum on time-series data requires a Fast Fourier Transform (FFT). The open-source ARM CMSIS DSP libraries are baked into Zephyr. You can enable specific features in the application proj.conf file.

```
CONFIG_CMSIS_DSP=y
CONFIG_CMSIS_DSP_TRANSFORM=y
CONFIG_CMSIS_DSP_TABLES_ALL_FFT=n
CONFIG_CMSIS_DSP_TABLES_RFFT_FAST_F32_2048=y
```

CMSIS DSP is structured such that you can enable only the bits you want. In the case of the FFT implementation, you can enable only the lengths that you plan on using, saving flash memory used by the twiddle lookup tables, etc. My implementation of a power spectrum estimation used 2048 point FFTs.

Once enabled in the **proj.conf** file, using CMSIS DSP is straightforward. You need to allocate an instance of the FFT struct:

```
arm_rfft_fast_instance_f32  MyFFT;
```

Initialize it:

```
arm_rfft_fast_init_f32(&MyFFT, 2048);
```

and then use it!

```
arm_rfft_fast_f32(&MyFFT, &AnalysisWindow[0], &FFT_Out[0], 0);
```

This is an example of the floating-point version of the CMSIS DSP FFT. CMSIS DSP also supports fixed point versions for 16-bit and 32-bit data types. The Cortex M4F core in the nRF9160 has hardware floating point capabilities. Several years ago, I did a comparison study of the CMSIS DSP FFT performance across the different input sizes and data types. In terms of speed the floating-point implementation is faster than the 32-bit fixed point. The 16-bit version is the quickest as the compiler can make use of some 16-bit SIMD instructions that efficiently compute the FFT butterflies. However, make sure to read the CMSIS DSP documentation if you use the fixed point versions as there are details about the integer scaling through the FFT pipeline that must be taken into consideration.

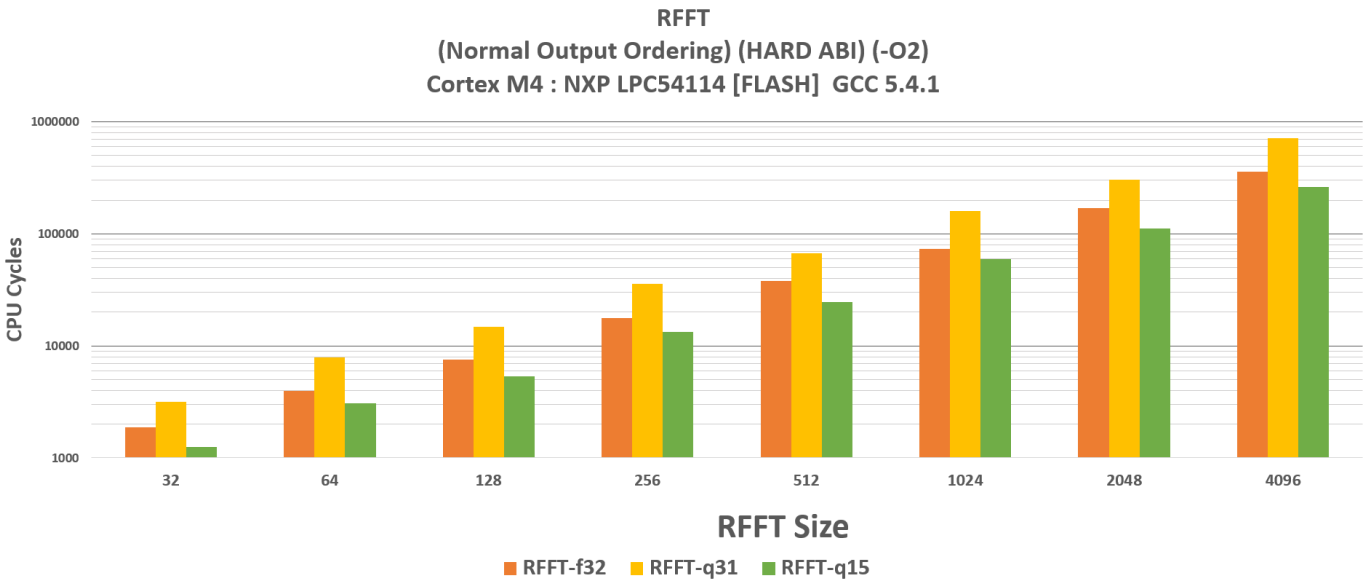

FIGURE 12. CMSIS DSP FFT COMPARISON [SOURCE: HTTPS://GITHUB.COM/EHUGHES/ESC-M4]

For the MachineMailbox application, processed data was transmitted to a backend using a secure, stateless protocol over UDP. Sending frequency domain vibration data to a cloud backend can be challenging if one does not set reasonable boundaries on the size of the data. Since this is a low bandwidth cellular CAT-M application, it was important to send enough data to enable machine monitoring algorithms in the cloud backend while keeping an eye on the cellular data costs. It was also beneficial for a single vibration spectrum to fit into one Ethernet V2 MTU of 1500 bytes, so the data set did not have to be split over multiple frames.

The output of the power spectrum estimation can yield a wide dynamic range through the FFT and averaging process. For the purposes of processing frequency domain data, we can use a priori knowledge of the measurement physics to compress the data a bit.

The unit of the acceleration power spectrum is g^2 over the width of the frequency bin. Given the squared nature of the magnitude, we can transform bins of data into the log domain:

$$10\ log(P_n)$$

This computation gives us an acceleration value in dB scaling (dBg). We can round and store as a signed 8-bit integer. This process will yield a possible power range of -128dBg to +127dBg with 1dBg steps. This is an incredibly wide dynamic range while being able to resolve nearly imperceptible acceleration in a frequency domain bin (-128dBg is *very* small).

From the perspective of machine monitoring / anomaly detection, a 1dB step is a good tradeoff for resolution vs data compression. I personally have never observed any real-world anomaly detection models for rugged/ industrial processes that required triggers less than 1dBg. It is also my opinion that log scaled power spectrum data is one of the most efficient data sources for neural net based processing. Physics-informed, pre-processed data can greatly simplify both the training and real-time crunching of anomaly detection models.

Circuit Current Measurements for Data Acquisition/Processing

Once the data capture and DSP pieces were functional, I measured the total current of the MachineMailbox circuit assembly. This data was fed into a power consumption model to make assessments of battery life. I used a [Joulescope](#) DC energy analyzer to monitor current consumption to profile the system.

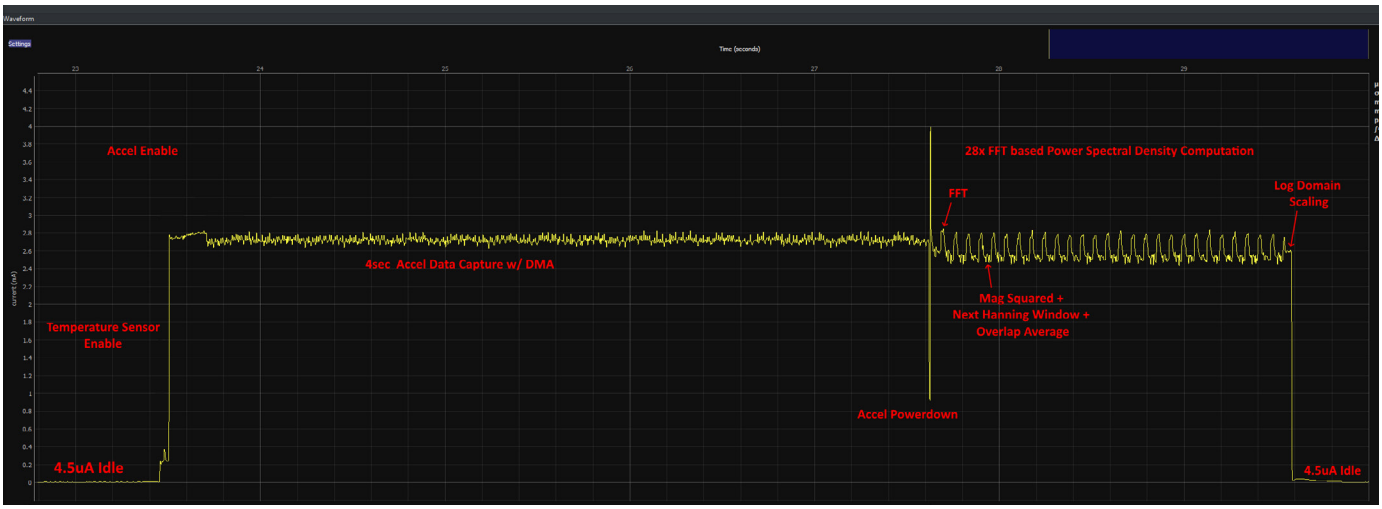

FIGURE 13. POWER PROFILE OF THE FIRST IMPLEMENTATION OF THE VIBRATION DATA CAPTURE AND DSP

Once I got the initial pipeline working, I could more easily identify where I should be focusing time to optimize. For example, to improve battery life, I could choose to overlap some of the power spectrum computations with the data capture. Since the power spectrum requires fixed size slices of data, the FFT and averaging could be started as soon as the 1st slice is available. Because we are using highly tuned interrupt-based data capture, it is possible to get a lot of work completed in an efficient manner and get the NRF9160 back to sleep quickly.

In the MachineMailbox use case, I found that most of the optimization needed to be directly at queueing up vibration spectrums to minimize the number of transmits to the cloud backend. Even when taking CAT-M PSM states into consideration, there was a delicate balance of how often to measure data and how often to transmit to achieve multi-year battery life.

Conclusion

The combination of direct IRQ , multi-peripheral interface combined with embedded DSP and high-level Zephyr network APIs enabled me to get pretty vibration / temperature pictures such as this:

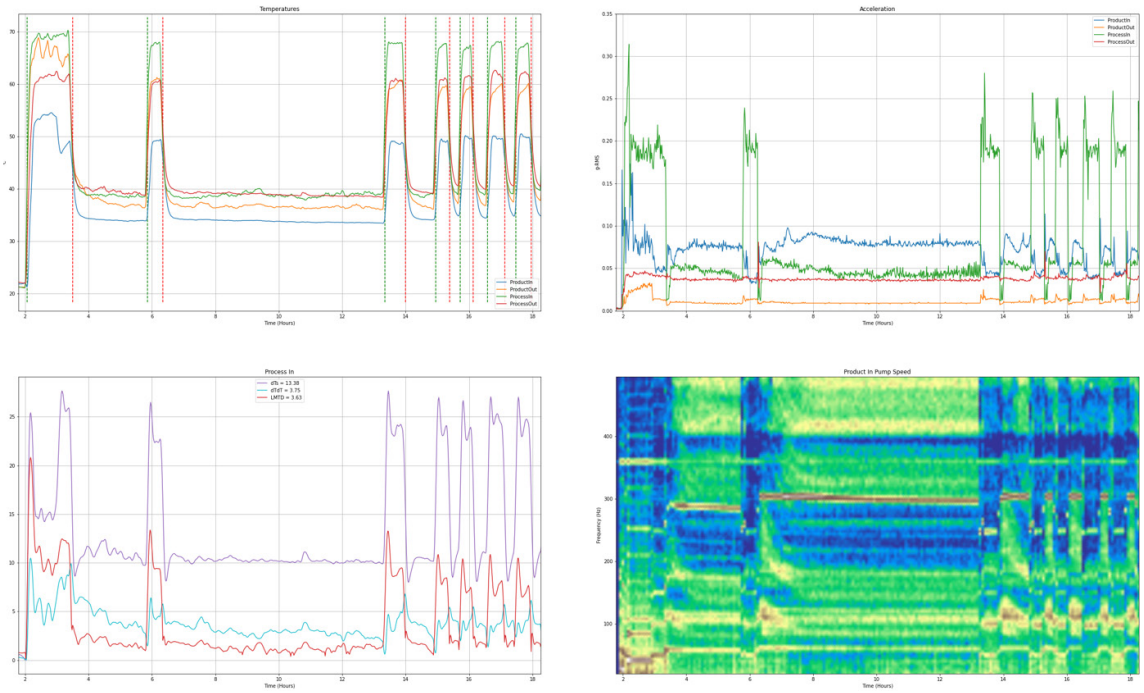

FIGURE 14. PROCESSED VIBRATION AND TEMPERATURE DATA FROM THE MACHINEMAILBOX

The vibration power spectrums can accumulate in the backend over time to be stacked to form 2D images. It is easy to see where AI / machine learning can come in to extract trends and patterns.

Using Zephyr enabled me to get the best of all worlds. I could operate as close to the “metal” as I wanted while getting all the benefits of the build system, high level RTOS APIs, network drivers and common system functions such logging, shells and flash settings. The result for the MachineMailbox was a vibration sensing system that could be easily installed and deliver edge processed data securely to a cloud backend via the cellular networks. All of this while achieving multi-year battery life in a real-world acoustic machine monitoring application.