Real-Time Adaptive Digital Filter on STM32 with FreeRTOS

Abstract:

This project implements a real-time digital signal processing system on the STM32F446RE microcontroller, using FreeRTOS. Analog signals are acquired via ADC & DMA, processed using an Infinite Impulse filter, and transmitted over UART. A MATLAB script running on the host PC receives the data stream, parses it, and plots the output of the filter in real time.

The system demonstrates the integration of peripherals with FreeRTOS scheduling, CMSIS-DSP for efficient filter execution, and MATLAB for visualization. A major design choice for this project was to only use IIR filtering, justified by the efficiency it provides.

This document explains my design choices, as well as the technical problems I faced, how they were resolved, and the learning outcomes of this project.

Introduction and Motivation:

Embedded systems are often required to perform signal processing in real time. Some microcontroller applications and other projects I have done use a super-loop architecture. While this is a sufficient design for simple tasks, this approach struggles with concurrency, scalability, and responsiveness in real-time systems that handle multiple asynchronous data streams.

This project demonstrates how FreeRTOS enables a more structured and maintainable solution for DSP tasks by providing deterministic task scheduling and inter-task communication. The key motivations behind this project are:

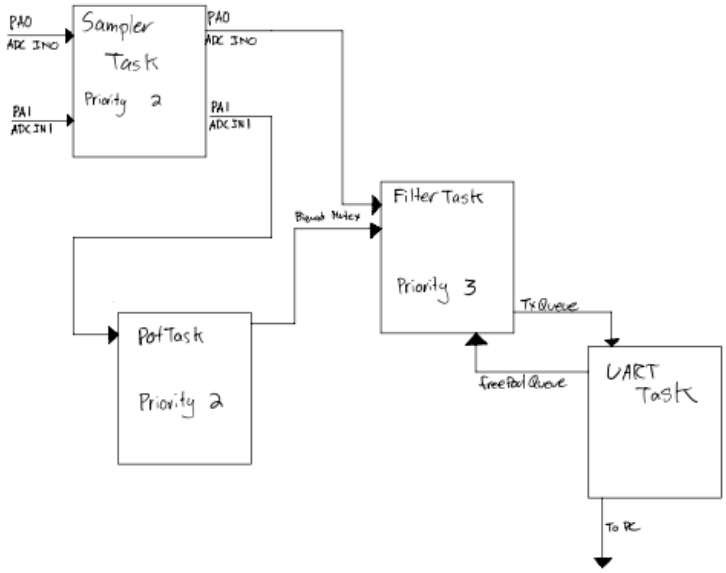
1. Real-Time Responsiveness: ADC samples must be processed as they arrive without dropping any data.
2. Concurrency: The system must simultaneously acquire data, process the data with filters, and transmit the results.
3. Flexibility: The potentiometer dynamically adjusts the DSP filter characteristics, requiring adaptive control.
4. Extensibility: The design should allow new sensors to be integrated with minimal redesign.

This semester I took ENSC 320, where I learned about analog filter design. Through further discussion with Rodney Vaughan, I became very curious about filters and their design. At the time, I was looking for a project that would allow me to learn development in an RTOS environment as well as multi-threaded development. I decided to work on this project as it would allow me to explore filters inside the digital realm as well as gain the experience with an RTOS that I was seeking. After some discussion with my peers and Rodney, this project seemed like a perfect fit. This project can then later be scaled and utilized in later projects and courses. It can be a building block for projects done in, Digital Signal Processing (ENSC 429) and even my capstone project.

System Overview:

This project is designed to be a modular component of a larger design/system. Thus, this project as a standalone is constrained by CPU cycles, deterministic timing, and memory use. For these reasons the filter I chose to implement was a second-order IIR filter, with the coefficients updated dynamically based on the potentiometer input. An IIR filter is a better choice because of the number of coefficients needed vs the FIR filter. The need for a filter and the difference between IIR and FIR are defined below.

Block Diagram:



Task Structure:

1. SamplerTask:
    - Sets up ADC data acquisition via DMA.
    - Triggers only once on start up of schedular.
2. FilterTask:
    - Applies the IIR filter and fills the TxBlock buffers.
3. UARTTask:
    - Formats the data into CSV format and streams the data over UART.
4. PotTask:
    - Periodically samples injected ADC channel, updates filter coefficients.

Queues:

In this project there are two queues between the tasks, specifically the SamplerTask and the UARTTask. The first queue holds TxBlock ready to transmit, these blocks are populated by the FilterTask that fills the buffer with samples. The UART task pulls from this queue, formats it, then sends it over UART to the host PC to be plotted.

The second queue is for the pool and manages the available empty buffers the FilterTask can take and fill. This block is filled and then sent over the TxQueue and sent to the

UART task. The UART task will transmit the data, then it will return the buffer back to the pool queue.

This design allows the system to not need malloc and free at runtime which can cause problems in the system, such as memory fragmentation and unwanted latency. The memory is pre-allocated in the set-up and is recycled through the pipeline.

This design choice was made using my prior experience with embedded systems through the SFU robot soccer design team, ENSC 254 (Introduction to Computer Organization), and the FreeRTOS documentation. From these experiences, I knew that dynamic memory allocation during runtime is not a good design choice in real-time systems because it can lead to fragmentation, unpredictable latencies, and memory leaks. To avoid these issues, I knew it was best to reuse allocated memory and allocate all resources once at the startup of the system.

If I had designed the UART pipeline using dynamic memory allocation, memory management could have become one of the major technical challenges of this project, especially under the load of continuous data streaming. While dynamic allocation might have functioned under lighter loads, it would not be best practices in embedded system design.

Task Scheduling & Timing Analysis:

- MCU/clock: STM32F446RE @ 180MHz
- Sampling rate: $f_s = 1000Hz$
- DMA capture buffer: length of 1024 (half/full transfer IRQs at 512 sample boundaries)
- Streaming block size: The size is 128, so there is one block every 128ms

Task Set and Priorities

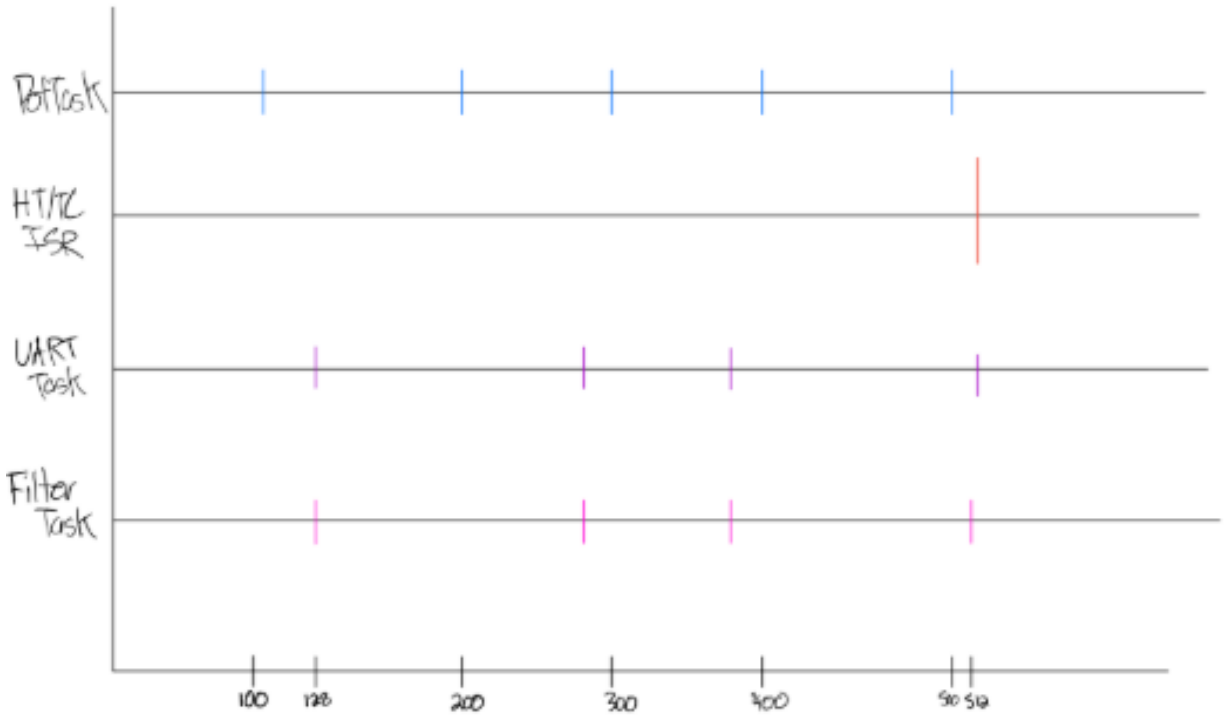
Task/ISR	Trigger	Period	Priority
ADC & DMA Half/Full ISR	**DMA HT/TC events**	**512ms**	**IRQ level**
FilterTask	**Notified by ISR**	**128ms (per TxBlock)**	**High**
UarTTask	**TxQueue not empty**	**128ms**	**Medium**
PotTask	**Periodic**	**100ms**	**Low**

At a sampling rate of 1kHz, the 1024 sample buffer corresponds to 1024ms. The DMA is configured to generate interrupts on the half transfer and transfer complete events. Each half buffer is then 512ms of data. The DMA ISR signals the FilterTask to process the completed half. This forms a ping-pong buffer which results in continuous, non-blocking data acquisition.

Rationale:

- DMA ISR must be highest to keep acquisition deterministic.
- FilterTask should preempt UARTTask to filter the new data.
- UARTTask runs whenever blocks are available.

Task Scheduling Timing Diagram:



Filter Design:

In embedded digital signal processing systems, filtering is one of the most fundamental operations. The purpose of a digital filter is to shape the frequency content of sampled signals in order to remove noise, reject unwanted components, or condition the data before further processing. In this project, an LDR and a potentiometer are connected to the ADC subsystem. The data is streamed into a FreeRTOS application for real time signal processing and visualization on a host PC via UART.

FIltering in this context is motivated by the following needs:

1. Noise reduction: Raw sensor data from analog components is inherently noisy. A digital low-pass filter can reduce high frequency fluctuation, yielding smoother signals for display and analysis.
2. Bandwidth control: Different sensors or operating conditions may require different effective bandwidths.
3. Real-time adaptability: Instead of designing a fixed filter for a specific sensor and specifications, this project provides a tunable cutoff frequency that is controlled by the potentiometer. This allows exploration of the filter's performance at different cut-off frequencies in real time.

Digital filters can be categorized into finite impulse response (FIR) and infinite impulse response (IIR) filters.

1. FIR filters compute each output as a weighted sum of a finite number of past inputs. They are always stable and can be designed to have perfectly linear phases. However, achieving sharp cutoff would require a large number of taps, which will consume more CPU cycles and memory, which I want to limit the consumption of.
2. IIR filters compute each output as a weighted sum of past inputs and past outputs. Because the recursion feeds output back into the computation, the impulse response extends infinitely. IIR filters achieve sharp roll-off with fewer coefficients. But, it does introduce nonlinear phase distortion.

RBJ Biquads Background:

When I was in the design stage of this project and learning about digital filters. ChatGPT 4.0 suggested the RBJ cookbook equations. This provides closed-form formulas for designing second-order digital IIR filters such as low-pass, high-pass and bandpass filters. These are simple, stable, and directly produce digital coefficients without requiring the bilinear transform. The stability of the filter in the Z-domain requires all poles to lie inside the unit circle.

The general form of a Biquad filter is as follows:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$$

This corresponds to the transfer function,

$H(z) = \frac{b_0+b_1z^{-1}+b_2z^{-2}}{1+a_1z^{-1}+a_2z^{-2}}$, where b_0, b_1, b_2 are the numerator feed-forward coefficients and a_1, a_2 are the denominator feedback coefficients.

For the RBJ equations there are some key intermediate variables that are defined as,

$\omega_0 = 2\pi\frac{f_c}{f_s}$, $\alpha = \frac{sin(\omega_0)}{2Q}$, $c = cos(\omega_0)$, where

ω_0: $Normalized\ digital\ resonance\ frequency$

α: $Controls\ bandwidth\ and\ damping$

c: $Simplifies\ formulas$

For this project, I wanted a lowpass filter, so the coefficients for a low pass filter section are:

$b_0 = \frac{1-c}{2}$

$b_1 = 1 - c$

$b_2 = \frac{1-c}{2}$

$a_0 = 1 + \alpha$

$a_1 = -2c$

$a_2 = 1 - \alpha$

The coefficients then need to be normalized by a_0 so that the denominator's leading term is 1. After this normalization, the filter has unity gain in the passband and is ready for implementation.

This filter design also allowed me to learn about the z-transform and the z-domain that I have heard about through my analog filter design class. I learnt that the z-transform is the mathematical tool that allows us to analyze discrete time systems in the frequency domain. It plays the same role as the Laplace transform in continuous time systems.

A major technical challenge in this project was determining an appropriate sampling frequency that would not overwhelm the filter task and UART interface. Initially, I considered much higher sampling rates, from 10KHz all the way up to 500KHz, under the assumption that faster sampling rates would yield better accuracy. When testing the project modularly, specifically testing the ADC & DMA in isolation, 500KHz was reasonable, but once the filter was implemented and analyzing the block diagram and timing sequence that such a high rate would cause the filter task to preempt other FreeRTOS tasks and saturate the UART bandwidth, which would make other task execution and reliable real-time streaming almost impossible. After using the resources available to me such as concepts from ENSC 320 and discussions with the professor, Rodney, I came to the conclusion that the filter only needs to handle signals with frequencies below 500Hz. Which means a 1KHz sampling frequency is sufficient according to the Nyquist criterion. This process taught me the importance of system design choices, The importance of integration design between modular components, and assuming higher sampling speeds is better. By utilizing the resources around me, I was able to solve this challenge efficiently and effectively in a timely manner.

An additional note for the choice of the sampling frequency, is that it would need to be increased if the sensor to be implemented with the filter was an accelerometer, gyroscope, and audio. This would require an increase in sampling frequency which would in turn increase the CPU load and possible design changes for the real-time streaming and task preemption. My next step in this project is having the project work with all sensors, and possibly have a tunable sampling frequency. I can likely do this alongside my ENSC 320 professor Rodney Vaughan, as his research works with audio and RF applications.

The sampling frequency I set is 1000Hz, which gives a Nyquist frequency of 500Hz. Since the maximum cutoff frequency of any digital filter must lie below Nyquist, the tunable range I defined was from 2Hz to 400Hz. This provides flexibility in the filter to provide

meaningful output, I can learn and observe in the MATLAB script. At very low cutoff frequencies the filter smooths the noisy analog signal, while at high cutoffs it becomes nearly transparent, allowing the raw sensor behavior and noise to be observed. Choosing 400Hz rather than 500Hz leaves a small gap band that avoids instability and inaccuracies in the calculations near Nyquist.

Nyquist-Shannon Sampling Theorem:
To avoid aliasing, the sampling rate must be at least twice the highest frequency present in the input signal:

$f_s \geq 2f_{max}$

Hardware Peripherals:

ADC + DMA:

For this project to have a user-adjustable control for the filter, I connected a potentiometer to PA1 which is sampled using the MCU's injected ADC channel. Unlike the regular ADC channel that runs continuously with the DMA to acquire sensor data streams, the injected channel allows me to have on demand, single shot conversions that will not interfere with the DMA buffer that is being used for the other ADC channel. This design allows the potentiometer to be read at a low-rate task period of 100ms and used to dynamically adjust the cutoff frequency of the filter. By isolating the data acquisition of the potentiometer into the injected channel path, the system separates the continuous sensor sampling ensuring deterministic DMA operation while supporting user-tunable filter parameters.

UART Streaming:

For the UART streaming, I went with a design where the transmission over UART is done in blocks. This is because sending one sample at a time would be very inefficient, there would be too many interrupts and context switching. By grouping the samples into blocks, I could reduce the overhead and improve the throughput.

The pool was designed to have 6 buffers. So, there is a maximum of 6 blocks between the FilterTask producer and UARTTask consumer. The queue can also only have a maximum of 8 pending blocks waiting for the UART task. This design choice was made because UART is very slow compared to ADC sampling and filtering. If the UART task falls behind, the procedure can keep queuing more blocks. This provides a safety margin between fast producers and the slow UART transmission lines.

I chose these specific values for the number of buffers and queue depth, as a practical balance that is large enough to smooth our rate mismatches, but small enough to respect the CPU and memory constraints.

Potentiometer:

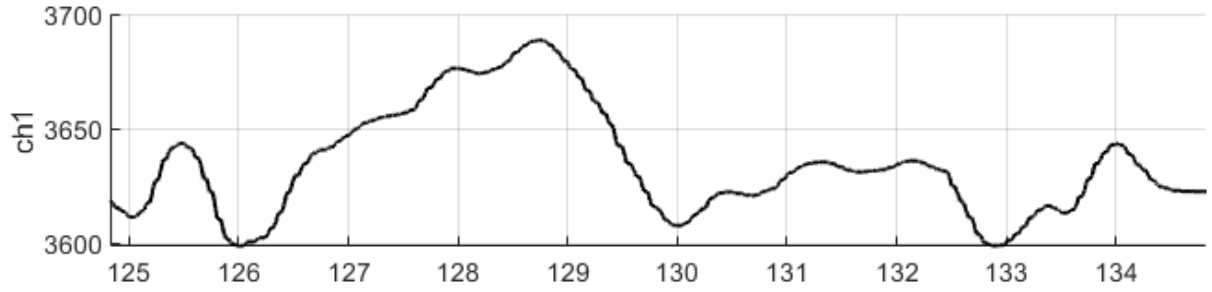
One of the major technical challenges in this project was ensuring the potentiometer provided an effective control response. My initial design used linear mapping, but testing and

observing the output on the host PC revealed large changes at low cutoff frequencies and small changes at higher frequencies. Correcting this disproportionate behavior was important, since adaptive control is a core feature of this project, and inconsistent responsiveness would have created significant problems when tuning the filter while integrating with different sensors. In my ENSC 320 class, I learned why logarithmic scales are used, and how they are used to represent wide ranges of data because they compress large variations into smaller groups of data.. So, to address this issue, I decided to implement logarithmic interpolation which is described below.

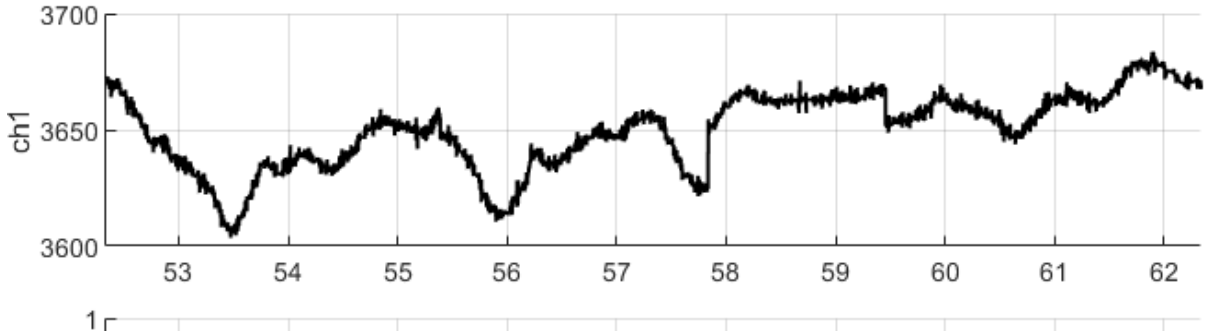
For this design, the potentiometer's 12 bit ADC reading is mapped to the filter cutoff frequency using logarithmic interpolation between 2Hz and maximum of 400Hz to ensure small knob adjustments have consistent effect from low to high frequencies. I chose this frequency range so the behavior of the filter at different cutoff frequencies will be meaningful to observe. I also chose it to make sure it is clamped below the Nyquist frequency. This ensures that the cutoff frequency remains numerically well-behaved across the whole range of the potentiometer. With the logarithmic mapping, each step corresponds to a ratio of frequencies instead of a fixed portion of a Hertz. This means every adjustment of the potentiometer corresponds to the same proportion of cutoff frequency change.

Results:

Low Cut-off:



High Cut-off:



The system behaviour was validated by sweeping the potentiometer across its full range and observing the filter's output. With the potentiometer fill turned left, the cutoff frequency is low and the filter strongly attenuates the higher frequencies. The output of this configuration appears smooth, with only small variations visible demonstrating effective noise suppression.

When the potentiometer is fully turned right, the cutoff frequency approaches the Nyquist limit. In this configuration, the entire signal bandwidth lies within the passband, so the filter's transfer function is effectively unity and the output closely models the raw sensor data with high frequency fluctuations. Which can be seen above.

These results confirm the potentiometer mapping successfully tunes the cutoff frequency of the filter and the system behaves as expected.

Conclusion:

This project successfully demonstrated the implementation of a real-time adaptive digital filter on an STM32 microcontroller under FreeRTOS. By using DMA drive ADC data acquisition, IIR filter processing, UART streaming, and adaptive control, the system successfully achieved real-time operation with a tunable cutoff frequency. Through this project, I learned not only technical details about digital filter design, but also the system level considerations of concurrency and memory management.

A key outcome of this project was that the design decisions must be balanced across subsystems. Higher sampling rates and simplistic mappings appeared sufficient in isolated tests, but can create problems when integrated together in the full system. By integrating them together and iterating through the design, I was able to identify potential bottlenecks and change the design.

A key learning outcome that this project provided me was experience in multi-threaded RTOS development, including inter-task communication and deterministic scheduling. This project complemented my coursework and prepares me for future projects using DSP and embedded systems.

A continuation of this project would be to extend the system to operate with higher bandwidth sensors such as accelerometers, gyroscopes, and microphones. This would require possibly tuning the sample frequencies and changing the filtering portion of the system.

1. Add/test unit test to the freertos.c file
2. Fix MATLAB script
3. Add code to github