SS2025

ESPS – Lab 3 Report

SUBMISSION DEADLINE: 6.7.2025

EDAET BOLOBAN – 1111858, KHANH HOANG TRIEU - 772906

Inhaltsverzeichnis

1. RT EMA-Filter using Q15 Fixed-Point Arithmetic

1.1. Introduction

In this project, an Exponential Moving Average (EMA) filter is implemented in two different formats: Q1.15 fixed-point arithmetic and floating-point arithmetic. The system operates on an STM32 microcontroller platform running at a sampling rate of 48 kHz.

A sinusoidal input signal with a 1.8 V peak-to-peak amplitude and a 1 V DC offset is sampled via the integrated 12-bit ADC. The sampled data is processed in real time within an interrupt service routine, applying the EMA filter in both floating-point and fixed-point implementations.

The filtered output signals from both approaches are then converted back to analog form and output via the microcontroller's integrated DAC channels. This setup allows for a direct comparison of the numerical accuracy and performance of the two filter implementations under identical conditions.

The exercise aims to illustrate the practical differences between floating-point and fixed-point processing, including considerations such as scaling, saturation, and execution time in an embedded environment.

1.2. Setup

The system is configured to acquire and process the analog input signal connected to PA0 using the integrated 12-bit ADC. The ADC operates in single conversion mode with right-aligned data and a resolution from 0 to 4095. Conversions are not performed continuously but are instead triggered externally by Timer 6, which generates a trigger output (TRGO) event on each rising edge. This setup ensures that sampling occurs deterministically at a fixed rate of 48 kHz, corresponding to one conversion every 20.83 microseconds. Overrun data is preserved to avoid data loss if the ISR is delayed.

Mode	Independent mode
ADC_Settings	
Clock Prescaler	Synchronous clock mode divided by 4
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Gain Compensation	0
Scan Conversion Mode	Disabled
End Of Conversion Selection	End of single conversion
Low Power Auto Wait	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
Overrun behaviour	Overrun data preserved
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Enable Regular Oversampling	Disable
Number Of Conversion	1
External Trigger Conversion Source	Timer 6 Trigger Out event
External Trigger Conversion Edge	Trigger detection on the rising edge

Figure 1: Configuration for ADC

To generate this trigger, Timer 6 is configured with a prescaler of 4 and a counter period of 708 minus 1. Each update event triggers the ADC conversion automatically without CPU intervention. This timer-based approach guarantees a precise and stable sampling frequency.

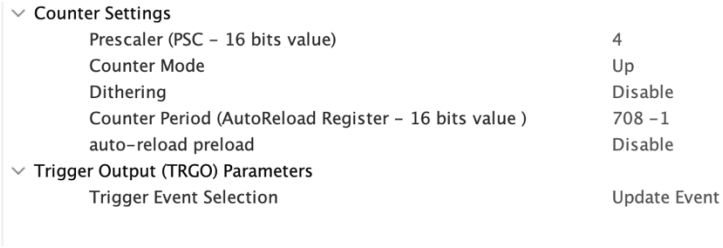

Figure 2: Configuration for TIM 6

The output of the filter implementations is provided via the DAC channels. DAC Out1 on PA4 outputs the original input signal as received from the ADC, allowing a direct comparison between the unfiltered and filtered waveforms. DAC Out2 on PA5 outputs the filtered signal computed by the active implementation, either the floating-point EMA or the fixed-point (Q1.15) EMA filter. Both DAC channels operate in normal mode with the output buffer enabled and no hardware trigger, meaning the outputs are updated directly by the software routine after each ADC conversion. High-frequency mode is set to automatic, and the signed format is disabled to match the standard unsigned DAC output range.

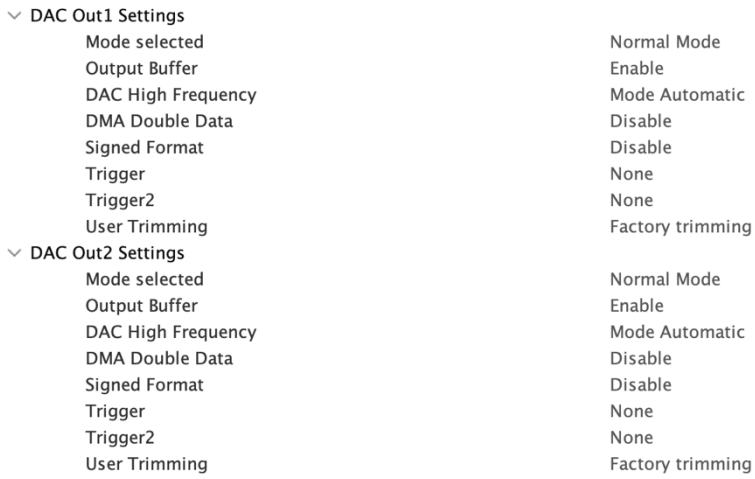

Figure 3: Configuration for DAC

Additionally, the GPIO pin PA9 is configured as a digital output. Inside the interrupt service routine, this pin is toggled each time a sample is processed. By monitoring PA9 with an oscilloscope or logic analyzer, the actual sampling rate and the execution time per ISR call can be accurately verified.

1.3. Implementation in Matlab

The Exponential Moving Average filter was evaluated in MATLAB using a sinusoidal input signal with a frequency of 500 Hz, a peak-to-peak amplitude of 1.8 V, and a DC

offset of 1 V. The smoothing factor was set to α=0.5. The figure below shows a direct comparison between the input signal and the outputs of the floating-point and Q1.15 fixed-point implementations. Both filters were executed on the sampled data at a sampling rate of 48 kHz.
The plotted waveforms demonstrate that the EMA filter effectively smooths the input signal while maintaining its overall shape and amplitude. The floating-point output and the Q1.15 fixed-point output are almost identical and closely follow the original waveform. The measured peak-to-peak values confirm the excellent agreement: the floating-point implementation yields 1.792 V, and the fixed-point implementation shows 1.792 V as well, compared to 1.800 V for the unfiltered input. This indicates that the scaling, quantization, and fixed-point arithmetic have been implemented correctly and introduce only minimal numerical error.

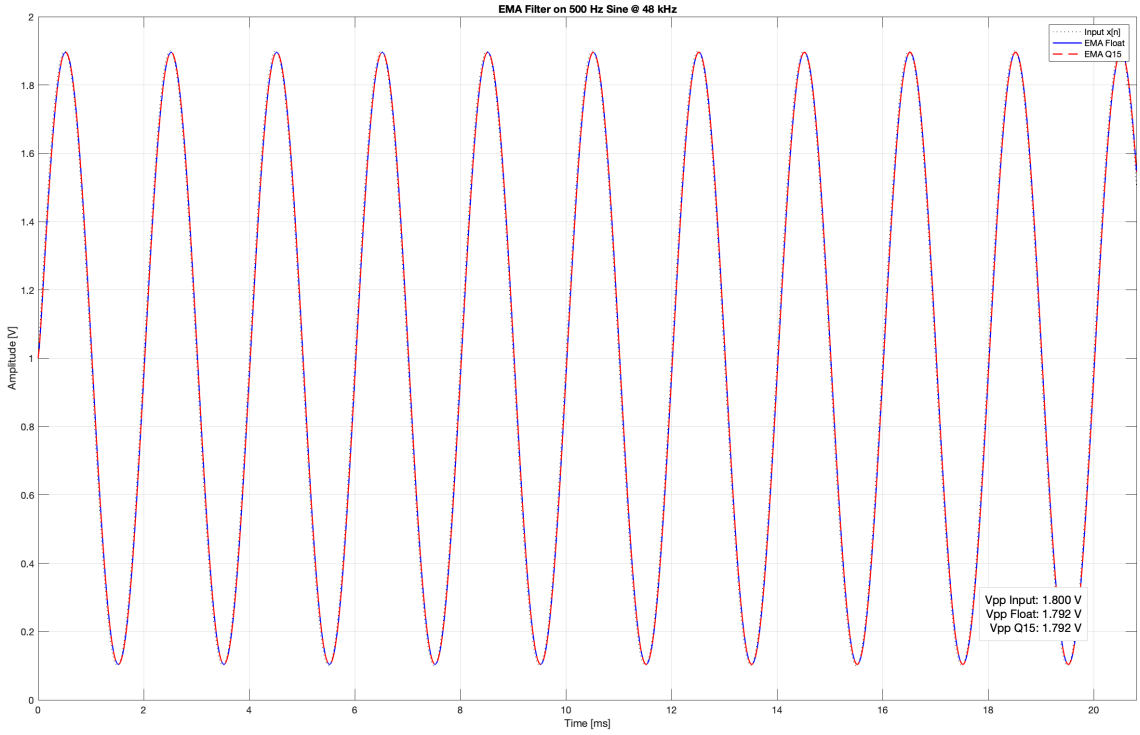

Figure 4: Simulation of the unfiltered and filtered signal (500Hz)

To further quantify the deviation between the floating-point and fixed-point implementations, the figure below shows the error signal computed as the difference between the floating-point output and the Q1.15 output over time. The error remains very small throughout the entire duration of the signal. Its magnitude stays within the range of approximately 10×10^{-5}, which corresponds to the quantization noise of the fixed-point arithmetic and the limited resolution of the Q1.15 format.
The error waveform exhibits a periodic pattern synchronized with the sine wave input, reflecting the small quantization steps that occur during multiplication and scaling. Despite this, the error is negligible in practice and demonstrates that the Q1.15 EMA filter is highly accurate for this application.

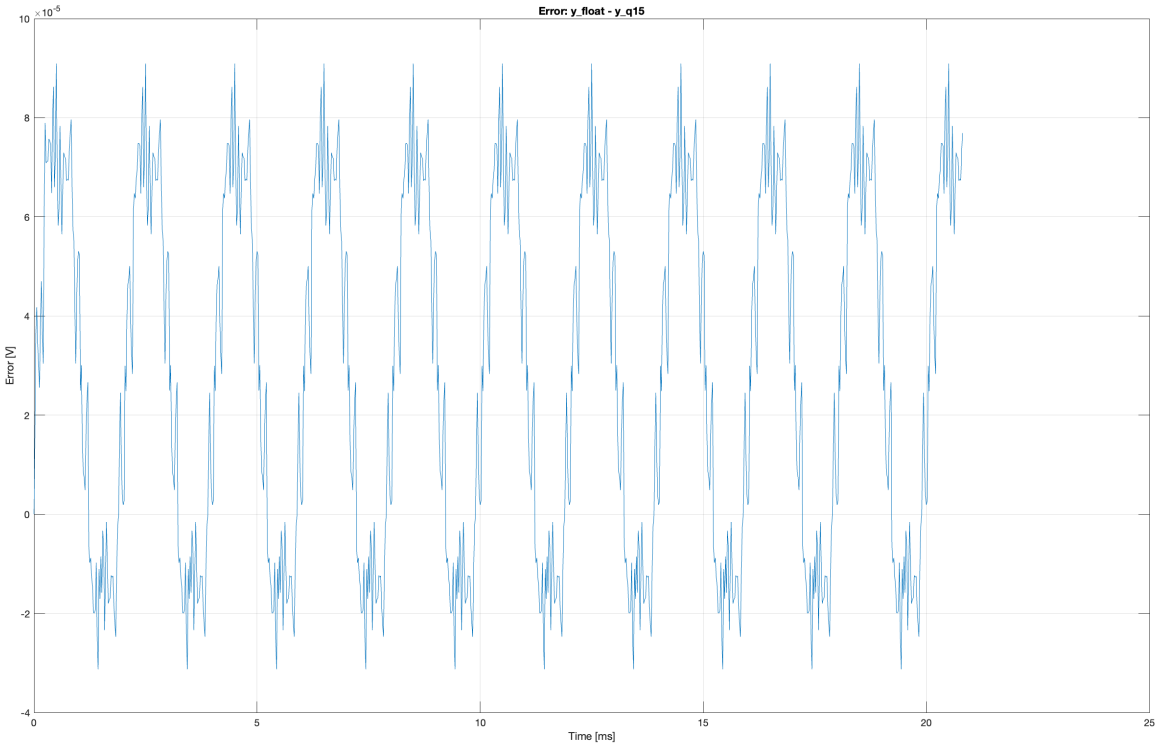

Figure 5: Error between float and fixed-point EMA outputs

1.4. Implementation on STM32-G474RE

Figure 6 and Figure 7 show oscilloscope measurements of the filtered 500 Hz sine wave output from the STM32 microcontroller. Channel 1 (yellow) displays the original input signal, while Channel 2 (blue) shows the filtered output generated by the EMA implementation running in real time.

In Figure 6, the filter was implemented using Q1.15 fixed-point arithmetic. The output waveform follows the input closely, with only a slight attenuation in amplitude, which is expected for the applied smoothing factor $\alpha = 0.5$. The measured peak-to-peak voltage of the filtered signal is approximately 1.82 V compared to 1.86 V for the input, confirming that the fixed-point processing preserves the signal characteristics with minimal deviation.

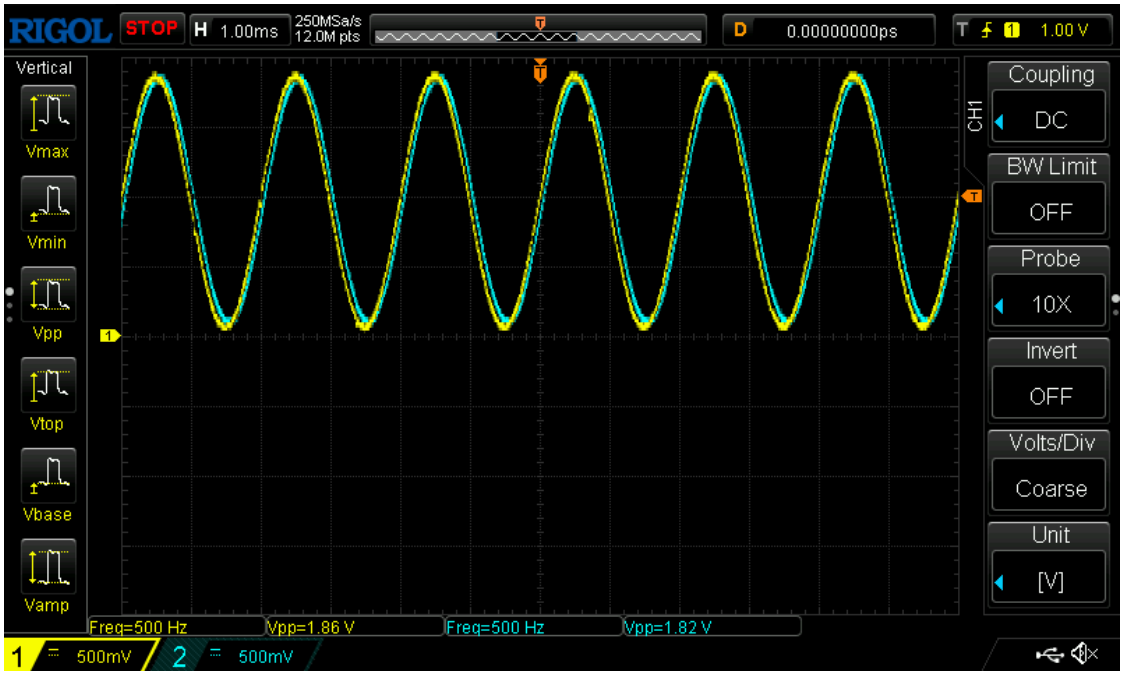

Figure 6: Input (yellow) and output (blue) signal (Q1.15)

In Figure 7, the same filter was executed using floating-point arithmetic on the STM32's hardware FPU. Again, the filtered output is nearly identical to the input signal. The measured peak-to-peak voltage in this case is 1.82 V, matching the fixed-point result within measurement accuracy.

The comparison demonstrates that both implementations achieve the intended smoothing effect with negligible differences in amplitude or phase. This validates the correctness of the fixed-point scaling and saturation logic implemented in the Q1.15 version, and highlights that, for this application, fixed-point filtering can deliver results virtually identical to floating-point processing while reducing computational load and memory requirements.

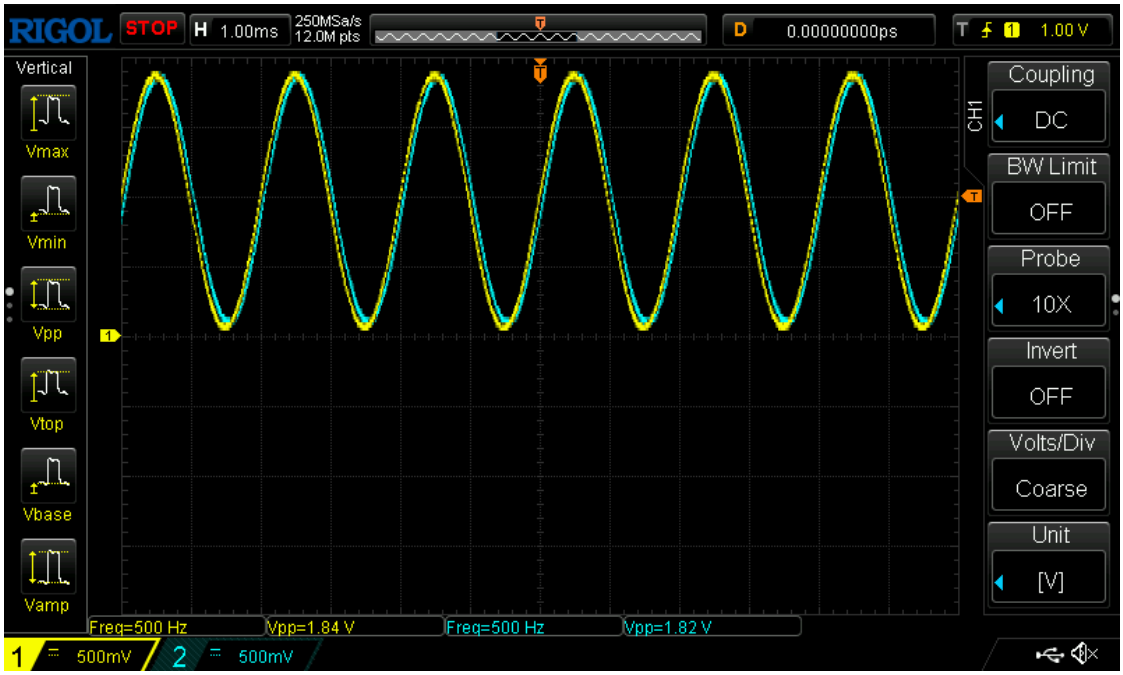

Figure 7: Input (yellow) and output (blue) signal (floating-point)

1.5. Discussion

The following oscilloscope screenshots show the execution time of the EMA filter measured on the STM32 microcontroller by toggling a GPIO pin inside the interrupt service routine. In both cases, the time, was measured at a sampling rate of 48 kHz and corresponds to the total processing time required per sample.

Figure 8 shows the Q1.15 fixed-point implementation. The measured execution time is approximately 800 ns. This very short duration demonstrates the high efficiency of fixed-point arithmetic on the Cortex-M4 core, which can perform integer multiplications, shifts, and additions with minimal instruction cycles.

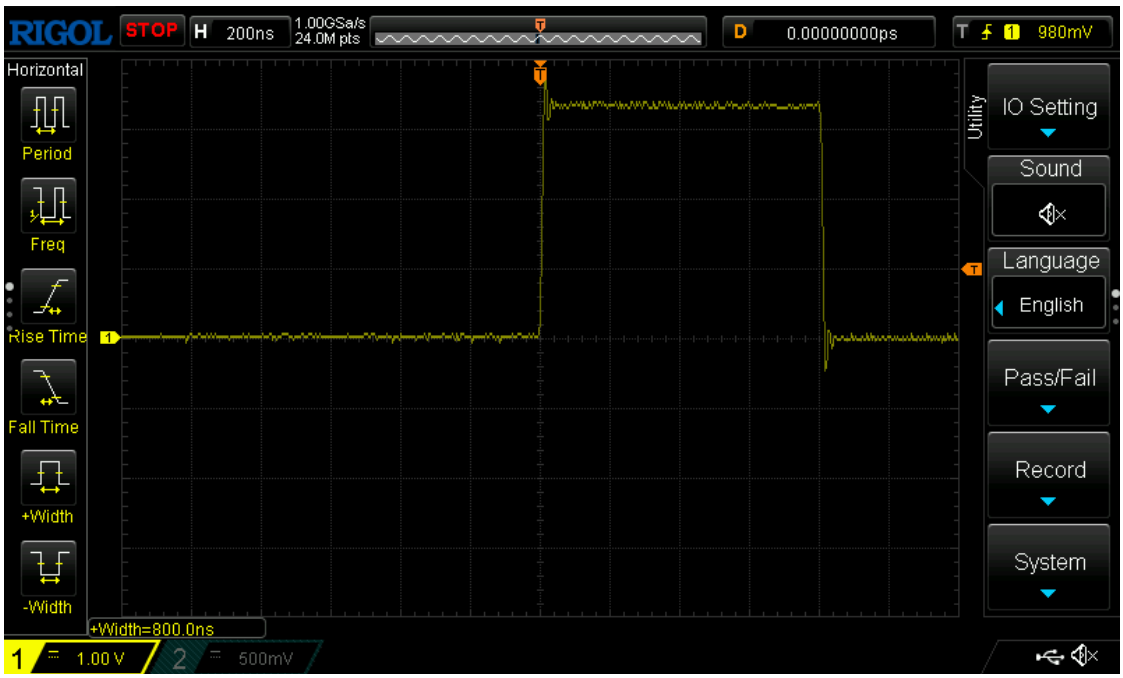

Figure 8: Execution time for the Q1.15 implementation

Figure 9 shows the floating-point implementation. Here, the execution time is slightly lower, measured at approximately 776 ns. The Cortex-M4 features a hardware floating-point unit (FPU) that accelerates single-precision calculations. As a result, the floating-point version in this specific case even outperforms the fixed-point version slightly, since no manual scaling or saturation logic is required and the filter operation is implemented using straightforward arithmetic.

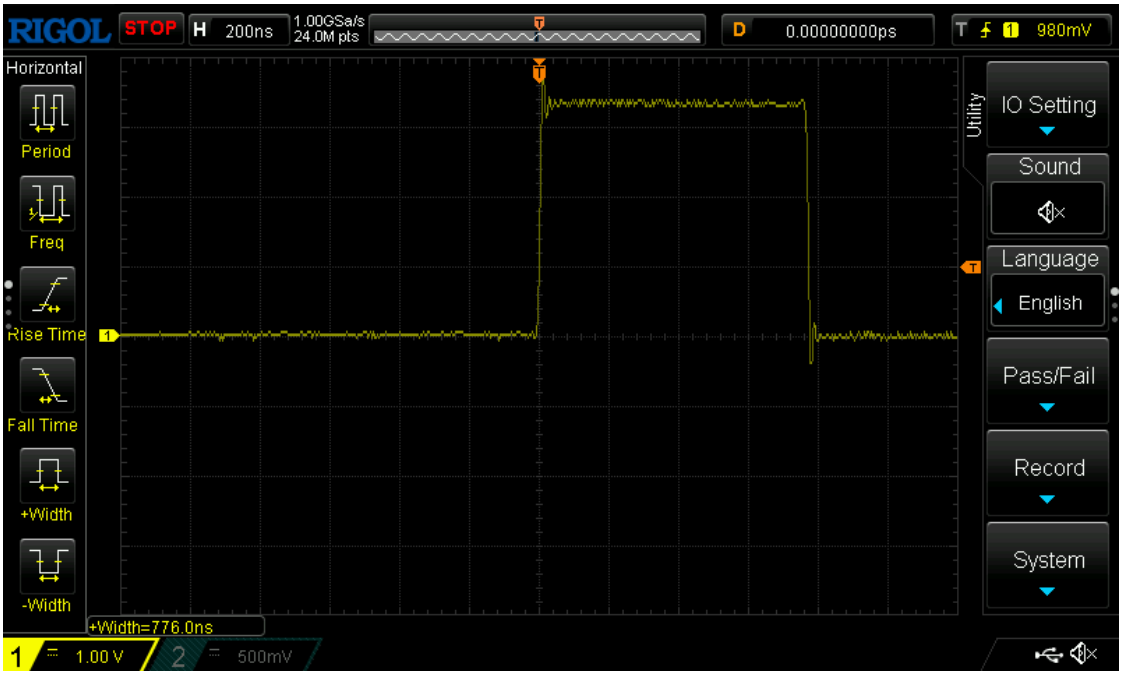

Figure 9: Execution time for the fixed-point implementation

This observation underlines that the performance advantage of fixed-point arithmetic depends heavily on the processor architecture. On older or low-power microcontrollers without an FPU, floating-point operations can require significantly more clock cycles, making Q1.15 or other fixed-point formats the preferred choice. In such environments, fixed-point filters are essential to meet strict real-time constraints, especially at high sampling rates or when multiple filters must run concurrently.
On modern MCUs with hardware floating-point support, like the STM32G4 series used here, the choice between floating-point and fixed-point implementations can be guided more by other considerations, such as:

- **Memory footprint**: Fixed-point buffers and variables require only 16 bits, reducing RAM consumption.
- **Deterministic behavior**: Fixed-point arithmetic guarantees fully predictable rounding and saturation.
- **Development complexity**: Floating-point code is often easier to implement and less prone to scaling errors.

In summary, both implementations demonstrated excellent performance, with execution times well below 1 microsecond. Under the test conditions, the floating-point filter is marginally faster thanks to FPU acceleration. However, fixed-point arithmetic remains an important alternative in embedded systems without floating-point hardware or in applications that require minimal memory usage and strict determinism.

2. Implementation of high-order FIR filters

2.1. Introduction

In this project, a high-order finite impulse response (FIR) filter is designed, implemented, and evaluated on an STM32G474RE microcontroller platform. The main objective is to develop a real-time filtering solution that processes sampled analog input signals using floating-point arithmetic.

The filter coefficients are generated in MATLAB with the fir1 function, and these coefficients are then exported as C code to integrate them into the microcontroller firmware. The designed filter is implemented in two different versions on the microcontroller: a naive approach with manual array shifting of input samples, and an optimized version using a circular buffer.

Both implementations are tested at a sampling rate of $48\,kHz$ to assess their performance, memory usage, and execution time. Additionally, the impact of compiler optimization levels and different filter orders is investigated. This exercise provides insight into the trade-offs between implementation complexity, computational efficiency, and resource requirements in embedded digital signal processing applications.

2.2. Implementation in Matlab

In MATLAB, the filter coefficients are generated using the fir1 function, which creates a low-pass FIR filter based on order N = 16 and cutoff frequency $f_C = 2\,kHz$. These coefficients are then used to plot the frequency response of the filter, illustrating both the magnitude and phase characteristics of the designed system (see Figure 10).

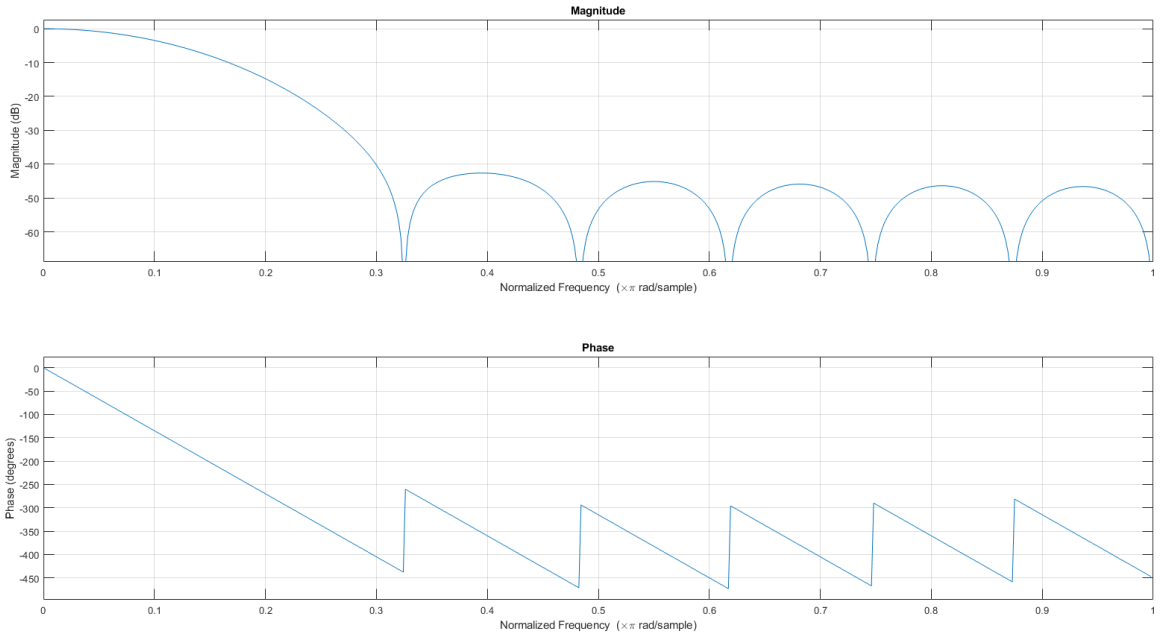

Figure 10: Frequency response of the FIR filter

2.3. Naive Implementation

In this implementation, the FIR filter is realized using a straightforward approach based on array shifting. For every new input sample, all existing buffer entries are shifted by one position to the right, and the newest sample is stored at index 0. This ensures that the convolution always multiplies the coefficients in the intended order, with the most recent sample at the start of the buffer. Although this method is simple to understand and easy to implement, it is computationally inefficient, since every sampling cycle requires moving all 16 buffer elements.

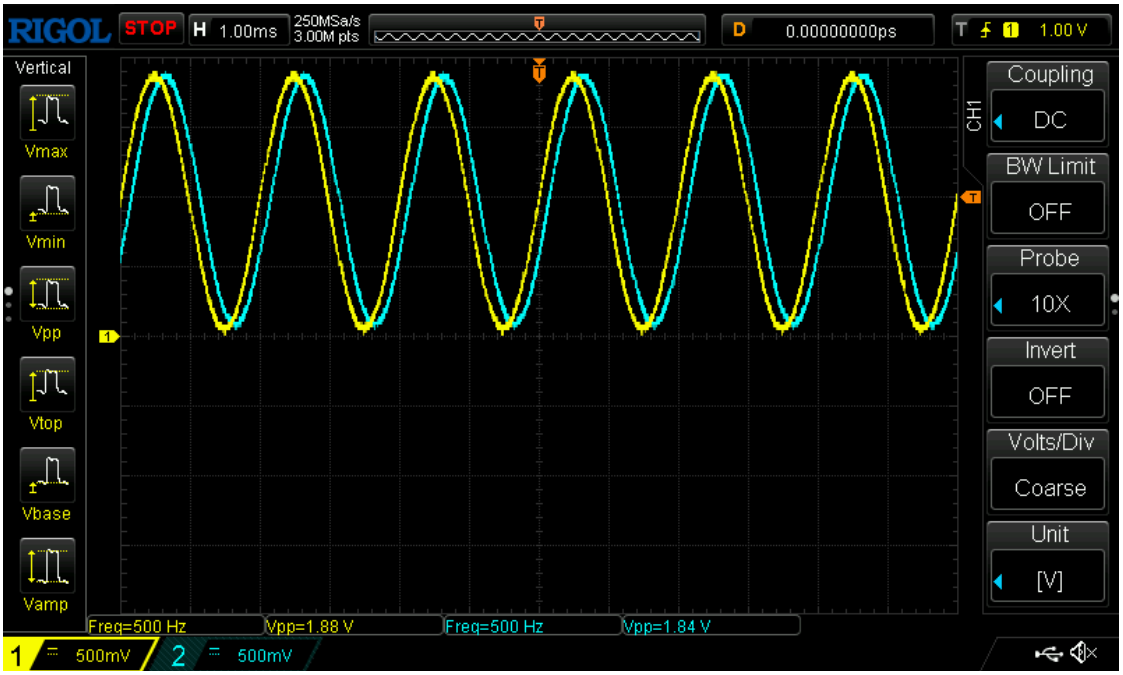

Figure 11: Input (yellow) and output (blue) @ 500Hz (Naive approach)

To validate the filtering behavior, measurements were performed using a signal generator and an oscilloscope. The designed filter is a low-pass FIR filter with a cutoff frequency set to $2kHz$. This is clearly visible in the recorded waveforms.

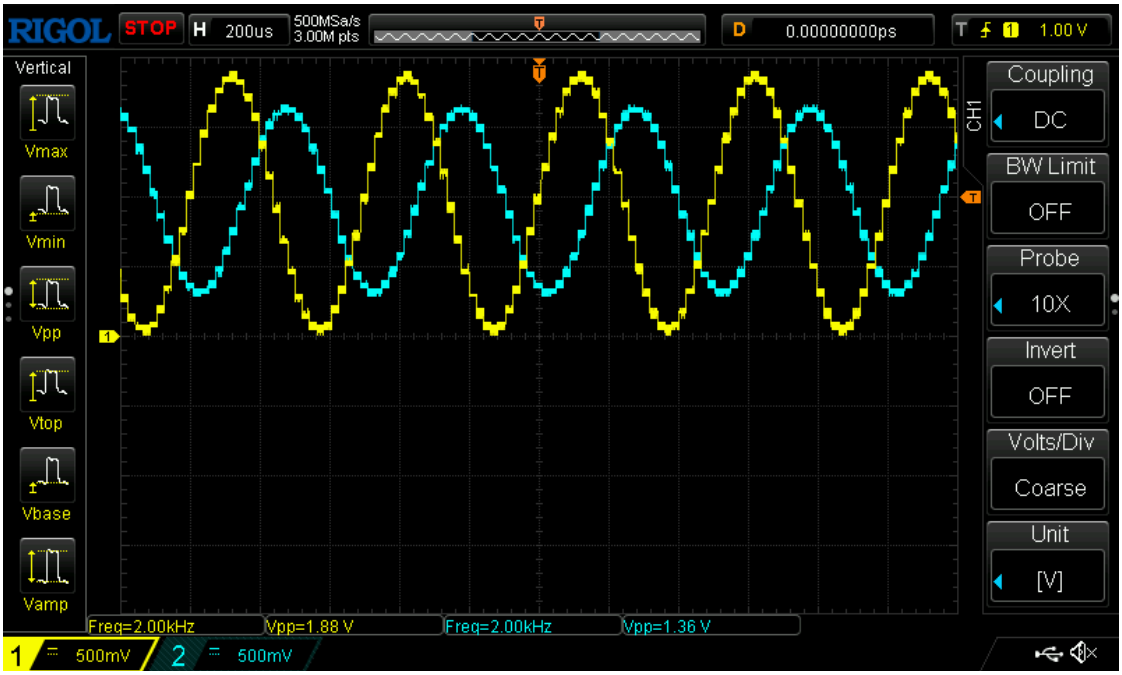

Figure 12: Input (yellow) and output (blue) @ 2kHz (Naive approach)

At a test frequency of 500 Hz, the input signal is passed almost unattenuated through the filter. In contrast, when the input frequency is increased to 2 kHz, the amplitude is reduced by approximately a factor of 0.707, corresponding to the expected −3 dB attenuation. This demonstrates that the naive approach produces the intended frequency response. Additionally, at the higher frequency of 2 kHz, the phase of the output signal is visibly shifted compared to the input; this behavior also matches expectations when considering the phase response shown in Figure 10.

2.4. Circular Buffer Approach

In this implementation, the FIR-filter was realized using a circular buffer to avoid the inefficient array shifting required in the naive approach. Each new input sample is written to the buffer at the current write index, and the index is incremented with wrap-around after each operation. This technique keeps the buffer elements in chronological order without physically moving the stored data. The filter multiplies the coefficients with the stored samples by iterating through the buffer in the correct sequence, ensuring the newest sample is combined with the first coefficient.

A key advantage of this approach is that the filter order was deliberately chosen as a power of two (in this case, 16). This allows the index wrap-around to be implemented efficiently using bit masking (& (N - 1)), rather than the modulo operation. When the filter order is not a power of two, the compiler typically generates a division instruction for the modulo, which incurs significantly higher execution time on the microcontroller. By contrast, the bitmasking approach requires only a single fast logical operation, making the circular buffer implementation highly efficient even for higher filter orders.

The behavior of this implementation was validated using the same test setup and the same filter configuration with a cutoff frequency of 2 kHz. The measurements clearly demonstrate that the functional performance is identical to the naive approach. For a

$500\,Hz$ input signal (see figure below), the output amplitude remains almost unchanged, showing that frequencies well below the cutoff are passed through without significant attenuation.

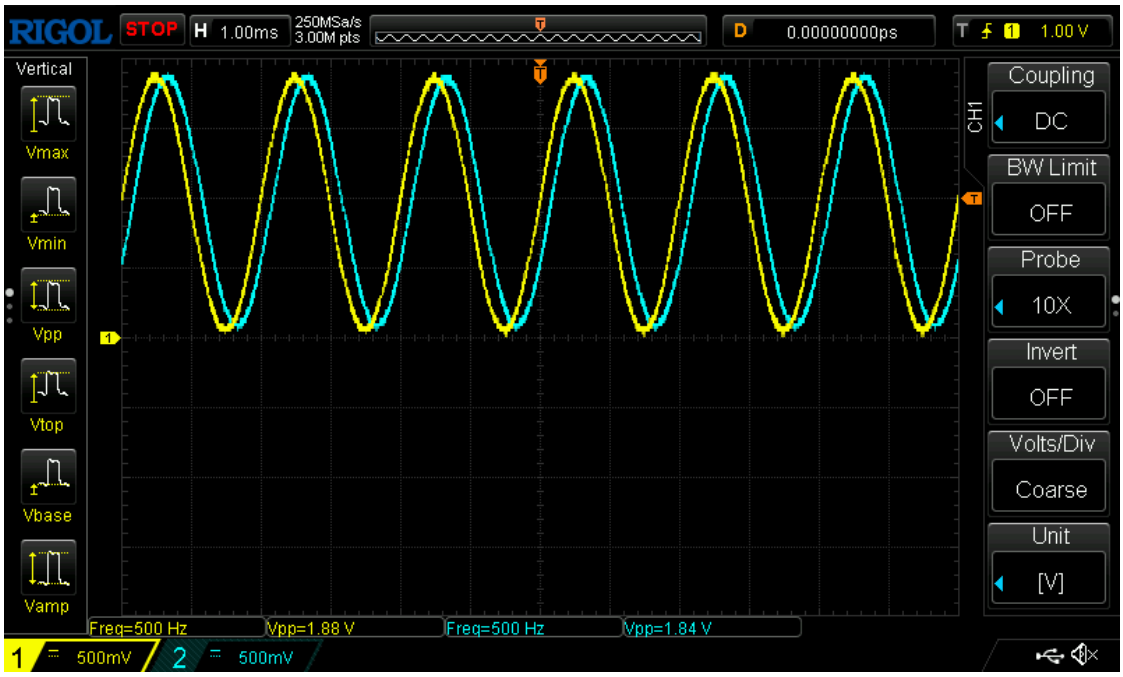

Figure 13: Input (yellow) and output (blue) @ 500Hz (Circular buffer)

For the $2\,kHz$ input signal (see Figure 14), the amplitude is attenuated by approximately 70.7%, which corresponds to the expected −3 dB cutoff point. This confirms that the circular buffer approach provides the same filtering characteristics while significantly reducing CPU load and memory operations.

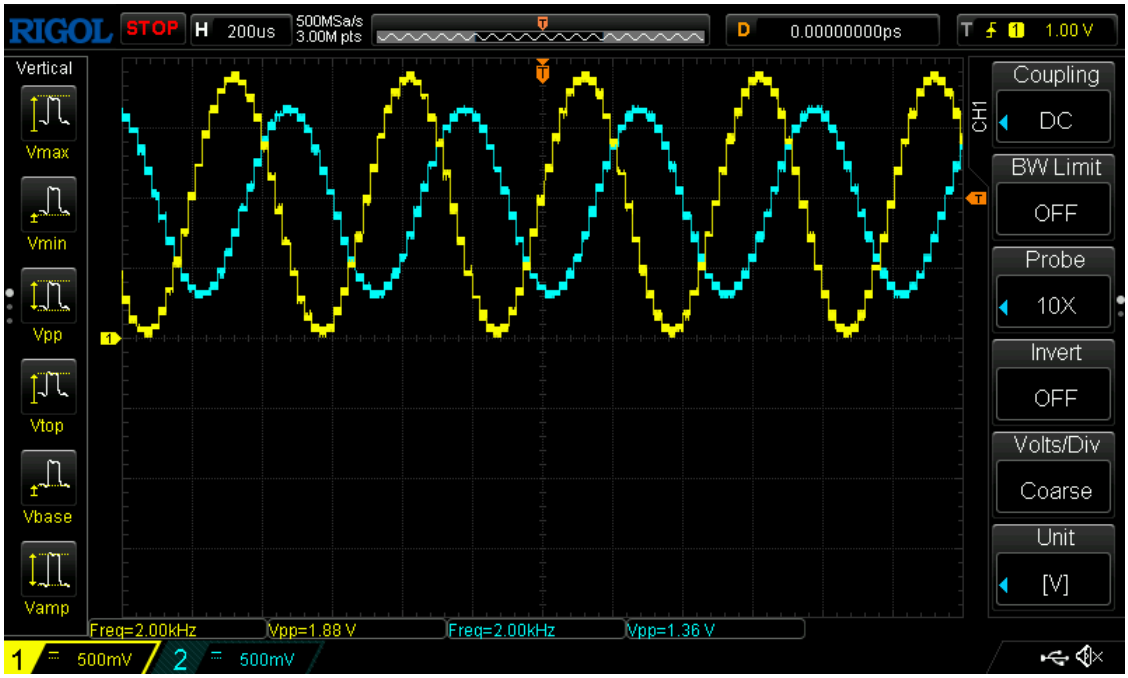

Figure 14: Input (yellow) and output (blue) @ 2kHz (Circular buffer)

2.5. Performance Analysis

To compare the computational efficiency of both implementations, the execution time of a single filtering operation was measured under different compiler optimization levels. Table 1 summarizes the measured runtimes for the naive array shifting approach and the circular buffer approach.

Table 1: Measured execution time per filtering operation under different compiler optimization levels

Optimization	Naive approach	Circular buffer approach
-O0	7.36 μs	5.08 μs
-O1	1.74 μs	1.7 μs
-O2	4.12 μs	1.44 μs
-O3	3.76 μs	1.66 μs
-Os (size)	4.12 μs	1.44 μs
-Ofast (speed)	3.76 μs	1.66 μs
-Oz (more size)	4.12 μs	1.44 μs

It is clearly visible that the circular buffer implementation consistently achieves significantly shorter execution times across all optimization settings. Especially at -O0 (no optimization), the difference is very pronounced, as the naive approach must perform explicit memory shifting of all buffer elements, while the circular buffer only updates a single index. At higher optimization levels such as -O2 and -O3, both implementations benefit from compiler optimizations. However, compared to -O1, these higher levels do not lead to further improvements in runtime performance in this case; on the contrary, the measured execution time even slightly increases, likely due to the compiler applying more aggressive code transformations that are not beneficial for this specific workload.

Another observation is that optimization settings focusing on speed (e.g., -Ofast) or size (-Os) have a measurable impact on performance. Overall, the circular buffer approach provides the most efficient solution, especially for high filter orders, and should be preferred in resource-constrained embedded systems.

2.6. Discussion

The comparison between the naive array shifting implementation and the circular buffer approach demonstrates that, while both methods produce equivalent filtering results, they differ considerably in computational efficiency and suitability for embedded systems. The naive approach is easy to implement and straightforward to understand, which makes it useful for validation or educational purposes. However, its linear time complexity and high memory operation overhead become prohibitive as the filter order increases.

In contrast, the circular buffer approach leverages efficient index handling through bitmasking when the buffer size is a power of two. This design eliminates the need for explicit memory shifting and significantly reduces execution time, as confirmed by the

performance analysis across different compiler optimization levels. Even at the lowest optimization level (-O0), the circular buffer implementation requires less processing time, and the relative advantage becomes even more apparent for higher filter orders or in real-time processing scenarios.

Overall, the results clearly indicate that for practical embedded applications where execution speed and determinism are critical, the circular buffer approach should be preferred. Selecting a power-of-two filter length further optimizes performance by allowing fast wrap-around operations without costly modulo divisions. These considerations highlight how algorithmic design decisions directly impact the efficiency and feasibility of digital signal processing on microcontrollers.