University of Seoul

Seoul Campus

School of Electrical and Computer Engineering

Lab 3: Filter Implementation and Design

Subject:

Embedded Systems Applications Design

Student:

Roberto Raymundo Gómez Vargas
2025020338

Professor:

Ye Gu Kang

Seoul, South Korea, November 2025

Abstract

This project presents the design, implementation, and experimental validation of five fundamental digital filters: Low-Pass, High-Pass, Band-Pass, Band-Stop, and All-Pass. Each filter was modeled in three different representations: the continuous-time S-domain, the discrete-time Z-domain, and a real-time C implementation executed through a C Caller block in Simulink. The continuous and discrete transfer functions were analyzed to understand frequency behavior, stability, and dynamic response, while the C code version enabled deployment on an STM32 microcontroller. All implementations were evaluated in Simulink using generated test signals and subsequently tested on hardware, where the filter outputs were observed using an oscilloscope. Results from the three domains were compared to assess accuracy, discretization effects, and real-time performance. The project demonstrates the complete workflow from theoretical filter design to embedded implementation and experimental verification.

Key Words: Digital Filters, S-Domain, Z-Domain, Embedded Systems, STM32, Simulink, C Caller, Oscilloscope Measurements.

Contents

List of Figures

Chapter 1

Project Overview

1.1 Introduction

Digital filters are essential components in modern signal processing, enabling engineers to manipulate, condition, and analyze signals across a wide variety of applications. Whether filtering noise from sensor measurements, shaping the frequency content of audio signals, or preparing data for control algorithms, filters serve as the foundation for many embedded and real-time systems. This project focuses on understanding, designing, simulating, and implementing five classical Infinite Impulse Response (IIR) filters: low-pass, high-pass, all-pass, band-pass, and band-stop.

The work begins with a theoretical study of each filter and then proceeds to their implementation in multiple domains. Each filter is developed in the analog s-domain, converted into a digital z-domain equivalent using the Tustin (bilinear) transform, and finally implemented using C code inside Simulink via the C Caller block. By validating all versions against each other, the project ensures the mathematical consistency of the transformations and prepares the models for real-world deployment.

The complete source code for this project can be access at:

https://github.com/raymundo140/Simulink-to-STM32-Embedded-DSP-Filters

1.2 Project Objective

The main objective of this project is to design and validate five IIR filters in simulation and later implement them in real hardware. The project is divided into two main stages:

1. **Simulation Stage**
    - Develop the analog transfer functions $H(s)$ for all five filters.
    - Convert each filter into its discrete $H(z)$ representation using the bilinear transform with pre-warping.

- Implement each filter's difference equation using custom C code.
- Integrate the C code in Simulink using the C Caller block.
- Test all three implementations (S-domain, Z-domain, C implementation) using sinusoids, chirps, noise, and mixed-frequency inputs.
- Verify that all three outputs behave consistently.

2. **Hardware Implementation Stage** After the simulation stage is validated, the filters will be implemented on an embedded microcontroller platform:
   - Use the STM32 NUCLEO-G474RE microcontroller board to deploy the C implementation.
   - Generate real analog test signals using a bench signal generator.
   - Use a digital oscilloscope to observe, measure, and compare the filtered outputs.
   - Validate the real-time behavior of each filter using physical hardware.

Through this process, the project bridges the gap between theoretical filter design, numerical simulation, and real-time embedded deployment.

1.3 Project Description

The first phase of the project consists of creating accurate filter models in Simulink. Each filter is implemented three times in parallel: an analog version using a Transfer Function block in the s-domain, a digital version using a discrete Transfer Function block in the z-domain, and a custom digital implementation written in C and executed through the C Caller block. All three versions are fed by the same input signals, allowing direct comparison of their time-domain responses.

The input test signals include:

- fixed-frequency sine waves at various frequencies,
- chirp signals sweeping across specific ranges,
- white noise,
- and mixed-frequency signals to evaluate selectivity and attenuation.

The second phase of the project consists of implementing the filters on the STM32 NUCLEO-G474RE board. The verified C code from Simulink will be ported to the STM32 environment using STM32CubeIDE. The microcontroller will process real-time ADC samples from a function generator acting as the input source. The filtered output signal will be observed on an oscilloscope to verify gain, cutoff behavior, frequency rejection, stability, and overall response.

By completing both stages, the project provides a full workflow, from theoretical modeling and digital simulation, all the way to real hardware implementation and experimental validation.

Chapter 2

Fundamentals of Digital Filters

This chapter presents the theoretical foundation behind the five filters implemented in the project. Each section explains the purpose, mathematical formulation, digital conversion process, and typical applications of the low-pass, high-pass, all-pass, band-pass, and band-stop filters. The explanations follow standard digital signal processing theory and align with the course material provided by the professor.

2.1 Overview of Filters

A digital filter is a system that selectively modifies the frequency content of an input signal. Depending on the design, a filter may amplify, attenuate, or phase-shift specific frequency components. Digital filters are categorized primarily into two groups:

- **Finite Impulse Response (FIR) filters:**
  - Contain only zeros in their transfer function.
  - Always stable and can achieve linear phase.
  - Often require many coefficients for sharp transitions.
- **Infinite Impulse Response (IIR) filters:**
  - Include both poles and zeros.
  - Provide sharper frequency responses with fewer coefficients.
  - Typically exhibit nonlinear phase.
  - Frequently obtained from analog prototypes using transformations such as the bilinear transform.

All filters used in this project belong to the IIR category, implemented as either first-order or second-order systems.

2.2 Low-Pass Filter (LPF)

2.2.1 Theory

A low-pass filter (LPF) allows low-frequency components to pass while attenuating higher frequencies. It is widely used in signal smoothing, noise reduction, and extracting slowly varying information. The analog first-order LPF is defined by:

$$H_{\text{LPF}}(s) = \frac{\omega_c}{s + \omega_c}, \tag{2.1}$$

where $\omega_c = 2\pi f_c$ is the cutoff frequency. At f_c, the magnitude response drops by -3 dB.

2.2.2 Digital Implementation

Applying the bilinear transform,

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}, \tag{2.2}$$

produces the digital transfer function

$$H_{\text{LPF}}(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}. \tag{2.3}$$

Low-pass filters are essential in sensor conditioning, audio processing, and control systems where high-frequency noise must be suppressed.

2.3 High-Pass Filter (HPF)

2.3.1 Theory

A high-pass filter (HPF) attenuates low-frequency components while allowing higher-frequency components to pass. It is commonly used to remove slow drifts, DC offsets, or low-frequency disturbances. The analog first-order HPF is:

$$H_{\text{HPF}}(s) = \frac{s}{s + \omega_c}. \tag{2.4}$$

2.3.2 Digital Implementation

The bilinear transform yields a discrete transfer function of the form:

$$H_{\text{HPF}}(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}. \tag{2.5}$$

High-pass filters are widely used in biomedical signal processing, vibration monitoring, AC coupling, and communication signal conditioning.

2.4 All-Pass Filter (APF)

2.4.1 Theory

An all-pass filter maintains a constant magnitude across all frequencies but alters the phase response. The analog form is given by:

$$H_{\text{APF}}(s) = \frac{s - \omega_0}{s + \omega_0}. \tag{2.6}$$

The magnitude satisfies:

$$|H_{\text{APF}}(j\omega)| = 1, \tag{2.7}$$

while the phase shift varies strongly around the characteristic frequency ω_0.

2.4.2 Digital Implementation

The digital version maintains the same structure as a first-order IIR filter:

$$H_{\text{APF}}(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}. \tag{2.8}$$

All-pass filters are widely used in audio effects (phasers), delay compensation, phase equalization, and control systems.

2.5 Band-Pass Filter (BPF)

2.5.1 Theory

A band-pass filter (BPF) passes frequencies within a desired range while attenuating those outside the band. A second-order analog BPF is defined as:

$$H_{\text{BPF}}(s) = \frac{\frac{\omega_0}{Q} s}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2}, \tag{2.9}$$

where ω_0 is the center frequency and Q controls the bandwidth.

2.5.2 Digital Implementation

Using the bilinear transform results in the discrete transfer function:

$$H_{\text{BPF}}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{2.10}$$

Band-pass filters are essential in audio equalization, radio communication channel selection, and vibration analysis.

2.6 Band-Stop Filter (BSF)

2.6.1 Theory

A band-stop filter (BSF), also known as a notch filter, removes a narrow band of frequencies around a center frequency while allowing all other frequencies to pass. Its analog representation is:

$$H_{\text{BSF}}(s) = \frac{s^2 + \omega_0^2}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2}. \tag{2.11}$$

2.6.2 Digital Implementation

The bilinear transform yields:

$$H_{\text{BSF}}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{2.12}$$

Band-stop filters are commonly used to remove electrical hum (50/60 Hz), eliminate narrowband interference, and suppress mechanical resonances in feedback systems.

2.7 Illustrative representation

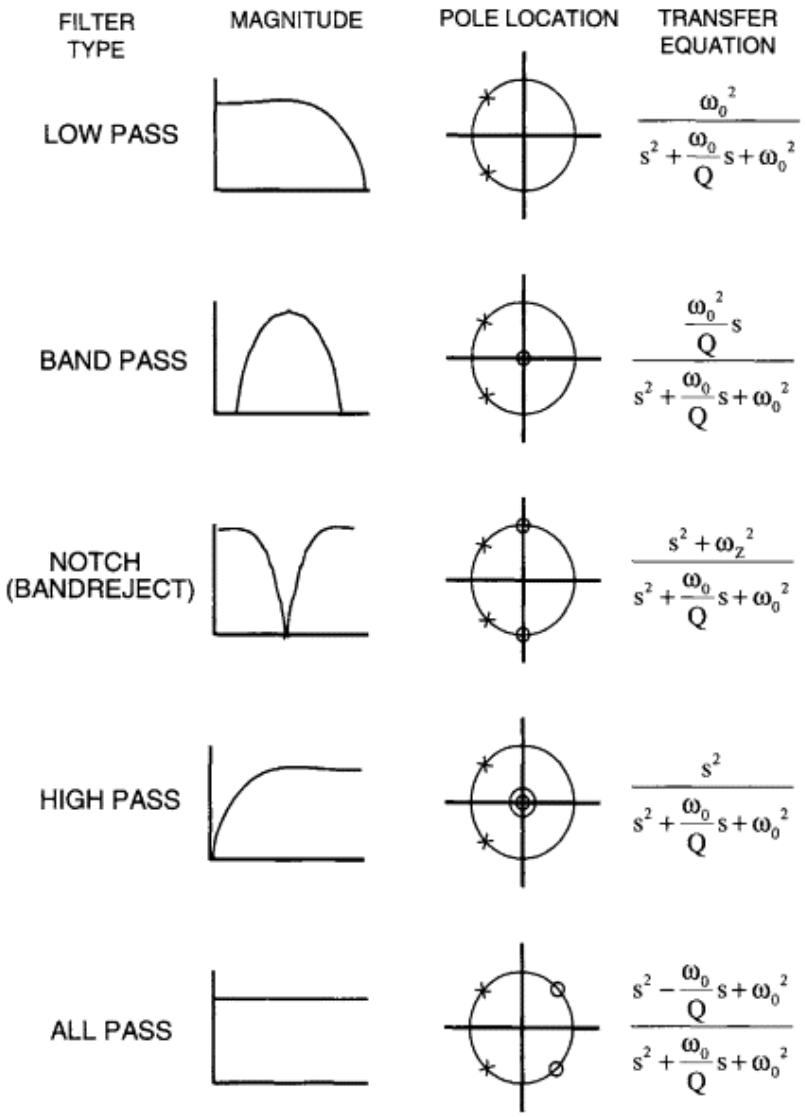

Figure 2.1: Summary of classical second order analog filter types and their corresponding transfer functions [1].

Figure 2.1 presents an illustrative comparison of the five classical **second–order** filters: low–pass (LPF), high–pass (HPF), band–pass (BPF), band–stop or notch (BSF), and all–pass (APF). Although the filters have distinct frequency responses and applications, they all share a common biquadratic structure and differ only in the configuration of their zeros and the form of their numerators. In this project, the filters are implemented as **first–order** systems, but the second–order diagram is still useful because it shows the general relationships between poles, zeros and magnitude response that also apply to higher orders.

Magnitude Response

The left column of Figure 2.1 shows the qualitative magnitude response of each filter:

- **Low–Pass (LPF):** Attenuates high–frequency components, allowing only low frequencies to pass.
- **Band–Pass (BPF):** Passes a narrow frequency band centered around ω_0 and rejects frequencies outside this region.
- **Band–Stop / Notch (BSF):** Strongly rejects a narrow frequency band around ω_0 while passing the rest of the spectrum.
- **High–Pass (HPF):** Attenuates low–frequency components and allows higher frequencies to pass.
- **All–Pass (APF):** Maintains a flat magnitude response (unity gain) but introduces a frequency–dependent phase shift.

Pole–Zero Location

The central column depicts the poles and zeros on the complex s-plane:

- All filters share a pair of complex–conjugate poles located at radius ω_0 with quality factor Q, which determine the resonance and selectivity of the system.
- The primary difference lies in the placement of zeros:
  - LPF: zeros at infinity.
  - HPF: a double zero at the origin.
  - BPF: zeros at the origin and at infinity.
  - BSF: zeros exactly at $\pm j\omega_0$, producing a spectral notch.
  - APF: zeros mirror the poles across the imaginary axis, yielding phase shifting without magnitude change.

Transfer–Function Forms

The right column lists the standard second–order transfer functions associated with each filter. All share the common denominator:

$$s^2 + \frac{\omega_0}{Q}s + \omega_0^2,$$

while the numerators vary according to the desired behavior:

$$\text{LPF:} \quad \frac{\omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

$$\text{BPF:} \quad \frac{\omega_0 s/Q}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

$$\text{BSF:} \quad \frac{s^2 + \omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

$$\text{HPF:} \quad \frac{s^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

$$\text{APF:} \quad \frac{s^2 - \frac{\omega_0}{Q}s + \omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

Effect of Filter Order

To understand why the order of a filter matters, Figure 2.2 shows the magnitude response of low–pass filters of different orders, all with the same cutoff frequency. The first–order response decays relatively slowly, while second, third, and fourth order filters exhibit increasingly sharper transitions between the passband and the stopband.

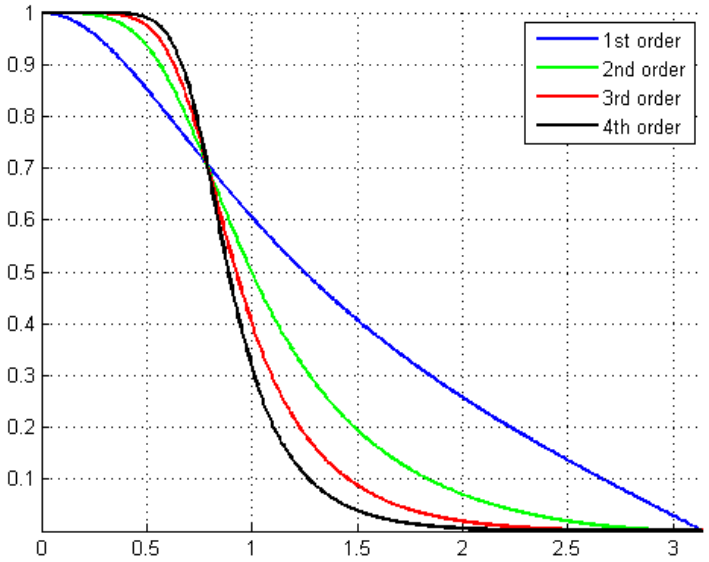

Figure 2.2: Comparison of low–pass filters of different orders. Higher order produces a steeper roll–off around the cutoff frequency. [2].

For an ideal low–pass filter, the transition from passband to stopband would be perfectly vertical (a "brick–wall" response). In practice, each pole contributes approximately -20dB/decade of attenuation beyond the cutoff frequency. Therefore:

- A first–order filter provides about -20dB/decade roll–off.
- A second–order filter provides about -40dB/decade.

- A third–order filter reaches about -60dB/decade, and so on.

From this point of view, a second–order design is "better" than a first–order one because it can separate frequencies more aggressively and approximate the ideal response more closely. A third or fourth order filter would be even sharper, which is why higher–order designs are used in audio equalizers, communication channel filters, and anti–aliasing filters when strict specifications are required.

However, there is a trade off:

- Each additional order introduces more poles and zeros, which means more states, more coefficients, and more arithmetic operations.
- Higher order filters are more sensitive to coefficient quantization and numerical errors, which can lead to degraded performance or even instability in fixed–point implementations.
- On a microcontroller such as the STM32 NUCLEO–G474RE, more complex filters consume additional CPU time and memory, which may not be acceptable in real–time applications with tight timing constraints.

For these reasons, practical designs often strike a balance between performance and complexity. In this project, first order filters are implemented to focus on the complete workflow, from continuous time modeling to digital implementation and finally embedded realization, while the second–order diagram is kept as a conceptual reference to illustrate how increasing the filter order sharpens the frequency response.

Chapter 3

Simulink Design and Implementation

3.1 Global Model Architecture

The first stage of the project consists of building a modular Simulink model that can be reused for all five filters. Figure 3.1 shows the top–level diagram: on the left there is an `Input` subsystem that generates the test signals, and on the right there are five subsystems corresponding to the low–pass, high–pass, all–pass, band–pass, and band–stop filters (`lpf`, `hpf`, `apf`, `bpf`, `bsf`). Each filter subsystem contains three parallel implementations of the same filter: an s–domain transfer function, a z–domain transfer function, and a C implementation using the `C Caller` block.

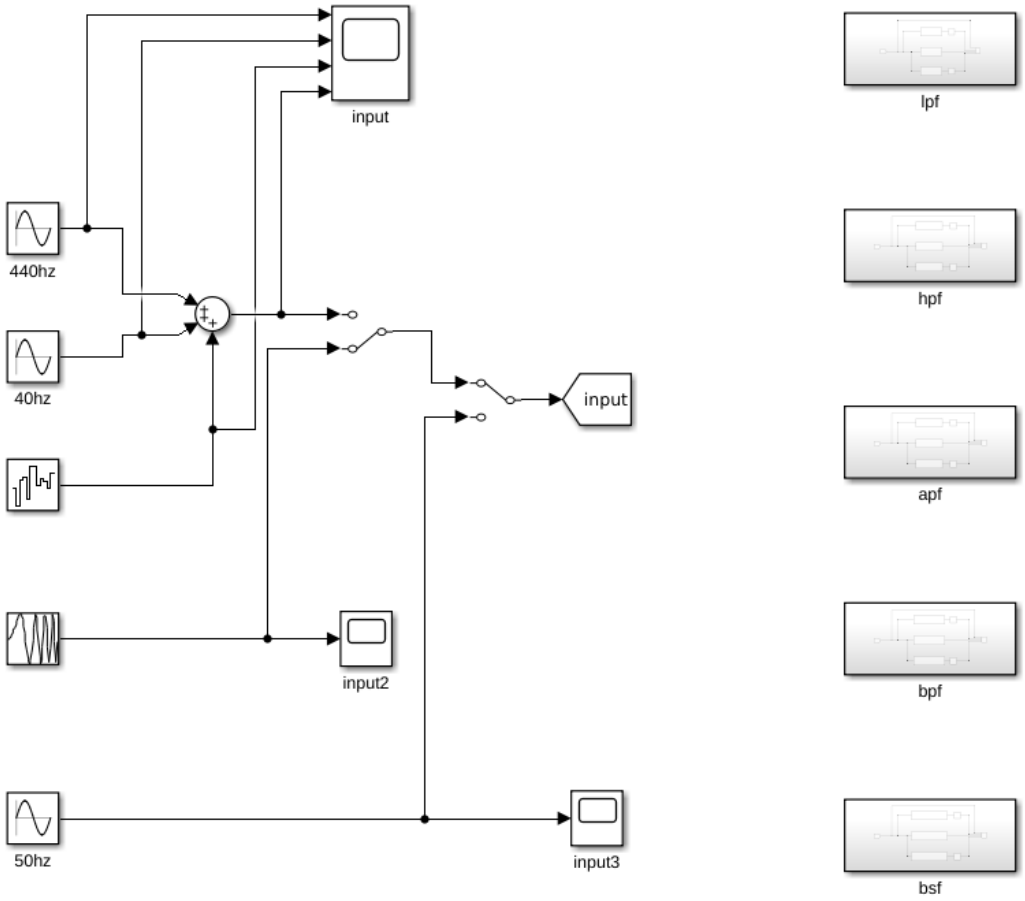

Figure 3.1: Top–level Simulink model: input subsystem (left) and filter subsystems (right).

This organization makes it possible to feed the same signal to all filters and compare, for each one, the behavior of the three domains. It also mirrors the structure of the later embedded implementation: the same C difference equation that runs inside each Simulink subsystem will later be executed on the STM32 NUCLEO–G474RE board.

3.1.1 Global Parameters in MATLAB

To keep the model consistent and easy to modify, the main numerical parameters are defined in a separate MATLAB script and used in the Simulink blocks through workspace variables. The following code snippet shows the base configuration:

```
%% Global parameters
fs = 10000;          % sampling frequency [Hz]
Ts = 1/fs;           % sampling period [s]

fc = 100;            % LPF cutoff [Hz]
wc = 2*pi*fc;        % LPF cutoff [rad/s]
```

The parameters have the following roles:

- `fs`: global sampling frequency of the digital system (10kHz). All discrete blocks in the model, including the `C Caller` filters, operate at this rate.
- `Ts`: sampling period, used as the sample time for discrete blocks and for generating time vectors when needed.
- `fc`: nominal cutoff frequency of the first–order low–pass and high–pass filters (100Hz). This value is also reused as the central frequency for the band–pass and band–stop filters.
- `wc`: angular cutoff frequency $\omega_c = 2\pi f_c$, used directly in the continuous–time transfer–function blocks in the s–domain.

Defining these parameters in MATLAB has two advantages. First, it guarantees that the s–domain, z–domain, and C implementations always share exactly the same numerical values. Second, it allows quick experiments (for example, changing the cutoff frequency or sampling rate) without having to edit multiple blocks manually in Simulink.

3.1.2 Input Subsystem

The `Input` subsystem is responsible for generating the test signals that will be processed by each filter. Its structure is shown in Figure 3.3. Several signal sources are included:

- A 40Hz sine wave.
- A 440Hz sine wave.
- A white–noise source.
- A chirp signal whose frequency sweeps linearly from 50Hz up to 300Hz.
- A separate 50Hz sine wave used as an additional reference.

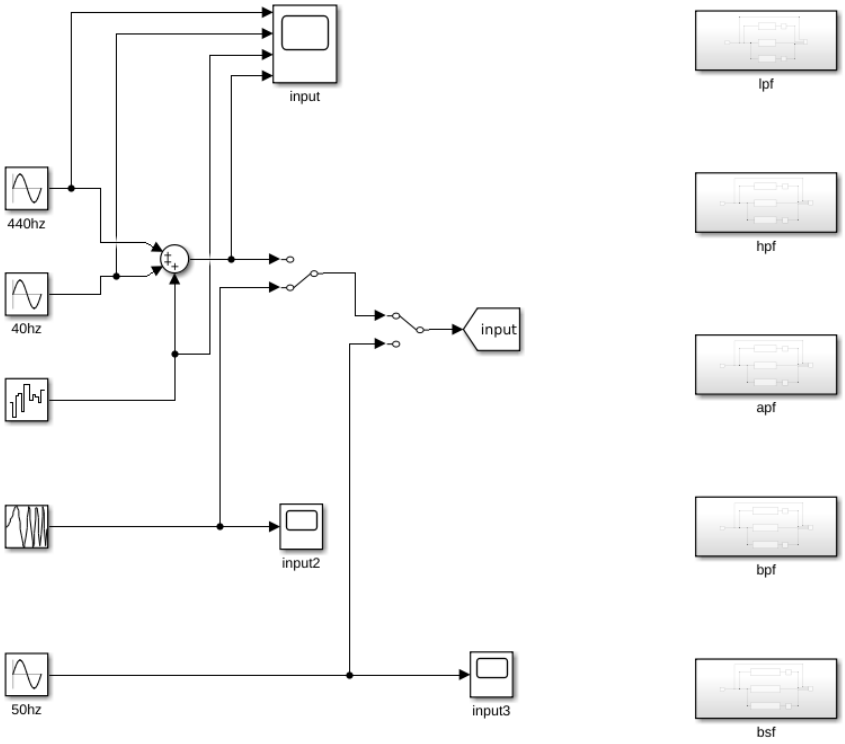

Figure 3.2: Input subsystem used to generate test signals for all filters.

The two sine waves and the white–noise source are summed using an `Add` block, forming a composite signal that contains both low and high frequencies plus stochastic components. Manual switches allow the user to route either this composite signal or the chirp to the main `input` port that feeds the filters. Additional outputs (`input2`, `input3`) are connected to scopes for monitoring the chirp and the standalone 50Hz sine wave.

This configuration makes it straightforward to evaluate how each filter behaves under different types of excitation: pure sinusoids, mixed–frequency signals, broadband noise, and frequency sweeps.

3.1.3 Filter Subsystem Organization

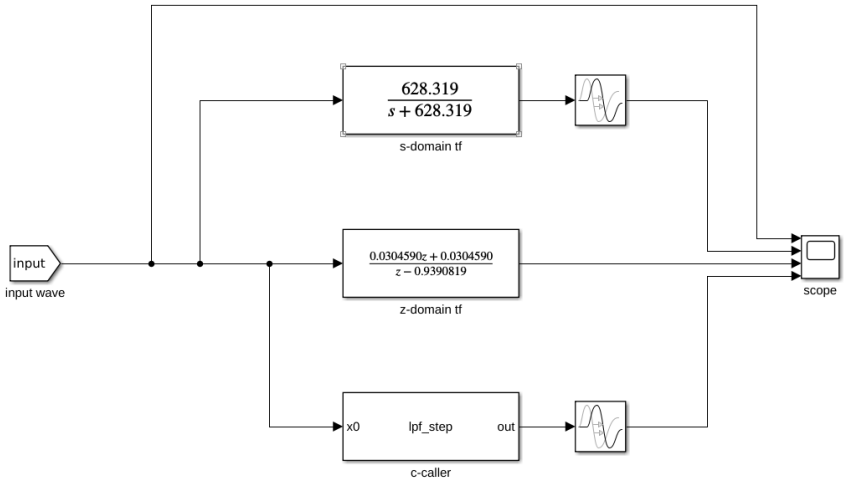

Figure 3.3: Subsystem organization

On the right side of the top–level model, each filter is encapsulated in its own subsystem (`lpf`, `hpf`, `apf`, `bpf`, `bsf`). All five subsystems share the same internal structure:

- An s–domain `Transfer Fcn` block implementing the analog transfer function $H(s)$ using the corresponding ω_c or (ω_0, Q) parameters.
- A discrete `Transfer Fcn` block implementing the z–domain transfer function $H(z)$ obtained with the bilinear transform and pre–warping.
- A `C Caller` block that executes the hand–written C function (for example, `lpf_step`, `hpf_step`, etc.), which implements the difference equation using internal state variables.

Each of the three outputs is routed to a common scope so that the analog model, the discrete transfer function, and the C implementation can be compared sample by sample. This layout directly reflects the main objective of the project: to ensure that the mathematical design, the discrete–time implementation, and the final C code intended for the STM32 microcontroller are all consistent and produce nearly identical responses under the same excitation.

3.2 Low-pass filter (LPF)

The first filter implemented in the project is a first–order low–pass filter with cutoff frequency $f_c = 100Hz$ and sampling frequency $f_s = 10kHz$. This filter is realized in three equivalent forms: an analog transfer function in the s–domain, a discrete transfer function in the z–domain obtained by the bilinear transform, and a C implementation based on the corresponding difference equation.

3.2.1 Analog s-domain transfer function

For a first–order RC low–pass filter, the standard transfer function is

$$H(s) = \frac{\omega_c}{s + \omega_c}, \tag{3.1}$$

where $\omega_c = 2\pi f_c$ is the cutoff angular frequency in rad/s. Using $f_c = 100Hz$, the cutoff becomes

$$\omega_c = 2\pi \cdot 100 \approx 628.319\ rad/s.$$

Substituting this value into (3.1) gives

$$H(s) = \frac{628.319}{s + 628.319}. \tag{3.2}$$

In Simulink this expression is implemented with a `Transfer Fcn` block, where the numerator and denominator vectors are

$$\mathrm{Num}_s = [\,628.319\,], \qquad \mathrm{Den}_s = [\,1 \quad 628.319\,].$$

This block forms the upper path of the LPF subsystem (labelled “s-domain tf” in Figure 3.4).

3.2.2 Discrete z-domain transfer function

To obtain a digital IIR implementation that approximates the analog response, the bilinear (Tustin) transform is applied to (3.1). The bilinear transform maps the s-plane into the z-plane through

$$s = \frac{2}{T_s}\,\frac{1 - z^{-1}}{1 + z^{-1}}, \tag{3.3}$$

where $T_s = 1/f_s$ is the sampling period. With $f_s = 10kHz$, one has $T_s = 0.0001\ s$.

Substituting (3.3) into (3.1) and simplifying gives

$$\begin{aligned} H(z) &= \frac{\omega_c}{\dfrac{2}{T_s}\dfrac{1-z^{-1}}{1+z^{-1}} + \omega_c} \\ &= \frac{\omega_c(1+z^{-1})}{\dfrac{2}{T_s}(1-z^{-1}) + \omega_c(1+z^{-1})}. \end{aligned} \tag{3.4}$$

Defining $A = \dfrac{2}{T_s}$, the denominator becomes

$$A(1-z^{-1}) + \omega_c(1+z^{-1}) = (A+\omega_c) + (-A+\omega_c)\,z^{-1}.$$

Normalizing by $(A+\omega_c)$ leads to the standard IIR form

$$H(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}, \tag{3.5}$$

with coefficients

$$b_0 = \frac{\omega_c}{A+\omega_c}, \qquad b_1 = \frac{\omega_c}{A+\omega_c}, \tag{3.6}$$

$$a_1 = \frac{-A+\omega_c}{A+\omega_c}. \tag{3.7}$$

Using $T_s = 0.0001$ and $\omega_c \approx 628.319$ yields

$$b_0 = b_1 \approx 0.0304590, \tag{3.8}$$

$$a_1 \approx -0.9390819. \tag{3.9}$$

Therefore, the discrete transfer function implemented in Simulink is

$$H(z) = \frac{0.0304590 + 0.0304590\,z^{-1}}{1 - 0.9390819\,z^{-1}}. \tag{3.10}$$

The corresponding `Transfer Fcn` block uses

$$\text{Num}_z = [\,0.0304590 \quad 0.0304590\,], \qquad \text{Den}_z = [\,1 \quad -0.9390819\,].$$

3.2.3 Simulink LPF subsystem

Figure 3.4 shows the Simulink subsystem for the low–pass filter. The incoming signal `input` is split into three parallel paths:

- the continuous–time s–domain transfer function (3.2),
- the discrete–time z–domain transfer function (3.10),

- and the C implementation called through the `C Caller` block.

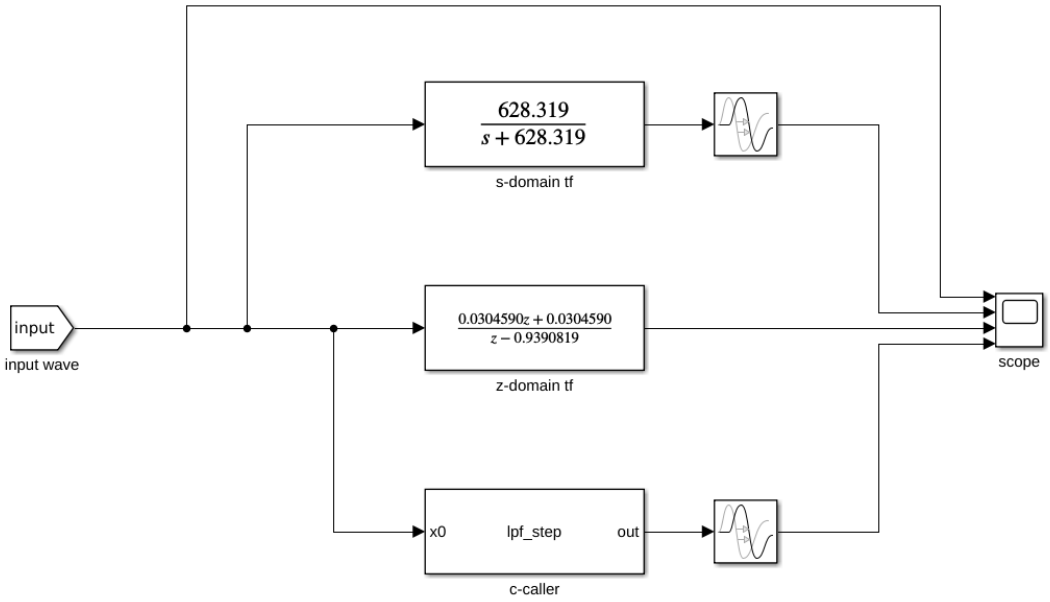

Figure 3.4: LPF subsystem with s–domain, z–domain and C implementation in parallel.

To make the three waveforms easier to compare in the scope, small `Transport Delay` blocks are inserted in the upper (s–domain) and lower (C implementation) paths. These delays shift the signals slightly in time on the horizontal axis, avoiding perfect overlap while preserving amplitude and phase relationships. They are only for visualization purposes and do not modify the filter design itself.

3.2.4 C implementation: header and source files

The C implementation of the LPF is based directly on the difference equation associated with (3.10):

$$y[n] = b_0 x[n] + b_1 x[n-1] - a_1 y[n-1], \tag{3.11}$$

where $x[n]$ is the input, $y[n]$ the output, and $x[n-1]$, $y[n-1]$ are the internal state variables.

In code, this implementation is split into a header file `lpf_c_caller.h` and a source file `lpf_c_caller.c`. The header declares the public interface:

LPF Header File

```
/* lpf_c_caller.h
 * Header file for 1st-order IIR LPF
 */
#ifndef __LPF_FILTER_H__
#define __LPF_FILTER_H__

void   lpf_reset(void);
double lpf_step(double x0);

#endif /* __LPF_FILTER_H__ */
```

Two functions are exposed:

- `lpf_reset()` initializes the internal state variables (previous input and output) to zero.
- `lpf_step(double x0)` processes one new sample $x_0 = x[n]$ and returns the corresponding output $y[n]$.

The source file contains the actual difference–equation implementation:

LPF Header File

```
/* lpf_c_caller.c
 * 1st-order IIR Low-Pass Filter
 * fs = 10 kHz, fc  100 Hz
 */

#include <stddef.h>
#include "lpf_c_caller.h"

/* Coefficients for H(z) = (b0 + b1 z^-1) / (1 + a1 z^-1) */
static const double b0 = 0.0304590;
static const double b1 = 0.0304590;
static const double a1 = -0.9390819;  /* note the sign */
```

```
/* Filter state: x[n-1], y[n-1] */
static double x1 = 0.0;
static double y1 = 0.0;

/* Reset filter state */
void lpf_reset(void)
{
    x1 = 0.0;
    y1 = 0.0;
}

/* One sample step */
double lpf_step(double x0)
{
    /* Difference equation:
       y[n] = b0 * x[n] + b1 * x[n-1] - a1 * y[n-1]
    */
    double y0 = b0 * x0 + b1 * x1 - a1 * y1;

    /* Update states */
    x1 = x0;
    y1 = y0;

    return y0;
}
```

The filter coefficients b_0, b_1 and a_1 are stored as `static const` variables so they are allocated once and cannot be modified at run time. The previous input and output (`x1` and `y1`) are stored as `static` variables as well, which allows `lpf_step()` to maintain internal memory between successive calls, mimicking the behavior of the IIR filter.

In Simulink, the `C Caller` block is configured to call `lpf_step()` with one scalar input port (the current sample $x[n]$) and one scalar output port (the result $y[n]$). During simulation, this C code is compiled and linked automatically, so the output of the `C Caller` block can be compared directly with the outputs of the s–domain and z–domain transfer–function blocks. The close agreement between the three traces confirms that the mathematical design, discrete filter coefficients and C implementation are consistent.

3.2.5 Simulation Results

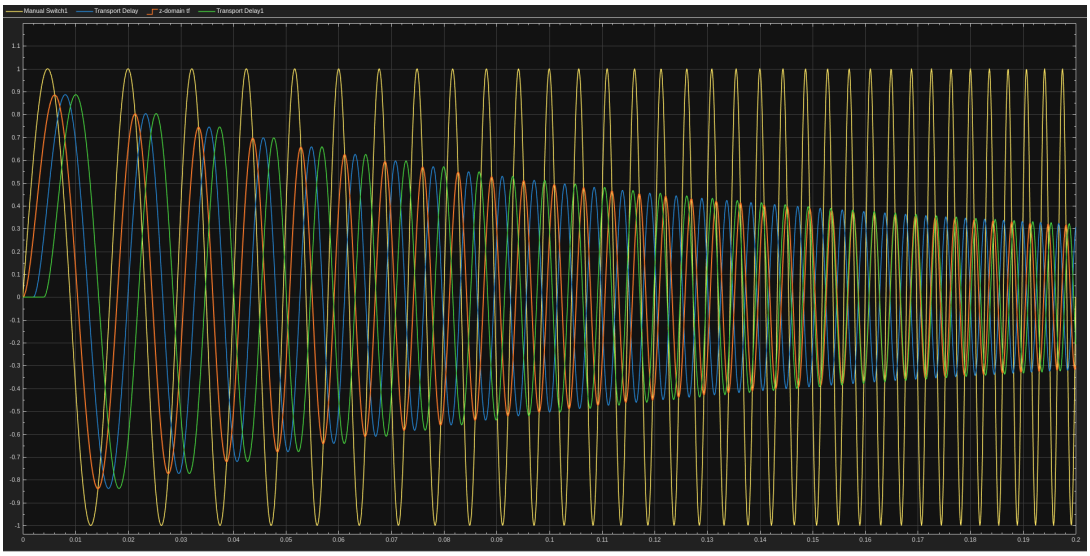

Figure 3.5: LPF Results

Figure 3.5 shows the comparative time–domain responses of the three LPF implementations: the continuous–time s-domain [blue] transfer function, the discrete–time z-domain [orange] transfer function obtained via the bilinear transform, and the custom C implementation [green] executed through the `C Caller` block. A frequency–swept input signal ranging from 50 Hz to 300 Hz was used to clearly highlight the filter's frequency–selective behavior.

At the beginning of the simulation, where the input frequency is low, all three implementations produce outputs with large amplitude, as expected for a low–pass filter operating well below its cutoff frequency of $f_c = 100$ Hz. As the input frequency increases, the outputs begin to attenuate progressively. This attenuation becomes more pronounced after the sweep surpasses the cutoff region, with the output converging toward zero amplitude for higher frequencies.

The slight horizontal offsets visible between the curves correspond to small transport delays deliberately introduced in the Simulink diagram. These delays were added only for visualization purposes, ensuring that the three traces do not overlap completely on the scope and allowing clearer comparison of the dynamic responses. Despite this shift, all three implementations show nearly identical amplitude envelopes and decay characteristics. This confirms that:

1. the analog prototype behavior is accurately preserved in the digital equivalent,
2. the bilinear–transformed discrete filter matches the expected mathematical response, and
3. the manually coded C filter behaves exactly like its theoretical z-domain counterpart.

Overall, the consistency among the three responses validates both the design process and the correctness of the custom filter implementation. The LPF meets its intended objective by allowing low–frequency components to pass while progressively attenuating frequencies above the cutoff.

3.3 High-pass filter (HPF)

The second filter implemented in this project is a first–order high–pass filter (HPF), also designed with a cutoff frequency of $f_c = 100Hz$ and sampling frequency $f_s = 10kHz$. As with the LPF, three parallel realizations are compared: the analog s–domain transfer function, the discrete z–domain transfer function derived through the bilinear transform, and the custom C implementation called through the `C Caller` block in Simulink.

3.3.1 Analog s-domain transfer function

The standard first–order analog high–pass filter has the transfer function

$$H(s) = \frac{s}{s + \omega_c}, \tag{3.12}$$

where $\omega_c = 2\pi f_c$ is the cutoff angular frequency. For $f_c = 100Hz$,

$$\omega_c = 2\pi \cdot 100 \approx 628.319 \; rad/s.$$

Substituting this into (3.12), the numerical analog transfer function becomes

$$H(s) = \frac{s}{s + 628.319}. \tag{3.13}$$

In Simulink, this is implemented using the `Transfer Fcn` block with

$$\text{Num}_s = [\,1 \quad 0\,], \qquad \text{Den}_s = [\,1 \quad 628.319\,].$$

3.3.2 Discrete z-domain transfer function

To obtain the corresponding digital high–pass filter, the bilinear transform

$$s = \frac{2}{T_s}\,\frac{1 - z^{-1}}{1 + z^{-1}}$$

is substituted into (3.12). Using $T_s = 0.0001$ s and defining again $A = \frac{2}{T_s}$, the transformed numerator becomes

$$s(1 + z^{-1}) = A(1 - z^{-1}),$$

while the denominator becomes

$$A(1 - z^{-1}) + \omega_c(1 + z^{-1}) = (A + \omega_c) + (-A + \omega_c)z^{-1}.$$

Normalizing by $(A + \omega_c)$ yields the standard IIR form

$$H(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}, \tag{3.14}$$

with coefficients

$$b_0 = \frac{A}{A+\omega_c}, \qquad b_1 = -\frac{A}{A+\omega_c}, \tag{3.15}$$

$$a_1 = \frac{-A+\omega_c}{A+\omega_c}. \tag{3.16}$$

Numerically,

$$b_0 \approx 0.96954097, \tag{3.17}$$

$$b_1 \approx -0.96954097, \tag{3.18}$$

$$a_1 \approx -0.93908194. \tag{3.19}$$

Thus, the discrete HPF transfer function becomes

$$H(z) = \frac{0.96954097 - 0.96954097z^{-1}}{1 - 0.93908194z^{-1}}. \tag{3.20}$$

In Simulink, the corresponding `Transfer Fcn` block uses

$$\text{Num}_z = [\,0.96954097 \quad -0.96954097\,], \qquad \text{Den}_z = [\,1 \quad -0.93908194\,].$$

3.3.3 Simulink HPF subsystem

Figure 3.6 shows the complete HPF subsystem. The incoming signal is branched into three paths:

- the continuous s-domain transfer function (3.13),
- the discrete z-domain transfer function (3.20),
- and the custom C implementation executed through a `C Caller` block.

As done previously with the LPF subsystem, small `Transport Delay` blocks were inserted in the upper and lower paths purely for visualization. These delays prevent the three outputs from perfectly overlapping, making it easier to visually compare the responses.

3.3.4 C implementation: header and source files

The difference equation corresponding to (3.20) is

$$y[n] = b_0x[n] + b_1x[n-1] - a_1y[n-1], \tag{3.21}$$

where $x[n]$ and $y[n]$ are the current input and output, while $x[n-1]$ and $y[n-1]$ are the stored filter states.

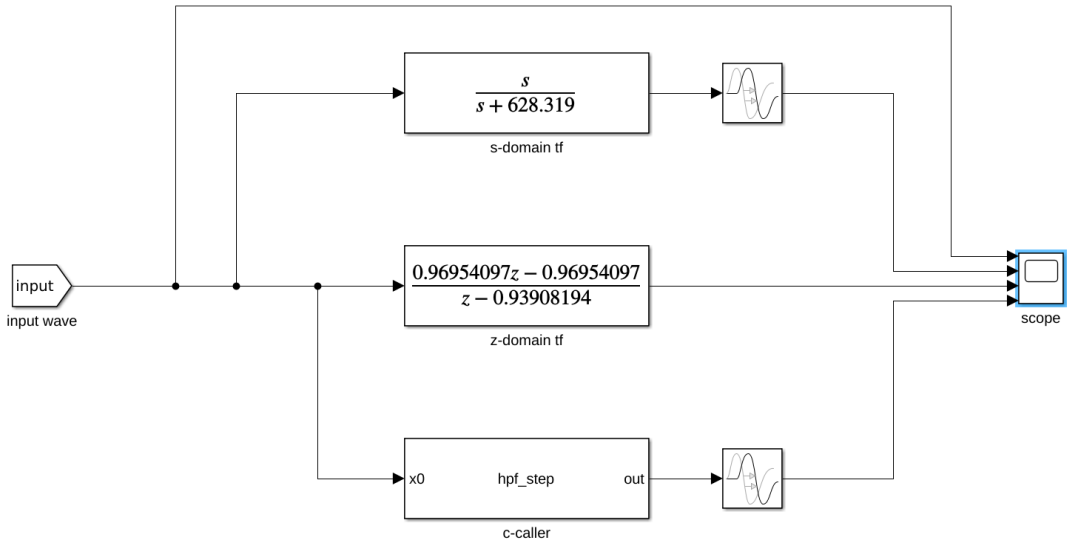

Figure 3.6: HPF subsystem with s–domain, z–domain and C implementation.

The header file defines the public functions:

HPF Header File

```
/* hpf_c_caller.h
 * 1st-order IIR High-Pass Filter
 * fs = 10 kHz, fc  100 Hz
 */

#ifndef HPF_C_CALLER_H
#define HPF_C_CALLER_H

void   hpf_reset(void);
double hpf_step(double x0);

#endif /* HPF_C_CALLER_H */
```

The source file implements the filter:

HPF Source File

```
/* hpf_c_caller.c
 * 1st-order IIR High-Pass Filter
 * fs = 10 kHz, fc  100 Hz
 */

#include <stddef.h>
#include "hpf_c_caller.h"
```

```
/* Coefficients for H(z) = (b0 + b1 z^-1) / (1 + a1 z^-1) */
static const double b0 = 0.96954097;
static const double b1 = -0.96954097;
static const double a1 = -0.93908194;

/* Filter state */
static double x1 = 0.0;
static double y1 = 0.0;

void hpf_reset(void)
{
    x1 = 0.0;
    y1 = 0.0;
}

double hpf_step(double x0)
{
    double y0 = b0 * x0 + b1 * x1 - a1 * y1;

    x1 = x0;
    y1 = y0;

    return y0;
}
```

The implementation closely matches the mathematical model derived in the z–domain. The use of `static` internal variables allows the C filter to maintain memory between function calls, exactly replicating the recursive nature of an IIR filter.

3.3.5 Simulation Results

Figure 3.7 shows the responses of the three HPF implementations to a frequency–swept input ranging from 50 Hz to 300 Hz.

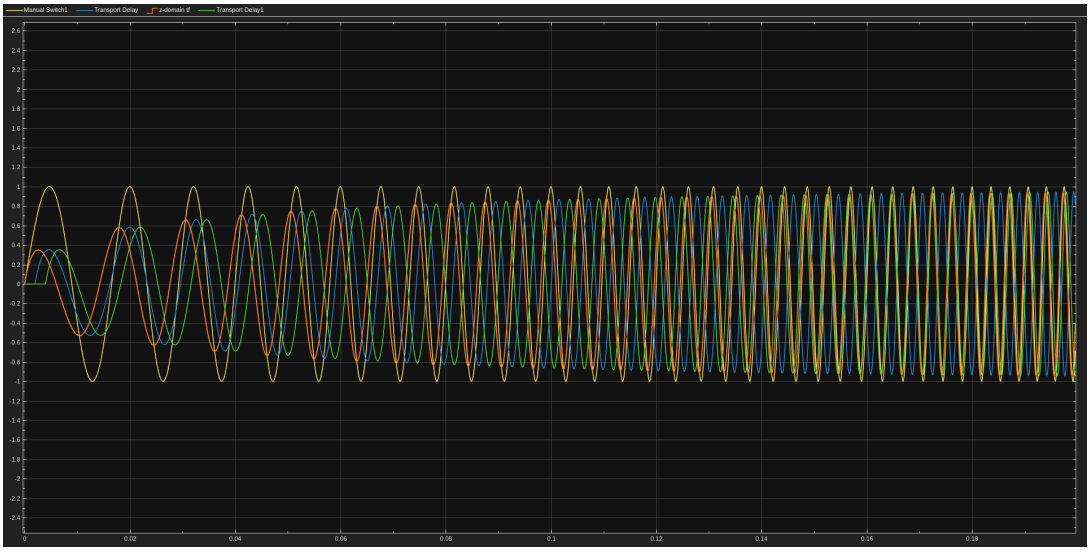

Figure 3.7: HPF Results

[Input from 50Hz to 300Hz]. At low frequencies (below the cutoff), the HPF strongly attenuates the input, resulting in small output amplitudes. As the sweep frequency increases, the output amplitude rises significantly — demonstrating the expected high–pass behavior.

All three implementations (theoretical s–domain, digital z–domain and C code) match closely in amplitude and dynamic response. Again, the small horizontal offsets are intentional transport delays for visualization only. The consistency across all implementations validates the high–pass design, the correctness of the bilinear–transformed coefficients, and the fidelity of the custom C implementation.

3.4 All-pass filter (APF)

The third filter implemented in the project is the first–order all–pass filter (APF). Unlike the low–pass and high–pass filters presented earlier, the APF does not attenuate or amplify any part of the frequency spectrum. Instead, it applies a frequency–dependent *phase shift* while maintaining a constant magnitude of unity across all frequencies. This makes all–pass filters essential in applications such as phase equalization, group–delay shaping, feedback systems, and IIR filter design.

As with the previous filters, the APF was implemented in three equivalent forms:

1. an analog s–domain transfer function,
2. a discrete z–domain transfer function obtained using the bilinear transform,
3. and a C implementation using the corresponding difference equation.

3.4.1 Analog s-domain transfer function

The standard first–order analog all–pass filter has the form

$$H(s) = \frac{s - \omega_c}{s + \omega_c}, \tag{3.22}$$

where ω_c is a tuning constant that determines the frequency at which the phase shift crosses zero degrees.

Using the previously established cutoff

$$\omega_c = 2\pi \cdot 100 \approx 628.319 \; rad/s,$$

the numeric expression becomes

$$H(s) = \frac{s - 628.319}{s + 628.319}. \tag{3.23}$$

In Simulink, the corresponding `Transfer Fcn` block uses:

$$\text{Num}_s = [\,1 \quad -628.319\,], \qquad \text{Den}_s = [\,1 \quad 628.319\,].$$

This block forms the upper branch of the APF subsystem.

3.4.2 Discrete z-domain transfer function

To obtain a realizable digital implementation, the bilinear transform

$$s = \frac{2}{T_s}\,\frac{1 - z^{-1}}{1 + z^{-1}}$$

is applied to (3.22). Using $T_s = 0.0001$ (from $f_s = 10kHz$), let

$$A = \frac{2}{T_s} = 20000.$$

Substituting the bilinear expression into (3.22) gives

$$H(z) = \frac{A(1 - z^{-1}) - \omega_c(1 + z^{-1})}{A(1 - z^{-1}) + \omega_c(1 + z^{-1})}.$$

Expanding numerator and denominator:

$$\text{Num} = (A - \omega_c) + (-A - \omega_c)\, z^{-1},$$

$$\text{Den} = (A + \omega_c) + (-A + \omega_c)\, z^{-1}.$$

Normalizing by $(A + \omega_c)$ produces the standard IIR form

$$H(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}.$$

Substituting $A = 20000$ and $\omega_c = 628.319$ yields:

$$b_0 = \frac{A - \omega_c}{A + \omega_c} \approx 0.93908194,$$

$$b_1 = \frac{-A - \omega_c}{A + \omega_c} = -1,$$

$$a_1 = \frac{-A + \omega_c}{A + \omega_c} \approx -0.93908194.$$

This matches the Simulink implementation:

$$H(z) = \frac{0.93908194\, z - 1}{z - 0.93908194}. \tag{3.24}$$

The `Transfer Fcn` block uses:

$$\text{Num}_z = [\, 0.93908194 \quad -1 \,], \qquad \text{Den}_z = [\, 1 \quad -0.93908194 \,].$$

3.4.3 Simulink APF subsystem

Figure 3.8 presents the subsystem used to evaluate the three APF implementations. The input is routed through:

- the s–domain transfer function (3.23),
- the z–domain transfer function (3.24),
- and the C implementation through the `C Caller` block.

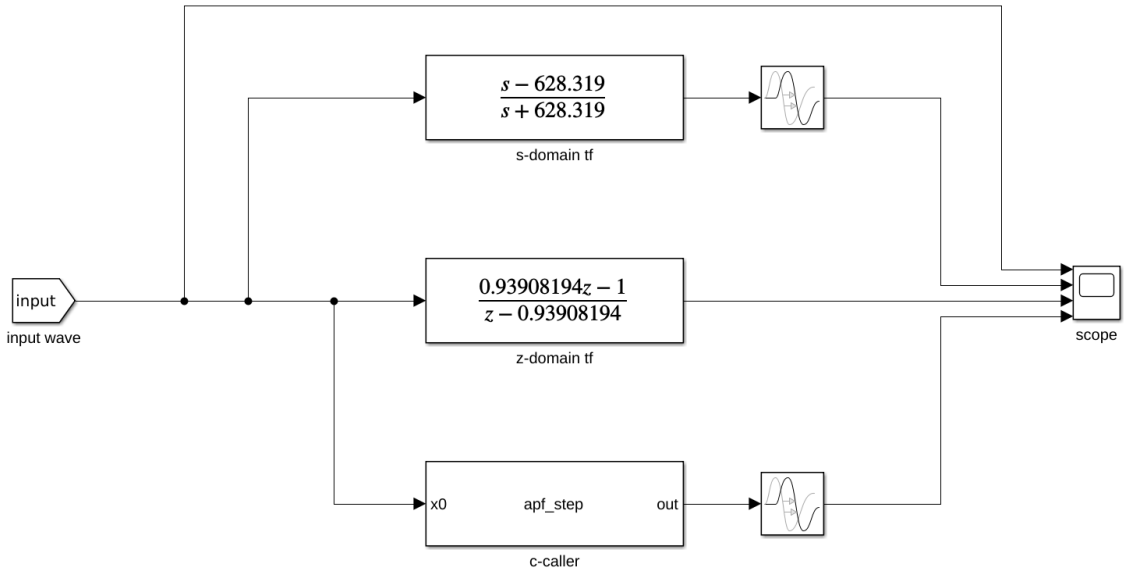

Figure 3.8: APF subsystem with continuous, discrete, and C implementations in parallel.

As with the LPF and HPF, small `Transport Delay` blocks were added to the upper and lower branches purely for visualization. These delays shift the traces slightly so that the three waveforms are distinguishable in the scope but do not affect the correctness of the filter.

3.4.4 C implementation: header and source files

The C code directly implements the difference equation derived from (3.24):

$$y[n] = b_0\, x[n] + b_1\, x[n-1] - a_1\, y[n-1].$$

The header file is:

APF Header File

```
/* apf_c_caller.h
 * 1st-order IIR All-Pass Filter
 * fs = 10 kHz, fc  100 Hz
```

```
 */

#ifndef APF_C_CALLER_H
#define APF_C_CALLER_H

void   apf_reset(void);
double apf_step(double x0);

#endif /* APF_C_CALLER_H */
```

The source file contains the filter logic:

APF Source File

```
/* apf_c_caller.c
 * 1st-order IIR All-Pass Filter
 * fs = 10 kHz, fc  100 Hz
 */

#include <stddef.h>
#include "apf_c_caller.h"

/* Coefficients for H(z) = (b0 + b1 z^-1) / (1 + a1 z^-1) */
static const double b0 = 0.93908194;
static const double b1 = -1.0;
static const double a1 = -0.93908194;

/* Filter state: x[n-1], y[n-1] */
static double x1 = 0.0;
static double y1 = 0.0;

void apf_reset(void)
{
    x1 = 0.0;
    y1 = 0.0;
}

double apf_step(double x0)
{
    /* Difference equation:
       y[n] = b0*x[n] + b1*x[n-1] - a1*y[n-1]
    */
    double y0 = b0 * x0 + b1 * x1 - a1 * y1;
```

```
    /* Update states */
    x1 = x0;
    y1 = y0;

    return y0;
}
```

The structure is identical to the LPF and HPF implementations, ensuring consistency. Only the coefficient set changes, reflecting the unique phase–shifting behavior of the all-pass filter.

3.4.5 Simulation Results

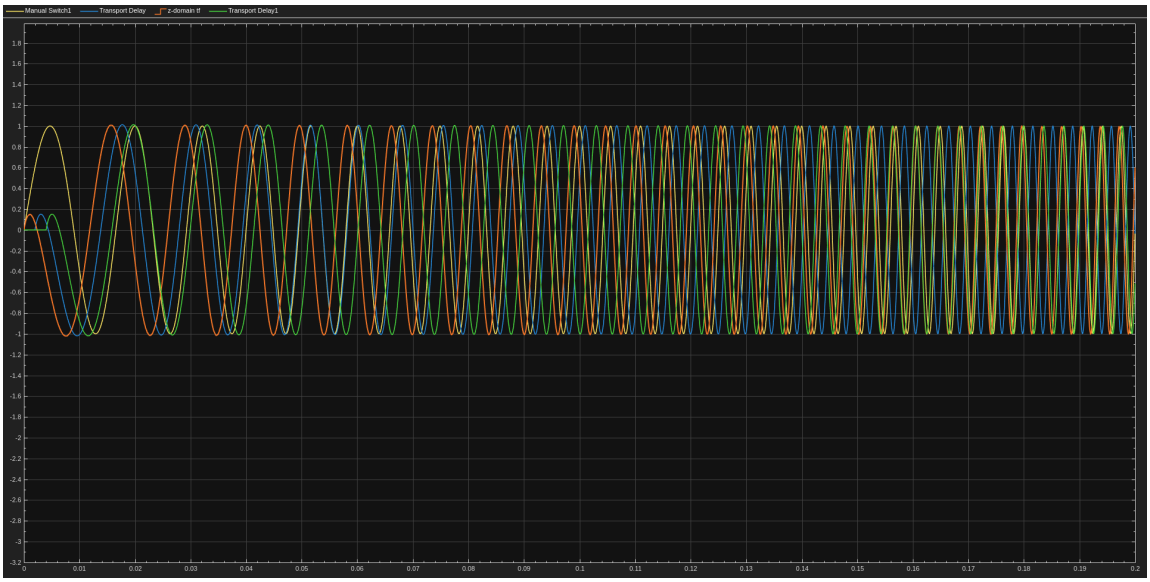

Figure 3.9: APF simulation results for s–domain, z–domain and C implementations.

Figure 3.9 compares the responses of the three APF realizations under a frequency–swept input signal. Since an all–pass filter preserves the magnitude of the input, the amplitude of all three outputs remains approximately identical throughout the frequency sweep. However, the phase response varies significantly with frequency, which is clearly visible in the growing phase offset between the different delayed paths of the Simulink subsystem.

At low frequencies (near 50 Hz), the phase shift is small, so the filtered output initially aligns closely with the input. As frequency increases toward and beyond the design frequency of $f_c = 100$ Hz, the phase shift grows, producing the characteristic time displacement of the waveform while leaving its amplitude unchanged.

The three traces, continuous, discrete, and C-based, overlap almost perfectly aside from the small visualization delays inserted in the diagram. This confirms that:

1. the analog APF behavior is preserved through the bilinear transform,

2. the discrete implementation matches the theoretical phase response,

3. and the C implementation reproduces the exact digital filter behavior sample-by-sample.

The APF thus behaves exactly as expected: producing frequency-dependent phase shifts while maintaining unit magnitude across all frequencies.

3.5 Band-pass filter (BPF)

The fourth filter implemented is a second–order band–pass filter (BPF) centered at $f_c = 100Hz$ with a quality factor of $Q = 5$. Compared to the previous first–order filters (LPF, HPF, APF), this filter requires a second–order biquad structure to produce a selective band–pass response with a narrow bandwidth. The BPF is implemented in three parallel forms: an analog s–domain transfer function, its discrete z–domain equivalent obtained via the bilinear transform, and a custom C implementation executed through the `C Caller` block.

3.5.1 Analog s-domain transfer function

The standard form of a second–order band–pass filter is:

$$H(s) = \frac{\frac{\omega_0}{Q} s}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2}, \tag{3.25}$$

where:

$$\omega_0 = 2\pi f_c, \qquad Q = 5.$$

Substituting $f_c = 100Hz$ gives:

$$\omega_0 = 2\pi \cdot 100 = 628.319 \; rad/s.$$

The numerator becomes:

$$\frac{\omega_0}{Q} s = \frac{628.319}{5} s = 125.6637061 \, s.$$

The denominator is:

$$s^2 + \frac{\omega_0}{Q} s + \omega_0^2 = s^2 + 125.6637061 s + 394784.1760.$$

Thus, the analog transfer function is:

$$H(s) = \frac{125.6637061 \, s}{s^2 + 125.6637061 s + 394784.1760}. \tag{3.26}$$

This expression is implemented in Simulink using a `Transfer Fcn` block with:

$$\text{Num}_s = [\,0 \quad 125.6637061 \quad 0\,], \qquad \text{Den}_s = [\,1 \quad 125.6637061 \quad 394784.1760\,].$$

3.5.2 Discrete z-domain transfer function

To derive a digital equivalent, the bilinear transform

$$s = \frac{2}{T_s}\frac{1 - z^{-1}}{1 + z^{-1}}$$

is applied to the analog prototype (3.25). With $f_s = 10kHz$, the sampling period is

$$T_s = 0.0001 \ s.$$

After substitution, algebraic expansion, and normalization into the standard biquad form,

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}, \tag{3.27}$$

the resulting coefficients are:

$$\begin{aligned} &b_0 = 0.00623784, && b_1 = 0, && b_2 = -0.00623784, \\ &a_1 = -1.98360498, && a_2 = 0.98752433. \end{aligned}$$

Thus, the discrete transfer function implemented in Simulink is:

$$H(z) = \frac{0.00623784 - 0.00623784\, z^{-2}}{1 - 1.98360498\, z^{-1} + 0.98752433\, z^{-2}}. \tag{3.28}$$

These coefficients define a narrow second–order band–pass centered near 100 Hz with $Q = 5$.

3.5.3 Simulink BPF subsystem

Figure 3.10 shows the Simulink subsystem for the band–pass filter. The input waveform is routed simultaneously through:

- the continuous–time s–domain transfer function (3.26),
- the discrete–time z–domain transfer function (3.28),
- and the C implementation through a `C Caller` block.

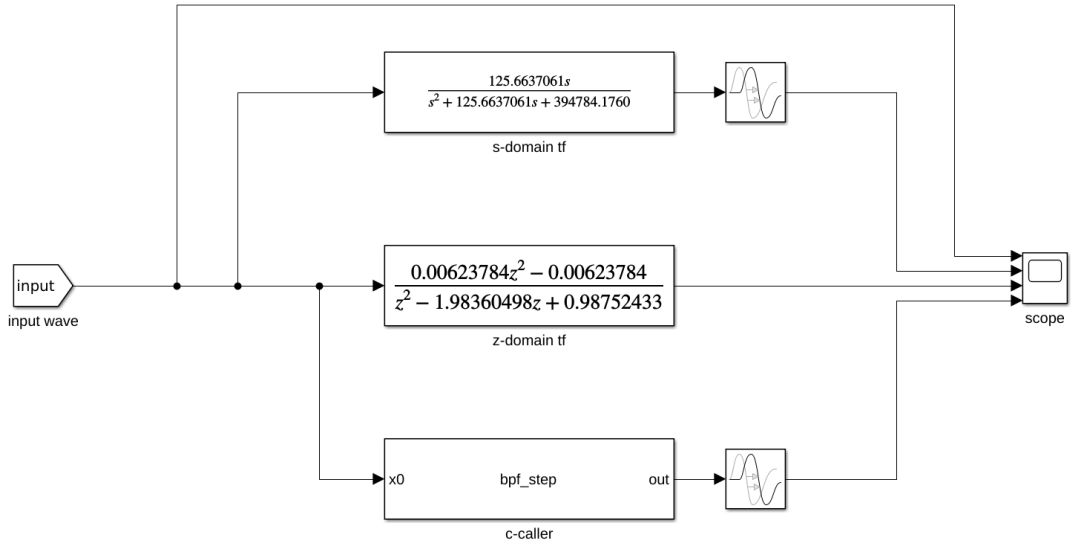

Figure 3.10: BPF subsystem with parallel s–domain, z–domain, and C implementations.

As done for the previous filters, small transport delays were added to the top (analog) and bottom (C Caller) paths to avoid perfect overlap on the scope and improve visual clarity. These delays do not modify the filter behavior.

3.5.4 C implementation: header and source files

The C implementation directly follows the biquad difference equation:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]. \tag{3.29}$$

Header file

BPF Header File

```
/* bpf_c_caller.h
 * 2nd-order IIR Band-Pass Filter
 * fs = 10 kHz, fc  100 Hz, Q = 5
 */

#ifndef BPF_C_CALLER_H
#define BPF_C_CALLER_H

void   bpf_reset(void);
double bpf_step(double x0);

#endif /* BPF_C_CALLER_H */
```

Source file

BPF Source File

```
/* bpf_c_caller.c
 * 2nd-order IIR Band-Pass Filter
 * fs = 10 kHz, fc  100 Hz, Q = 5
 */

#include <stddef.h>
#include "bpf_c_caller.h"

/* Coefficients for biquad filter */
static const double b0 =  0.00623784;
static const double b1 =  0.0;
static const double b2 = -0.00623784;

static const double a1 = -1.98360498;
static const double a2 =  0.98752433;

/* Filter state: x[n-1], x[n-2], y[n-1], y[n-2] */
static double x1 = 0.0, x2 = 0.0;
static double y1 = 0.0, y2 = 0.0;

void bpf_reset(void)
{
    x1 = x2 = 0.0;
    y1 = y2 = 0.0;
}

double bpf_step(double x0)
{
    double y0 = b0 * x0 + b1 * x1 + b2 * x2
              - a1 * y1 - a2 * y2;

    x2 = x1;
    x1 = x0;
    y2 = y1;
    y1 = y0;

    return y0;
}
```

3.5.5 Simulation Results

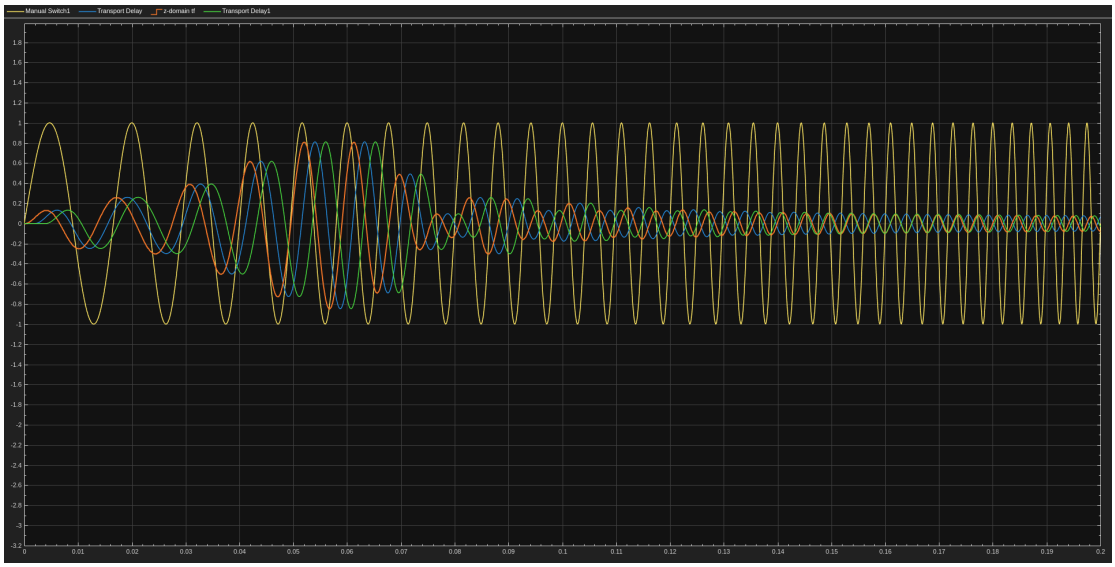

Figure 3.11: BPF simulation results for the s–domain, z–domain, and C implementations.

Figure 3.11 shows the outputs of the three BPF implementations when excited by a swept frequency input ranging from 50 Hz to 300 Hz.

At low frequencies (below $\approx$ 70 Hz), all implementations produce a very small amplitude because the input lies outside the passband. As the frequency approaches the center frequency of 100 Hz, the amplitude increases significantly for all three filters. After crossing the resonant region, the outputs decay again as the frequency continues to rise, which is characteristic of a properly tuned band–pass response.

The analog, digital, and C–based implementations match closely in amplitude and shape. Slight horizontal offsets are due only to the intentionally added transport delays. The consistency confirms the correctness of:

1. the analog filter formulation,
2. the bilinear–transform conversion,
3. and the custom C implementation.

This validates that the BPF design behaves as intended, passing only a narrow band around the center frequency while rejecting frequencies outside the selected range.

3.6 Band-stop filter (BSF)

The last filter implemented in Simulink is a second–order band–stop (notch) filter with center frequency $f_c = 100Hz$, quality factor $Q = 5$, and sampling frequency $f_s = 10kHz$. Its purpose is to strongly attenuate a narrow band around f_c while leaving the rest of the spectrum (both below and above the notch) practically unchanged. As with the BPF, the BSF is realized in three parallel forms: an analog s–domain transfer function, a digital z–domain transfer function obtained via the bilinear transform, and a C implementation based on the corresponding second–order difference equation.

3.6.1 Analog s-domain transfer function

The standard second–order band–stop (notch) prototype with center angular frequency ω_0 and quality factor Q is

$$H_{\mathrm{BSF}}(s) = \frac{s^2 + \omega_0^2}{s^2 + \dfrac{\omega_0}{Q}s + \omega_0^2}. \tag{3.30}$$

For this project the center frequency is $f_c = 100Hz$, so

$$\omega_0 = 2\pi f_c = 2\pi \cdot 100 \approx 628.319\ rad/s,$$

and

$$\omega_0^2 \approx (628.319)^2 \approx 394784.176.$$

With a quality factor $Q = 5$, the damping term becomes

$$\frac{\omega_0}{Q} \approx \frac{628.319}{5} \approx 125.6637.$$

Substituting these numerical values into (3.30) yields the analog transfer function actually implemented in Simulink:

$$H_{\mathrm{BSF}}(s) = \frac{s^2 + 394784.1760}{s^2 + 125.6637061\,s + 394784.1760}. \tag{3.31}$$

In the `Transfer Fcn` block, the numerator and denominator are therefore configured as

$$\mathrm{Num}_s = [\,1 \quad 0 \quad 394784.1760\,], \qquad \mathrm{Den}_s = [\,1 \quad 125.6637061 \quad 394784.1760\,].$$

3.6.2 Discrete z-domain transfer function

To obtain the digital IIR version of the notch filter, the bilinear transform

$$s = \frac{2}{T_s}\,\frac{1 - z^{-1}}{1 + z^{-1}},$$

with $T_s = 1/f_s = 0.0001\ s$, is applied to (3.30). The exact algebra for the second–order case is lengthy, so the coefficients were computed with MATLAB's `c2d` function (bilinear method), starting from (3.31). The resulting discrete transfer function takes the standard biquadratic form

$$H_{\mathrm{BSF}}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}, \tag{3.32}$$

with numerical coefficients

$$b_0 = 0.99376216, \qquad b_1 = -1.98360498, \qquad b_2 = 0.99376216, \tag{3.33}$$

$$a_1 = -1.98360498, \qquad a_2 = 0.98752433. \tag{3.34}$$

Thus, the discrete transfer function used in Simulink is

$$H_{\mathrm{BSF}}(z) = \frac{0.99376216 + (-1.98360498) z^{-1} + 0.99376216 z^{-2}}{1 - 1.98360498 z^{-1} + 0.98752433 z^{-2}}. \tag{3.35}$$

The corresponding `Transfer Fcn` block is configured with

$$\mathrm{Num}_z = [\,0.99376216 \quad -1.98360498 \quad 0.99376216\,], \quad \mathrm{Den}_z = [\,1 \quad -1.98360498 \quad 0.98752433\,].$$

3.6.3 Simulink BSF subsystem

Figure 3.12 shows the Simulink subsystem used to validate the band–stop filter. As with the previous filters, the input signal is split into three branches that feed:

- the analog s–domain model of (3.31),
- the discrete z–domain model of (3.35),
- and the custom C implementation invoked by the `C Caller` block.

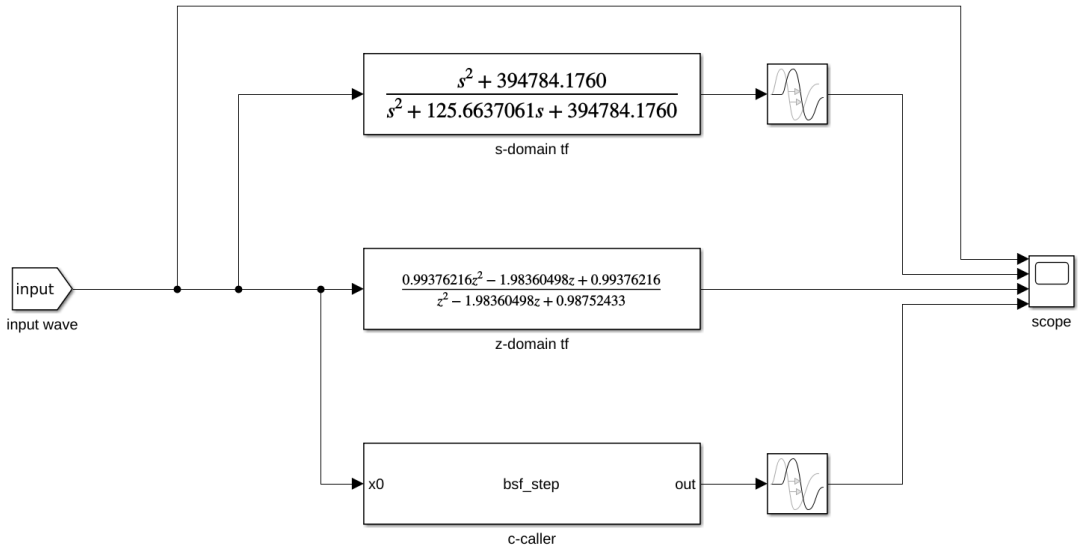

Figure 3.12: BSF subsystem with s–domain, z–domain and C implementation in parallel.

As in the other subsystems, small `Transport Delay` blocks are added in the upper and lower branches solely to separate the waveforms horizontally in the scope. They do not affect the magnitude response or the effective filter design; they simply prevent the three curves from overlapping exactly, making visual comparison easier.

3.6.4 C implementation: header and source files

The discrete transfer function (3.35) corresponds to the following second–order difference equation:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2], \tag{3.36}$$

where $x[n]$ is the input sequence and $y[n]$ the output. To implement this equation in C, two previous inputs ($x[n-1]$, $x[n-2]$) and two previous outputs ($y[n-1]$, $y[n-2]$) must be stored as internal state.

The public interface of the BSF is defined in the header file `bsf_c_caller.h`:

BSF Header File

```
/* bsf_c_caller.h
 * 2nd-order IIR Band-Stop (Notch) Filter
 * fs = 10 kHz, fc  100 Hz, Q = 5
 */

#ifndef BSF_C_CALLER_H
#define BSF_C_CALLER_H

void   bsf_reset(void);
double bsf_step(double x0);
```

```
#endif /* BSF_C_CALLER_H */
```

As before, two functions are provided: `bsf_reset()` clears the internal memory, and `bsf_step()` processes one sample at a time.

The implementation of the difference equation is contained in `bsf_c_caller.c`:

BSF Source File

```
/* bsf_c_caller.c
 * 2nd-order IIR Band-Stop (Notch) Filter
 * fs = 10 kHz, fc  100 Hz, Q = 5
 */

#include <stddef.h>
#include "bsf_c_caller.h"

/* H(z) = (b0 + b1 z^-1 + b2 z^-2) / (1 + a1 z^-1 + a2 z^-2) */

static const double b0 =  0.99376216;
static const double b1 = -1.98360498;
static const double b2 =  0.99376216;

static const double a1 = -1.98360498;
static const double a2 =  0.98752433;

/* States: x[n-1], x[n-2], y[n-1], y[n-2] */
static double x1 = 0.0, x2 = 0.0;
static double y1 = 0.0, y2 = 0.0;

void bsf_reset(void)
{
    x1 = x2 = 0.0;
    y1 = y2 = 0.0;
}

double bsf_step(double x0)
{
    /* Difference equation
       y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2]
              - a1*y[n-1] - a2*y[n-2];
    */
    double y0 = b0 * x0 + b1 * x1 + b2 * x2
```

```
                - a1 * y1 - a2 * y2;

    /* Update states */
    x2 = x1;
    x1 = x0;
    y2 = y1;
    y1 = y0;

    return y0;
}
```

The coefficients are stored as `static const` so they are fixed at compile time, while the state variables `x1`, `x2`, `y1` and `y2` are `static` to preserve their values between successive calls to `bsf_step()`. The `C Caller` block in Simulink passes the current sample $x[n]$ to this function and receives the filtered output $y[n]$, which is then displayed alongside the s–domain and z–domain responses.

3.6.5 Simulation Results

Figure 3.13: BSF simulation results for a swept–frequency input.

Figure 3.13 presents the time–domain responses of the three BSF implementations when excited by the swept–frequency input signal (from 50 Hz up to 300 Hz). At low and high frequencies, far away from the notch, the three responses have amplitudes comparable to the original signal, indicating that the filter behaves almost like an all–pass system in those regions. Around the center frequency $f_c = 100$ Hz, however, a clear attenuation window appears: the amplitude of the filtered output decreases markedly, showing the rejection of that narrow band.

Again, small transport delays were added in the s–domain and C branches only to separate the curves visually; they do not alter the effective magnitude response. The strong agreement among the blue (s–domain), orange (z–domain) and green (C implementation) traces confirms that the analog prototype, the digital biquadratic realization and the hand–written C code are consistent and correctly implement the intended notch behavior.

3.7 Simulink video walkthrough

Simulink video walkthrough

Chapter 4

STM32 Real-Time Filter Implementation

4.1 STM32 Implementation of Digital Filters

In the second phase of this project, real–time digital signal filtering was implemented on an embedded microcontroller platform. The STM32G474RE microcontroller was used to deploy the same five filters previously designed and validated through simulation: a Low–Pass Filter (LPF), High–Pass Filter (HPF), All–Pass Filter (APF), Band–Pass Filter (BPF), and Band–Stop / Notch Filter (BSF). Each filter was executed directly on the hardware to evaluate its performance and verify correct analog input–output behavior.

The STM32CubeIDE development environment was used to configure the peripherals and implement the firmware. The microcontroller samples an external analog signal using its ADC (Analog–to–Digital Converter), processes the sample through the selected digital filter algorithm, and outputs the filtered result through the DAC (Digital–to–Analog Converter). Timer 1 was configured to ensure precise synchronization of the sampling operation.

4.1.1 Description of the Embedded Implementation

The objective of this stage was to port the previously validated filters to the STM32 microcontroller and verify their real–time analog performance using laboratory measurement equipment.

Unlike MATLAB/Simulink simulations, this stage processes actual analog signals. The workflow executed on the hardware is described below:

1. Acquisition of the analog input signal using ADC1 on pin PA0.
2. Conversion of the digitized sample into a normalized value and filtering using one of the implemented digital algorithms.
3. Reconstruction of the filtered signal as an analog voltage through DAC1 on pin PA4.
4. Visualization of the input and filtered output on an oscilloscope.

This approach confirms that the filter behavior demonstrated in simulation is preserved when deployed on an embedded system.

4.1.2 STM32 Firmware Overview

A custom real–time signal processing firmware was developed in C. The system continuously performs the following operations:

1. ADC conversions triggered by Timer 1 at a sampling rate of 10 kHz.
2. Real–time filtering based on the currently selected filter algorithm.
3. DAC output update for continuous analog reconstruction of the filtered signal.

The firmware contains five source files and five corresponding header files (LPF, HPF, APF, BPF, BSF). These files contain the same difference equations used in the MATLAB/Simulink design. A filter selection variable determines the filter executed during operation.

4.1.3 Peripheral Configuration on STM32

Three hardware peripherals were configured using STM32CubeIDE to enable real–time digital filtering:

- **ADC1 — Analog Input**
- **DAC1 — Analog Output**
- **Timer 1 — Sampling Synchronization**

ADC1 — Analog Input (PA0) The signal generator output is connected to PA0, configured as `ADC1_IN1`. The ADC is configured with:

- 12-bit resolution
- Single-conversion mode
- Trigger source: TIM1 TRGO (rising edge)

Figure 4.1 to Figure 4.3 presents the ADC settings used.

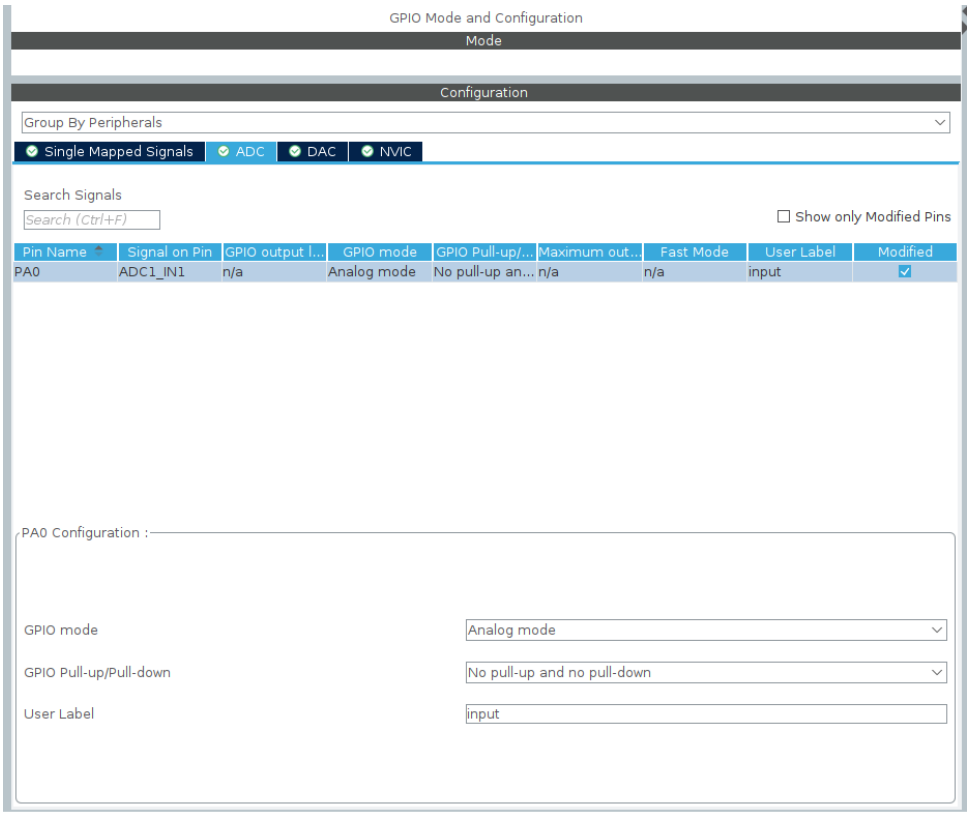

Figure 4.1: [1] ADC1 configuration: PA0 as analog input (single-ended) with TIM1 triggering.

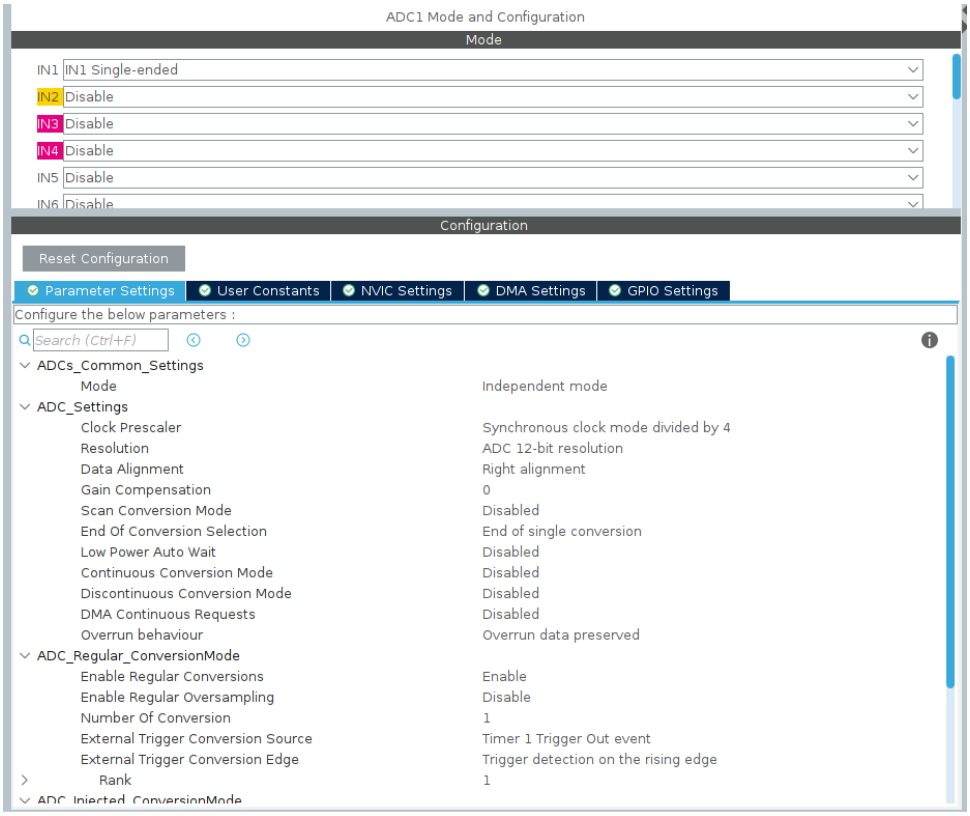

Figure 4.2: [2] ADC1 configuration: PA0 as analog input (single-ended) with TIM1 triggering.

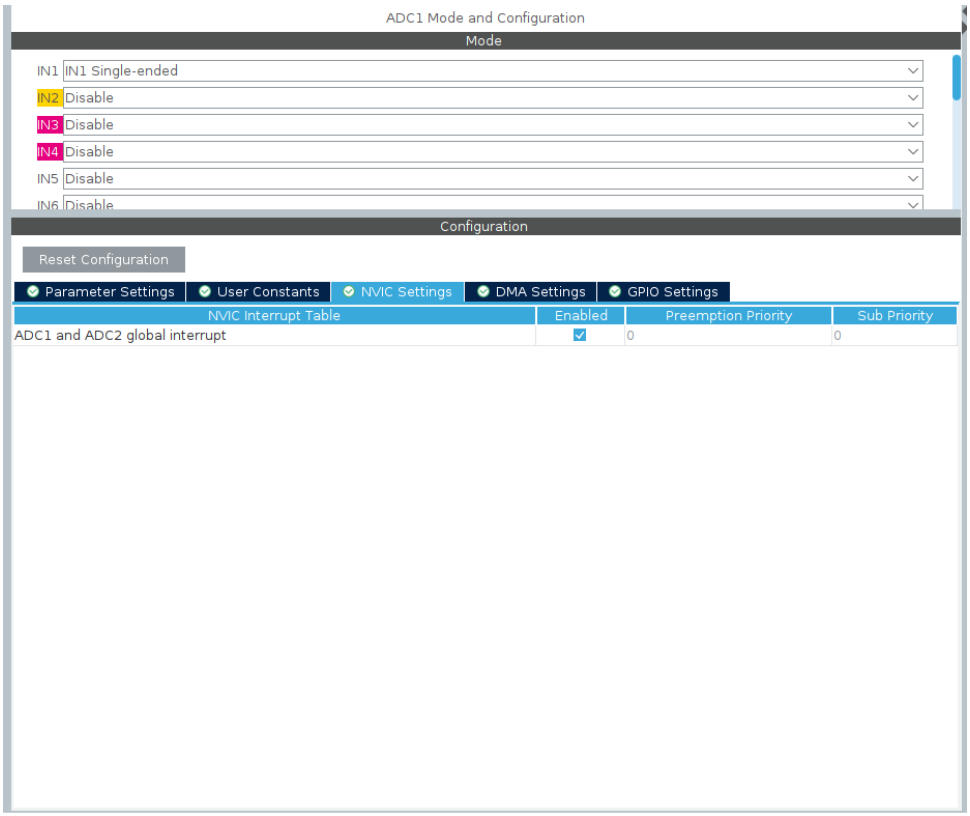

Figure 4.3: [3] ADC1 configuration: PA0 as analog input (single-ended) with TIM1 triggering.

DAC1 — Analog Output (PA4) The filtered signal is generated on PA4, configured as `DAC1_OUT1`. The DAC operates in:

- Normal mode
- Output buffer enabled
- Software-triggered (updated in the ADC interrupt)

Figure 4.4 and Figure 4.5 show the corresponding configuration.

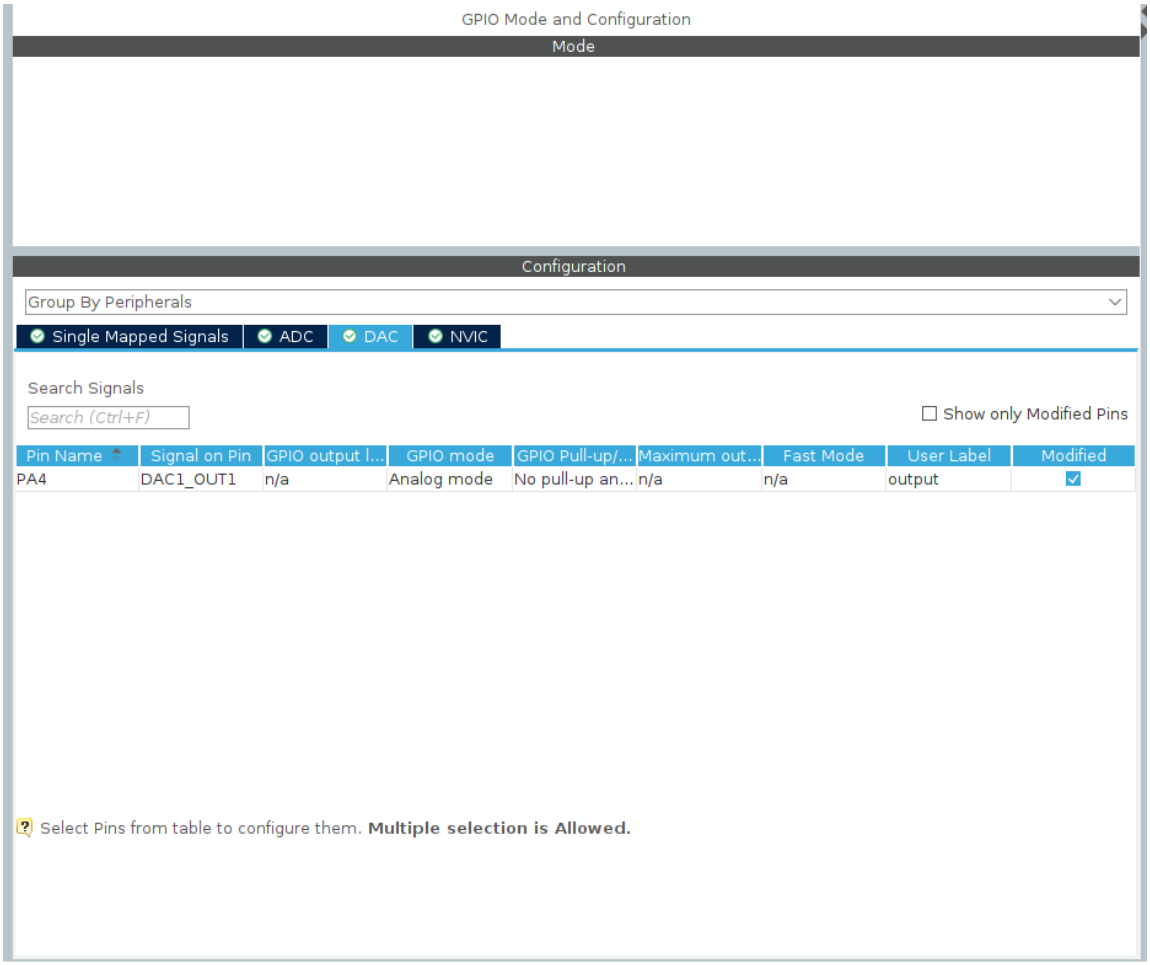

Figure 4.4: [1] DAC1 configuration: PA4 as analog filtered output signal.

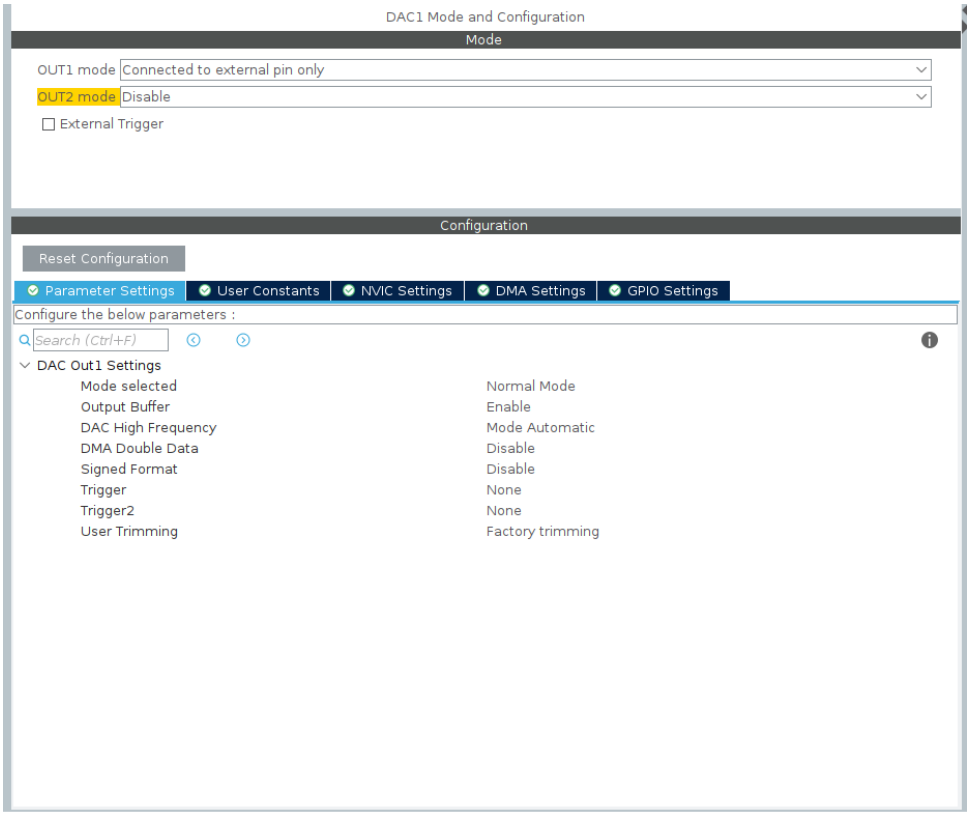

Figure 4.5: [2] DAC1 configuration: PA4 as analog filtered output signal.

Timer 1 — Sampling Trigger (10 kHz) Timer 1 is responsible for maintaining a sampling frequency of 10 kHz. Each Update Event generates a TRGO signal that triggers the ADC conversion. This ensures accurate and constant sampling without jitter.

Figure 4.6 illustrates the TIM1 configuration.

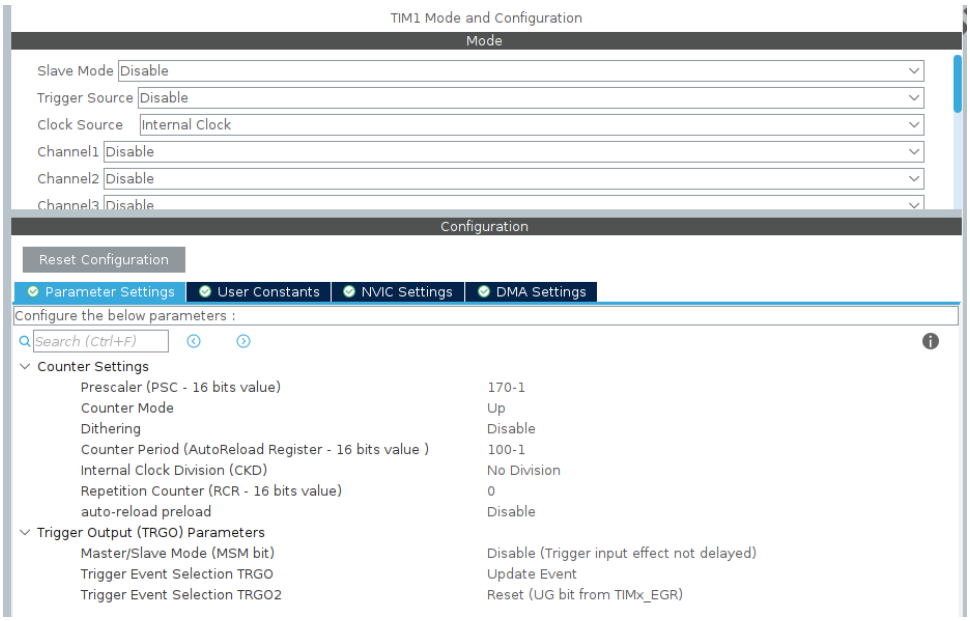

Figure 4.6: TIM1 configuration controlling the sampling frequency at 10 kHz.

4.1.4 Pinout, Clock Setup and Physical Connections

Figure 4.7 displays the selected pin configuration, where PA0 and PA4 are designated as analog input and output, respectively.

Figure 4.7 and Figure 4.8 show the system clock configuration in which the PLL is used to provide a 170 MHz CPU clock, enabling efficient digital signal processing.

- PA0 receives the input from the signal generator.
- PA4 outputs the filtered signal to the oscilloscope.
- Power and grounding follow standard laboratory practices.

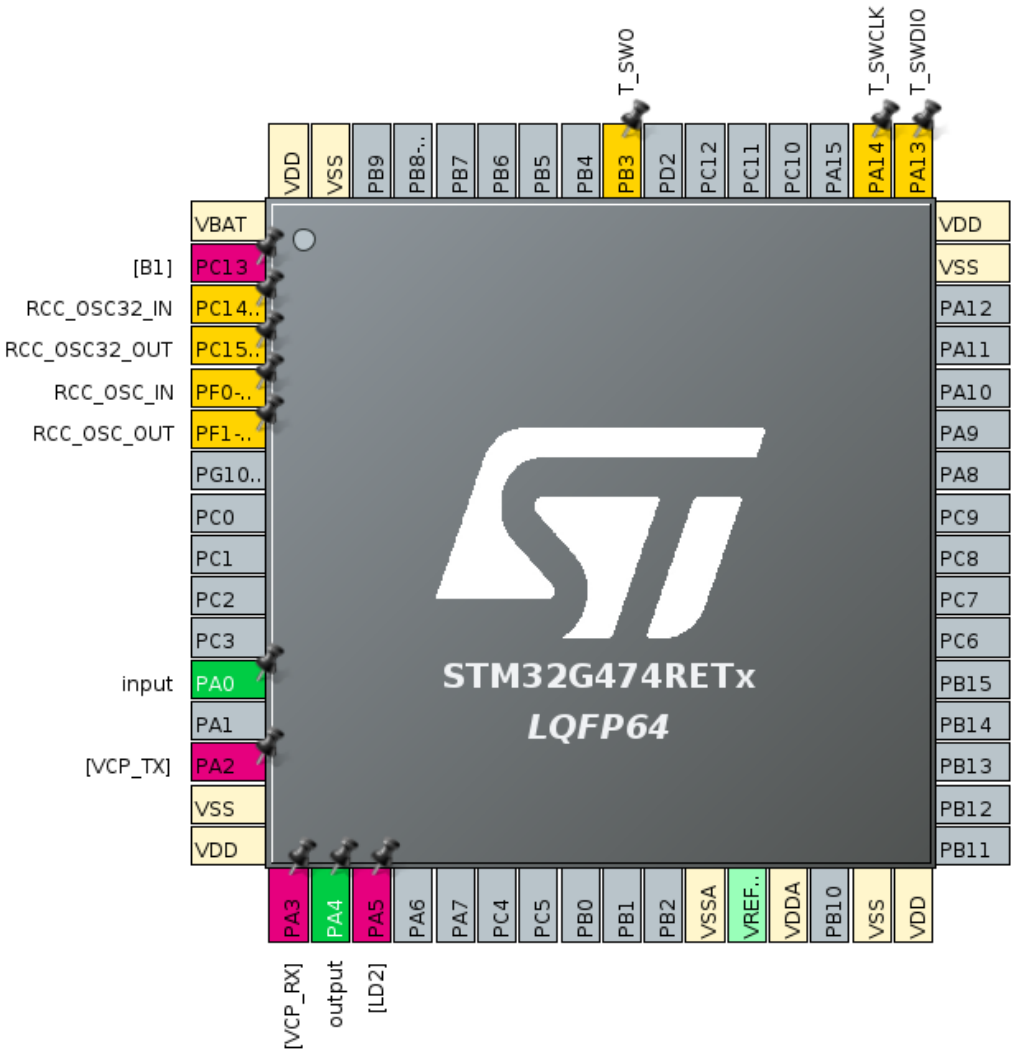

Figure 4.7: Pinout configuration showing ADC1 input on PA0 and DAC1 output on PA4.

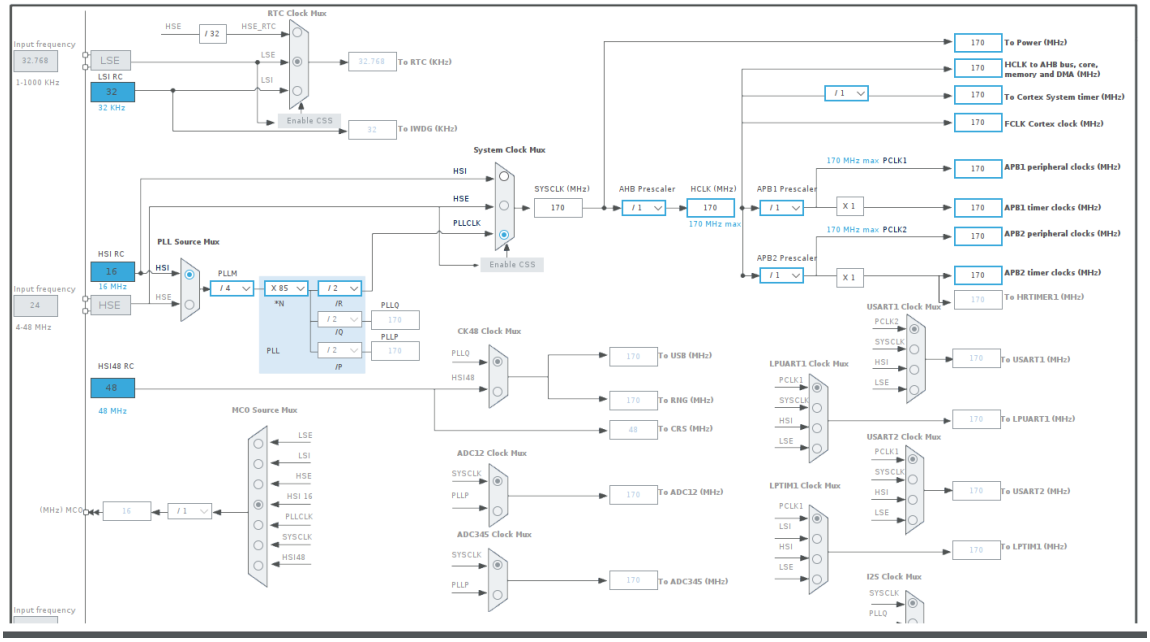

Figure 4.8: [1] Clock configuration at 170 MHz using PLL for stable real–time operation.

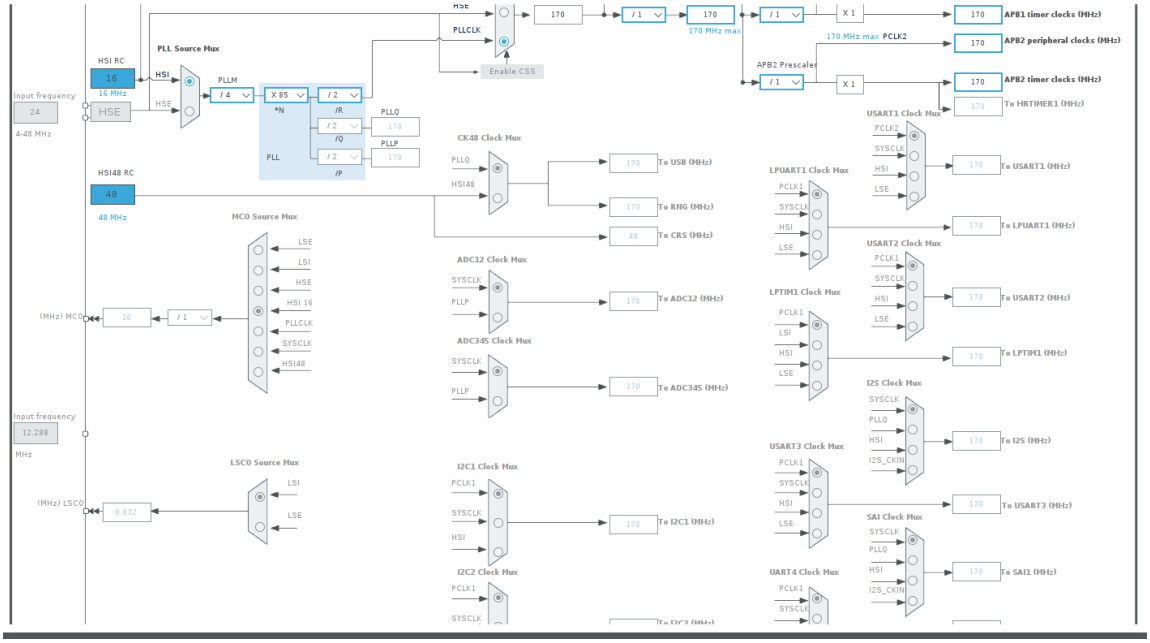

Figure 4.9: [2] Clock configuration at 170 MHz using PLL for stable real–time operation.

Demonstration Video of the Setup

A brief demonstration video is included to show the physical test environment and connections using laboratory instruments:

Video: STM32 + Function Generator + Oscilloscope Connections

4.2 Real-Time Filter Firmware Architecture

This section explains the structure and functionality of the firmware developed for real-time digital filtering on the STM32G474RE microcontroller. The program was implemented in C using STM32CubeIDE, where the ADC interrupt controls the sampling routine, the filter computation is executed, and the DAC generates the reconstructed analog output.

The code integrates five digital filters (LPF, HPF, APF, BPF, BSF), previously designed and validated in simulation. Each filter has its own `.c` and `.h` file pair, containing the same recursive difference equations derived from the Simulink implementation. A global variable determines which filter is active during execution.

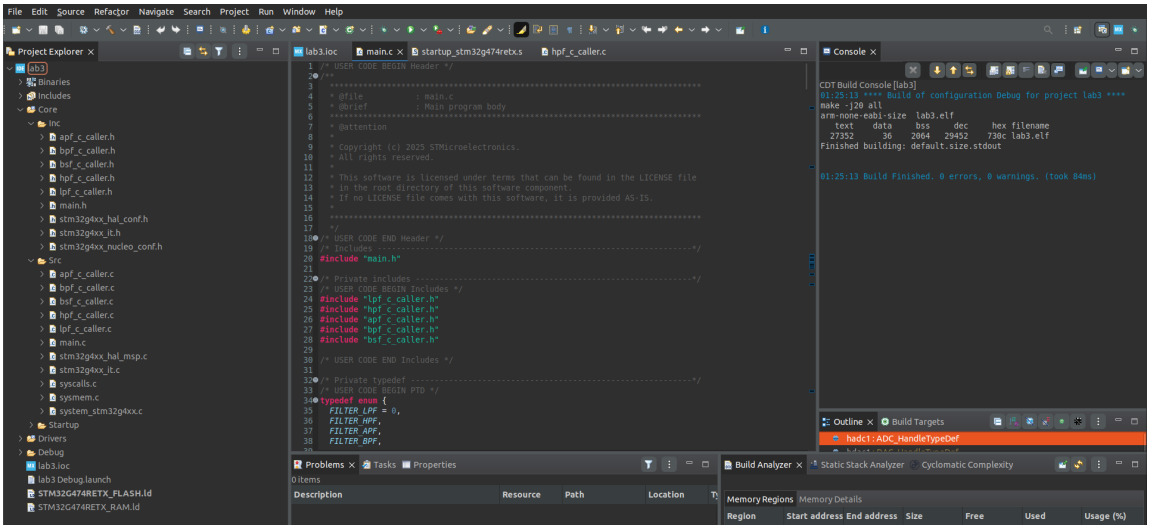

Figure 4.10: Each filter has its own `.c` and `.h` file pair

The most relevant portions of the firmware are described below.

4.2.1 Filter Selection and Initialization

All five filters are included at the top of the program through their respective header files. An enumeration type was defined to simplify filter switching during tests.

Filter Selection Enumeration

```
typedef enum {
  FILTER_LPF = 0,
  FILTER_HPF,
  FILTER_APF,
  FILTER_BPF,
  FILTER_BSF
} filter_t;

volatile filter_t current_filter = FILTER_BPF;  // Select active filter
```

Each filter contains an initialization function "reset()" that clears internal memory states before processing begins. These reset functions are called once in the `main()` function after peripheral initialization.

4.2.2 Peripheral Activation for Real-Time DSP

The embedded system requires three peripherals to operate cooperatively:

- ADC1: Samples the analog input at 10 kHz.
- TIM1: Generates a periodic trigger for ADC sampling.
- DAC1: Reconstructs the filtered output as an analog voltage.

Once configured, the firmware activates all peripherals before entering the main loop:

Peripheral Startup

```
// Start DAC output on PA4
HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);

// Start ADC with interrupt triggered by TIM1
HAL_ADC_Start_IT(&hadc1);

// Start TIM1 base counter for 10 kHz triggering
HAL_TIM_Base_Start(&htim1);
```

After this stage, the infinite loop remains empty because all real-time work is interrupt-driven.

4.2.3 ADC Interrupt Callback: Real-Time DSP Execution

Digital filtering is executed inside the ADC conversion complete interrupt routine. Each time TIM1 triggers a conversion, the ADC generates an interrupt and the following processing pipeline occurs:

1. Acquire latest analog sample from ADC register.
2. Convert ADC integer value to a normalized voltage.
3. Execute selected IIR filter difference equation.
4. Saturate output to valid range.
5. Convert normalized output back to DAC digital value.
6. Write result to DAC1_OUT1 (PA4).

The callback implementation is shown below:

ADC Interrupt Filter Processing

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
  if (hadc->Instance == ADC1)
  {
    uint32_t raw = HAL_ADC_GetValue(&hadc1);
    double x = ((double)raw) / 4095 * 3.3 - 1;

    double y = 0.0;
    switch (current_filter)
    {
      case FILTER_LPF: y = lpf_step(x) + 1; break;
      case FILTER_HPF: y = hpf_step(x) + 1; break;
      case FILTER_APF: y = apf_step(x) + 1; break;
      case FILTER_BPF: y = bpf_step(x) + 1; break;
      case FILTER_BSF: y = bsf_step(x) + 1; break;
      default: y = x; break;
    }

    if (y > 3.3) y = 3.3;
    if (y < 0)   y = 0;

    uint32_t dac_val = (uint32_t)(4095 / 3.3 * y);
    if (dac_val > 4095) dac_val = 4095;

    HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1,
                     DAC_ALIGN_12B_R, dac_val);
  }
}
```

This interrupt executes at 10 kHz, enabling continuous real-time digital signal processing.

4.3 Real-Time Filtering Results

This section presents the results obtained from the real–time implementation of the digital filters on the STM32G474RE. Each filter was tested individually by applying a swept–frequency sinusoidal input signal from the function generator. The resulting input and output waveforms were captured using an oscilloscope to evaluate the frequency-selective behavior of each filter.

The following videos correspond to each implemented filter:

- **Low–Pass Filter (LPF)** **Video: LPF Real–Time Output**

- **High–Pass Filter (HPF)** **Video: HPF Real–Time Output**

- **All–Pass Filter (APF)** **Video: APF Real–Time Output**

- **Band–Pass Filter (BPF)** **Video: BPF Real–Time Output**

- **Band–Stop Filter (BSF)** **Video: BSF Real–Time Output**

Each demonstration evaluates real–time performance by observing amplitude response changes as the input frequency increases and decreases. The expected theoretical behavior was validated correctly in all tests.

4.4 Conclusion

This project successfully demonstrated the complete workflow for designing, simulating, and implementing real–time digital filters using an embedded platform. Five fundamental IIR filters were developed: Low–Pass (LPF), High–Pass (HPF), Band–Pass (BPF), Band–Stop / Notch (BSF), and All–Pass (APF). Initially, each filter was modeled analytically using its transfer function in the s–domain, then discretized using the Bilinear Transform method to obtain the corresponding z–domain representation. Simulation in MATLAB/Simulink confirmed theoretical expectations regarding frequency response and temporal behavior.

The second stage focused on real–time deployment using the STM32G474RE microcontroller. By utilizing the ADC, DAC, and Timer 1, the system was able to acquire an analog input signal, digitally process each sample using the selected filter algorithm, and generate an analog filtered output at a sampling frequency of 10 kHz. The real–time behavior was verified experimentally using a function generator and oscilloscope.

The experimental results demonstrated that the amplitude and frequency response of each filter in hardware closely matched the simulation results, therefore validating the correct implementation of the difference equations on the embedded system. The work confirms that the selected microcontroller provides sufficient computational performance to execute IIR filters in real–time with low latency and stable operation.

Overall, this project strengthened the understanding of digital signal processing fundamentals, discretization techniques, embedded C programming, and peripheral configuration for real–time signal acquisition and generation. The successful integration of theoretical analysis, simulation, and physical implementation highlights the reliability of the designed filters and the suitability of the STM32 platform for real–time DSP applications.

Future improvements may include extending the design to higher–order filters, implementing fixed–point arithmetic for optimization, evaluating computational load and energy efficiency, or integrating DMA and dedicated DSP accelerators to enhance system performance.

Bibliography

[1] ScienceDirect Topics, “All-pass filters,” 2024. Accessed: 2025-11-19.

[2] DSP StackExchange Community, “Higher order butterworth filters,” 2024. Accessed: 2025-11-19.