POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE
MASTER DEGREE IN AUTOMATION ENGINEERING

DISTRIBUTED MEASUREMENT AND DATA ACQUISITION SYSTEMS

Prof. Attilio Di Nisio

Title:

Development of a Low-Cost STM32-Based Embedded System for Signal Generation and Acquisition

Student:
Domenico D'Introno

ACADEMIC YEAR 2025-2026

Contents

1 Introduction

Professional electronic instrumentation, such as digital oscilloscopes and function generators, represents a significant financial investment, often limiting accessibility in educational laboratories and for hobbyists. To address this challenge, this project presents the design and implementation of a cost-effective embedded system capable of performing both signal generation and acquisition tasks.

Based on a general-purpose microcontroller, the system leverages internal Digital-to-Analog (DAC) and Analog-to-Digital (ADC) converters to synthesize standard waveforms—such as sine, sawtooth, and step signals—and simultaneously digitize incoming analog signals in real-time. A custom-developed Graphical User Interface (GUI) running on a host PC communicates with the hardware, allowing users to control parameters, visualize waveforms, and record data for further analysis.

The result is a compact, low-cost platform that effectively replicates the core functionalities of benchtop equipment, providing a practical solution for signal analysis and system identification in didactic environments.

2 Hardware

The hardware architecture of the proposed system is built upon the STMicroelectronics STM32 ecosystem. The choice of a 32-bit ARM Cortex-M microcontroller ensures sufficient computational power to handle signal generation, data acquisition, and high-speed communication simultaneously without blocking the CPU.

2.1 The NUCLEO-F446RE Development Board

The core of the project is the **NUCLEO-F446RE** development board, based on the **STM32F446RE** microcontroller. This MCU features an ARM Cortex-M4 core with a Floating Point Unit (FPU).

Key specifications of the board relevant to this project include:

- **Core:** ARM Cortex-M4 running at up to 180 MHz.
- **Memory:** 512 KB of Flash memory and 128 KB of SRAM, providing ample space for circular buffers and data storage.
- **Analog Peripherals:** Three 12-bit ADCs and one 12-bit DAC, essential for the oscilloscope and function generator emulation.

The Nucleo board was selected for its high performance-to-cost ratio.

2.2 Peripherals Configuration

To achieve real-time performance, the firmware configuration relies heavily on Direct Memory Access (DMA) and hardware timers, minimizing CPU intervention during data transfer in order to ensure high data transfer rate and interactivity.

The peripherals are configured as follows based on the application requirements:

Signal Generation (DAC)

The internal DAC is used to generate the stimulus signals. To ensure precise timing, the DAC conversion is triggered by **TIM8**.

- **Mode:** DMA-driven. During system initialization, the CPU pre-computes two 1000-element Look-Up Tables (LUTs), representing one period of the Sine and Sawtooth waveforms respectively. The DMA controller (DMA1 Stream 5) is configured to automatically transfer these values to the DAC output register at each timer update. By utilizing the DMA's circular mode, the system ensures a continuous data feed to the DAC, resulting in the generation of uninterrupted signals.
- **Pinout:** The analog signal is output on pin PA4.

Signal Acquisition (ADC)

Two distinct ADCs are configured to handle the different operating modes:

- **ADC1 (Step Response):** Configured in "One-Shot" mode. It is triggered by **TIM2** at a fixed frequency of 10 kHz. Once the buffer of 2000 samples is full, the acquisition stops, and data is sent to the PC.
- **ADC2 (Frequency Response):** Configured in Circular mode. It is triggered by **TIM8**, ensuring **perfect synchronization with the DAC generation**. This synchronous approach eliminates phase drift between the generated and measured signals. The DMA (DMA2 Stream 2) fills a circular buffer, which is then streamed to the host PC using a **ping-pong approach** to avoid data loss.

- **Pinout:** ADC1 is connected to **PA0**, while ADC2 is connected to **PA1**.

Timers

Hardware timers are the heartbeat of the system, determining both generation and acquisition sampling rates. The project leverages two timers:

- **TIM2:** Prescaler set to 83 and Period to 99 (with 84 MHz clock), resulting in a **fixed 10 kHz trigger frequency** for step response analysis.
- **TIM8:** Used for waveform generation. The Auto-Reload Register (ARR) is dynamically updated by the firmware to vary the signal frequency based on user commands.

Both TIM2 and TIM8 are configured with the **Trigger Event Selection** parameter set to **Update Event**, generating a precise hardware trigger signal for the ADC and DAC peripherals upon every timer overflow.

Communication (UART)

Data transfer to the host PC is managed by **USART2**. To sustain the high data throughput required by the oscilloscope (streaming voltage values in real-time), the interface is configured at a **high baud rate of 921600 bps.**

- **Transmission (TX):** Handled by DMA (DMA1 Stream 6) to send large data chunks without CPU overhead.
- **Reception (RX):** Interrupt-based to promptly parse commands (Start, Stop, Frequency Change, Waveform Selection) from the GUI.

Peripheral	**MCU Pin**	**Function**
ADC1_IN0	PA0	Step Response Input
ADC2_IN1	PA1	Frequency Response Input
USART2_TX	PA2	Serial Data Output
USART2_RX	PA3	Serial Command Input
DAC_OUT1	PA4	Waveform Output
GPIO_OUTPUT	PA15	Step Signal Output

Table 1: Pinout Configuration Summary

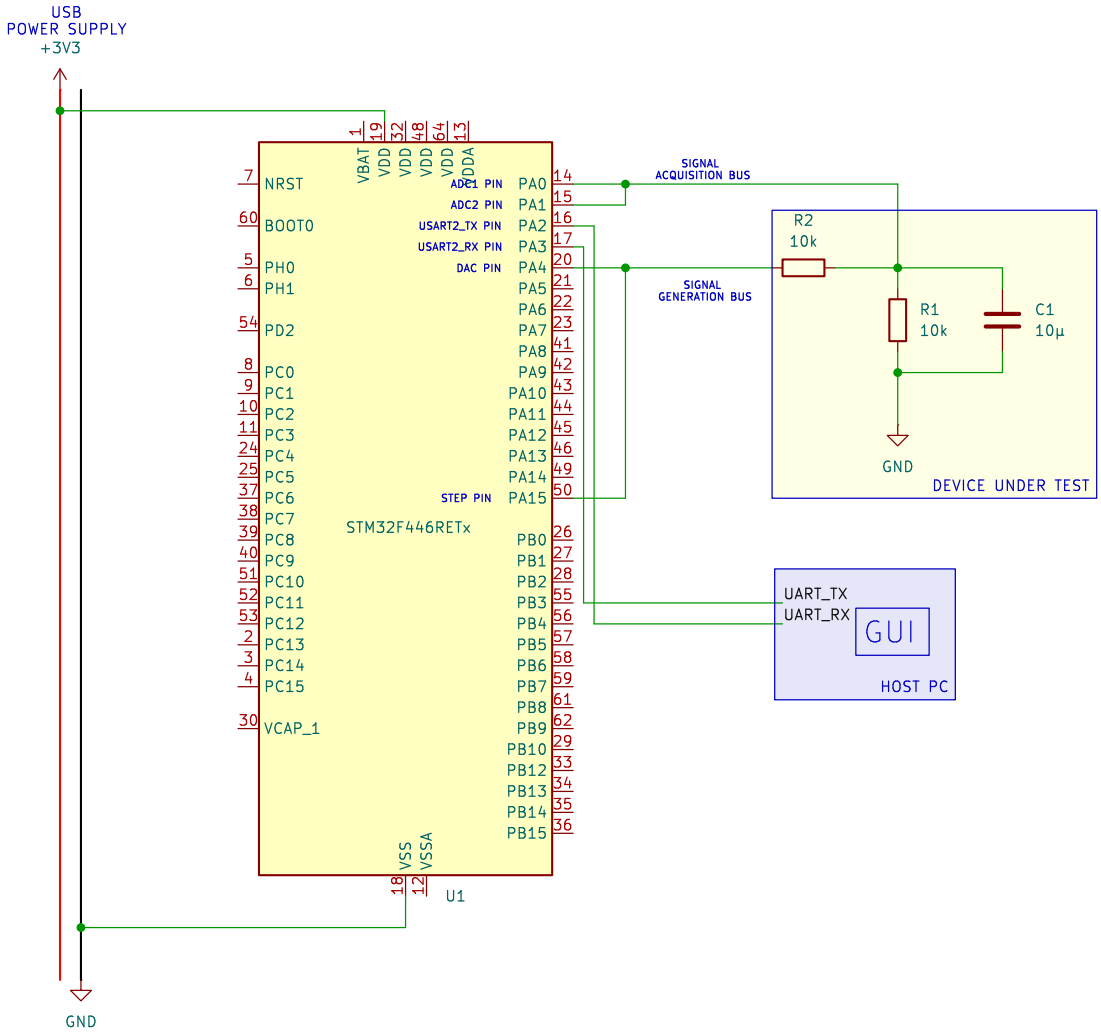

Figure 1: Hardware scheme, including host PC and Device Under Test

3 Firmware

The firmware logic is designed to maximize real-time performance by offloading data transfer and timing tasks to the hardware peripherals (DMA and Timers), keeping the CPU load minimal during signal acquisition. The code is written in C using the STM32 HAL library.

3.1 Initialization and System Startup

Upon system reset, after the low-level hardware configuration (System Clock and Peripherals), the firmware initializes the application-specific variables.

A fundamental part of the logic is the definition of the **Finite State Machine** (FSM) states via the `SystemState_t` enumeration, which governs the transition between Idle, Step Response, and Frequency Analysis modes. Of course, at boot, `SystemState_t` is set to Idle.

Listing 1: System FSM definition

```
typedef enum {
    MODE_IDLE,
    MODE_STEP_LAUNCH,
    MODE_WAVE_LAUNCH,
    MODE_STEP_RUNNING,
    MODE_FREQ_RUNNING
} SystemState_t;
```

Critically, the firmware generates the waveforms dynamically at startup. Instead of storing large static arrays in Flash memory, the CPU computes the samples for the Sine and Sawtooth waves using mathematical functions ($\sin(x)$ and linear ramps) and stores them in RAM arrays (`DAC_Sine` and `DAC_Saw`). This approach allows for flexibility in changing waveform resolution or types in the future.

Finally, the reception interrupt for the UART is enabled, allowing the system to listen for incoming commands from the host PC immediately.

Listing 2: Waveforms LUT generation and UART initialization

```
for (int i = 0; i < LUT_SAMPLES; i++){
    DAC_Sine[i] = (uint16_t)((4095.0f/2.0f) * (1.0f + sin(2.0f*M_PI*i/
        ↪ LUT_SAMPLES)));
}

for(int i = 0; i < LUT_SAMPLES; i++){
    DAC_Saw[i] = (uint16_t)(4095.0f/(LUT_SAMPLES-1)*i);
}

HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
```

3.2 Main Loop

The `while(1)` loop is intentionally minimalistic. Since the heavy lifting of signal generation and data acquisition is handled by the DMA controllers and Interrupt Service Routines (ISRs) in the background, the main loop acts solely as a dispatcher.

It monitors the `current_state` variable. When an interrupt changes the state to a "LAUNCH" status (e.g., `MODE_STEP_LAUNCH`), the main loop calls the appropriate setup function once and then waits. This non-blocking design ensures that the CPU is not occupied by polling loops, reducing jitter in the control signals.

Listing 3: The minimal Main Loop

```
while (1)
{
    if (current_state == MODE_STEP_LAUNCH) stepResponse();

    if (current_state == MODE_WAVE_LAUNCH) waveformResponse();
}
```

3.3 Custom Functions

Two **helper functions** were implemented to abstract the hardware complexity and manage the measurement routines.

- `setPinMode`: since the Step response and Frequency analysis require different pin configurations, a wrapper function was created to reconfigure GPIO modes dynamically without resetting the MCU.

Listing 4: setPinMode function

```
void setPinMode(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, uint32_t Mode)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    GPIO_InitStruct.Pin = GPIO_Pin;
    GPIO_InitStruct.Mode = Mode;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

    HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
}
```

- `sendData`: This function encapsulates the binary protocol. It waits for the UART to be ready, sends the synchronization header (`0xAA 0xBB`), the mode byte, and finally triggers the DMA to transfer the ADC buffer content to the serial port.

Listing 5: sendData function

```
void sendData(volatile uint16_t* buffer, uint16_t buffer_length, uint8_t
    ↪ mode)
{
    if (huart2.gState != HAL_UART_STATE_READY) return;

    HAL_UART_Transmit(&huart2, (uint8_t*)header_seq, 2, 1);
    HAL_UART_Transmit(&huart2, &mode, 1, 1);
    HAL_UART_Transmit_DMA(&huart2, (uint8_t*)buffer, buffer_length * 2);
}
```

- `changeFreq`: this function modifies the generated waveforms frequencies by acting on the frequency of **TIM8**, the timer that clocks the DAC. Some detailed explanation is required, as it implements one core feature of the system. TIM8, is clocked by the internal APB2 bus clock ($f_{clk} = 84$ MHz). The frequency at which the timer generates an Update Event (f_{update}), and consequently triggers a DAC conversion, is determined by the formula:

$$f_{update} = \frac{f_{clk}}{(PSC + 1) \times (ARR + 1)} \tag{1}$$

Since the prescaler (PSC) is set to 0, the equation simplifies to $f_{update} = \frac{f_{clk}}{ARR+1}$. To generate one complete cycle of the waveform, the system must output all N_{LUT} samples. Therefore, the final output frequency is:

$$f_{signal} = \frac{f_{update}}{N_{LUT}} = \frac{f_{clk}}{N_{LUT} \times (ARR + 1)} \tag{2}$$

By solving Equation (2) for the ARR value, we obtain the formula used by the firmware to configure the timer:

$$ARR = \left(\frac{f_{clk}}{f_{signal} \times N_{LUT}} \right) - 1 \tag{3}$$

The firmware implements this calculation, then sets the ARR.

Listing 6: changeFreq function

```
void changeFreq(uint16_t sine_freq)
{
        if (sine_freq > 2 && sine_freq <= 200){

                uint16_t new_ARR = (uint16_t) ((MCU_TIMER_CLOCK_FREQ / ((
                    ↪ uint32_t)sine_freq*LUT_SAMPLES)) - 1);
                __HAL_TIM_SET_AUTORELOAD(&htim8, new_ARR);
                htim8.Instance->EGR = TIM_EGR_UG;
        }

}
```

The system **main functionality**, simultaneous signal generation and acquiring, is handled by two crucial routines:

- `stepResponse`: Prepares the system for a one-shot measurement. It discharges the circuit capacitor by grounding it for half a second, then starts the ADC in linear mode, starts the timer, and finally sets the GPIO pin high to generate the step stimulus.

Listing 7: The Step Response launch routine

```
void stepResponse(void)
{
    current_state = MODE_STEP_RUNNING;

    setPinMode(DAC_GPIO_Port, DAC_Pin, GPIO_MODE_INPUT);
    setPinMode(STEP_GPIO_Port, STEP_Pin, GPIO_MODE_OUTPUT_PP);

    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_RESET);
    HAL_Delay(500);

    HAL_ADC_Start_DMA(&hadc1, (uint32_t*)ADC_buffer, STEP_SAMPLES);

    HAL_TIM_Base_Start(&htim2);

    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_15, GPIO_PIN_SET);
}
```

- `waveformResponse`: Configures the system for continuous streaming. It starts the DAC (circular DMA) and the ADC (circular DMA) simultaneously via `TIM8`, ensuring synchronization between stimulus and acquisition.

Listing 8: The Waveform launch routine

```
    void waveformResponse(void)
{
        current_state = MODE_FREQ_RUNNING;

        setPinMode(STEP_GPIO_Port, STEP_Pin, GPIO_MODE_INPUT);

        setPinMode(DAC_GPIO_Port, DAC_Pin, GPIO_MODE_ANALOG);

        HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t*)
            ↪ current_waveform_ptr, LUT_SAMPLES, DAC_ALIGN_12B_R);

        HAL_ADC_Start_DMA(&hadc2, (uint32_t*)ADC_buffer,
            ↪ TOTAL_BUFFER_SIZE);

        HAL_TIM_Base_Start(&htim8);
}
```

Note how both `stepResponse` and `waveformResponse` implement a **safety feature**: when one signal source is active, the other pin is effectively disconnected from the circuit by placing it in a high-impedance state (Hi-Z). Since both the digital GPIO pin (used for the step stimulus) and the analog DAC output (used for frequency analysis) are physically coupled to the same input node, without this precaution, if one pin were to drive a logic high level while the other drove a logic low level, a low-resistance path would form between the supply voltage and ground, causing damaging currents to flow through the microcontroller's output drivers.

3.4 Interrupt Handlers

The system relies on **three key Interrupt Service Routines** (ISRs) to maintain real-time operation.

- `HAL_UART_RxCpltCallback`: this callback parses incoming ASCII characters. It accumulates bytes until a newline character is detected. The command string is then analyzed (e.g., 'S' for Step, 'F' for Sine Waveform, 'T' for Sawtooth Waveform, 'A' for Abort) and the appropriate routine is triggered. The code also features a buffer overflow protection.

Listing 9: UART Interrupt Handler for command parsing

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
        if(huart->Instance == USART2){

                if (rx_byte == '\n' || rx_byte == '\r'){
                        rx_buffer[rx_index] = '\0';

                        if (rx_buffer[0] == 'S'){
                                current_state = MODE_STEP_LAUNCH;
                        }

                        if (rx_buffer[0] == 'F'){
                                int freq_val = atoi(&rx_buffer[1]);
                                changeFreq(freq_val);
                                current_waveform_ptr = DAC_Sine;
```

```
                                current_state = MODE_WAVE_LAUNCH;
                        }
                        if(rx_buffer[0] == 'T'){
                                int freq_val = atoi(&rx_buffer[1]);
                                changeFreq(freq_val);
                                current_waveform_ptr = DAC_Saw;
                                current_state = MODE_WAVE_LAUNCH;
                        }

                        if (rx_buffer[0] == 'A'){

                                current_state = MODE_IDLE;
                                HAL_TIM_Base_Stop(&htim2);
                                HAL_TIM_Base_Stop(&htim8);
                                HAL_ADC_Stop_DMA(&hadc1);
                                setPinMode(STEP_GPIO_Port, STEP_Pin,
                                    ↪ GPIO_MODE_INPUT);
                                HAL_ADC_Stop_DMA(&hadc2);
                                HAL_DAC_Stop_DMA(&hdac, DAC_CHANNEL_1);
                                setPinMode(DAC_GPIO_Port, DAC_Pin,
                                    ↪ GPIO_MODE_INPUT);

                        }
                        rx_index = 0;
                }
                else{
                        if (rx_index < RX_BUFFER_SIZE){
                                rx_buffer[rx_index] = rx_byte;
                                rx_index++;
                        }
                }
                HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
        }
}
```

- **ADC Conversion (`ConvHalfCplt` and `ConvCplt`):** these callbacks manage the double-buffering (Ping-Pong) mechanism for continuous streaming. In particular, `HAL_ADC_ConvHalfCpltCallback` is triggered when the first half of the circular buffer is full. It sends the first half of data to the PC while the ADC continues filling the second half, while `HAL_ADC_ConvCpltCallback` is triggered when the buffer is completely full, for continuous modes (Frequency response), it sends the second half of the data, instead, for One-Shot modes (Step response), it stops the timers and ADCs, as the acquisition is complete.

Listing 10: ADCs half and full complete conversion Interrupt Handler for data transmission

```
     void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
        if (current_state == MODE_IDLE) return;

    if (hadc->Instance == ADC1)
    {
        HAL_TIM_Base_Stop(&htim2);
        HAL_ADC_Stop_DMA(&hadc1);

        sendData(&ADC_buffer[0], STEP_SAMPLES, BIN_MODE_STEP);

```

```
12          current_state = MODE_IDLE;
13      }
14
15      if (hadc->Instance == ADC2)
16      {
17          sendData(&ADC_buffer[TOTAL_BUFFER_SIZE/2], TOTAL_BUFFER_SIZE/2,
                ↪ BIN_MODE_FREQ);
18      }
19  }
20
21  void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
22  {
23      if (hadc->Instance == ADC2)
24      {
25          sendData(&ADC_buffer[0], TOTAL_BUFFER_SIZE/2, BIN_MODE_FREQ);
26      }
27  }
```

4 Graphical User Interface

To provide the user with full control over the embedded hardware and to visualize the acquired signals in real-time, a custom Graphical User Interface (GUI) was developed using the **Python** programming language. The application leverages the standard `tkinter` library for window management and widgets, integrated with `matplotlib` for scientific plotting.

The software architecture is designed to handle high-speed data streams without compromising the responsiveness of the interface, employing a multi-threaded approach.

4.1 Software Architecture and Threading

A critical challenge in real-time data acquisition is preventing the User Interface (UI) from freezing while waiting for incoming data. To address this, the application adopts a **Producer-Consumer** model implemented via multi-threading:

- **Main Thread (GUI):** Handles the `tkinter` event loop, user interactions (button clicks, inputs), and graphical updates. It acts as the consumer, reading data from memory to update the plot.
- **Background Thread (Serial Listener):** A dedicated *daemon thread* handles the communication with the microcontroller. It acts as the producer, continuously listening to the serial port, parsing binary packets, and storing valid measurements into a shared memory buffer.

This decoupling ensures that the oscilloscope remains responsive to commands even when processing high-bandwidth data streams.

4.2 Serial Communication Interface

The communication between the Host PC and the STM32 microcontroller is UART based, bidirectional but asymmetrical:

- **Transmission**: control commands sent from the PC to the MCU are lightweight and event-driven. When the user interacts with the control panel (e.g., initiating a Step Response), on backend, the GUI sends short **ASCII strings** to the MCU, which are processed in the firmware `HAL_UART_RxCpltCallback` handler.
- **Reception**: the data stream from the MCU is high-bandwidth and continuous. To maximize throughput, the application implements a binary protocol parser. The listener thread scans the incoming byte stream for a specific 2-byte Synchronization Header (`0xAA 0xBB`). Upon detection, it reads the subsequent "Mode" byte and the fixed-size payload (2000 samples). The raw bytes are deserialized using the `struct` library into 16-bit integers, converted to floating-point voltage values ($V = ADC_{val} \times \frac{3.3}{4095}$), and pushed into the data buffer for subsequent visualization.

4.3 Data Management and Real-Time Plotting

Displaying thousands of points at 20 frames per second is computationally expensive. Some optimizations are required to achieve fluid visualization:

- **Circular Buffering:** A `collections.deque` with a fixed maximum length (10,000 points) is used as the primary data structure. This acts as a Rolling Buffer (FIFO): when new data arrives, old data is automatically ejected, creating a seamless **sliding window effect** without manual memory management.

- **Optimized Rendering:** Instead of clearing and redrawing the entire figure at every frame (which would cause flickering and high CPU usage), the application uses the `set_data()` method of the `matplotlib` line object. The graph axes and grid are drawn only once at initialization. Subsequently, only the XY coordinate arrays of the signal trace are updated.
- **Timer-based Animation:** The **graphical update is decoupled from the data reception**. A periodic timer triggers the refresh routine every 50 ms (20 FPS), ensuring the GUI is not overwhelmed by the potentially faster incoming serial data rate.

4.4 Data Logging

The GUI includes a **recording feature** that allows users to save acquisitions for analysis and post-processing. When recording is active, the incoming data chunks are not only displayed but also written to a CSV (Comma-Separated Values) file. To minimize I/O latency, the software buffers the converted voltage values and writes them to the disk in bulk operations, preventing the file system access times from slowing down the serial listener thread. Each sample is timestamped based on the theoretical sampling interval derived from the selected signal frequency.

5 Experimental Results

To validate the system's performance and the reliability of the acquisition pipeline, a hardware test bench was developed. The objective was to compare the theoretical behavior of a known physical system with the real-time measurements acquired by the STM32-based oscilloscope.

5.1 Test Circuit Description

To ensure the safety of the ADC inputs and to evaluate the system's response under specific impedance conditions, a modified RC Low-Pass Filter was used as the Device Under Test (DUT). The circuit topology includes a voltage divider configuration:

- A series resistor $R_s = 10k\Omega$ connected to the signal source (V_{in}).
- A parallel resistor $R_p = 10k\Omega$ connected across the capacitor.
- A capacitor $C = 10\mu F$ connected to ground.

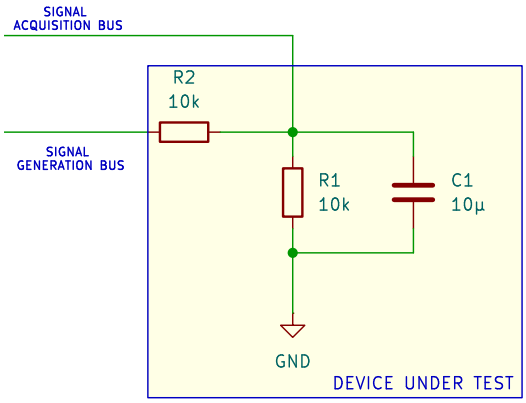

Figure 2: Test circuit schematics

This configuration alters the theoretical dynamics compared to a standard series RC circuit. The equivalent resistance seen by the capacitor is the parallel combination of the two resistors:

$$R_{eq} = R_s \parallel R_p = \frac{R_s \cdot R_p}{R_s + R_p} = 5k\Omega \tag{4}$$

Consequently, the expected time constant τ is:

$$\tau = R_{eq} \cdot C = 5k\Omega \cdot 10\mu F = 0.05s \tag{5}$$

This yields a theoretical cutoff frequency of $f_c = \frac{1}{2\pi\tau} \approx 3.18$ Hz.

Additionally, the steady-state voltage is scaled by the divider ratio:

$$V_{max} = V_{in} \cdot \frac{R_p}{R_s + R_p} = 3.3V \cdot 0.5 = 1.65V \tag{6}$$

5.2 Results Analysis

The following figures present the data collected via the Python GUI. The experiments involved stimulating the circuit with Step, Sine, and Sawtooth signals.

Step Response

The system was stimulated with a voltage step (0V to 3.3V). Figure 3 shows the acquired transient. Two key observations confirm the validity of the model:

- **Exponential Growth:** The curve follows the characteristic charging equation of a capacitor.
- **Final Voltage Value:** As predicted by the voltage divider equation (Eq. 3), the signal stabilizes at approximately **1.65V** rather than the full 3.3V input. This confirms the correct scaling effect of the resistive network.

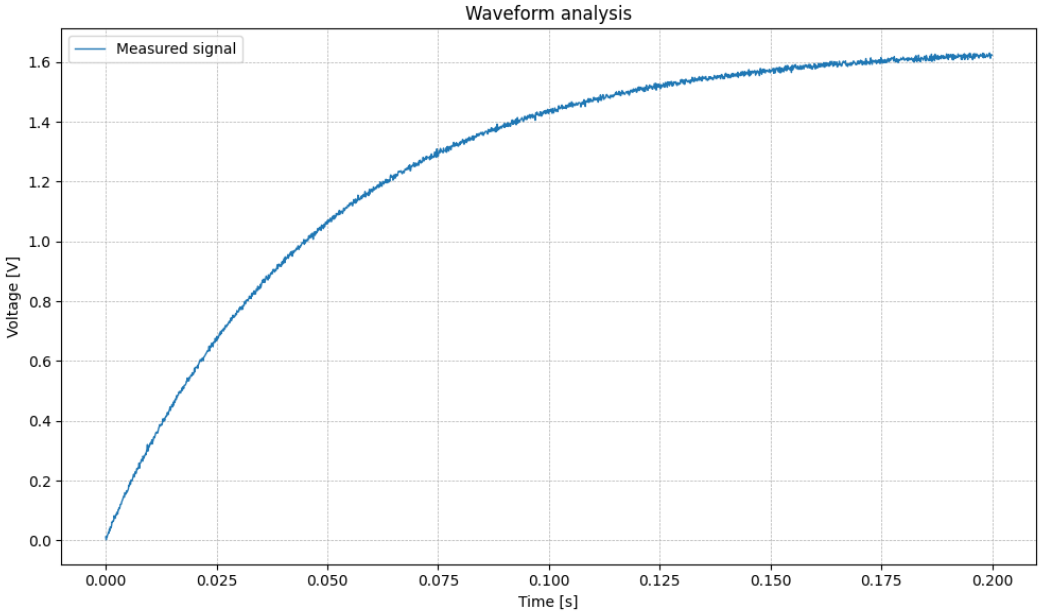

Figure 3: Step Response acquisition. The voltage rises exponentially and settles at 1.65V due to the voltage divider effect ($R_s = R_p = 10k\Omega$).

Frequency Response (Sinusoidal Input)

To test the dynamic behavior, a sinusoidal signal was injected into the DUT. Figure 4 illustrates the system response to two different frequencies applied in sequence: first 3 Hz, followed by 30 Hz.

- **Low Frequency (3 Hz):** In the first part of the plot ($t < 4.5s$), the signal frequency is close to the circuit's cutoff frequency ($f_c \approx 3.18$ Hz). The amplitude is significant, as the filter allows the signal to pass with minimal attenuation (approx. -3dB point).
- **High Frequency (30 Hz):** In the second part ($t > 4.5s$), the frequency is increased to 30 Hz. As expected from a first-order low-pass filter, the amplitude is drastically attenuated. This visual evidence aligns with the theoretical harmonic response, where the gain decreases by 20dB/decade beyond the cutoff frequency.

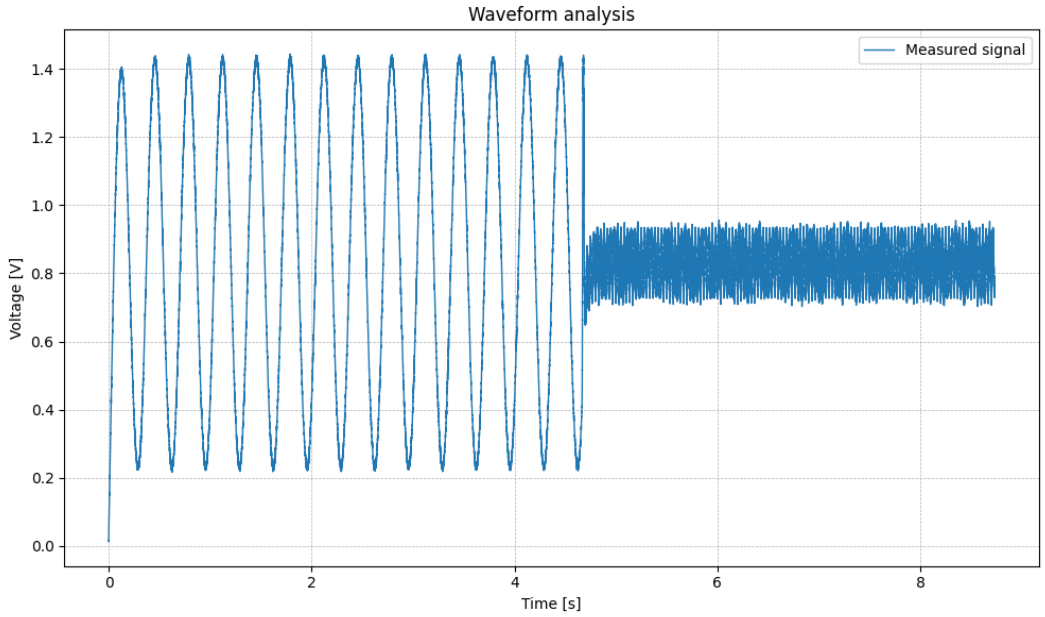

Figure 4: Harmonic response. The DUT is driven first at 3 Hz (pass-band/cutoff region), then at 30 Hz (stop-band). The attenuation at higher frequency is evident and consistent with the Low-Pass filter theory.

Sawtooth response (Sawtooth Input)

A similar test was performed using a Sawtooth waveform, as shown in Figure 5. At 3 Hz, the ramp is clearly visible and maintains good linearity. When the frequency switches to 30 Hz, the waveform loses its triangular shape and amplitude due to the filtering of higher-order harmonics, further confirming the bandwidth limitations imposed by the RC circuit.

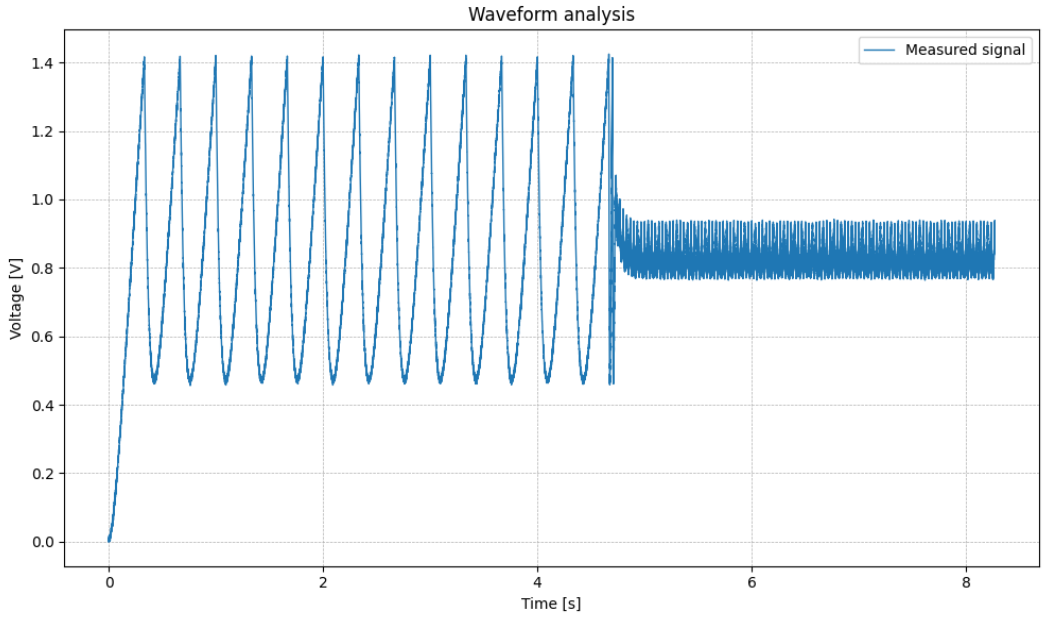

Figure 5: Sawtooth response (3 Hz vs 30 Hz). At lower frequencies, the linearity is preserved. at higher frequencies, the signal is heavily filtered, resulting in reduced amplitude and shape distortion.

6 Conclusions

This project successfully demonstrated the design and implementation of a cost-effective embedded system capable of performing simultaneous signal generation and data acquisition. By leveraging the high-performance peripherals of the STM32F446RE microcontroller—specifically the synchronization between DAC, ADC, and hardware timers via DMA—the system effectively replicates the core functionalities of professional benchtop equipment, such as oscilloscopes and function generators, at a fraction of the cost.

The reliability of the proposed architecture was rigorously verified through experimental testing on a passive RC circuit. The results obtained from the tests confirmed the accuracy of the device.

Finally, a significant advantage of this software-defined approach is its inherent flexibility. Since the signal generation logic relies on RAM-based Lookup Tables (LUTs) populated at startup, the system is **highly extensible**. Adding new waveform types—such as triangular waves, square waves, or complex arbitrary signals—does not require any hardware modification. New signal profiles can be implemented simply by defining the corresponding mathematical arrays in the firmware initialization routine, making this platform a versatile and adaptable tool for educational laboratories and signal processing applications.