Design Document

Logic Analyzer/Bus Sniffer

Team Members:

Furat Alhafez

Sultan Alghamdi

Sam Nepal

Julian Nigg

Thomas Senai

Shahroz Shahbaz

Date of Submission: 12/06/2024

Faculty Advisor: Dr. Jens-Peter Kaps

Associate Faculty: Dr. Craig Lorie

Course Coordinator: Dr. Tolga Soyota

1. Problem Statement

1.1 Motivation and Identification of Need

The logic analyzers available on the market today are either too expensive to be considered affordable or very cheap, but not very efficient. The limitations of the current practice are the cost of the materials. Our approach is focusing our model on an affordable developmental board to keep costs low and to program all the needed features. It will be able to perform similar tasks to the ADALM2000. This simplified logic analyzer is targeted for college and university students to help them understand the fundamentals of digital circuits. The logic analyzer can help school and university student have hands on experience on the logic analyzer, which will help them understand the basics of circuit analysis without any worries or shorting the circuit and destroying the board as they are inexpensive. This will significantly reduce the financial burden on the students as this project is aimed at producing a logic analyzer quarter the price of logic analyzer available in the market. Additionally, with high-speed data transmission port and high clock speed this logic analyzer can pick the signal with greater accuracy and precision.

1.2 Market Review

The high-end costs for logic analyzers are the ADLAM2000, Analog Discovery 2 (AD2) and Saleae Logic 8, which cost around \$236, \$299, and \$499 respectively. These have many important functionalities, but the cost is not realistic for many students. There also exists a logic analyzer that costs as cheap as \$20 such as the SparkFun USB Logic Analyzer. The issue with cheaper logic analyzers is that many of them do not provide all the necessary functionalities needed for Computer and Electrical engineering students.

Model	Saleae Logic 8	Analog Discovery 2 (AD2)	Advanced Active Learning Module (ADALM 2000)	Sparkfun USB Logic Analyzer
Device Picture				
Power connection	USB Type 2.0	USB Type 2.0	USB Type 2.0	USB Type C
Number of Digital Channels	8	16	16	8
Maximum Sampling Rate (MS/s)	100	100	100	24
Supported Logic Levels (V)	1.8 - 5.5	1.8 - 5.0	0.0 - 5.0	2.0 - 5.25
Software	Saleae Logic	Digilent Waveforms	ADALM Scopy	Open-source Sigork
Price	\$499.00	\$299.00	\$236.25	\$19.95

Figure 1: Comparison Table of examples of existing Logic Analyzers

2. Project Requirement Specification

2.1 Mission Requirement

The device shall offer an affordable solution for analyzing digital signals. It shall be user-friendly, capable of connecting to any PC running macOS, Windows, or Linux through a USB connection, and display a graphical user interface on the PC.

2.2 Operation Requirements

Input/output requirements:

- This device **shall** have 8 channels that can accept input signals.
- The device **shall** support 3.3V and 5.0V logic.

PC Interface requirement:

- The device **shall** use USB to communicate with the user's device.
- The device **shall** be powered using USB.

Functional Specification:

- The device **shall** sample at a maximum rate of 5 MHz and display the rate on the user's device.
- The device **shall** be compatible to run on Windows, Linux, and macOS.
- This device **shall** display digital signal in the form of squares waves.
- This device **shall** decode input I2C signal and SPI.
- This device **should** decode asynchronous UART signals.
- The device **shall** adjust the sampling rate and adjust buffer size through the GUI upon the user's input.

3. System Decomposition & Architecture

3.1 Level Zero Decomposition

In this level Zero Design, it describes the essence of a logic Analyzer. It takes in logic signals that might be potentially encoded in one of the various communication protocols (I2C, SPI, etc.) or already decoded logic signals, samples them, and presents them on a plot. It uses the user's help to decipher what communication protocol the signal is using so that it may plot a more accurate description of the logic signal being sampled. It also presents a user interface and setting options to help the user to easily provide the necessary information needed for decoding the logic signals.

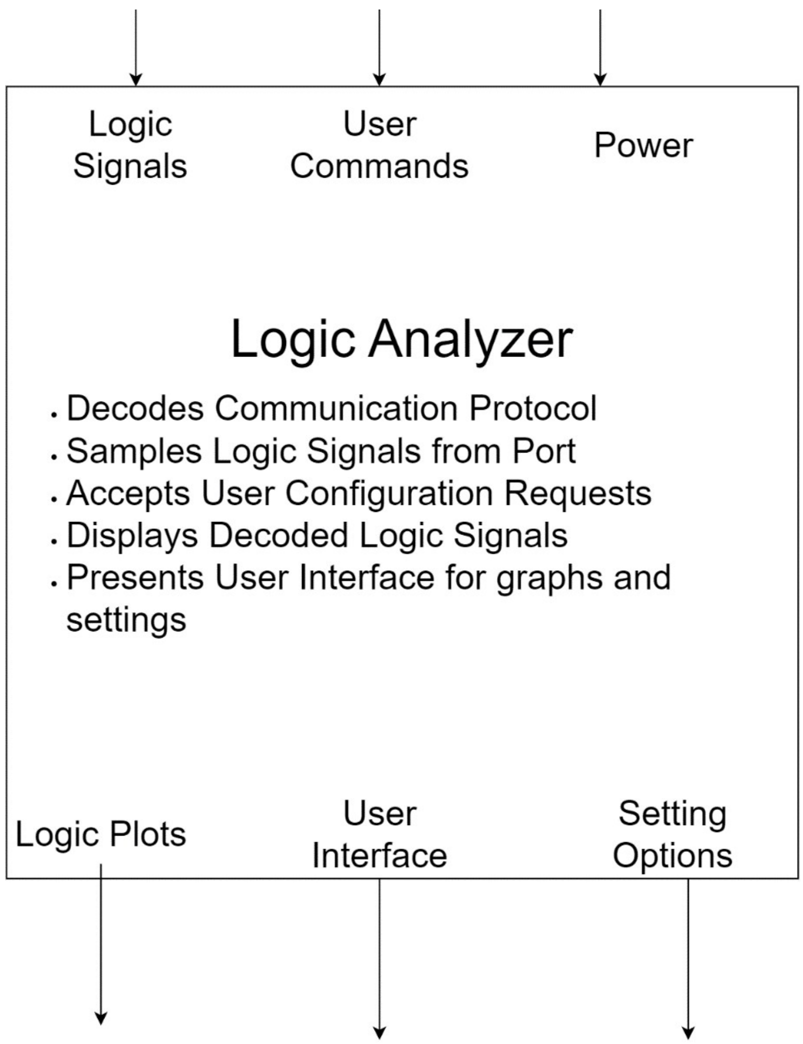


Figure 2: Level 0 Design

3.2 Level One decomposition

In level one design, it delves deeper into the plan of building a logic analyzer. A 16 pin I/O port is used to offer the user an abundance of pins to use with the ability to plot all 16 logic signals coming from those pins. The logic signals pass through the Connection Interface (CI), ensuring that the voltage protection built into the CI will prevent damaging the vital components of the Microcontroller (MCU). The MCU takes in User Commands that were given from the software running on the desktop computer to sample the logic signals at the right time with the correct frequency. The sampled data is sent back to the computer where it is decoded if necessary and displayed on a plot for the user to analyze.

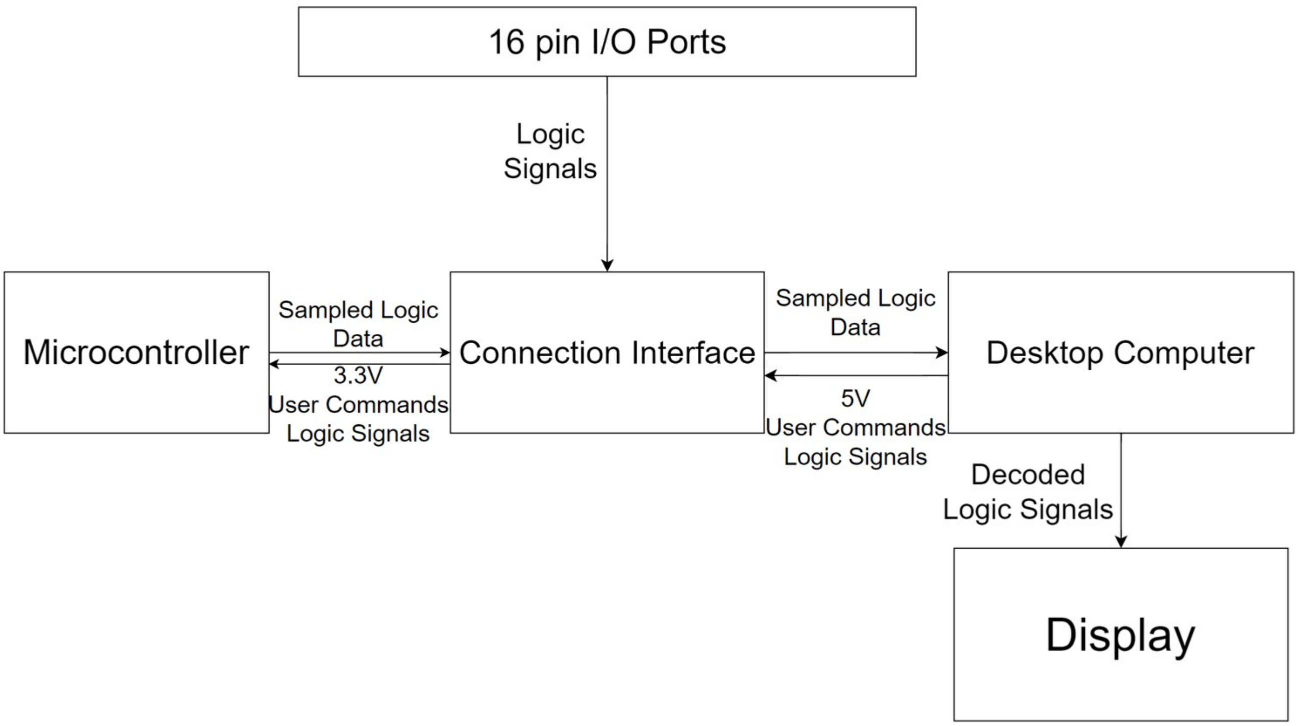


Figure 2: Level 1

3.3 Level Two Decomposition

The Level Two decomposition goes deeper into each individual element that is needed to build a functional Logic Analyzer. At the left side of the assembly we place our STM32 board, having the MCU(Microcontroller) on top of it, which will be configured to meet the requirements of our task. In the center, our PCB acts as the crucial connection point which links our MCU to PC. This board will also have a mounting spot for our Nucleo board. The PCB design we have is to protect our setup from any spikes or over voltages that might be directed to the MCU. On the right, the user's PC will be used to visualize the logic signals such as I2C, SPI, or UART. This arrangement enables the user to both view the data and issue commands to the MCU through a USB cable.

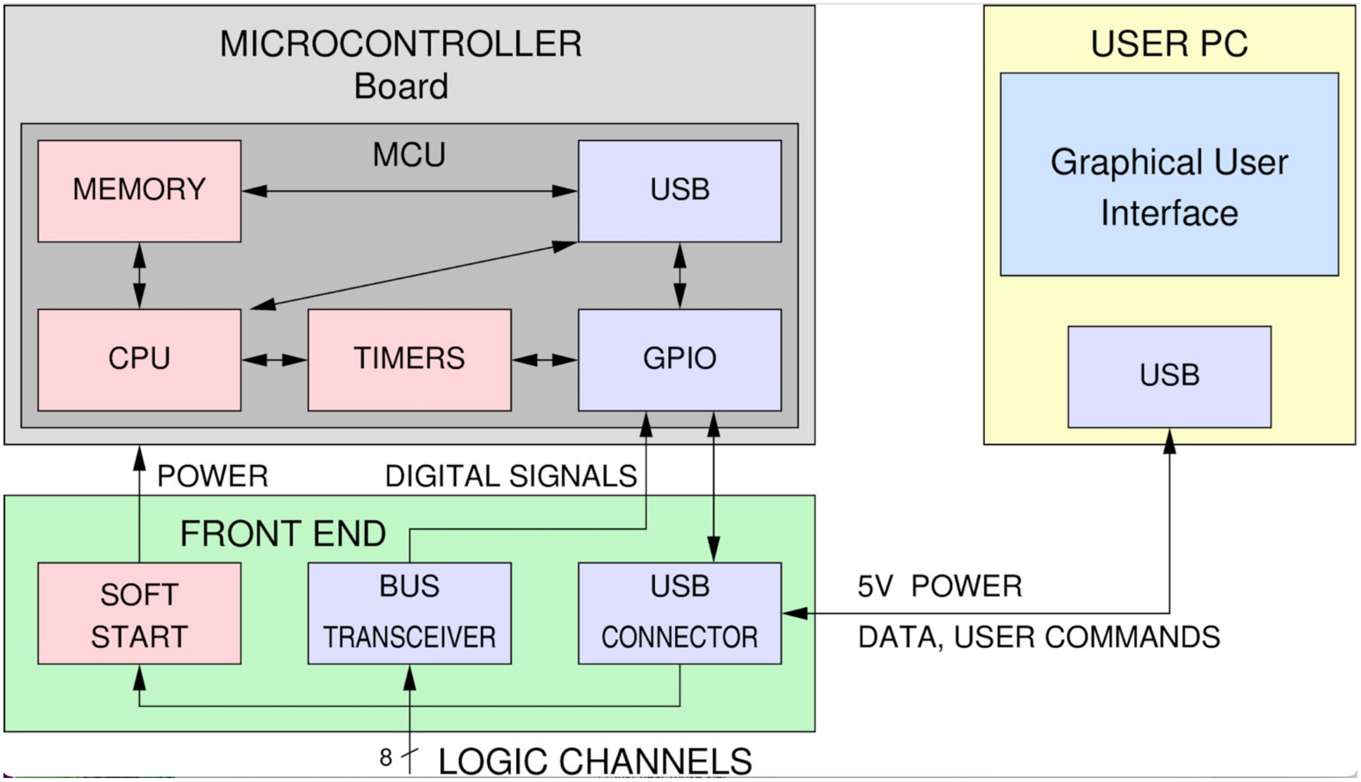


Figure 5: Level 2

4. Background Knowledge

4.1 Microcontroller (MCU)

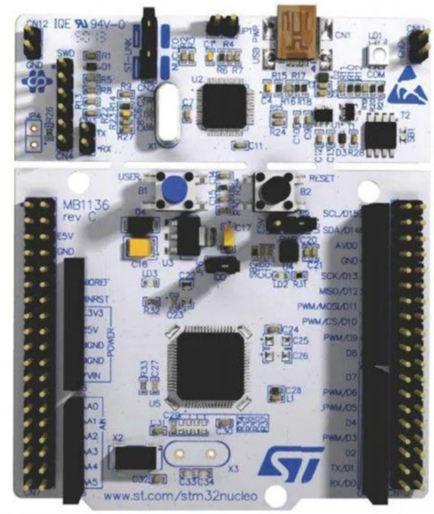


Figure 3: STM32-Nucleo-F303RE

A microcontroller (MCU) is a tiny integrated circuit that can be used in embedded systems to control operations without the need for a sophisticated operating system. Microcontrollers are customized tiny computers designed for specific tasks. Numerous devices, such as cars, robots, medical equipment, and household appliances, contain them.

A microcontroller's main parts are its memory, which stores data and programs, its CPU, which carries out instructions, and its input/output (I/O) interfaces, which allow it to interact with other devices. The CPU manages logic, I/O, and computations. Temporary data and long-term program code are both kept in memory. Peripherals for input/output facilitate communication with external components. The capabilities of the microcontroller are further enhanced by additional features like buses, serial ports, and analog-digital converters.
Microcontrollers can interface with sensors and effectively carry out specific tasks within embedded systems thanks to a variety of processor architectures, memory types, and programming languages like C, Python, and JavaScript. They are perfect for controlling individual functions in a variety of applications due to their dedicated, compact design.

4.1.1 STM32CubeIDE

STM32CubeIDE is an all-in-one multi-OS development tool, which is part of the STM32Cube software ecosystem. STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. It is based on the Eclipse/CDT framework and GCC toolchain for the development, and GDB for the debugging.

4.1.2 HAL Drivers

The Hardware Abstraction Layer (HAL) driver layer provides a simple, generic multi-instance set of APIs (application programming interfaces) to interact with the upper layer (application, libraries, and stacks). The HAL driver APIs are split into two categories: generic APIs, which provide common and generic functions for all the STM32 series and extension APIs, which include specific and customized functions for a given line or part number. The HAL drivers include a complete set of ready-to-use APIs that simplify the user application implementation. For example, the communication peripherals contain APIs to initialize and configure the peripheral, manage data transfers in polling mode, handle interruptions or DMA, and manage communication errors. The HAL drivers are feature-oriented instead of IP- oriented. For example, the timer APIs are split into several categories following the IP functions, such as basic timer, capture, and pulse width modulation (PWM). The HAL driver layer implements run-time failure detection by checking the input values of all functions. Such dynamic checking enhances the firmware robustness. Run-time detection is also suitable for user application development and debugging.

4.2 Graphical User Interface (GUI)

The GUI development aspect of this project aims to provide a user-friendly interface that allows students to interact with the device efficiently. Our group will be using PyQt6 and PySide6, which are well known frameworks for cross-platform GUI applications, to develop both a sophisticated and straightforward interface.

a) Programming fundamentals

There are a few concepts and skills needed to create a functional GUI. The first of which is having a good grasp of the Python programming fundamentals. Python's simplicity and readability makes it an ideal language for GUI development, especially for people who are trying to do this for the first time.

b) Signal Processing Basics

A basic understanding of how digital signals work, including PWM, SPI, and 12C protocols, is needed. A good understanding of these will allow us to accurately display and interpret signals within the GUI.

c) PyQt and PySide Frameworks

We will be using the 6th version of this framework which gives us comprehensive tools for creating GUI applications in Python. These frameworks Include a variety of modules that can be used for graphical elements, event handling, and more.

d) UI Design Principles

One of the most important things about creating a GUI is making it easy for the user to traverse it. We need to have a basic understanding of interface design, including layout, color theory, and user experience (UX) best practices. The interface needs to be both aesthetically pleasing, and functional, with clear presentation of information.

e) Cross-Platform Development

Fortunately, PyQt6 allows us to create a GUI that works for the three different operating systems we care about (Windows, Linux, MacOS). We will still need to be able to test if it functions exactly the way we need it to on all platforms. Mac and Windows testing will be easier to do since we have group members with those OS. We might need to set up a virtual machine for Linux testing.

4.3 Hardware Approach

We plan to integrate the USB connector with a PCB to control the voltage input to protect the data from being corrupted when going into the microcontroller. The USB will be connected to a computer using the logic analyzer software.

5. Detailed Design

5.1 MCU Design

For the detailed design of MCU, there are three parts. First, the controller layer is the microcontroller processor configuration. The function of the MCU is required for the GUI. This function will include RAM, Timer, GPIO and USB transmits/receives. STM32CubeIDE will be used to support these functions. STM32CubeIDE is used for setup MCU configuration, and for coding part of the MCU. For the MCU hardware, this will include the Hardware abstraction layer (HAL) driver and the low layer (LL) driver. The HAL driver is used to perform functions such as GPIO, Timer, RAM, and USB. LL driver is used for adjusting or changing in register level while the MCU is running without reinitialization by STM32CubeMX.

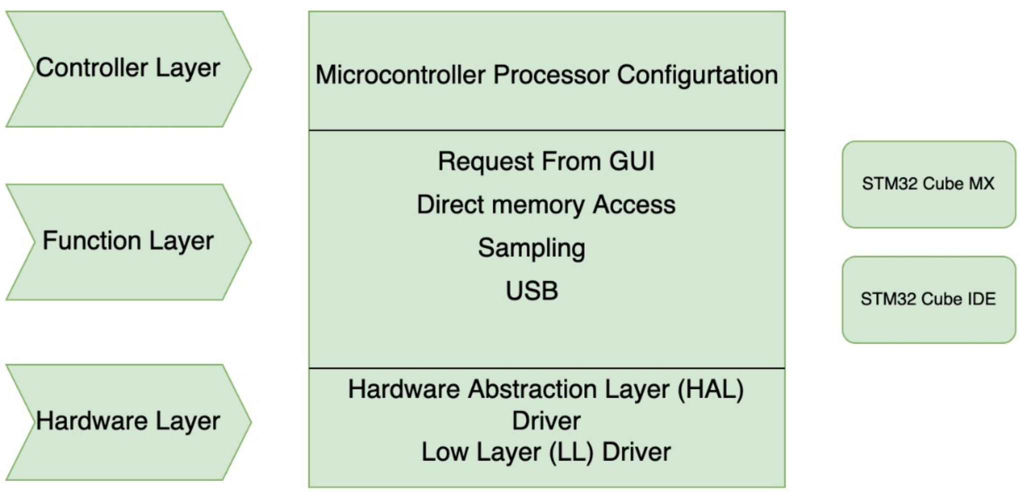


Figure 4: MCU Detailed Design

5.1.1 MCU Pinout:

This section provides the pinout view of our STM32 microcontroller. Which outlines the configuration used in our project. The diagram below shows what functionality has been assigned to each PIN.

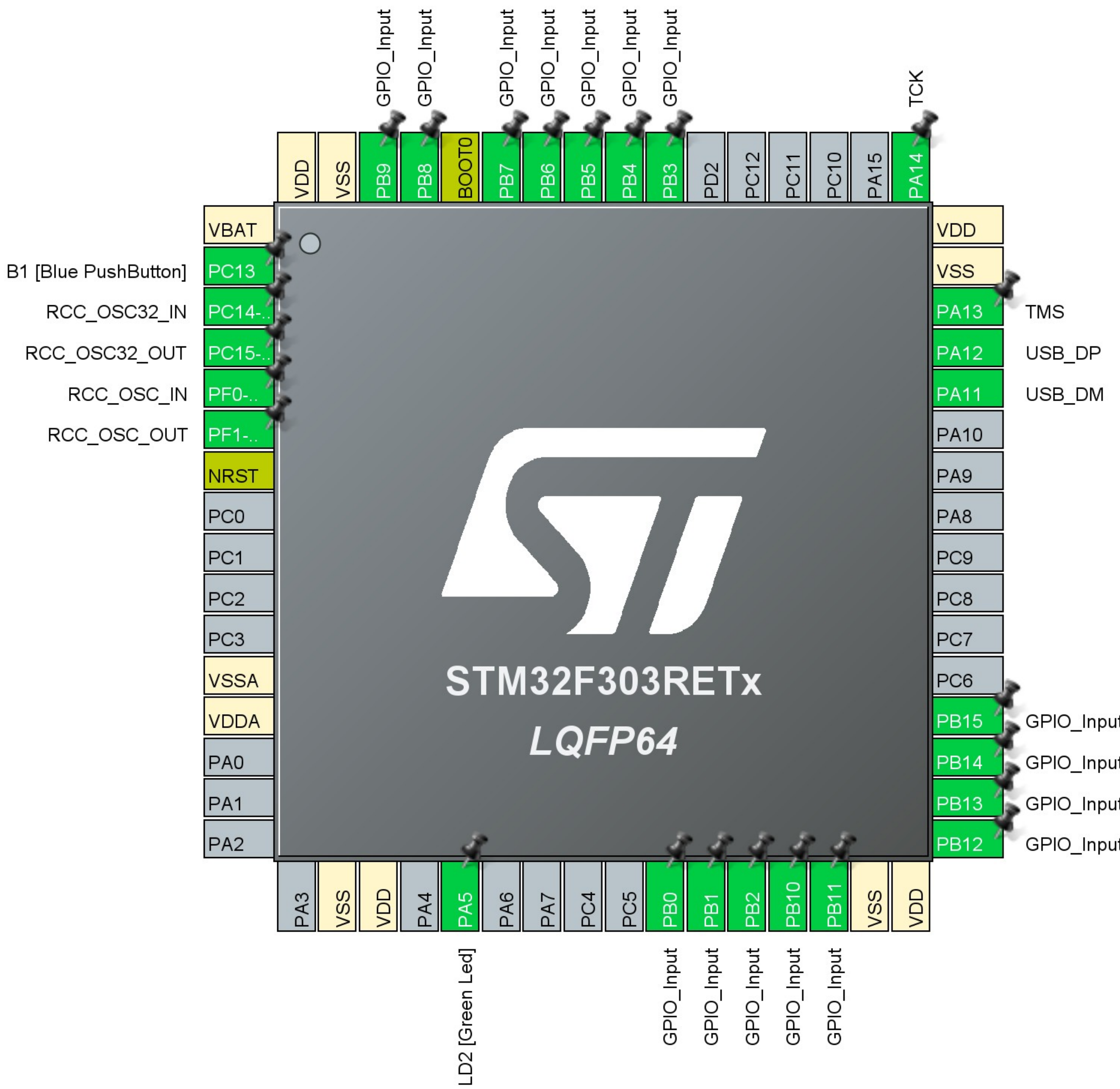


Figure 5: MCU Pinout from STM Cube IDE

- **PB0 to PB7:** In our application these pins serve as input channels for receiving the logic signals from Channel 1 to all the way Channel 8.

- **PB8 to PB15:** These pins serve as additional channels but currently not assigned to any channels. Can we integrate with other projects.
- **PA5:** This PIN serves as an indicator of our MCU is working properly and sampling the data continuously.

- **PA11 and PA12:** These pins form the USB data interface, with PA11 as USB_DM (Data

Minus) and PA12 as USB_DP (Data Plus), allowing for USB communication essential for data transfer and device control.

- **PF0 and PF1:** These pins are used for connecting an external crystal oscillator, which allows USB connectivity.

5.1.2 MCU Hardware Utilization:

The STM32 Microcontroller leverages an array of integrated hardware components to achieve efficient logic signal capturing and processing. Timer inside the MCU is responsible to sample the data so the accurate and efficient data has been captured without errors. MCU leverage RAM as buffer to store the data and transmit the data over to the GUI front. Using internal timer, based on the user specified sampling rate MCU capture logic signals and stored them into this buffer in circular motion and ensure seamless flow of data. The Use of Timer, GPIOs, RAM enables the STM32 to function as a robust and precise logic analyzer.

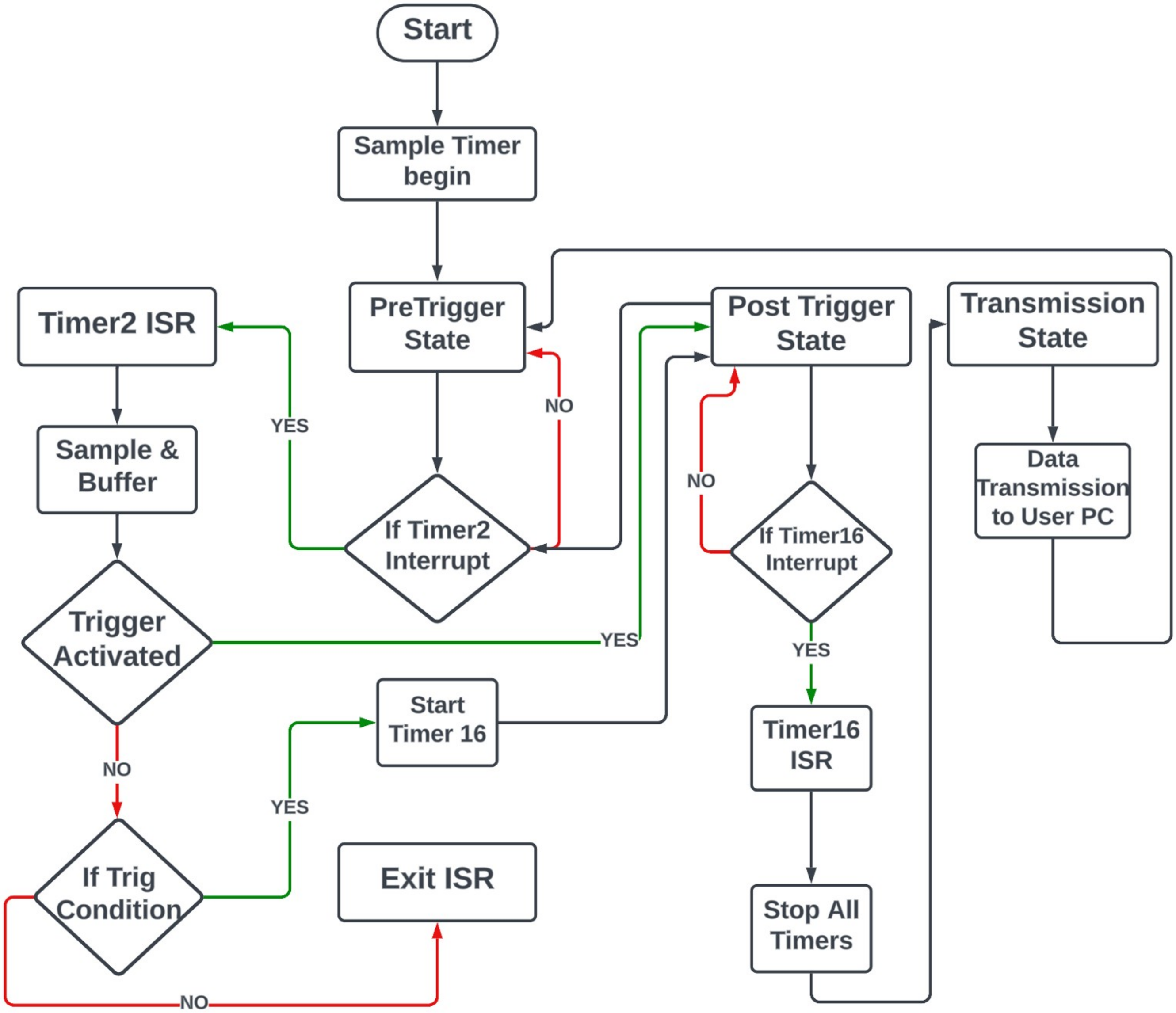


Figure 6.1: State Diagram for our MCU configuration

5.1.3 MCU Trigger Functionality:

The trigger functionality is based on user configuration on which channel has active trigger condition, on which edge (rising/falling edge), and the size of the post trigger samples for the MCU to handle. The MCU will sample the pre-trigger sample size, which is calculated by taking the buffer size and subtracting it by the post trigger size set by the user and then proceed to check for the trigger condition. If the buffer is filled before trigger condition is met, it will circle back to the start of the buffer and overwrite old pre-trigger data with new pre-trigger data. Once trigger condition is met, Timer 16 will begin, which has been adjusted to set off after a certain number of samples have been completed which is specified by the user as post trigger samples. Both timers are stopped, and the buffer is transmitted to the GUI.

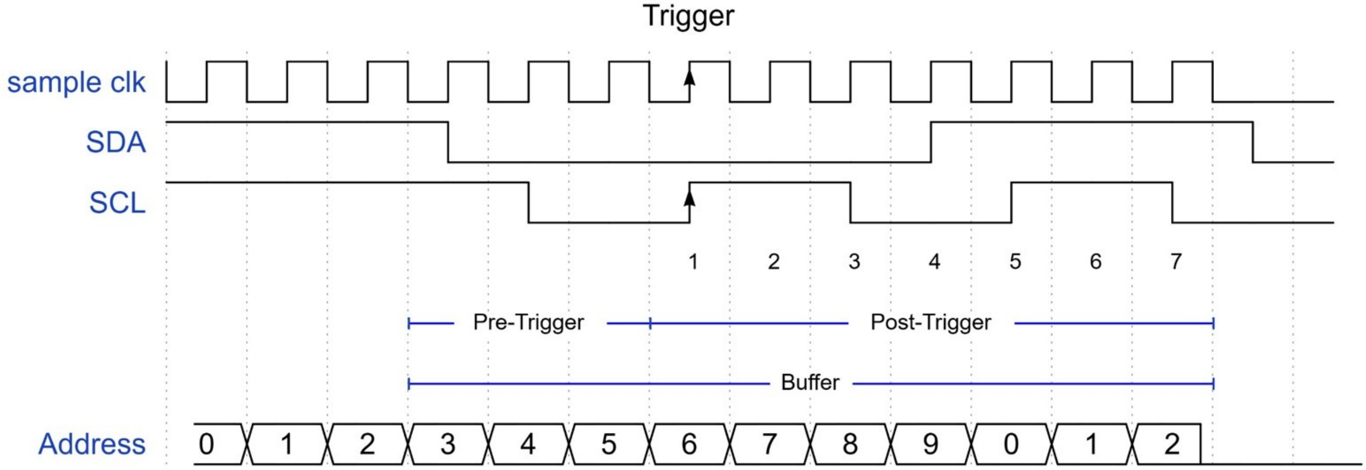


Figure 7.2: I2C Digital Wave Form

5.1.4 MCU/GUI communication:

For the GUI to send commands and configurations to the MCU, it must send three 16 bit (word) transmissions for each command. The first word (Command) is always saved in the MCU, while the second word (Value 1) is overwritten by the third word (Value 2) for the trigger edge and trigger pin commands, or the first and second word is combined such as the timer 16 period and timer 16 prescalar. Timer 2 has a period size of 32 bits, therefore, two commands are required to modify the upper 16 bits and the lower 16 bits. The MSB of trigger edge refers to PB0, and the LSB refers to PB7. The 1 represents rising edge while 0 represents falling edge. A similar layout follows the trigger pin, where 1 represents the trigger condition will be checked for that pin while a 0 represents that the trigger will not be checked on that pin.

Command Name	Command	Value 1	Value 2
Start	0x00	0xXX	0xXX
Stop	0x01	0xXX	0xXX
Trigger Edge	0x02	0xXX	0x00-0xFF
Trigger Pin	0x03	0xXX	0x00-0xFF
Timer 16 period	0x04	0x00-0xFF	0x00-0xFF
Timer 2 upper Half Period	0x05	0x00-0xFF	0x00-0xFF
Timer 2 Lower Lower Period	0x06	0x00-0xFF	0x00-0xFF
Timer 16 Prescalar	0x07	0x00-0xFF	0x00-0xFF

Figure 7.3: MCU/GUI command table

5.2 GUI Design

Upon starting the executable, the GUI will check to see if the microcontroller is detectable via USB. Once a connection has been formed, the user will be introduced to a screen where they can start using the logic analyzer. The user can start sampling the data with the default parameters or they can adjust the sampling rate, trigger conditions, or channels being displayed before sampling. While in the sampling state, the GUI will actively decode the signal and display it in a human readable form.

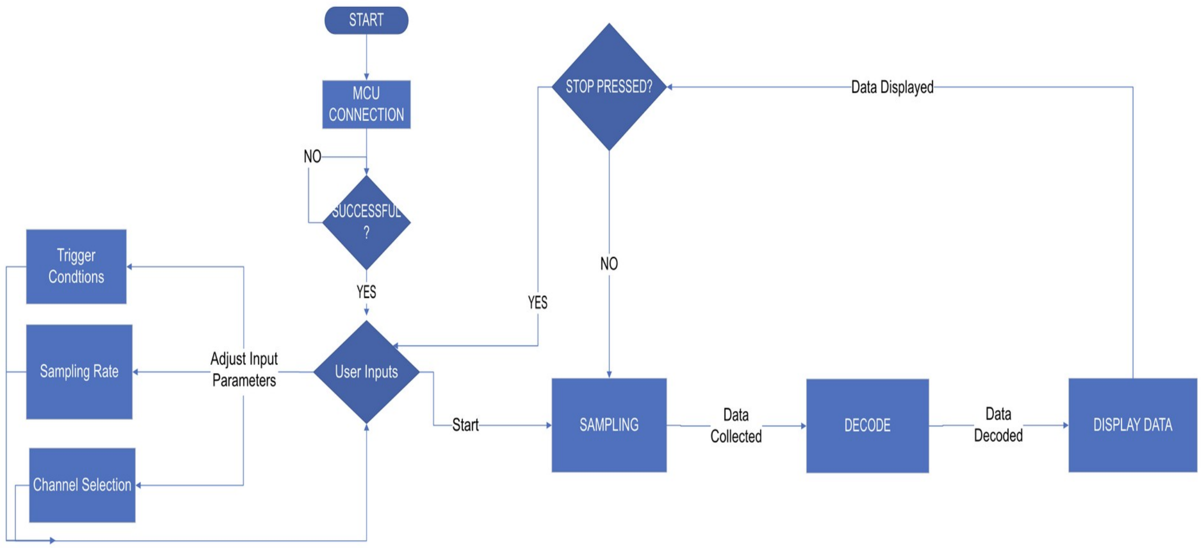


Figure 7: GUI Flow Chart

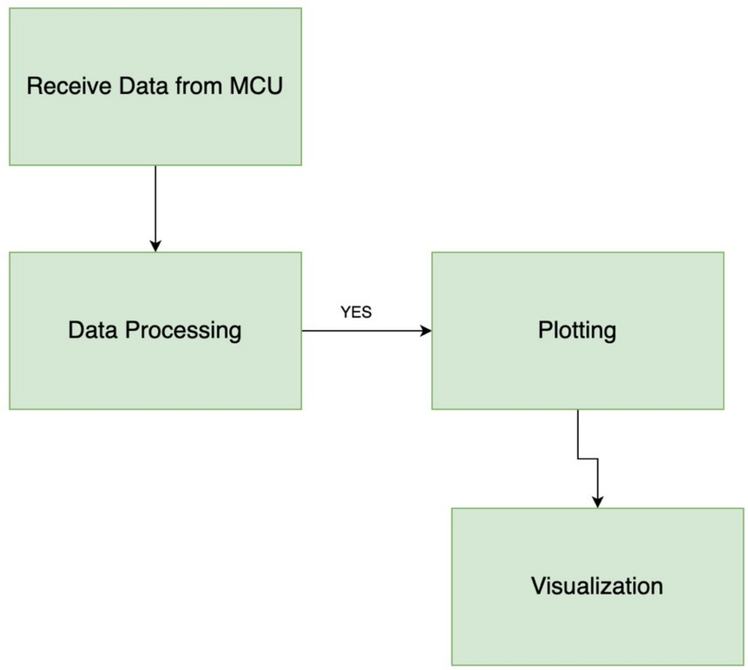


Figure 8: Detail Design of GUI

The GUI is designed in a way to allow the user to select their choice of decoding, by pressing one of the 4 buttons up top. On the right side of the graph, the user can choose which channels they want to be active on the graph and they can select what the trigger condition is going to be for the selected pin. The user also has the ability to choose a sample rate and the number of samples. The GUI has the capability to continuously sample by pressing the RUN button, additionally it performs the single capture. The configuration settings window can be reached, by right clicking the channel buttons on the right.

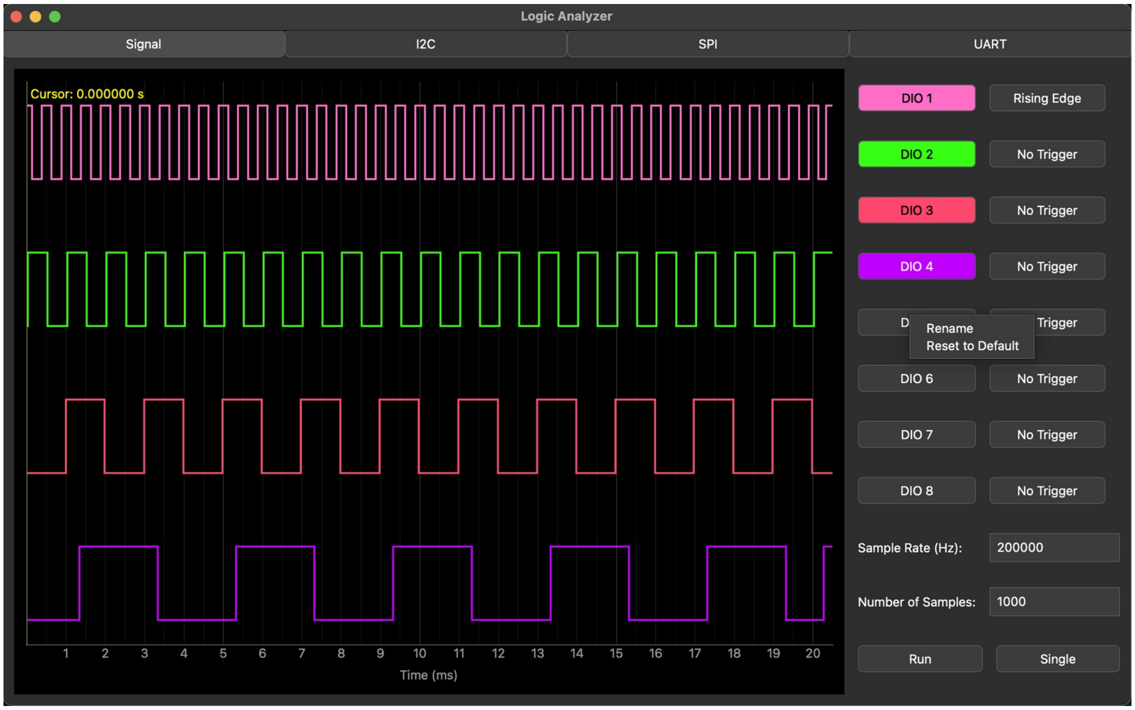


Figure 10: Singal Interface of GUI

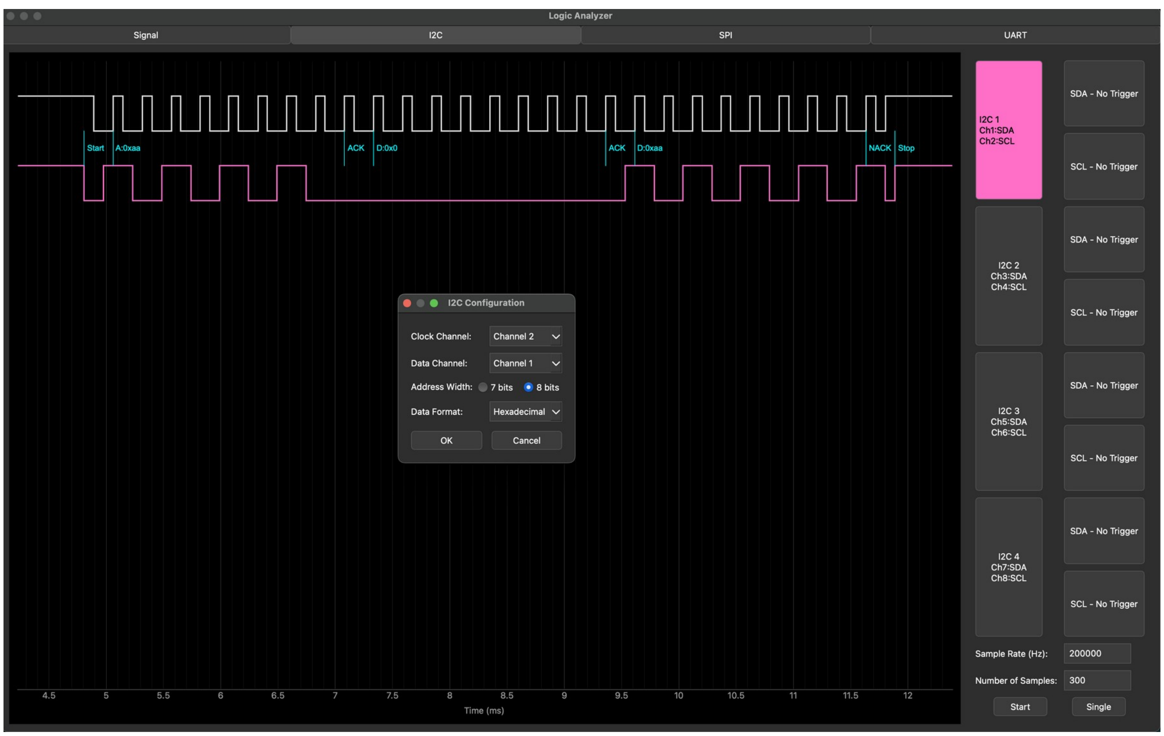


Figure 11: I2C Decoding Interface

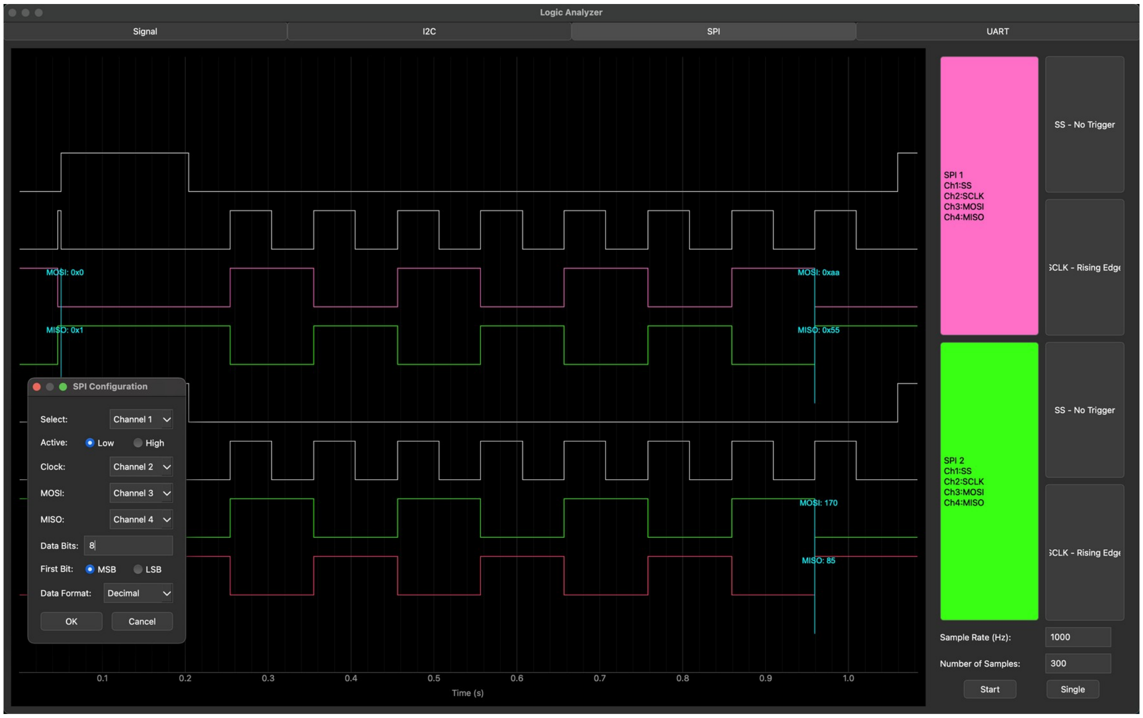


Figure 12: SPI Decoding Interface

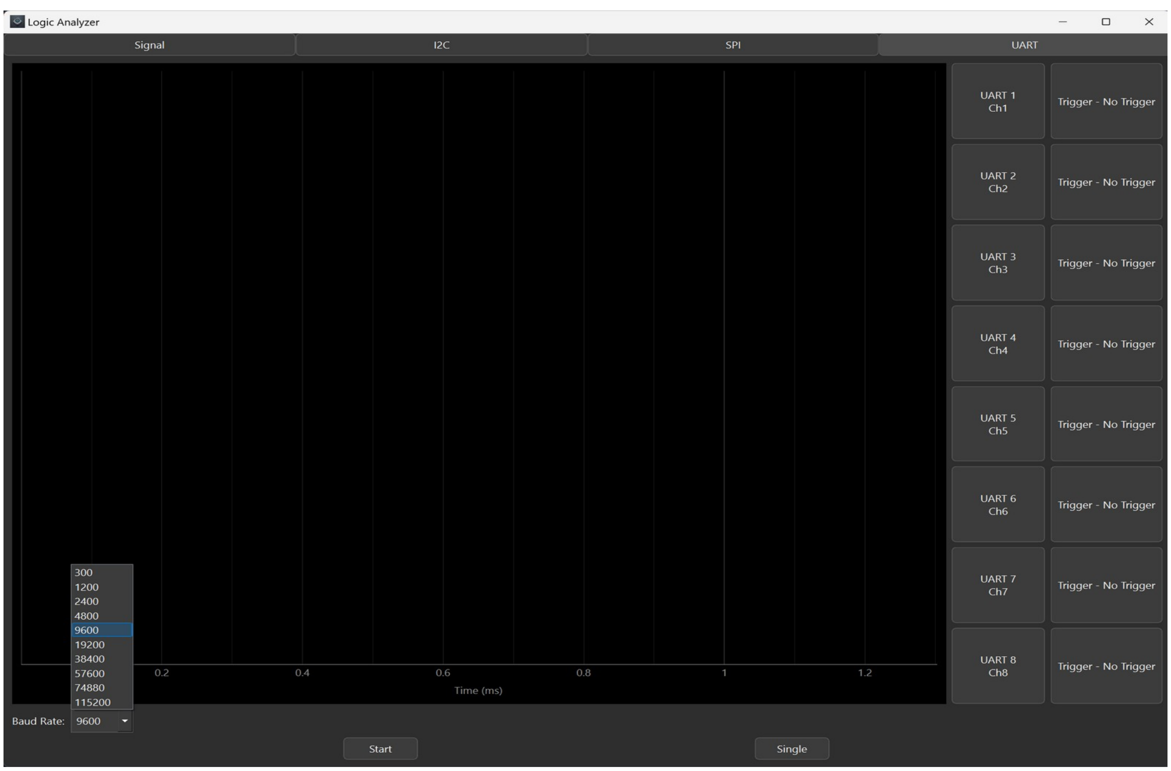


Figure 13: UART Decoding Interface

5.3 PCB/Circuit Design

5.3.1 Nucleo Board schematic

For this schematic, we have the Nucleo board with all its appropriate pins powered or grounded. Pins PB0-PB15 are the GPIO pins that take in the digital signals after being scaled down by the bus transceiver. Pins PA11 and PA12 are the Data lines + and – respectively. These pins are directly connected to the USB which in turn powers up the MCU. E5V is our 5V connection to draw power from the USB and power up the MCU. The 1.5k pull-up resistor is solely there for USB detection and the 22-ohm resistors are for data noise supression.

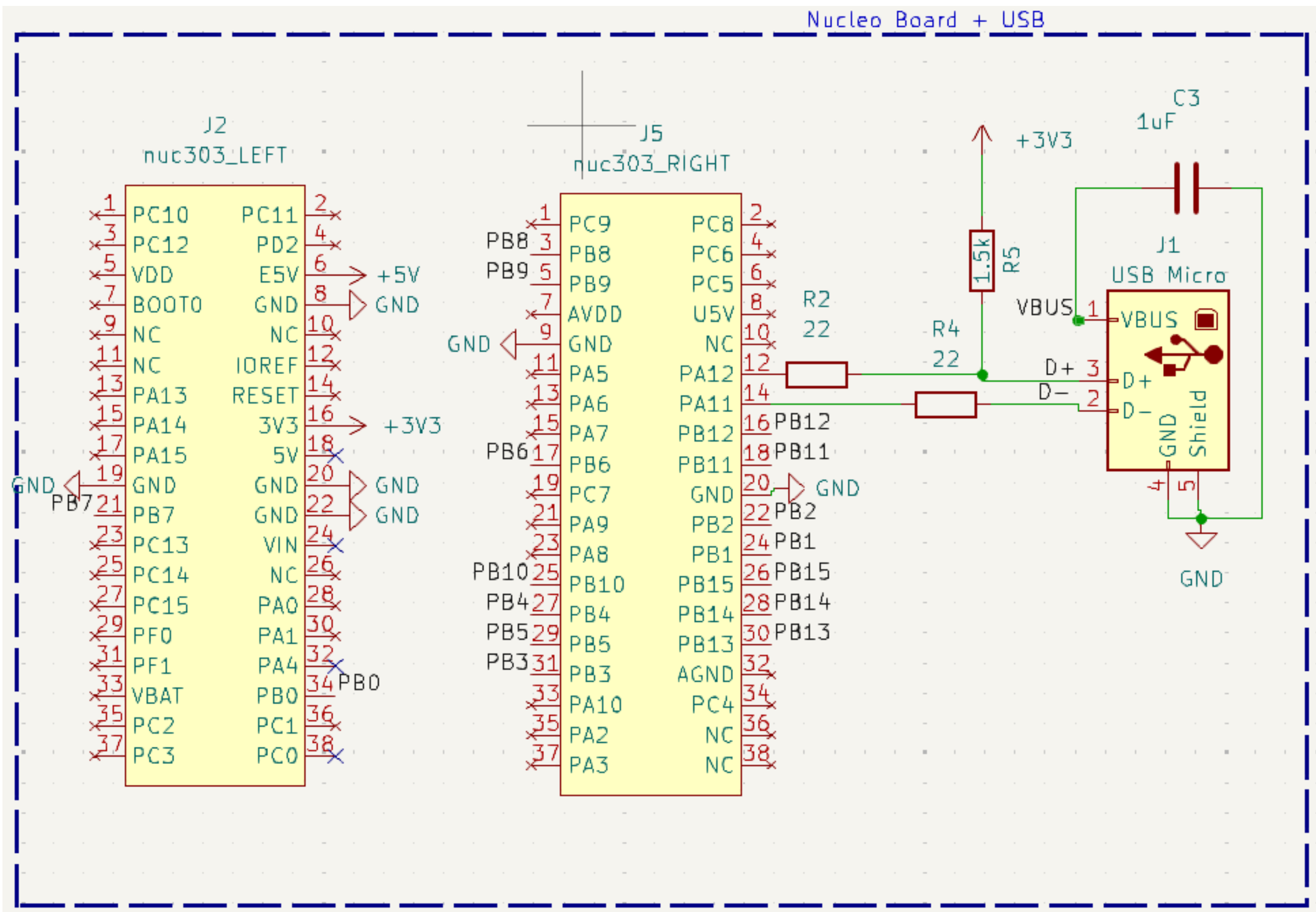


Figure 9: Microcontroller schematic.

5.3.2 Bus Transceiver Schematic

The purpose of the bus transceivers is to properly scale down the voltage without altering the quality of the digital signal. The intention for our bus transceivers is to take in 16 digital signals at side B and output those to GPIO PB0-PB15 on our Nucleo board from side A. The bus transceiver itself is powered on by 3.3V which is gathered from the MCU pin 16.

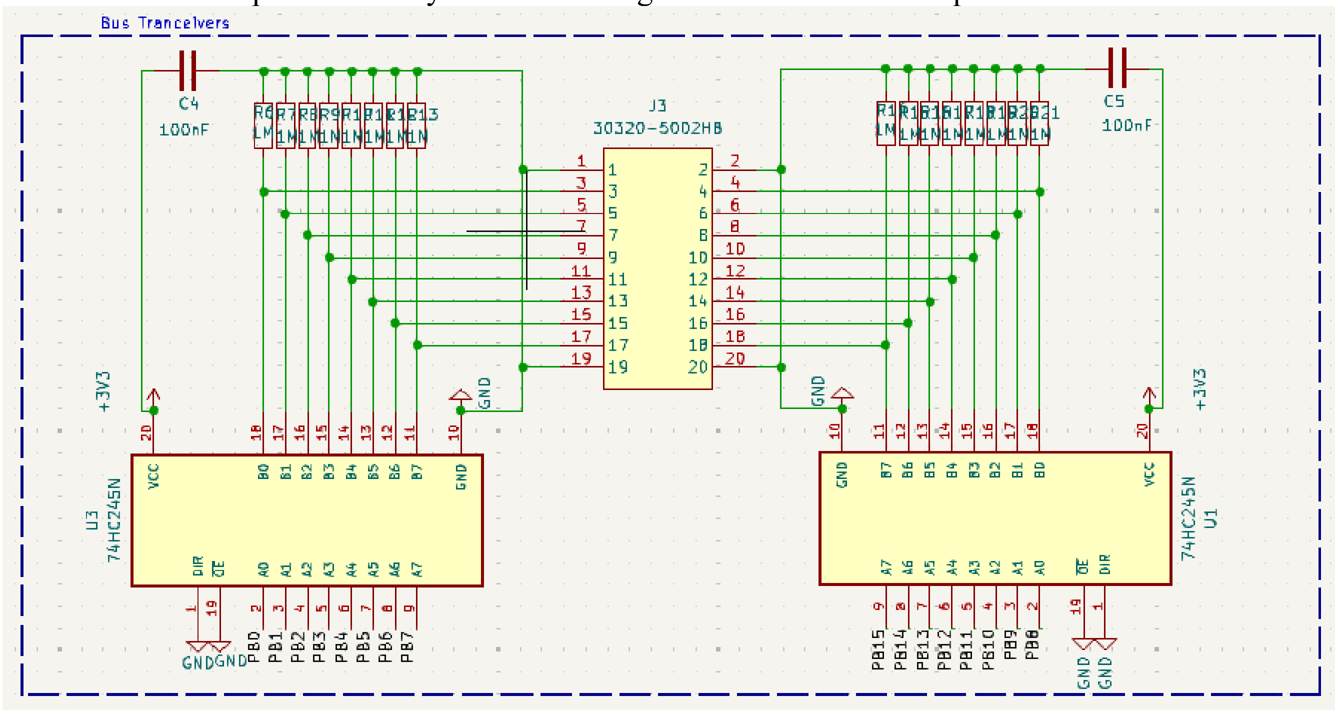


Figure 10: BUS TRANSCIVER

5.3.3 USB soft start

Soft start is a circuit designed to gradually power 'On' the system, preventing the sudden voltage surge caused during the initial startup reducing the stress on the electrical component. The soft start circuit uses an **RC** network to control the charging of a capacitor C1, which in turn regulates the MOSFET's gate voltage. This allows for a controlled and gradual increase of the output voltage supplied to the microcontroller circuit.

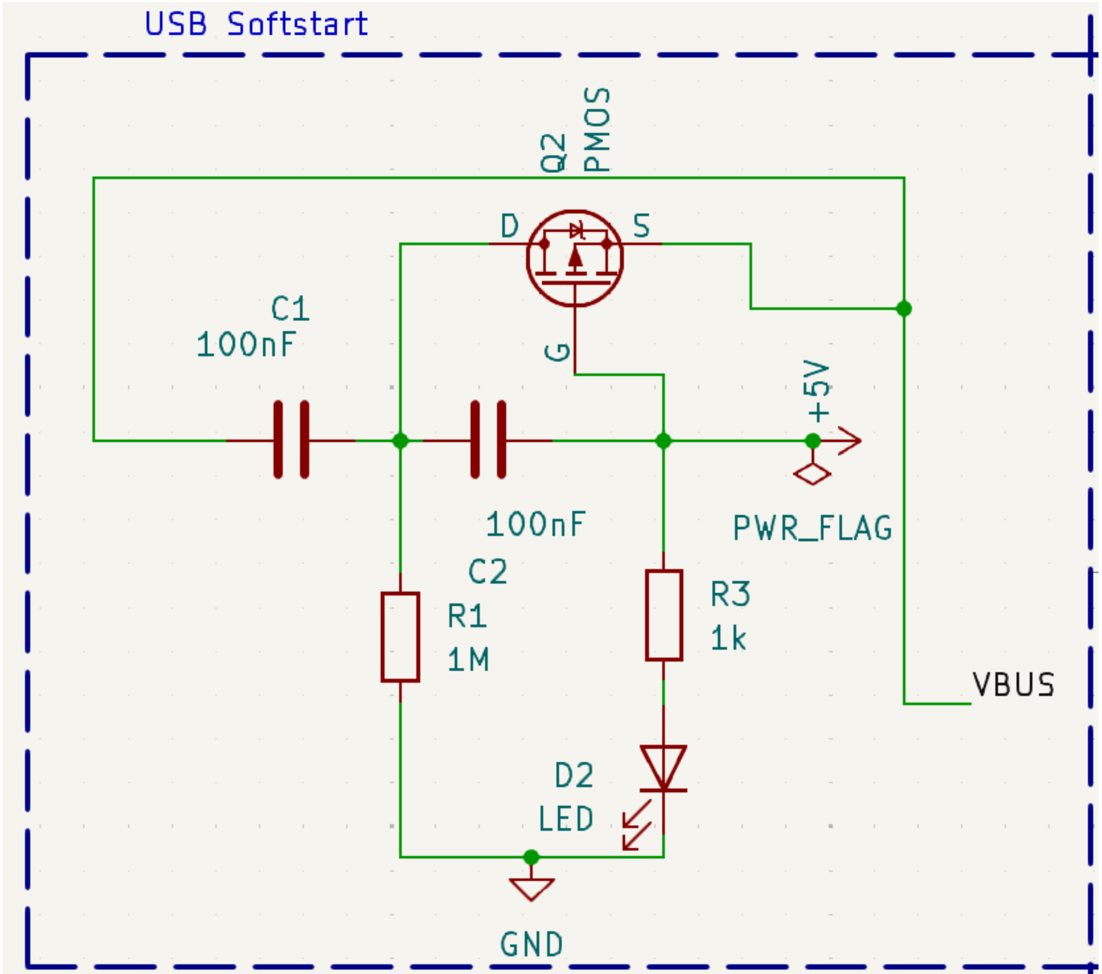


Figure 11: The USB Soft start circuit.

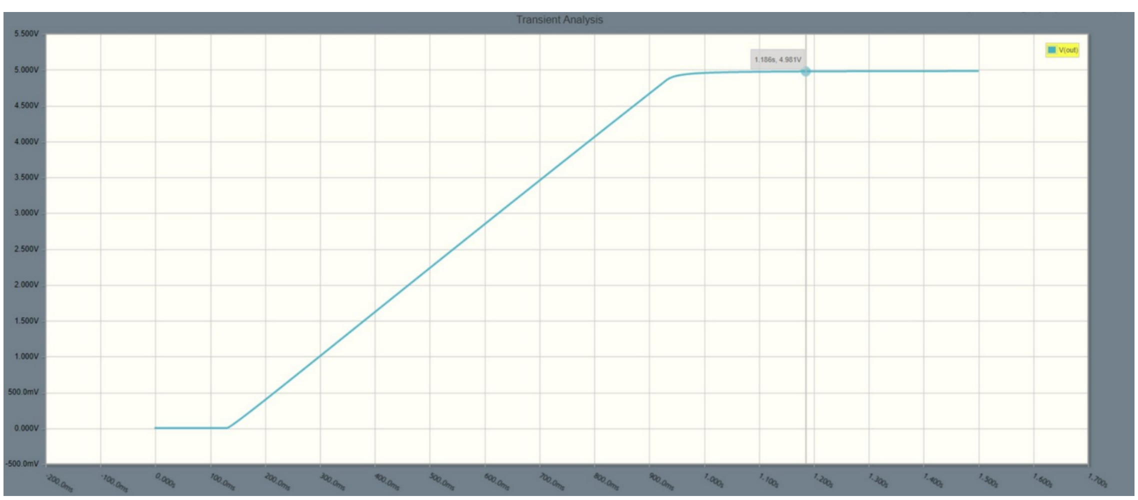


Figure 13: The USB Soft start circuit Simulation

The voltage for this simulation slowly increases until it reaches the max voltage (5V) at around 1.5s. This incrementing of the voltage protects against frying the pins due to a sudden change in voltage.

Vout(t) = Input voltage * [1 – ($e^{-t/RC1}$)]

= 5V * (1 – e $^{-1.5/ (10M * 25nF)}$

= 5 * (1 – $e^{-6.0}$) Volts

= 4.98Volts (after 1.5 seconds from initial start t = 0secs

Where, R = Resistor and C = Capacitor, t = any instance of time

The formula for calculating the output voltage of a soft start circuit considers input voltage, resistor value and the capacitor. The output voltage depends upon the capacitor charging and discharging behavior through resistor R1, which controls the gate voltage of MOSFET, resulting in controlling the drain-source current.

5.3.4 PCB Layout

We chose a 2-layer board for this PCB to keep our total device cost under $50. Most of our tracks use a width of 0.25mm (about 0.01 in); however, for our power traces, we used 0.4 mm. There is a common ground plane on the front and back of the board. A design choice we made with the data lines was to keep them as close as possible to each other on the PCB to avoid as little interference as possible. Lastly, there is silkscreen labeling the pins for testing use.

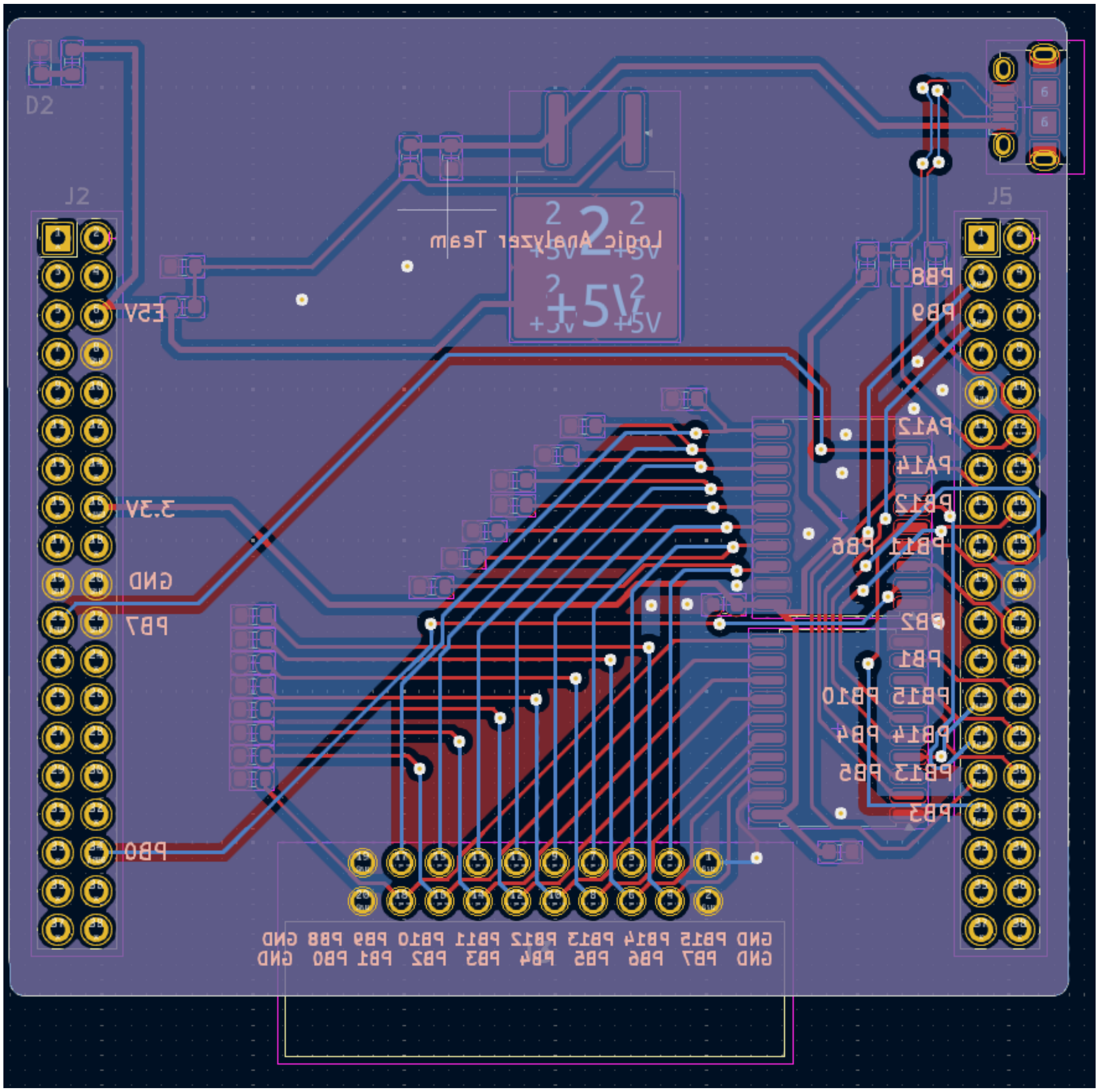


Figure 14: PCB layout

5.3.5 Final PCB

The PCB was manufactured through JLPCB. In addition, all the SMD components (resistors, bus transceivers, MOSFET, LED) and the micro-USB were soldered onto the PCB through JLPCB for a little extra (~$7 per board). The only components manually soldered onto the PCB were the four 1x19-pin headers and the twenty-pin connector.

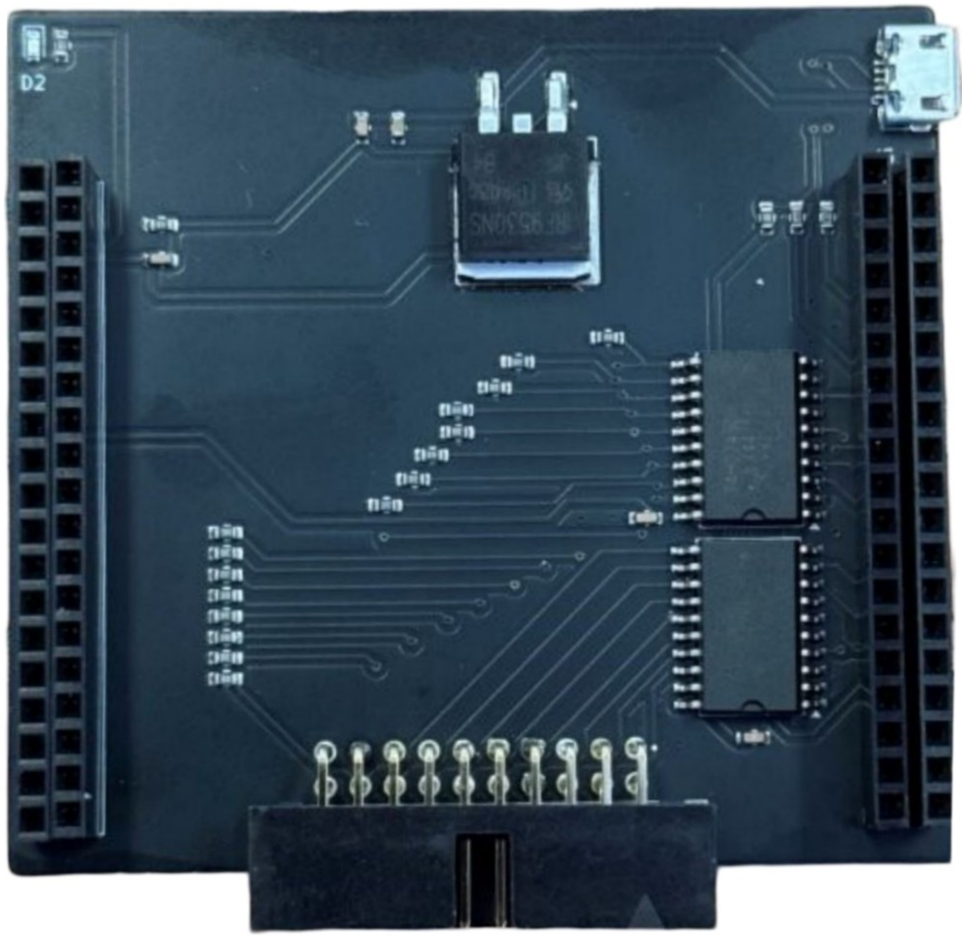


Figure 15: Final PCB

5.4 Device Case

5.4.1 Device Case Overview

The primary objective of designing a case for our logic analyzer was to protect the internal hardware components from physical damage while enhancing the device's overall visual appeal. Created using a Tinker CAD, the case features a robust and compact structure specifically designed to securely house the logic analyzer (PCB + Microcontroller). The design consists of two pieces: a base that firmly holds the logic analyzer and a lid that covers the top for additional protection. It incorporates precise cutouts for connectivity and ventilation, ensuring optimal functionality and cooling. The STL file for the design was exported and 3D printed using a Reality Ender 3 Pro, which is freely available to GMU students. The final version was fabricated using PLA plastic with a 25% infill density, offering an ideal balance between strength and lightweight construction.

5.4.2 Top View of Base and Lid

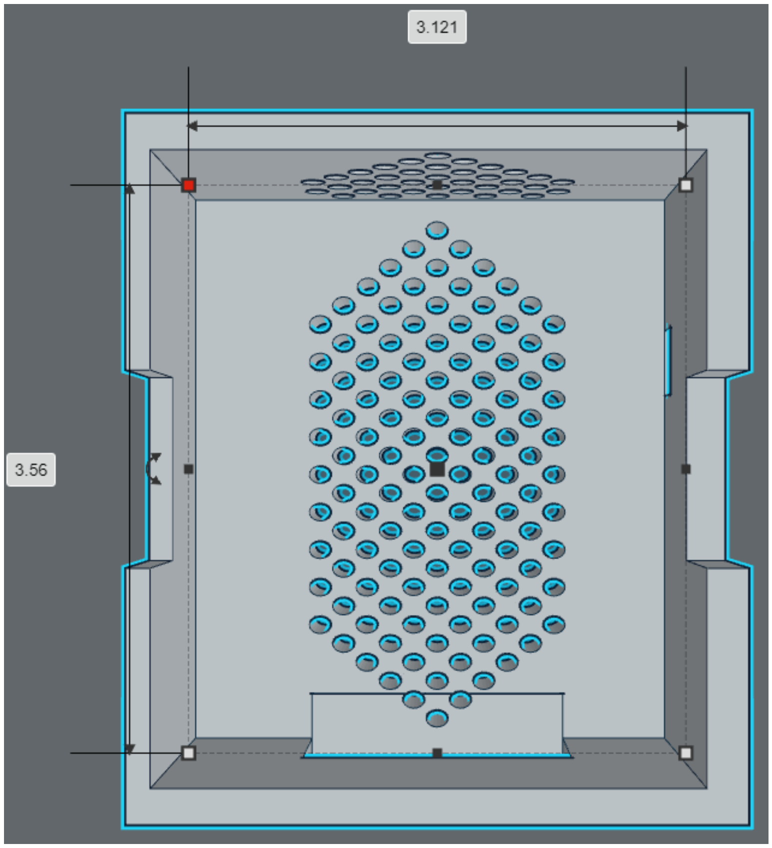


Figure 16 : Top12 View of Base

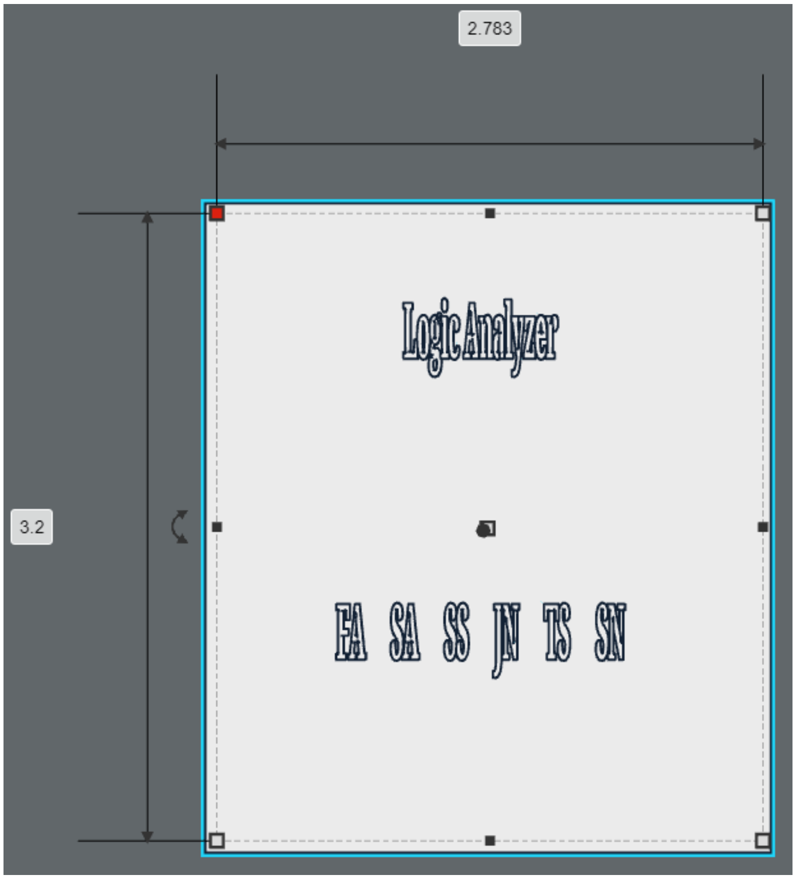


Figure 17 : Top13 View of Lid

5.4.3 Bottom View of Base and Lid

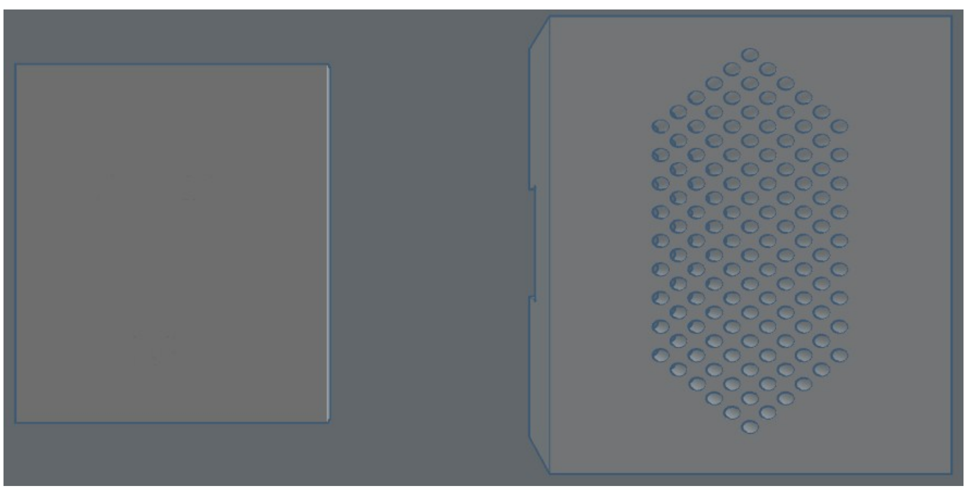


Figure 18 : Bottom14 View of Lid + Base

5.4.4 Top View of Assembled Case

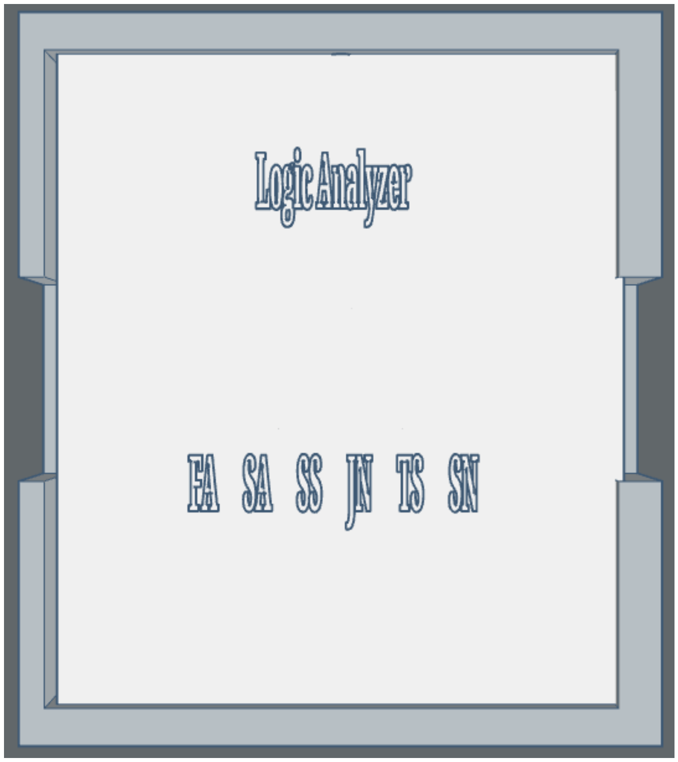


Figure 19 : Top15 View of Assembled Case

5.4.5 Bottom View of Assembled Case

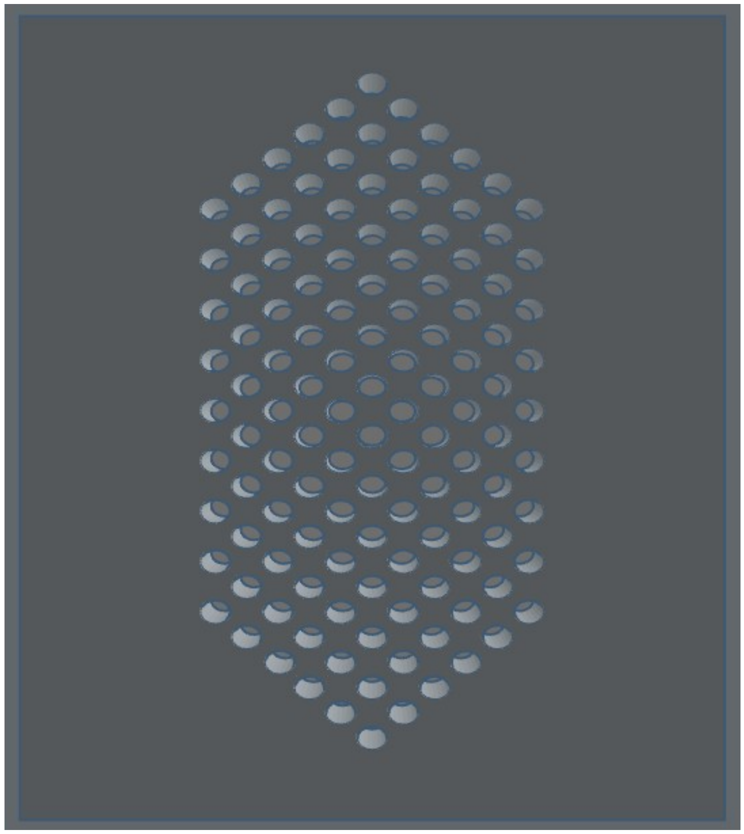


Figure 20: Bottom16 View of Assembled Case

5.4.6 Front View of Assembled Case

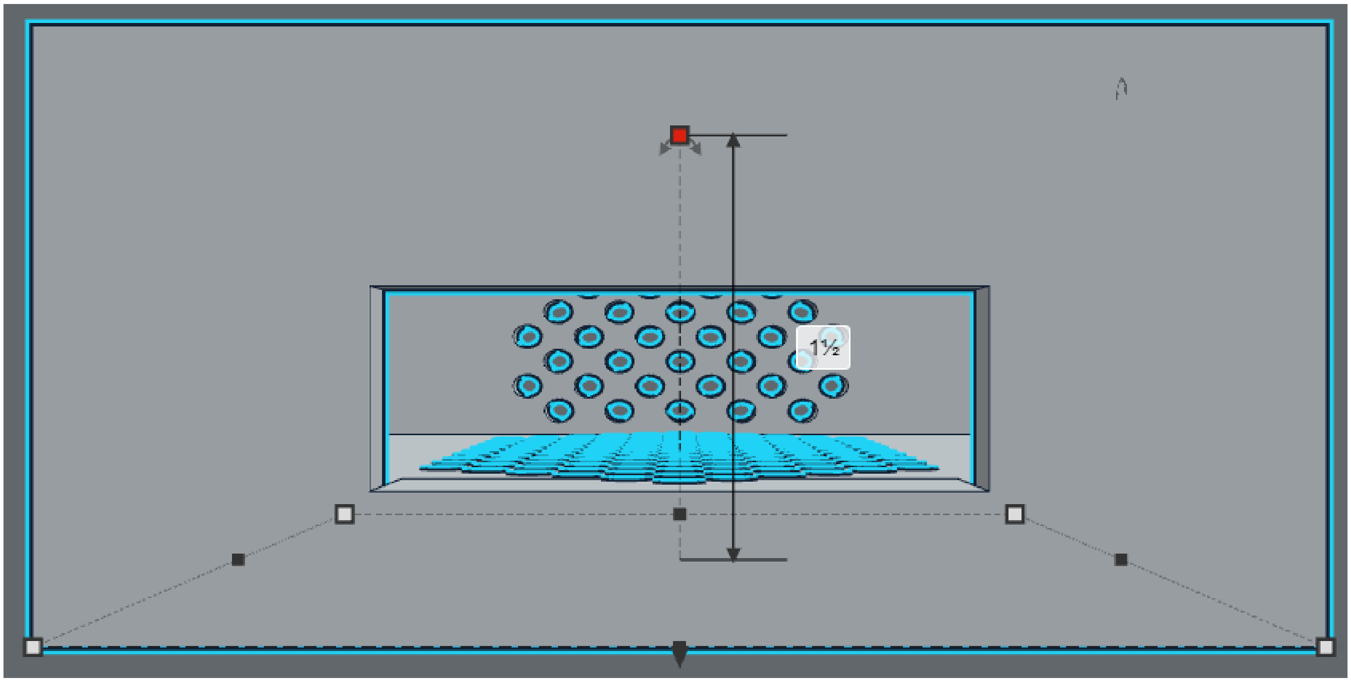


Figure 21: Front17 View of Assembled Case

5.4.7 Side View of Assembled Case

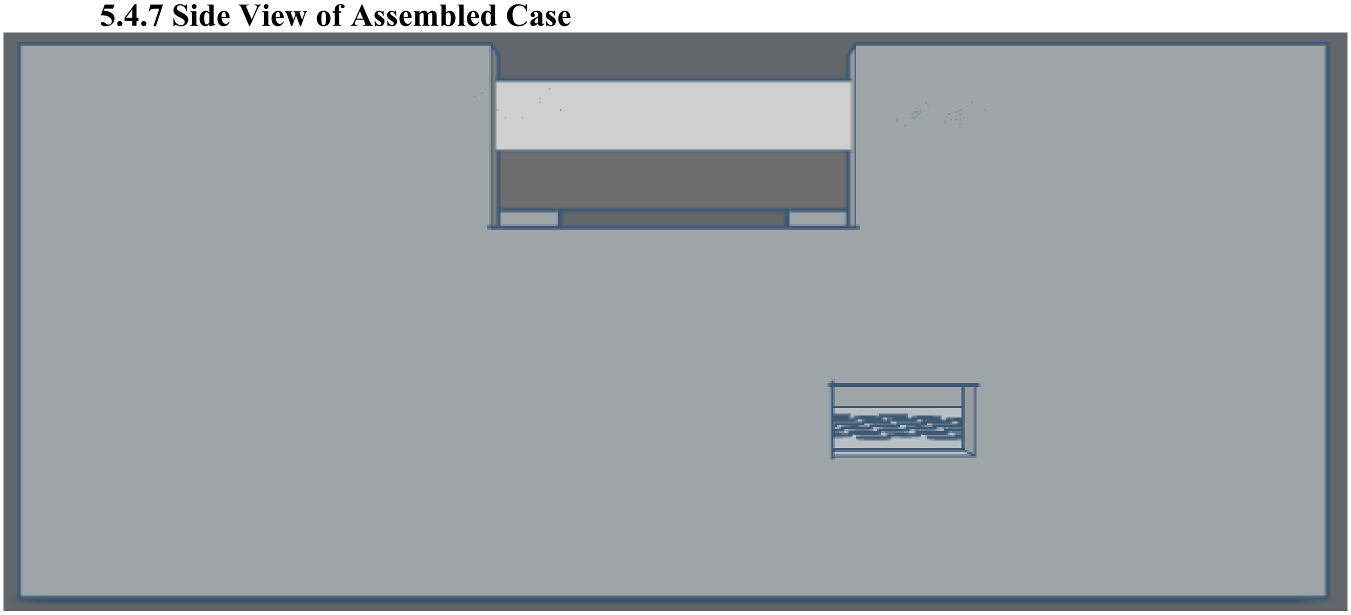


Figure 22:18 Side View of Assembled Case

5.4.8 Rear View of Assembled Case

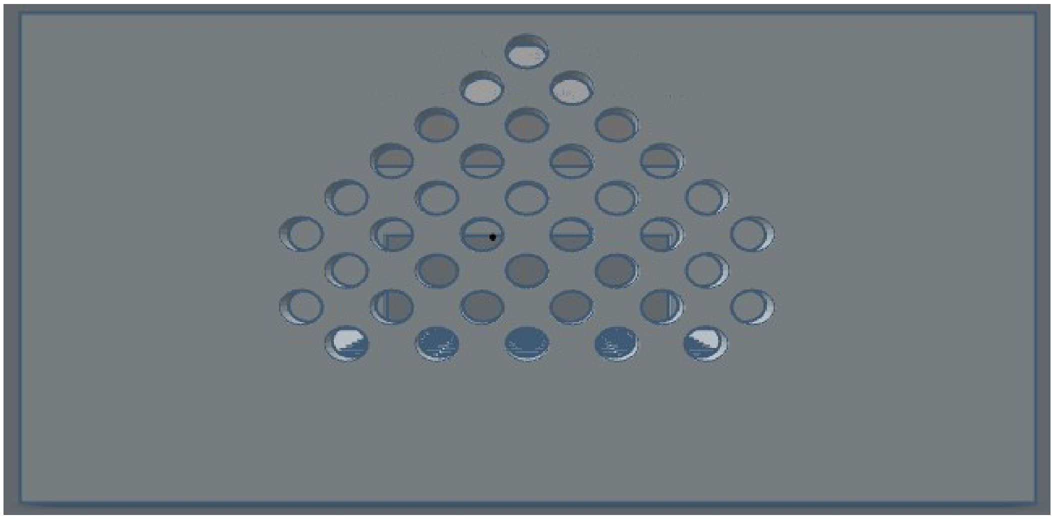


Figure 23:19 Rear View of Assembled Case

5.4.9 Isometric View of Assembled Case

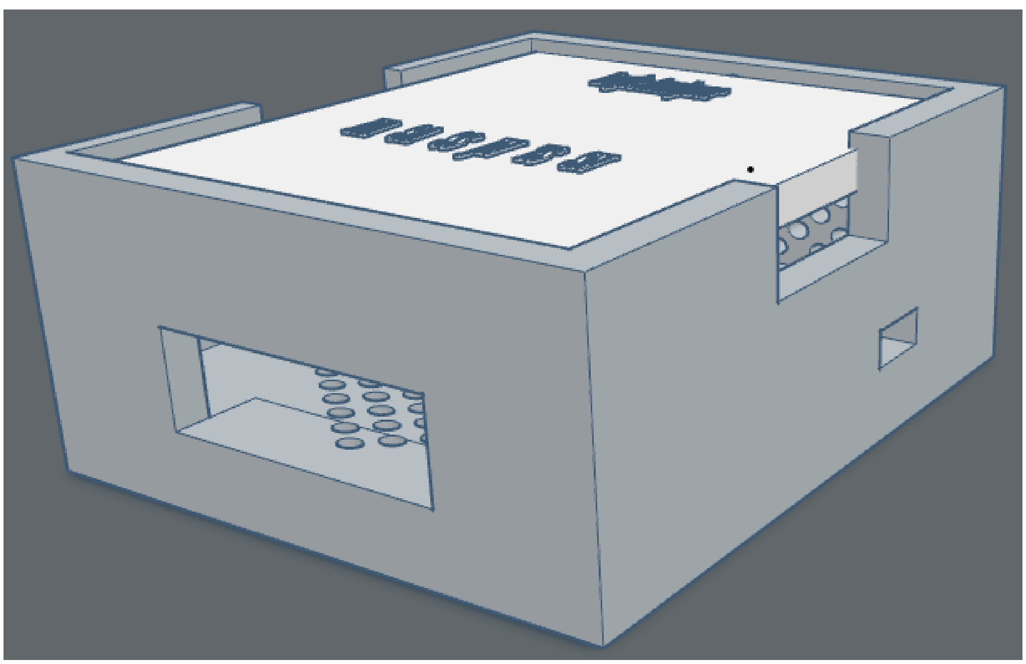


Figure 24: Isometric20 View of Assembled Case

5.4.10 Final Printed Case Model

Figure 215: 3D Printed Case for our Logic Analyzer

6: Preliminary Experimentation Plan

6.1: Preliminary Experiment

Testing for the prototype will be split into three core components, PCB module, Microcontroller unit and Graphical User Interface. Each of these components is tested individually to validate its functionality once the component is integrated.

6.2: Testing Procedures for Components

6.2.1 MCU Testing

The Nucleo-F303RE board is a widely used development board for prototyping and testing embedded systems. It is essential to confirm that our MCU operates effectively, which requires specific test cases to evaluate various components of the project.

Test #1: USB connection Test

The main purpose of the MCU is transmitting the logic signal from the MCU to GUI for signal graphing. To guarantee correct data transmission via USB, we added more components to the MCU, like capacitors and oscillators, and activated the USB connection. With the microcontroller being set up with USB functionality, we were able to transmit data via USB which was then displayed on the serial monitor of the PC. This test proved the data transfer from MCU to PC to be successful.

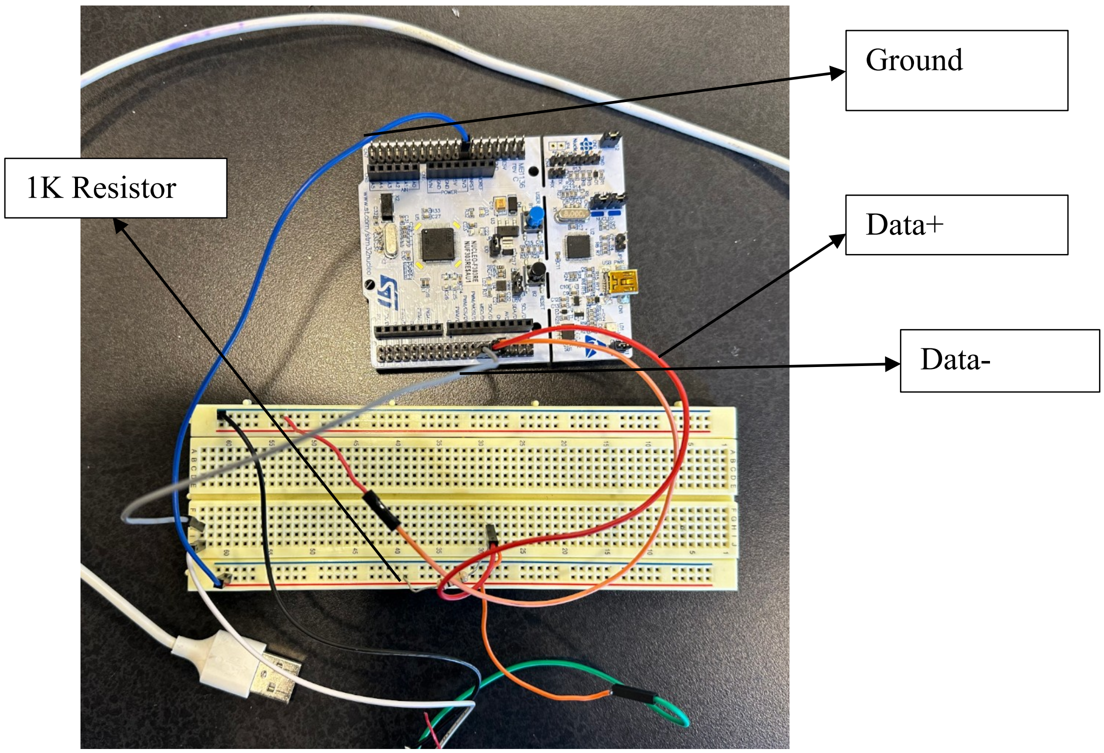


Figure 26 22: USB connection.

```
uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)
{
  uint8_t result = USBD_OK;
  /* USER CODE BEGIN 7 */
  USBD_CDC_HandleTypeDef *hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
  if (hcdc->TxState != 0){
    return USBD_BUSY;
  }
  USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
  result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);
  /* USER CODE END 7 */
  return result;
}
```

Figure 27:23 USB function to transmit data from MCU to PC

Test # 2: Recording GPIO Register values in circular buffer.

The implemented functionality records the register values of the PB port. To manage memory efficiently and ensure smooth operation, the recorded data is stored in a Circular Buffer. This Circular Buffer allows for continuous recording by overwriting old data with new data when the buffer is full, preventing memory overflow. By storing register values in this manner, the system can monitor changes in PB port registers over time while conserving memory resources. This approach is particularly useful for systems requiring real-time monitoring of hardware states or for debugging purposes, as it provides a historical record of register values for analysis. A debugger was used to verify the values of the buffers live as they were being modified and showed it was successfully rewriting old values with new ones after filling the buffer with samples from the GPIO pins. The values in the buffer did correctly relay the values that were fed to the GPIO pins.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){

    if ( htim == &htim16)
        {
            // Read GPIO port B value and store it
            buttonState = GPIOB->IDR;

            logicBuffer[i++] = buttonState;
            uint8_t binary = (uint16_t)buttonState;
            CDC_Transmit_FS(&binary,sizeof(binary));

            if(i == 20){
                i = 0;
            }

            HAL_GPIO_TogglePin(GPIOA,GPIO_PIN_5);
        }
```

Figure 28:24 Sample code Written to grab Register Values

Test # 3: MCU Trigger functionality

The trigger is crucial in delivering the digital signals that the user is looking for. The trigger functions like a filter, the user has a choice of having a rising edge or falling edge trigger, on which pin to activate it, and the number of samples taken after the trigger condition is met. To test this functionality, PB3 has an active rising edge trigger with 300 samples after the trigger condition. The first test involves setting PB3 to high and setting a breakpoint at post trigger state to verify that the post trigger set is never met. PB3 was connected to ground next to verify that falling edge does not pass the trigger checking functionality. Finally, PB3 was set back to high, and the post trigger state was met, showing that the trigger had occurred on PB3. A similar procedure was carried out with PB3 trigger set to falling edge, and ground was tested first, then high for rising edge, and finally back to ground to verify falling edge correctly triggers and switches states to post trigger.

Test # 4: MCU Timer configuration for data Sampling

Timer Configuration has been implemented to record the data timely into the circular buffer so there is no data loss. This setup allows for precise timing of register value recordings, ensuring that data is captured at specific intervals. This functionality is very crucial for several reasons. Firstly, they provide insight into the behavior of the system over time by capturing register values at regular intervals. This is essential for monitoring the stability and performance of the system. Secondly, timer-based synchronization ensures that data is captured consistently and reliably, reducing the likelihood of missing critical events or changes in the digital circuit. To meet our 5 MHz samples per second requirement, we carried out a test to verify the speed the MCU is sampling. An output pin is toggled after each the GPIO pins are sampled and buffered, and trigger checking procedures have been completed in the timer 2 ISR. The oscilloscope tool from the ADALM2000 was connected to the output pin to show the frequency of the output pin. It showed that the output was near the 5 MHz samples required for the project.

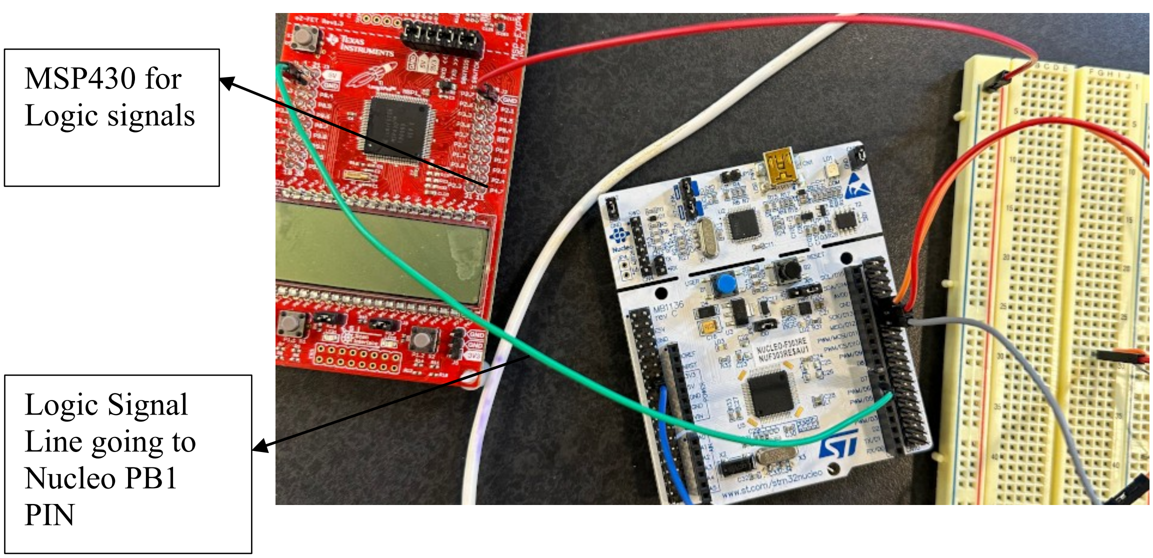


Figure 2925: Sending Logic signal to Nucleo using MSP430.

6.2.2: GUI

There are a variety of tests that will need to be done to make sure that the GUI is functional. We have performed several tests to verify the functionality of our Graphical User Interface.

- **Functional:**
  - Test to see if all eight signals are accurately displayed in real-time.
  - Compare the generated signal with known simulated signals to ensure that the signals generated by the microcontroller are displayed correctly.
  - Simulate signal loss or corruption to check if the GUI displays the desired error message.
  - Ensure that the GUI provides visual feedback/acknowledgement of a command being sent.
  -
- **Compatibility:**
  - We will need to run the software on Linux, MacOS, and windows to make sure it installs and launches with no issues.
  - This includes tests on virtual machines.
  - Check to see if there are any layout discrepancies or functional differences across platforms.
  - It should send an error message when the microcontroller is disconnected from the device.

- **Performance:**
  - The GUI should be able to display signals in real-time without significant lag or delay.

- **Usability:**
  - Ensure that the UI (user interface) is intuitive and easy to navigate for users.
  - Verify that the GUI indicates a successful connection to the microcontroller.

- **Longevity:**
  - Test to see how the GUI holds up when being run for long periods of time to check for crashes or performance degradation over time.

Some parts have been successful. An appropriate error message is displayed if a connection to the MCU can't be detected. The GUI can also successfully read serial data. It can also adjust the sampling rate and start/stop sampling data. The GUI can display UART messages and display it on the screen.

Written Code:

```python
main.py

This module serves as the entry point for the application. It initializes the PyQt6
application, applies aesthetic styles, searches for a specific serial device, and
launches the appropriate window based on whether the device is found.

Dependencies:
- sys
- serial.tools.list_ports
- PyQt6.QtWidgets.QApplication
- LogicDisplay from LogicDisplay module
- SerialApp from connection module
- apply_styles from aesthetic module
"""

import sys
import serial.tools.list_ports
from PyQt6.QtWidgets import QApplication
from LogicDisplay import LogicDisplay
from connection import SerialApp
from aesthetic import apply_styles

def main():
    """
    The main function initializes the PyQt6 application, applies styles, and attempts
    to connect to a serial device with specified VID and PID. Depending on whether the
    device is found, it either opens the LogicDisplay window or the SerialApp connection
    window.

    Steps:
    1. Initialize the QApplication with command-line arguments.
    2. Apply aesthetic styles to the application (e.g., dark mode, icons).
    3. Search for a serial device with VID=1155 and PID=22336.
    4. If the device is found:
        - Create and display a LogicDisplay window with the device's port.
        - Print a message indicating automatic connection.
    5. If the device is not found:
        - Create and display a SerialApp window to allow user connection.
        - Print a message indicating that the connection window is opening.
    6. Execute the application's event loop and exit when done.
    """

    app = QApplication(sys.argv)
    apply_styles(app)

    # Attempt to find the device with vid=1155 and pid=22336
    vid = 1155
    pid = 22336
    ports = serial.tools.list_ports.comports()
    target_port = None
    for port in ports:
        if port.vid == vid and port.pid == pid:
            target_port = port.device
            break

    if target_port:
        # Device found, directly create LogicDisplay
        window = LogicDisplay(port=target_port, baudrate=115200, bufferSize=4096, channels=8)
        window.show()
        print(f"Automatically connected to device on port {target_port}")
    else:
        # Device not found, show SerialApp
        window = SerialApp()
        window.show()
        print("Device not found. Opening connection window.")

    sys.exit(app.exec())

if __name__ == '__main__':
    main()
```

Figure 3026: Main.py

```python
"""
aesthetic.py

This module provides functions to apply aesthetic styles to the PyQt6 application.
It includes functionalities to set a dark mode stylesheet and configure the application icon.
"""

import os
from PyQt6.QtGui import QIcon
from PyQt6.QtWidgets import QApplication


def get_icon() -> QIcon:
    """
    Retrieves the application's icon.

    Constructs the path to the icon image relative to the current file's directory
    and returns a QIcon object.

    Returns:
        QIcon: The icon object to be used as the application icon.
    """
    # Construct the icon path relative to this file
    icon_path = os.path.join(os.path.dirname(__file__), 'images', 'logo.png')
    return QIcon(icon_path)
```

Figure 31:27 aesthetic.py - get_icone

```python
def apply_styles(app: QApplication) -> None:
    """
    Applies aesthetic styles to the PyQt6 application.

    This function sets a dark mode stylesheet for the entire application and
    configures the application's window icon.

    Args:
        app (QApplication): The PyQt6 application instance to style.
    """
    # Dark mode stylesheet
    dark_style = """
    QWidget {
        background-color: #2e2e2e;
        color: #ffffff;
    }
    QPushButton {
        background-color: #3c3c3c;
        color: #ffffff;
        border: 1px solid #555;
        border-radius: 5px;
        padding: 5px;
    }
    QPushButton:checked {
        background-color: #4d4d4d;
    }
    QPushButton:hover {
        background-color: #4d4d4d;
    }
    QComboBox {
        background-color: #3c3c3c;
        color: #ffffff;
        border: 1px solid #555;
        padding: 5px;
    }
    QComboBox QAbstractItemView {
        background-color: #3c3c3c;
        color: #ffffff;
        selection-background-color: #4d4d4d;
    }
    QLineEdit {
        background-color: #3c3c3c;
        color: #ffffff;
        border: 1px solid #555;
        padding: 5px;
    }
    QMenu {
        background-color: #3c3c3c;
        color: #ffffff;
        border: 1px solid #555;
    }
    QMenu::item:selected {
        background-color: #4d4d4d;
    }
    """
    # Apply the stylesheet
    app.setStyleSheet(dark_style)

    # Set the application icon
    app.setWindowIcon(get_icon())
```

Figure 32 28:Aesthetic.py - apply_style

```python
"""
connection.py

This module manages serial connections for the application. It provides the
SerialApp class, a PyQt6 QMainWindow that allows users to select, connect,
and disconnect from available serial COM ports. Upon successful connection,
it launches the LogicDisplay window to interact with the connected device.
"""

import sys
import serial.tools.list_ports
from PyQt6.QtWidgets import QMainWindow, QPushButton, QVBoxLayout, QWidget, QComboBox
from PyQt6.QtGui import QIcon
from typing import Optional
from aesthetic import get_icon
from LogicDisplay import LogicDisplay  # Ensure this is the correct file name


class SerialApp(QMainWindow):
    """
    SerialApp is a PyQt6 QMainWindow that provides a user interface for managing
    serial connections. It allows users to refresh available COM ports, connect
    to a selected port, and disconnect from the current connection.

    Attributes:
        logic_display_window (Optional[LogicDisplay]): Reference to the LogicDisplay window.
    """

    def __init__(self) -> None:
        """
        Initializes the SerialApp window, sets up the UI components, and configures
        the window's title and icon.
        """
        super().__init__()
        self.setWindowTitle("Serial Connection Manager")
        self.setWindowIcon(get_icon())
        self.logic_display_window: Optional[LogicDisplay] = None  # Reference to LogicDisplay window
        self.initUI()

    def initUI(self) -> None:
        """
        Sets up the user interface components, including the main widget, layout,
        COM ports dropdown, and control buttons (Refresh, Connect, Disconnect).
        """
        # Main widget and layout
        self.main_widget = QWidget()
        self.setCentralWidget(self.main_widget)
        layout = QVBoxLayout(self.main_widget)

        # Dropdown for COM ports
        self.combo_ports = QComboBox()
        self.refresh_ports()
        layout.addWidget(self.combo_ports)

        # Refresh button
        self.button_refresh = QPushButton("Refresh")
        self.button_refresh.clicked.connect(self.refresh_ports)
        layout.addWidget(self.button_refresh)

        # Connect button
        self.button_connect = QPushButton("Connect")
        self.button_connect.clicked.connect(self.connect_device)
        layout.addWidget(self.button_connect)

        # Disconnect button
        self.button_disconnect = QPushButton("Disconnect")
        self.button_disconnect.clicked.connect(self.disconnect_device)
        self.button_disconnect.setEnabled(False)
        layout.addWidget(self.button_disconnect)
```

Figure 3329:connection.py - SerialApp – init

```python
    def refresh_ports(self) -> None:
        """
        Refreshes the list of available serial COM ports by clearing the current
        dropdown and repopulating it with the latest COM port information.
        """
        self.combo_ports.clear()
        ports = serial.tools.list_ports.comports()
        for port in ports:
            self.combo_ports.addItem(port.device)

    def connect_device(self) -> None:
        """
        Attempts to establish a connection to the selected serial COM port.
        If successful, it disables the Connect button, enables the Disconnect
        button, and opens the LogicDisplay window. If a connection is already
        open, it closes the previous LogicDisplay window before opening a new one.

        Prints status messages to the console regarding the connection status.
        """
        port_name = self.combo_ports.currentText()
        try:
            self.button_connect.setEnabled(False)
            self.button_disconnect.setEnabled(True)
            print(f"Connected to {port_name}")

            # Close the previous LogicDisplay window if it exists
            if self.logic_display_window:
                self.logic_display_window.close()

            # Create a new LogicDisplay window
            self.logic_display_window = LogicDisplay(port=port_name, baudrate=115200, channels=8)
            self.logic_display_window.show()

        except Exception as e:
            print(f"Failed to connect to {port_name}: {str(e)}")
            self.button_connect.setEnabled(True)
            self.button_disconnect.setEnabled(False)

    def disconnect_device(self) -> None:
        """
        Disconnects from the currently connected serial COM port by closing the
        LogicDisplay window. It also resets the Connect and Disconnect buttons'
        enabled states and prints a status message to the console.
        """
        # Close the LogicDisplay window when disconnecting the device
        if self.logic_display_window:
            self.logic_display_window.close()
            self.logic_display_window = None  # Reset the reference

        self.button_connect.setEnabled(True)
        self.button_disconnect.setEnabled(False)
        print("Disconnected")
```

Figure 3430:connection.py - SerialApp – refresh, connect, disconnect

```python
# InterfaceCommands.py:

def get_trigger_edge_command(trigger_modes):
    """
    Determines the edge of buttons selected and returns the corresponding command integer.

    The LSB represents the edge of channel 1 while the MSB represents channel 8.
    If the button is on 'Rising Edge', the bit value will be 1.
    If it's on 'Falling Edge' or 'No Trigger', the bit will be 0.
    This 8-bit value is converted to an int and can be sent as a character.
    """
    command_value = 0
    for idx in range(8):
        mode = trigger_modes[idx]
        if mode == 'Rising Edge':
            command_value |= 1 << idx  # Set bit idx if Rising Edge
    return command_value


def get_trigger_pins_command(trigger_modes):
    """
    Determines which channels have triggers enabled and returns the corresponding command integer.

    The LSB represents channel 1, and the MSB represents channel 8.
    If the button is either 'Rising Edge' or 'Falling Edge', the bit value is 1.
    If it's 'No Trigger', the bit will be 0.
    This 8-bit value is converted to an int and can be sent as a character.
    """
    command_value = 0
    for idx in range(8):
        mode = trigger_modes[idx]
        if mode in ('Rising Edge', 'Falling Edge'):
            command_value |= 1 << idx  # Set bit idx if trigger is enabled
    return command_value

```

Figure 3531:InterfaceCommands.py

```python
"""
LogicDisplay.py

This module defines the LogicDisplay class, a PyQt6 QMainWindow that serves as the main interface
for the Logic Analyzer application. It allows users to select between different communication
protocols (Signal, I2C, SPI, UART) and manages the corresponding display modules. The LogicDisplay
handles the initialization of the user interface, loading of selected modules, and management of
serial communication parameters such as baud rate and buffer size.
"""

import sys
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QWidget,
    QVBoxLayout,
    QHBoxLayout,
    QButtonGroup,
    QPushButton,
)
from PyQt6.QtGui import QAction
from PyQt6.QtCore import Qt

from typing import Optional

from aesthetic import get_icon
from Signal import SignalDisplay
from I2C import I2CDisplay
from SPI import SPIDisplay
from UART import UARTDisplay
```

Figure 3632: LogicDisplay.py - Libraries

```python
class LogicDisplay(QMainWindow):
    """
    LogicDisplay is the main window of the Logic Analyzer application. It provides an interface
    for users to select different communication protocols and displays the corresponding modules.

    Attributes:
        port (str): The serial port to which the device is connected.
        default_baudrate (int): The default baud rate for serial communication.
        baudrate (int): The current baud rate for serial communication.
        channels (int): The number of channels used in the logic analyzer.
        bufferSize (int): The size of the buffer for serial communication.
        current_module (Optional[QWidget]): The currently active display module.
    """

    def __init__(self, port: str, baudrate: int, bufferSize: int = 4096, channels: int = 8) -> None:
        """
        Initializes the LogicDisplay window with the specified serial port, baud rate, buffer size,
        and number of channels. It sets up the user interface and loads the default module.

        Args:
            port (str): The serial port to connect to.
            baudrate (int): The baud rate for serial communication.
            bufferSize (int, optional): The size of the buffer for serial communication. Defaults to 4096.
            channels (int, optional): The number of channels for the logic analyzer. Defaults to 8.
        """
        super().__init__()
        self.port = port
        self.default_baudrate = baudrate  # Store default baud rate
        self.baudrate = baudrate
        self.channels = channels
        self.bufferSize = bufferSize

        self.setWindowTitle("Logic Analyzer")
        self.setWindowIcon(get_icon())

        self.current_module: Optional[QWidget] = None
        self.init_ui()

        # Load the default module (Signal)
        self.load_module('Signal')
```

Figure 3733: LogicDisplay.py - init

```python
    def init_ui(self) -> None:
        """
        Initializes the user interface components of the LogicDisplay window, including the
        mode selection buttons and the area where the selected module is displayed.
        """
        # Create a central widget with vertical layout
        central_widget = QWidget()
        central_layout = QVBoxLayout(central_widget)
        central_layout.setContentsMargins(0, 0, 0, 0)
        central_layout.setSpacing(0)

        # Create a widget for the buttons
        button_widget = QWidget()
        button_layout = QHBoxLayout(button_widget)
        button_layout.setContentsMargins(0, 0, 0, 0)
        button_layout.setSpacing(0)

        # Create buttons for each mode
        self.signal_button = QPushButton('Signal')
        self.i2c_button = QPushButton('I2C')
        self.spi_button = QPushButton('SPI')
        self.uart_button = QPushButton('UART')

        # Make buttons checkable
        self.signal_button.setCheckable(True)
        self.i2c_button.setCheckable(True)
        self.spi_button.setCheckable(True)
        self.uart_button.setCheckable(True)

        # Create a button group for exclusive checking
        self.mode_button_group = QButtonGroup()
        self.mode_button_group.setExclusive(True)
        self.mode_button_group.addButton(self.signal_button)
        self.mode_button_group.addButton(self.i2c_button)
        self.mode_button_group.addButton(self.spi_button)
        self.mode_button_group.addButton(self.uart_button)

        # Set the default checked button
        self.signal_button.setChecked(True)

        # Add buttons to the layout
        button_layout.addWidget(self.signal_button)
        button_layout.addWidget(self.i2c_button)
        button_layout.addWidget(self.spi_button)
        button_layout.addWidget(self.uart_button)

        # Connect buttons to the handler
        self.signal_button.clicked.connect(lambda: self.load_module('Signal'))
        self.i2c_button.clicked.connect(lambda: self.load_module('I2C'))
        self.spi_button.clicked.connect(lambda: self.load_module('SPI'))
        self.uart_button.clicked.connect(lambda: self.load_module('UART'))

        # Create a widget to hold the current module
        self.module_widget = QWidget()
        self.module_layout = QVBoxLayout(self.module_widget)
        self.module_layout.setContentsMargins(0, 0, 0, 0)
        self.module_layout.setSpacing(0)

        # Add the button widget and the module widget to the central layout
        central_layout.addWidget(button_widget)
        central_layout.addWidget(self.module_widget)

        # Set the central widget
        self.setCentralWidget(central_widget)
```

Figure 3834: LogicDisplay.py - init_ui

```python
    def load_module(self, module_name: str) -> None:

        Loads the specified module into the LogicDisplay window. It handles the cleanup of
        the existing module and initializes the new module based on the selected communication
        protocol.

        Args:
            module_name (str): The name of the module to load. Expected values are 'Signal',
                                'I2C', 'SPI', or 'UART'.
        """
        # Remove the existing module widget if any
        if self.current_module:
            self.current_module.close()
            self.current_module.deleteLater()
            self.current_module = None

        # Clear the module_layout
        while self.module_layout.count():
            item = self.module_layout.takeAt(0)
            widget = item.widget()
            if widget is not None:
                widget.deleteLater()

        # Reset baud rate to default when switching modes
        if module_name != 'UART':
            self.baudrate = self.default_baudrate

        # Load the selected module
        if module_name == 'Signal':
            self.current_module = SignalDisplay(self.port, self.baudrate, self.bufferSize, self.channels)
            self.signal_button.setChecked(True)
        elif module_name == 'I2C':
            self.current_module = I2CDisplay(self.port, self.baudrate, self.bufferSize)
            self.i2c_button.setChecked(True)
        elif module_name == 'SPI':
            self.current_module = SPIDisplay(self.port, self.baudrate, self.bufferSize)
            self.spi_button.setChecked(True)
        elif module_name == 'UART':
            # Update baud rate if changed in UART mode
            self.current_module = UARTDisplay(self.port, self.baudrate, self.bufferSize)
            self.uart_button.setChecked(True)

        if self.current_module:
            self.module_layout.addWidget(self.current_module)
            self.current_module.show()
        else:
            # Placeholder if the module is not implemented
            placeholder_widget = QWidget()
            self.module_layout.addWidget(placeholder_widget)

    def update_baudrate(self, baudrate: int) -> None:
        """
        Updates the baud rate for serial communication. This method can be called to change
        the baud rate dynamically based on user input or other conditions.

        Args:
            baudrate (int): The new baud rate to set.
        """
        self.baudrate = baudrate

    def closeEvent(self, event: Qt.QEvent) -> None:
        """
        Handles the close event of the LogicDisplay window. Ensures that the currently active
        module is properly closed before the window itself is closed.

        Args:
            event (Qt.QEvent): The close event triggered when the window is being closed.
        """
        if self.current_module:
            self.current_module.close()
        event.accept()
```

Figure 3935: LogicDisplay.py - load_module, update_baudrate, closeEvent

```python
"""
Signal.py

This module defines classes and functionalities related to handling serial communication,
data processing, and graphical display for a Logic Analyzer application. It includes:

- SerialWorker: A QThread subclass that manages serial data reading and triggering mechanisms.
- FixedYViewBox: A custom PyQtGraph ViewBox that restricts scaling and translation on the Y-axis.
- EditableButton: A QPushButton subclass that allows for context menu operations like renaming.
- SignalDisplay: A QWidget subclass that provides the main interface for displaying and interacting
  with signal data, including plotting, control buttons, and trigger configurations.

Dependencies:
- sys, serial, math, time, numpy, pyqtgraph
- PyQt6.QtWidgets, PyQt6.QtGui, PyQt6.QtCore
- collections.deque
- InterfaceCommands (custom module)
- aesthetic (custom module)
"""

import sys
import serial
import math
import time
import numpy as np
import pyqtgraph as pg
from PyQt6.QtWidgets import (
    QWidget,
    QVBoxLayout,
    QHBoxLayout,
    QGridLayout,
    QInputDialog,
    QMenu,
    QPushButton,
    QLabel,
    QLineEdit,
)
from PyQt6.QtGui import QIcon, QIntValidator
from PyQt6.QtCore import QTimer, QThread, pyqtSignal, Qt
from collections import deque
from typing import List, Optional

from InterfaceCommands import (
    get_trigger_edge_command,
    get_trigger_pins_command,
)
from aesthetic import get_icon

```

Figure 4036: Signal.py - Libraries

```python
class SerialWorker(QThread):
    """
    SerialWorker handles serial communication in a separate thread. It reads incoming data from
    the serial port, processes trigger conditions for multiple channels, and emits signals when
    data is ready for processing.

    Attributes:
        data_ready (pyqtSignal): Signal emitted when new data is ready. Carries a list of integers.
        is_running (bool): Flag indicating whether the worker is active.
        channels (int): Number of channels to monitor for triggers.
        trigger_modes (List[str]): List of trigger modes for each channel.
        bufferSize (int): Maximum size of the data buffer.
        serial (serial.Serial): Serial port instance for communication.
    """

    data_ready = pyqtSignal(list)

    def __init__(self, port: str, baudrate: int, bufferSize: int, channels: int = 8) -> None:
        """
        Initializes the SerialWorker thread with the specified serial port parameters.

        Args:
            port (str): The serial port to connect to (e.g., 'COM3', '/dev/ttyUSB0').
            baudrate (int): The baud rate for serial communication.
            bufferSize (int): The maximum number of data points to store in the buffer.
            channels (int, optional): The number of channels to monitor for triggers. Defaults to 8.
        """
        super().__init__()
        self.is_running = True
        self.channels = channels
        self.trigger_modes = ['No Trigger'] * self.channels
        self.bufferSize = bufferSize
        try:
            self.serial = serial.Serial(port, baudrate)
        except serial.SerialException as e:
            print(f"Failed to open serial port: {str(e)}")
            self.is_running = False

    def set_trigger_mode(self, channel_idx: int, mode: str) -> None:
        """
        Sets the trigger mode for a specific channel.

        Args:
            channel_idx (int): The index of the channel (0-based).
            mode (str): The trigger mode to set (e.g., 'No Trigger', 'Rising Edge', 'Falling Edge').
        """
        self.trigger_modes[channel_idx] = mode

```

Figure 4137: Signal.py - Serial Worker Init, trigger mode

```python
    def run(self) -> None:
        """
        The main loop of the worker thread. Continuously reads data from the serial port,
        processes trigger conditions, and emits data_ready signals when appropriate.
        """
        data_buffer = deque(maxlen=self.bufferSize - 24)
        triggered = [False] * self.channels

        while self.is_running:
            if self.serial.in_waiting:
                raw_data = self.serial.read(self.serial.in_waiting).splitlines()
                for line in raw_data:
                    try:
                        data_value = int(line.strip())
                        data_buffer.append(data_value)

                        for i in range(self.channels):
                            if not triggered[i] and self.trigger_modes[i] != 'No Trigger':
                                last_value = data_buffer[-2] if len(data_buffer) >= 2 else None
                                if last_value is not None:
                                    current_bit = (data_value >> i) & 1
                                    last_bit = (last_value >> i) & 1

                                    if self.trigger_modes[i] == 'Rising Edge' and last_bit == 0 and current_bit == 1:
                                        triggered[i] = True
                                        print(f"Trigger condition met on channel {i+1}: Rising Edge")
                                    elif self.trigger_modes[i] == 'Falling Edge' and last_bit == 1 and current_bit == 0:
                                        triggered[i] = True
                                        print(f"Trigger condition met on channel {i+1}: Falling Edge")
                        if any(triggered) or all(mode == 'No Trigger' for mode in self.trigger_modes):
                            self.data_ready.emit([data_value])

                    except ValueError:
                        continue

    def stop_worker(self) -> None:
        """
        Stops the worker thread by setting the running flag to False and closing the serial port.
        """
        self.is_running = False
        if self.serial.is_open:
            self.serial.close()
```

Figure 4238: Signal.py - Serial Worker Run & Stop

```python
class FixedYViewBox(pg.ViewBox):
    """
    FixedYViewBox is a custom PyQtGraph ViewBox that restricts scaling and translation
    along the Y-axis, allowing only horizontal scaling and movement.
    """

    def __init__(self, *args, **kwargs) -> None:
        """
        Initializes the FixedYViewBox with the provided arguments.
        """
        super(FixedYViewBox, self).__init__(*args, **kwargs)

    def scaleBy(self, s=None, center=None, x: Optional[float] = None, y: Optional[float] = None) -> None:
        """
        Overrides the scaleBy method to fix the Y-axis scaling to 1.0, preventing vertical scaling.

        Args:
            s: Scaling factor (unused for Y-axis).
            center: Center point for scaling.
            x (float, optional): Scaling factor for X-axis.
            y (float, optional): Scaling factor for Y-axis (fixed to 1.0).
        """
        y = 1.0
        if x is None:
            if s is None:
                x = 1.0
            elif isinstance(s, dict):
                x = s.get('x', 1.0)
            elif isinstance(s, (list, tuple)):
                x = s[0]
            else:
                x = s
        super(FixedYViewBox, self).scaleBy(x=x, y=y, center=center)

    def translateBy(self, t=None, x: Optional[float] = None, y: Optional[float] = None) -> None:
        """
        Overrides the translateBy method to fix the Y-axis translation to 0.0, preventing vertical movement.

        Args:
            t: Translation value (unused for Y-axis).
            x (float, optional): Translation value for X-axis.
            y (float, optional): Translation value for Y-axis (fixed to 0.0).
        """
        y = 0.0
        if x is None:
            if t is None:
                x = 0.0
            elif isinstance(t, dict):
                x = t.get('x', 0.0)
            elif isinstance(t, (list, tuple)):
                x = t[0]
            else:
                x = t
        super(FixedYViewBox, self).translateBy(x=x, y=y)

```

Figure 43 39: Signal.py - FixedYViewBox

```python
class EditableButton(QPushButton):
    """
    EditableButton is a QPushButton subclass that allows users to rename the button label
    via a context menu and reset it to its default label.
    """

    def __init__(self, label: str, parent: Optional[QWidget] = None) -> None:
        """
        Initializes the EditableButton with a given label.

        Args:
            label (str): The initial text label of the button.
            parent (QWidget, optional): The parent widget. Defaults to None.
        """
        super().__init__(label, parent)
        self.setContextMenuPolicy(Qt.ContextMenuPolicy.CustomContextMenu)
        self.customContextMenuRequested.connect(self.show_context_menu)
        self.default_label = label

    def show_context_menu(self, position: Qt.QPoint) -> None:
        """
        Displays a context menu with options to rename the button or reset it to the default label.

        Args:
            position (Qt.QPoint): The position where the context menu is requested.
        """
        menu = QMenu()
        rename_action = menu.addAction("Rename")
        reset_action = menu.addAction("Reset to Default")
        action = menu.exec(self.mapToGlobal(position))
        if action == rename_action:
            new_label, ok = QInputDialog.getText(
                self, "Rename Button", "Enter new label:", text=self.text()
            )
            if ok and new_label:
                self.setText(new_label)
        elif action == reset_action:
            self.setText(self.default_label)

```

Figure 4440: Signal.py - Editable Button

```python
class SignalDisplay(QWidget):
    """
    SignalDisplay provides the main interface for displaying and interacting with signal data.
    It includes graphical plots, control buttons, and configurations for triggers and sampling.

    Attributes:
        period (int): The period for sample timing.
        num_samples (int): Number of samples to capture.
        port (str): Serial port for communication.
        baudrate (int): Baud rate for serial communication.
        channels (int): Number of channels for the logic analyzer.
        bufferSize (int): Size of the data buffer.
        data_buffer (List[deque]): Data buffers for each channel.
        channel_visibility (List[bool]): Visibility status for each channel.
        is_single_capture (bool): Flag indicating if a single capture is active.
        current_trigger_modes (List[str]): Current trigger modes for each channel.
        trigger_mode_indices (List[int]): Indices representing trigger modes for each channel.
        sample_rate (int): Sampling rate in Hz.
        timer (QTimer): Timer for updating the plot.
        is_reading (bool): Flag indicating if data reading is active.
        worker (SerialWorker): Worker thread handling serial communication.
        graph_layout (pg.GraphicsLayoutWidget): Layout widget for graphs.
        plot (pg.PlotItem): Plot item for displaying data.
        colors (List[str]): List of colors for plotting each channel.
        curves (List[pg.PlotDataItem]): Plot curves for each channel.
        channel_buttons (List[EditableButton]): Buttons to toggle channel visibility.
        trigger_mode_buttons (List[QPushButton]): Buttons to toggle trigger modes.
        cursor (pg.InfiniteLine): Cursor for measurement on the plot.
        cursor_label (pg.TextItem): Label displaying cursor position.
    """

    def __init__(self, port: str, baudrate: int, bufferSize: int, channels: int = 8) -> None:
        """
        Initializes the SignalDisplay with the specified serial port parameters and sets up the UI.

        Args:
            port (str): Serial port for communication.
            baudrate (int): Baud rate for serial communication.
            bufferSize (int): Size of the data buffer.
            channels (int, optional): Number of channels for the logic analyzer. Defaults to 8.
        """
        super().__init__()
        self.period = 65454
        self.num_samples = 0
        self.port = port
        self.baudrate = baudrate
        self.channels = channels
        self.bufferSize = bufferSize

        self.data_buffer: List[deque] = [deque(maxlen=self.bufferSize) for _ in range(self.channels)]
        self.channel_visibility: List[bool] = [False] * self.channels

        self.is_single_capture = False
        self.current_trigger_modes: List[str] = ['No Trigger'] * self.channels
        self.trigger_mode_indices: List[int] = [0] * self.channels
        self.sample_rate = 1000  # Default sample rate in Hz

        self.setup_ui()
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_plot)

        self.is_reading = False

        self.worker = SerialWorker(self.port, self.baudrate, self.bufferSize, channels=self.channels)
        self.worker.data_ready.connect(self.handle_data)
        self.worker.start()
```

Figure 4541: Signal.py - SignalDisplay – init

```python
    def setup_ui(self) -> None:
        main_layout = QHBoxLayout(self)

        self.graph_layout = pg.GraphicsLayoutWidget()
        main_layout.addWidget(self.graph_layout)

        self.plot = self.graph_layout.addPlot(viewBox=FixedYViewBox())
        self.plot.setXRange(0, 200 / self.sample_rate, padding=0)
        self.plot.setLimits(xMin=0, xMax=self.bufferSize / self.sample_rate)
        self.plot.setYRange(-2, 2 * self.channels, padding=0)
        self.plot.enableAutoRange(axis=pg.ViewBox.XAxis, enable=False)
        self.plot.enableAutoRange(axis=pg.ViewBox.YAxis, enable=False)
        self.plot.showGrid(x=True, y=True)
        self.plot.getAxis('left').setTicks([])
        self.plot.getAxis('left').setStyle(showValues=False)
        self.plot.getAxis('left').setPen(None)
        self.plot.setLabel('bottom', 'Time', units='s')

        self.colors = ['#FF6EC7', '#39FF14', '#FF486D', '#BF00FF', '#FFFF33', '#FFA500', '#00F5FF', '#BFFF00']
        self.curves: List[pg.PlotDataItem] = []
        for i in range(self.channels):
            color = self.colors[i % len(self.colors)]
            curve = self.plot.plot(pen=pg.mkPen(color=color, width=4))
            curve.setVisible(self.channel_visibility[i])
            self.curves.append(curve)

        button_layout = QGridLayout()
        main_layout.addLayout(button_layout)

        self.channel_buttons: List[EditableButton] = []
        self.trigger_mode_buttons: List[QPushButton] = []
        self.trigger_mode_options = ['No Trigger', 'Rising Edge', 'Falling Edge']

        for i in range(self.channels):
            label = f"DIO {i+1}"
            button = EditableButton(label)
            button.setCheckable(True)
            button.setChecked(False)
            button.toggled.connect(lambda checked, idx=i: self.toggle_channel(idx, checked))
            button_layout.addWidget(button, i, 0)
            self.channel_buttons.append(button)

            trigger_button = QPushButton(self.trigger_mode_options[self.trigger_mode_indices[i]])
            trigger_button.clicked.connect(lambda _, idx=i: self.toggle_trigger_mode(idx))
            button_layout.addWidget(trigger_button, i, 1)
            self.trigger_mode_buttons.append(trigger_button)

        # Sample Rate input
        self.sample_rate_label = QLabel("Sample Rate (Hz):")
        button_layout.addWidget(self.sample_rate_label, self.channels, 0)

        self.sample_rate_input = QLineEdit()
        self.sample_rate_input.setValidator(QIntValidator(1, 5000000))
        self.sample_rate_input.setText("1000")
        button_layout.addWidget(self.sample_rate_input, self.channels, 1)
        self.sample_rate_input.returnPressed.connect(self.handle_sample_rate_input)

        # Number of Samples input
        self.num_samples_label = QLabel("Number of Samples:")
        button_layout.addWidget(self.num_samples_label, self.channels + 1, 0)

        self.num_samples_input = QLineEdit()
        self.num_samples_input.setValidator(QIntValidator(1, 1023))
        self.num_samples_input.setText("300")
        button_layout.addWidget(self.num_samples_input, self.channels + 1, 1)
        self.num_samples_input.returnPressed.connect(self.send_num_samples_command)

        # Control buttons layout
        control_buttons_layout = QHBoxLayout()
        self.toggle_button = QPushButton("Start")
        self.toggle_button.clicked.connect(self.toggle_reading)
        control_buttons_layout.addWidget(self.toggle_button)

        self.single_button = QPushButton("Single")
        self.single_button.clicked.connect(self.start_single_capture)
        control_buttons_layout.addWidget(self.single_button)
        button_layout.addLayout(control_buttons_layout, self.channels + 2, 0, 1, 2)

        # Cursor for measurement
        self.cursor = pg.InfiniteLine(pos=0, angle=90, movable=True, pen=pg.mkPen(color='y', width=2))
        self.plot.addItem(self.cursor)

        self.cursor_label = pg.TextItem(anchor=(0, 1), color='y')
        self.plot.addItem(self.cursor_label)
        self.update_cursor_position()
        self.cursor.sigPositionChanged.connect(self.update_cursor_position)
```
Figure 4642: Signal.py - SignalDisplay – setup_ui

```python
    def handle_sample_rate_input(self) -> None:
        Handles the event when the sample rate input field receives a return key press.
        Validates and updates the sample rate, adjusts the plot range and timers accordingly.
        """
        try:
            sample_rate = int(self.sample_rate_input.text())
            if sample_rate <= 0:
                raise ValueError("Sample rate must be positive")
            self.sample_rate = sample_rate
            period = (72 * 10**6) / sample_rate
            print(f"Sample Rate set to {sample_rate} Hz, Period: {period} ticks")
            self.updateSampleTimer(int(period))
            self.plot.setXRange(0, 200 / self.sample_rate, padding=0)
            self.plot.setLimits(xMin=0, xMax=self.bufferSize / self.sample_rate)
        except ValueError as e:
            print(f"Invalid sample rate: {e}")

    def send_num_samples_command(self) -> None:
        """
        Sends the number of samples command to the serial device based on user input.
        """
        try:
            num_samples = int(self.num_samples_input.text())
            self.num_samples = num_samples
            self.updateTriggerTimer()
        except ValueError as e:
            print(f"Invalid number of samples: {e}")

    def send_trigger_edge_command(self) -> None:
        """
        Sends the trigger edge configuration to the serial device.
        """
        command_int = get_trigger_edge_command(self.current_trigger_modes)
        command_str = str(command_int)
        try:
            self.worker.serial.write(b'2')
            time.sleep(0.01)
            self.worker.serial.write(b'0')
            time.sleep(0.01)
            self.worker.serial.write(command_str.encode('utf-8'))
            time.sleep(0.01)
        except serial.SerialException as e:
            print(f"Failed to send trigger edge command: {str(e)}")

    def send_trigger_pins_command(self) -> None:
        """
        Sends the trigger pins configuration to the serial device.
        """
        command_int = get_trigger_pins_command(self.current_trigger_modes)
        command_str = str(command_int)
        try:
            self.worker.serial.write(b'3')
            time.sleep(0.001)
            self.worker.serial.write(b'0')
            time.sleep(0.001)
            self.worker.serial.write(command_str.encode('utf-8'))
        except serial.SerialException as e:
            print(f"Failed to send trigger pins command: {str(e)}")
```

Figure 4743: Signal.py - SignalDisplay – sampleRate, numSamples, trigger

```python
    def updateSampleTimer(self, period: int) -> None:
        """
        Updates the sample timer configuration on the serial device.

        Args:
            period (int): The period value to set for the sample timer.
        """
        self.period = period
        try:
            self.worker.serial.write(b'5')
            time.sleep(0.001)
            # Send first byte
            selected_bits = (period >> 24) & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.001)
            # Send second byte
            selected_bits = (period >> 16) & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.001)
            self.worker.serial.write(b'6')
            time.sleep(0.001)
            # Send third byte
            selected_bits = (period >> 8) & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.001)
            # Send fourth byte
            selected_bits = period & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.001)
        except Exception as e:
            print(f"Failed to update sample timer: {e}")

    def updateTriggerTimer(self) -> None:
        """
        Updates the trigger timer configuration on the serial device based on the number of samples.
        """
        sampling_freq = 72e6 / self.period
        trigger_freq = sampling_freq / self.num_samples
        period16 = 72e6 / trigger_freq
        prescaler = 1
        if period16 > 2**16:
            prescaler = math.ceil(period16 / (2**16))
            period16 = int((72e6 / prescaler) / trigger_freq)
            print(f"Period timer 16 set to {period16}, Timer 16 prescaler is {prescaler}")
        try:
            self.worker.serial.write(b'4')
            time.sleep(0.01)
            # Send high byte
            selected_bits = (period16 >> 8) & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.01)
            # Send low byte
            selected_bits = period16 & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.01)
            # Update Prescaler
            self.worker.serial.write(b'7')
            time.sleep(0.01)
            # Send high byte
            selected_bits = (prescaler >> 8) & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.01)
            # Send low byte
            selected_bits = prescaler & 0xFF
            self.worker.serial.write(str(selected_bits).encode('utf-8'))
            time.sleep(0.01)
        except Exception as e:
            print(f"Failed to update trigger timer: {e}")
```

Figure 4844: Signal.py - SignalDisplay – Sample & Trigger Timer

```python
    def toggle_trigger_mode(self, channel_idx: int) -> None:
        """
        Toggles the trigger mode for a specific channel cyclically through predefined options.

        Args:
            channel_idx (int): The index of the channel (0-based).
        """
        self.trigger_mode_indices[channel_idx] = (self.trigger_mode_indices[channel_idx] + 1) % len(self.trigger_mode_options)
        mode = self.trigger_mode_options[self.trigger_mode_indices[channel_idx]]
        self.trigger_mode_buttons[channel_idx].setText(mode)
        self.current_trigger_modes[channel_idx] = mode
        if self.worker:
            self.worker.set_trigger_mode(channel_idx, mode)
        self.send_trigger_edge_command()
        self.send_trigger_pins_command()

    def is_light_color(self, hex_color: str) -> bool:
        """
        Determines if a given hex color is light based on its luminance.

        Args:
            hex_color (str): The hex color string (e.g., '#FF6EC7').

        Returns:
            bool: True if the color is light, False otherwise.
        """
        hex_color = hex_color.lstrip('#')
        r, g, b = tuple(int(hex_color[i:i+2], 16) for i in (0, 2, 4))
        luminance = (0.299 * r + 0.587 * g + 0.114 * b) / 255
        return luminance > 0.5

    def toggle_channel(self, channel_idx: int, is_checked: bool) -> None:
        """
        Toggles the visibility of a specific channel's data on the plot.

        Args:
            channel_idx (int): The index of the channel (0-based).
            is_checked (bool): Whether the channel should be visible.
        """
        self.channel_visibility[channel_idx] = is_checked
        self.curves[channel_idx].setVisible(is_checked)

        button = self.channel_buttons[channel_idx]
        if is_checked:
            color = self.colors[channel_idx % len(self.colors)]
            text_color = 'black' if self.is_light_color(color) else 'white'
            button.setStyleSheet(f"QPushButton {{ background-color: {color}; color: {text_color}; "
                                 f"border: 1px solid #555; border-radius: 5px; padding: 5px; }}")
        else:
            button.setStyleSheet("")

    def toggle_reading(self) -> None:
        """
        Toggles the data reading state between active and inactive. Starts or stops data acquisition
        and updates the UI accordingly.
        """
        if self.is_reading:
            self.send_stop_message()
            self.stop_reading()
            self.toggle_button.setText("Run")
            self.single_button.setEnabled(True)
            self.toggle_button.setStyleSheet("")
        else:
            self.is_single_capture = False
            self.send_start_message()
            self.start_reading()
            self.toggle_button.setText("Running")
            self.single_button.setEnabled(True)
            self.toggle_button.setStyleSheet("background-color: #00FF77; color: black;")
```

Figure 4945: Signal.py - SignalDisplay –toggle and is_light_color

```python
    def send_start_message(self) -> None:
        """
        Sends a 'start' command to the serial device to begin data acquisition.
        """
        if self.worker.serial.is_open:
            try:
                self.worker.serial.write(b'0')
                time.sleep(0.001)
                self.worker.serial.write(b'0')
                time.sleep(0.001)
                self.worker.serial.write(b'0')
                print("Sent 'start' command to device")
            except serial.SerialException as e:
                print(f"Failed to send 'start' command: {str(e)}")
        else:
            print("Serial connection is not open")

    def send_stop_message(self) -> None:
        """
        Sends a 'stop' command to the serial device to halt data acquisition.
        """
        if self.worker.serial.is_open:
            try:
                self.worker.serial.write(b'1')
                time.sleep(0.001)
                self.worker.serial.write(b'1')
                time.sleep(0.001)
                self.worker.serial.write(b'1')
                print("Sent 'stop' command to device")
            except serial.SerialException as e:
                print(f"Failed to send 'stop' command: {str(e)}")
        else:
            print("Serial connection is not open")

    def start_reading(self) -> None:
        """
        Starts the data reading process by activating the timer.
        """
        if not self.is_reading:
            self.is_reading = True
            self.timer.start(1)

    def stop_reading(self) -> None:
        """
        Stops the data reading process by deactivating the timer.
        """
        if self.is_reading:
            self.is_reading = False
            self.timer.stop()
```

Figure 5046: Signal.py - SignalDisplay –Start & Stop

```python
    def start_single_capture(self) -> None:
        """
        Initiates a single data capture by clearing existing data buffers, sending a start message,
        and starting the reading process. Disables relevant UI buttons during capture.
        """
        if not self.is_reading:
            self.clear_data_buffers()
            self.is_single_capture = True
            self.send_start_message()
            self.start_reading()
            self.single_button.setEnabled(False)
            self.toggle_button.setEnabled(False)
            self.single_button.setStyleSheet("background-color: #00FF77; color: black;")

    def stop_single_capture(self) -> None:
        """
        Stops a single data capture by stopping the reading process, sending a stop message,
        and re-enabling relevant UI buttons.
        """
        self.is_single_capture = False
        self.stop_reading()
        self.send_stop_message()
        self.single_button.setEnabled(True)
        self.toggle_button.setEnabled(True)
        self.toggle_button.setText("Start")
        self.single_button.setStyleSheet("")

    def clear_data_buffers(self) -> None:
        """
        Clears all data buffers for each channel.
        """
        self.data_buffer = [deque(maxlen=self.bufferSize) for _ in range(self.channels)]

    def handle_data(self, data_list: List[int]) -> None:
        """
        Handles incoming data emitted by the SerialWorker. Appends data to buffers and manages
        single capture logic.

        Args:
            data_list (List[int]): List of incoming data values.
        """
        if self.is_reading:
            for data_value in data_list:
                for i in range(self.channels):
                    bit_value = (data_value >> i) & 1
                    self.data_buffer[i].append(bit_value)
            if self.is_single_capture and all(len(buf) >= self.bufferSize for buf in self.data_buffer):
                self.stop_single_capture()
```

Figure 5147: Signal.py - Signal Display –Single Start & Stop, Buff Clear and Data Handle

```python
    def update_plot(self) -> None:
        """
        Updates the graphical plot with the latest data from the buffers.
        """
        for i in range(self.channels):
            if self.channel_visibility[i]:
                inverted_index = self.channels - i - 1
                num_samples = len(self.data_buffer[i])
                if num_samples > 1:
                    t = np.arange(num_samples) / self.sample_rate
                    square_wave_time = []
                    square_wave_data = []
                    for j in range(1, num_samples):
                        square_wave_time.extend([t[j-1], t[j]])
                        level = self.data_buffer[i][j-1] + inverted_index * 2
                        square_wave_data.extend([level, level])
                        if self.data_buffer[i][j] != self.data_buffer[i][j-1]:
                            square_wave_time.append(t[j])
                            level = self.data_buffer[i][j] + inverted_index * 2
                            square_wave_data.append(level)
                    self.curves[i].setData(square_wave_time, square_wave_data)

    def update_cursor_position(self) -> None:
        """
        Updates the position and label of the cursor on the plot based on user interaction.
        """
        cursor_pos = self.cursor.pos().x()
        self.cursor_label.setText(f"Cursor: {cursor_pos:.6f} s")
        self.cursor_label.setPos(cursor_pos, self.channels * 2 - 1)

    def closeEvent(self, event: Qt.QEvent) -> None:
        """
        Handles the close event of the SignalDisplay widget. Ensures that the worker thread is
        properly stopped before closing.

        Args:
            event (Qt.QEvent): The close event triggered when the widget is being closed.
        """
        self.worker.stop_worker()
        self.worker.quit()
        self.worker.wait()
        event.accept()
```

Figure 5248: Signal.py - Signal Display –Update Plot and Cursor, Close Event

```python
"""
I2C.py

This module defines classes and functionalities related to handling I2C communication,
data processing, and graphical display for a Logic Analyzer application. It includes:

- SerialWorker: A QThread subclass that manages serial data reading, I2C decoding, and triggering mechanisms.
- FixedYViewBox: A custom PyQtGraph ViewBox that restricts scaling and translation on the Y-axis.
- EditableButton: A QPushButton subclass that allows for context menu operations like renaming.
- I2CChannelButton: An EditableButton subclass specific to I2C channels, with additional signals for configuration.
- I2CConfigDialog: A QDialog subclass that provides a user interface for configuring I2C channel settings.
- I2CDisplay: A QWidget subclass that provides the main interface for displaying and interacting with I2C data,
  including plotting, control buttons, and trigger configurations.

Dependencies:
- sys, serial, math, time, numpy, pyqtgraph
- PyQt6.QtWidgets, PyQt6.QtGui, PyQt6.QtCore
- collections.deque
- InterfaceCommands (custom module)
- aesthetic (custom module)
"""

import sys
import serial
import math
import time
import numpy as np
import pyqtgraph as pg
from PyQt6.QtWidgets import (
    QWidget,
    QVBoxLayout,
    QHBoxLayout,
    QGridLayout,
    QInputDialog,
    QMenu,
    QPushButton,
    QLabel,
    QLineEdit,
    QComboBox,
    QDialog,
    QRadioButton,
    QButtonGroup,
    QSizePolicy,
    QTextEdit,
    QGroupBox,
)
from PyQt6.QtGui import QIcon, QIntValidator, QTextCursor, QFont
from PyQt6.QtCore import QTimer, QThread, pyqtSignal, Qt, QPoint
from collections import deque
from typing import List, Dict, Optional

from InterfaceCommands import (
    get_trigger_edge_command,
    get_trigger_pins_command,
)
from aesthetic import get_icon

```

Figure 5349: I2C.py - Libraries

```python
class SerialWorker(QThread):
    """
    SerialWorker handles I2C serial communication in a separate thread. It reads incoming data from
    the serial port, decodes I2C messages, processes trigger conditions for multiple I2C groups,
    and emits signals when data or decoded messages are ready for processing.

    Attributes:
        data_ready (pyqtSignal): Signal emitted when new raw data is ready. Carries an integer value and sample index.
        decoded_message_ready (pyqtSignal): Signal emitted when a decoded I2C message is ready. Carries a dictionary with message details.
        is_running (bool): Flag indicating whether the worker is active.
        channels (int): Number of channels to monitor for triggers.
        group_configs (List[Dict]): Configuration settings for each I2C group.
        trigger_modes (List[str]): List of trigger modes for each channel.
        states (List[str]): Current state of the state machine for each I2C group.
        bit_buffers (List[List[int]]): Bit buffers for each I2C group.
        current_bytes (List[int]): Current byte being assembled for each I2C group.
        bit_counts (List[int]): Bit count for the current byte in each I2C group.
        decoded_messages (List[List[Dict]]): Decoded messages for each I2C group.
        scl_last_values (List[int]): Last sampled SCL values for edge detection.
        sda_last_values (List[int]): Last sampled SDA values for edge detection.
        messages (List[List[Dict]]): Accumulated messages for each I2C group.
        error_flags (List[bool]): Error flags for each I2C group.
        sample_idx (int): Global sample index counter.
    """

    data_ready = pyqtSignal(int, int)  # For raw data values and sample indices
    decoded_message_ready = pyqtSignal(dict)  # For decoded messages

    def __init__(self, port: str, baudrate: int, channels: int = 8, group_configs: Optional[List[Dict]] = None) -> None:
        """
        Initializes the SerialWorker thread with the specified serial port parameters and I2C group configurations.

        Args:
            port (str): The serial port to connect to (e.g., 'COM3', '/dev/ttyUSB0').
            baudrate (int): The baud rate for serial communication.
            channels (int, optional): The number of channels to monitor for triggers. Defaults to 8.
            group_configs (List[Dict], optional): Configuration settings for each I2C group. Defaults to None.
        """
        super().__init__()
        self.is_running = True
        self.channels = channels
        self.group_configs = group_configs if group_configs else [{} for _ in range(4)]
        self.trigger_modes = ['No Trigger'] * self.channels
        # Initialize I2C decoding variables for each group
        self.states = ['IDLE'] * len(self.group_configs)
        self.bit_buffers: List[List[int]] = [[] for _ in range(len(self.group_configs))]
        self.current_bytes = [0] * len(self.group_configs)
        self.bit_counts = [0] * len(self.group_configs)
        self.decoded_messages: List[List[Dict]] = [[] for _ in range(len(self.group_configs))]
        self.scl_last_values = [1] * len(self.group_configs)
        self.sda_last_values = [1] * len(self.group_configs)
        self.messages: List[List[Dict]] = [[] for _ in range(len(self.group_configs))]
        self.error_flags = [False] * len(self.group_configs)
        self.sample_idx = 0  # Initialize sample index

        try:
            self.serial = serial.Serial(port, baudrate)
        except serial.SerialException as e:
            print(f"Failed to open serial port: {str(e)}")
            self.is_running = False

        # Initialize sample index variables for each group
        self.addr_sample_idxs: List[Optional[int]] = [None] * len(self.group_configs)
        self.ack_sample_idxs: List[Optional[int]] = [None] * len(self.group_configs)
        self.data_sample_idxs: List[Optional[int]] = [None] * len(self.group_configs)
        self.stop_sample_idxs: List[Optional[int]] = [None] * len(self.group_configs)
```

Figure 5450: I2C.py - Serial Worker – init

```python
    def set_trigger_mode(self, channel_idx: int, mode: str) -> None:
        """
        Sets the trigger mode for a specific channel.

        Args:
            channel_idx (int): The index of the channel (0-based).
            mode (str): The trigger mode to set (e.g., 'No Trigger', 'Rising Edge', 'Falling Edge').
        """
        self.trigger_modes[channel_idx] = mode

    def run(self) -> None:
        """
        The main loop of the worker thread. Continuously reads data from the serial port,
        processes I2C decoding, and emits data_ready and decoded_message_ready signals when appropriate.
        """
        data_buffer = deque(maxlen=1000)

        while self.is_running:
            if self.serial.in_waiting:
                raw_data = self.serial.read(self.serial.in_waiting).splitlines()
                for line in raw_data:
                    try:
                        data_value = int(line.strip())
                        data_buffer.append(data_value)
                        self.data_ready.emit(data_value, self.sample_idx)  # Emit data_value and sample_idx
                        self.decode_i2c(data_value, self.sample_idx)
                        self.sample_idx += 1  # Increment sample index
                    except ValueError:
                        continue

```

Figure 5551:I2C.py - SerialWorker – triggerMode, run

```python
    def decode_i2c(self, data_value: int, sample_idx: int) -> None:
        """
        Decodes incoming serial data to interpret I2C messages based on configured groups.

        Args:
            data_value (int): The raw data value read from the serial port.
            sample_idx (int): The current sample index.
        """
        for group_idx, group_config in enumerate(self.group_configs):
            scl_channel = group_config.get('clock_channel', 2) - 1
            sda_channel = group_config.get('data_channel', 1) - 1
            address_width = group_config.get('address_width', 8)
            data_format = group_config.get('data_format', 'Hexadecimal')

            # Extract SCL and SDA values
            scl = (data_value >> scl_channel) & 1
            sda = (data_value >> sda_channel) & 1

            # Detect edges on SCL and SDA
            scl_last = self.scl_last_values[group_idx]
            sda_last = self.sda_last_values[group_idx]
            scl_edge = scl != scl_last
            sda_edge = sda != sda_last

            # State machine for I2C decoding
            state = self.states[group_idx]
            current_byte = self.current_bytes[group_idx]
            bit_count = self.bit_counts[group_idx]
            message = self.messages[group_idx]
            error_flag = self.error_flags[group_idx]

            # Retrieve stored sample indices
            addr_sample_idx = self.addr_sample_idxs[group_idx]
            ack_sample_idx = self.ack_sample_idxs[group_idx]
            data_sample_idx = self.data_sample_idxs[group_idx]
            stop_sample_idx = self.stop_sample_idxs[group_idx]

```

Figure 5652:I2C.py – Decode Start

```python
            if state == 'IDLE':
                if sda_edge and sda == 0 and scl == 1:
                    # Start condition detected
                    state = 'START'
                    current_byte = 0
                    bit_count = 0
                    message = []
                    error_flag = False
                    # Record the sample index for START
                    start_sample_idx = sample_idx
                    # Emit start condition immediately
                    self.decoded_message_ready.emit({
                        'group_idx': group_idx,
                        'event': 'START',
                        'sample_idx': start_sample_idx,
                    })
            elif state == 'START':
                if scl_edge and scl == 1:
                    if bit_count == 0:
                        # Record sample index at the start of address transmission
                        addr_sample_idx = sample_idx
                        self.addr_sample_idxs[group_idx] = addr_sample_idx
                    # Rising edge of SCL, sample SDA
                    current_byte = (current_byte << 1) | sda
                    bit_count += 1
                    if bit_count == expected_bits:
                        # Address byte received
                        if address_width == 7:
                            address = current_byte >> 1
                            rw_bit = current_byte & 1
                            message.append({'type': 'Address', 'data': address, 'rw': rw_bit})
                        else:
                            address = current_byte
                            rw_bit = None
                            message.append({'type': 'Address', 'data': address})
                        # Emit signal for address
                        self.decoded_message_ready.emit({
                            'group_idx': group_idx,
                            'event': 'ADDRESS',
                            'data': address,
                            'rw_bit': rw_bit,
                            'sample_idx': addr_sample_idx,  # Use recorded sample index
                        })
                        bit_count = 0
                        current_byte = 0
                        state = 'ACK'
                        # Reset address sample index
                        self.addr_sample_idxs[group_idx] = None
```

Figure 5753:I2C.py – Decode FSM – IDLE & START

```python
            elif state == 'ACK':
                if scl_edge and scl == 1:
                    # Record sample index at the start of ACK bit
                    ack_sample_idx = sample_idx
                    self.ack_sample_idxs[group_idx] = ack_sample_idx
                    # Sample ACK bit
                    ack = sda
                    message.append({'type': 'ACK', 'data': ack})
                    # Emit signal for ACK
                    self.decoded_message_ready.emit({
                        'group_idx': group_idx,
                        'event': 'ACK',
                        'data': ack,
                        'sample_idx': ack_sample_idx,
                    })
                    state = 'DATA'
                    # Reset ACK sample index
                    self.ack_sample_idxs[group_idx] = None
            elif state == 'DATA':
                if scl_edge and scl == 1:
                    if bit_count == 0:
                        # Record sample index at the start of data byte
                        data_sample_idx = sample_idx
                        self.data_sample_idxs[group_idx] = data_sample_idx
                    # Rising edge of SCL, sample SDA
                    current_byte = (current_byte << 1) | sda
                    bit_count += 1
                    if bit_count == 8:
                        # Data byte received
                        message.append({'type': 'Data', 'data': current_byte})
                        # Emit signal for DATA
                        self.decoded_message_ready.emit({
                            'group_idx': group_idx,
                            'event': 'DATA',
                            'data': current_byte,
                            'sample_idx': data_sample_idx,  # Use recorded sample index
                        })
                        bit_count = 0
                        current_byte = 0
                        state = 'ACK2'
                        # Reset data sample index
                        self.data_sample_idxs[group_idx] = None
            elif state == 'ACK2':
                if scl_edge and scl == 1:
                    # Record sample index at the start of ACK bit
                    ack_sample_idx = sample_idx
                    self.ack_sample_idxs[group_idx] = ack_sample_idx
                    # Sample ACK bit
                    ack = sda
                    message.append({'type': 'ACK', 'data': ack})
                    # Emit signal for ACK
                    self.decoded_message_ready.emit({
                        'group_idx': group_idx,
                        'event': 'ACK',
                        'data': ack,
                        'sample_idx': ack_sample_idx,
                    })
                    state = 'DATA'
                    # Reset ACK sample index
                    self.ack_sample_idxs[group_idx] = None
```

Figure 5854:I2C.py – Decode FSM – ACK, DATA, ACK2

```python
            if sda_edge and sda == 1 and scl == 1:
                # Stop condition detected
                stop_sample_idx = sample_idx  # Record sample index for STOP
                # Emit the decoded message
                self.decoded_message_ready.emit({
                    'group_idx': group_idx,
                    'event': 'STOP',
                    'message': message.copy(),
                    'sample_idx': stop_sample_idx,
                })
                # Reset state
                state = 'IDLE'
                current_byte = 0
                bit_count = 0
                message = []
                error_flag = False
                # Reset sample indices
                self.addr_sample_idxs[group_idx] = None
                self.ack_sample_idxs[group_idx] = None
                self.data_sample_idxs[group_idx] = None
                self.stop_sample_idxs[group_idx] = None

            # Update the stored states
            self.states[group_idx] = state
            self.current_bytes[group_idx] = current_byte
            self.bit_counts[group_idx] = bit_count
            self.messages[group_idx] = message
            self.error_flags[group_idx] = error_flag

            # Update last values
            self.scl_last_values[group_idx] = scl
            self.sda_last_values[group_idx] = sda

```

Figure 5955: I2C.py – Decode FSM – Idle Check

```python
    def reset_decoding_states(self) -> None:
        """
        Resets the I2C decoding state machines for all groups, clearing buffers and states.
        """
        # Reset I2C decoding variables for each group
        self.states = ['IDLE'] * len(self.group_configs)
        self.bit_buffers = [[] for _ in range(len(self.group_configs))]
        self.current_bytes = [0] * len(self.group_configs)
        self.bit_counts = [0] * len(self.group_configs)
        self.decoded_messages = [[] for _ in range(len(self.group_configs))]
        self.scl_last_values = [1] * len(self.group_configs)
        self.sda_last_values = [1] * len(self.group_configs)
        self.messages = [[] for _ in range(len(self.group_configs))]
        self.error_flags = [False] * len(self.group_configs)
        self.sample_idx = 0  # Reset sample index


    def stop_worker(self) -> None:
        """
        Stops the worker thread by setting the running flag to False and closing the serial port.
        """
        self.is_running = False
        if self.serial.is_open:
            self.serial.close()

```

Figure 6056:I2C.py – Serial Worker – reset decode & stop worker

```python
class I2CChannelButton(EditableButton):
    """
    I2CChannelButton is an EditableButton subclass specific to I2C channels. It emits additional
    signals for configuration and reset actions.

    Attributes:
        configure_requested (pyqtSignal): Signal emitted when the configure option is selected.
        reset_requested (pyqtSignal): Signal emitted when the reset to default option is selected.
    """

    configure_requested = pyqtSignal(int)  # Signal to notify when configure is requested
    reset_requested = pyqtSignal(int)      # Signal to notify when reset is requested

    def __init__(self, label: str, group_idx: int, parent: Optional[QWidget] = None) -> None:
        """
        Initializes the I2CChannelButton with a given label and group index.

        Args:
            label (str): The initial text label of the button.
            group_idx (int): The index of the I2C group this button represents.
            parent (QWidget, optional): The parent widget. Defaults to None.
        """
        super().__init__(label, parent)
        self.group_idx = group_idx  # Store the index of the I2C group

    def show_context_menu(self, position: QPoint) -> None:
        """
        Displays a context menu with options to rename the button, reset to default, or configure the group.

        Args:
            position (QPoint): The position where the context menu is requested.
        """
        menu = QMenu()
        rename_action = menu.addAction("Rename")
        reset_action = menu.addAction("Reset to Default")
        configure_action = menu.addAction("Configure")  # Add the Configure option
        action = menu.exec(self.mapToGlobal(position))
        if action == rename_action:
            new_label, ok = QInputDialog.getText(
                self, "Rename Button", "Enter new label:", text=self.text()
            )
            if ok and new_label:
                self.setText(new_label)
        elif action == reset_action:
            self.setText(self.default_label)
            self.reset_requested.emit(self.group_idx)  # Emit reset signal
        elif action == configure_action:
            self.configure_requested.emit(self.group_idx)  # Emit signal to open configuration dialog

```

Figure 6157:I2C.py – I2C Channel Button

```python
    def init_ui(self) -> None:
        """
        Sets up the user interface components of the configuration dialog.
        """
        layout = QVBoxLayout()

        # Clock Channel Selection
        clock_layout = QHBoxLayout()
        clock_label = QLabel("Clock Channel:")
        self.clock_combo = QComboBox()
        self.clock_combo.addItems([f"Channel {i+1}" for i in range(8)])
        self.clock_combo.setCurrentIndex(self.current_config.get('clock_channel', 2) - 1)
        clock_layout.addWidget(clock_label)
        clock_layout.addWidget(self.clock_combo)
        layout.addLayout(clock_layout)

        # Data Channel Selection
        data_layout = QHBoxLayout()
        data_label = QLabel("Data Channel:")
        self.data_combo = QComboBox()
        self.data_combo.addItems([f"Channel {i+1}" for i in range(8)])
        self.data_combo.setCurrentIndex(self.current_config.get('data_channel', 1) - 1)
        data_layout.addWidget(data_label)
        data_layout.addWidget(self.data_combo)
        layout.addLayout(data_layout)

        # Address Width Selection
        address_layout = QHBoxLayout()
        address_label = QLabel("Address Width:")
        self.address_group = QButtonGroup(self)
        self.address_7bit = QRadioButton("7 bits")
        self.address_8bit = QRadioButton("8 bits")
        self.address_group.addButton(self.address_7bit)
        self.address_group.addButton(self.address_8bit)
        address_layout.addWidget(address_label)
        address_layout.addWidget(self.address_7bit)
        address_layout.addWidget(self.address_8bit)
        layout.addLayout(address_layout)

        if self.current_config.get('address_width', 8) == 8:
            self.address_8bit.setChecked(True)
        else:
            self.address_7bit.setChecked(True)

        # Data Format Selection
        format_layout = QHBoxLayout()
        format_label = QLabel("Data Format:")
        self.format_combo = QComboBox()
        self.format_combo.addItems(["Binary", "Decimal", "Hexadecimal", "BCD", "ASCII"])
        self.format_combo.setCurrentText(self.current_config.get('data_format', 'Hexadecimal'))
        format_layout.addWidget(format_label)
        format_layout.addWidget(self.format_combo)
        layout.addLayout(format_layout)

        # Buttons
        button_layout = QHBoxLayout()
        ok_button = QPushButton("OK")
        cancel_button = QPushButton("Cancel")
        ok_button.clicked.connect(self.accept)
        cancel_button.clicked.connect(self.reject)
        button_layout.addWidget(ok_button)
        button_layout.addWidget(cancel_button)
        layout.addLayout(button_layout)

        self.setLayout(layout)
```

Figure 6258: I2C.py – I2C Config

```python
class I2CDisplay(QWidget):
    """
    I2CDisplay provides the main interface for displaying and interacting with I2C data.
    It includes graphical plots, control buttons, and configurations for multiple I2C groups.

    Attributes:
        period (int): The period for sample timing.
        num_samples (int): Number of samples to capture.
        port (str): Serial port for communication.
        baudrate (int): Baud rate for serial communication.
        channels (int): Number of channels for the logic analyzer.
        bufferSize (int): Size of the data buffer.
        data_buffer (List[deque]): Data buffers for each channel.
        sample_indices (deque): Sample indices buffer.
        total_samples (int): Total number of samples captured.
        is_single_capture (bool): Flag indicating if a single capture is active.
        current_trigger_modes (List[str]): Current trigger modes for each channel.
        trigger_mode_options (List[str]): Available trigger mode options.
        sample_rate (int): Sampling rate in Hz.
        group_configs (List[Dict]): Configuration settings for each I2C group.
        default_group_configs (List[Dict]): Default configuration settings for each I2C group.
        i2c_group_enabled (List[bool]): Flags indicating whether each I2C group is enabled.
        decoded_messages_per_group (Dict[int, List[str]]): Decoded messages for each I2C group.
        group_cursors (List[List[Dict]]): Cursors for each I2C group.
        setup_ui (method): Method to set up the user interface.
        timer (QTimer): Timer for updating the plot.
        is_reading (bool): Flag indicating if data reading is active.
        decoded_texts (List[QTextEdit]): Text edits for displaying decoded messages.
        worker (SerialWorker): Worker thread handling serial communication.
        group_curves (List[Dict[str, pg.PlotDataItem]]): Plot curves for SDA and SCL of each group.
        colors (List[str]): List of colors for plotting each group.
        channel_buttons (List[I2CChannelButton]): Buttons to toggle I2C group visibility and configuration.
        sda_trigger_mode_buttons (List[QPushButton]): Buttons to toggle trigger modes for SDA of each group.
        scl_trigger_mode_buttons (List[QPushButton]): Buttons to toggle trigger modes for SCL of each group.
        sample_rate_input (QLineEdit): Input field for sample rate.
        num_samples_input (QLineEdit): Input field for number of samples.
        toggle_button (QPushButton): Button to start/stop data acquisition.
        single_button (QPushButton): Button to initiate a single data capture.
    """
```

Figure 6359: I2C.py –I2CDisplay - DocString

```python
    def __init__(self, port: str, baudrate: int, bufferSize: int, channels: int = 8) -> None:
        """
        Initializes the I2CDisplay with the specified serial port parameters and sets up the UI.

        Args:
            port (str): Serial port for communication.
            baudrate (int): Baud rate for serial communication.
            bufferSize (int): Size of the data buffer.
            channels (int, optional): Number of channels for the logic analyzer. Defaults to 8.
        """
        super().__init__()
        self.period = 65454
        self.num_samples = 0
        self.port = port
        self.baudrate = baudrate
        self.channels = channels
        self.bufferSize = bufferSize

        self.data_buffer: List[deque] = [deque(maxlen=self.bufferSize) for _ in range(self.channels)]  # 8 channels
        self.sample_indices: deque = deque(maxlen=self.bufferSize)
        self.total_samples: int = 0

        self.is_single_capture: bool = False

        self.current_trigger_modes: List[str] = ['No Trigger'] * self.channels
        self.trigger_mode_options: List[str] = ['No Trigger', 'Rising Edge', 'Falling Edge']

        self.sample_rate: int = 1000  # Default sample rate in Hz

        # Initialize group configurations with default channels and address width
        self.group_configs: List[Dict] = [
            {'data_channel': 1, 'clock_channel': 2, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 3, 'clock_channel': 4, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 5, 'clock_channel': 6, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 7, 'clock_channel': 8, 'address_width': 8, 'data_format': 'Hexadecimal'},
        ]

        # Default group configurations for resetting
        self.default_group_configs: List[Dict] = [
            {'data_channel': 1, 'clock_channel': 2, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 3, 'clock_channel': 4, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 5, 'clock_channel': 6, 'address_width': 8, 'data_format': 'Hexadecimal'},
            {'data_channel': 7, 'clock_channel': 8, 'address_width': 8, 'data_format': 'Hexadecimal'},
        ]

        self.i2c_group_enabled: List[bool] = [False] * 4  # Track which I2C groups are enabled

        # Initialize decoded messages per group
        self.decoded_messages_per_group: Dict[int, List[str]] = {i: [] for i in range(4)}

        self.group_cursors: List[List[Dict]] = [[] for _ in range(4)]  # To store cursors per group

        self.setup_ui()
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_plot)

        self.is_reading: bool = False
        self.decoded_texts: List[QTextEdit] = []

        self.worker = SerialWorker(
            port=self.port,
            baudrate=self.baudrate,
            channels=self.channels,
            group_configs=self.group_configs
        )
        self.worker.data_ready.connect(self.handle_data_value)
        self.worker.decoded_message_ready.connect(self.display_decoded_message)
        self.worker.start()
```

Figure 6460: I2C.py –I2CDisplay - init

```python
    def send_stop_message(self) -> None:

        if self.worker.serial.is_open:
            try:
                self.worker.serial.write(b'1')
                time.sleep(0.001)
                self.worker.serial.write(b'1')
                time.sleep(0.001)
                self.worker.serial.write(b'1')
                print("Sent 'stop' command to device")
            except serial.SerialException as e:
                print(f"Failed to send 'stop' command: {str(e)}")
        else:
            print("Serial connection is not open")

    def start_reading(self) -> None:
        """
        Starts the data reading process by activating the timer.
        """
        if not self.is_reading:
            self.is_reading = True
            self.timer.start(1)

    def stop_reading(self) -> None:
        """
        Stops the data reading process by deactivating the timer.
        """
        if self.is_reading:
            self.is_reading = False
            self.timer.stop()

    def start_single_capture(self) -> None:
        """
        Initiates a single data capture by clearing existing data buffers, sending a start message,
        and starting the reading process. Disables relevant UI buttons during capture.
        """
        if not self.is_reading:
            self.clear_data_buffers()
            self.is_single_capture = True
            self.send_start_message()
            self.start_reading()
            self.single_button.setEnabled(False)
            self.toggle_button.setEnabled(False)
            self.single_button.setStyleSheet("background-color: #00FF77; color: black;")

    def stop_single_capture(self) -> None:
        """
        Stops a single data capture by stopping the reading process, sending a stop message,
        and re-enabling relevant UI buttons.
        """
        self.is_single_capture = False
        self.stop_reading()
        self.send_stop_message()
        self.single_button.setEnabled(True)
        self.toggle_button.setEnabled(True)
        self.toggle_button.setText("Start")
        self.single_button.setStyleSheet("")

    def clear_data_buffers(self) -> None:
        Clears all data buffers for each channel and removes all cursors from the plot.
        Also resets the worker's decoding states.
        """
        self.data_buffer = [deque(maxlen=self.bufferSize) for _ in range(self.channels)]
        self.total_samples = 0  # Reset total samples

        # Remove all cursors
        for group_idx in range(4):
            for cursor_info in self.group_cursors[group_idx]:
                self.plot.removeItem(cursor_info['line'])
                self.plot.removeItem(cursor_info['label'])
            self.group_cursors[group_idx] = []

        # Reset worker's decoding states
        self.worker.reset_decoding_states()

    def handle_data_value(self, data_value: int, sample_idx: int) -> None:
        """
        Handles incoming raw data emitted by the SerialWorker. Appends data to buffers and manages
        single capture logic.

        Args:
            data_value (int): The raw data value.
            sample_idx (int): The current sample index.
        """
        if self.is_reading:
            # Store raw data for plotting
            for i in range(self.channels):
                bit = (data_value >> i) & 1
                self.data_buffer[i].append(bit)
            self.total_samples += 1  # Increment total samples

            # Check if buffers are full
            if all(len(buf) >= self.bufferSize for buf in self.data_buffer):
                if self.is_single_capture:
                    # In single capture mode, stop acquisition
                    self.stop_single_capture()
                else:
                    # In continuous mode, reset buffers and cursors
                    self.clear_data_buffers()
                    self.clear_decoded_text()

    def display_decoded_message(self, decoded_data: Dict) -> None:
        """
        Displays a decoded I2C message by creating cursors and appending messages to the display.

        Args:
            decoded_data (Dict): A dictionary containing decoded message details.
        """
        group_idx: int = decoded_data.get('group_idx', -1)
        if group_idx == -1 or not self.i2c_group_enabled[group_idx]:
            return  # Do not display if the group is not enabled or invalid

        data_format: str = self.group_configs[group_idx].get('data_format', 'Hexadecimal')
        event: Optional[str] = decoded_data.get('event', None)
        sample_idx: Optional[int] = decoded_data.get('sample_idx', None)
```

Figure 6561:I2C.py – Start, Stop, handle data

```python
    def display_decoded_message(self, decoded_data: Dict) -> None:
        Displays a decoded I2C message by creating cursors and appending messages to the display.

        Args:
            decoded_data (Dict): A dictionary containing decoded message details.
        """
        group_idx: int = decoded_data.get('group_idx', -1)
        if group_idx == -1 or not self.i2c_group_enabled[group_idx]:
            return  # Do not display if the group is not enabled or invalid

        data_format: str = self.group_configs[group_idx].get('data_format', 'Hexadecimal')
        event: Optional[str] = decoded_data.get('event', None)
        sample_idx: Optional[int] = decoded_data.get('sample_idx', None)

        if event == 'START' and sample_idx is not None:
            # Create cursor for START condition
            self.create_cursor(group_idx, sample_idx, 'Start')
        elif event == 'ADDRESS':
            # Create cursor for Address
            address: int = decoded_data['data']
            rw_bit: Optional[int] = decoded_data.get('rw_bit', None)
            if data_format == 'Binary':
                addr_str = bin(address)
            elif data_format == 'Decimal':
                addr_str = str(address)
            elif data_format == 'Hexadecimal':
                addr_str = hex(address)
            elif data_format == 'ASCII':
                addr_str = chr(address)
            else:
                addr_str = hex(address)
            if rw_bit is not None:
                rw_str = 'Read' if rw_bit else 'Write'
                label_text = f"A:{addr_str} ({rw_str})"
            else:
                label_text = f"A:{addr_str}"
            self.create_cursor(group_idx, sample_idx, label_text)
        elif event == 'ACK':
            # Create cursor for ACK/NACK
            ack: int = decoded_data['data']
            ack_str: str = 'ACK' if ack == 0 else 'NACK'
            self.create_cursor(group_idx, sample_idx, ack_str)
        elif event == 'DATA':
            # Create cursor for Data byte
            data_byte: int = decoded_data['data']
            if data_format == 'Binary':
                data_str = bin(data_byte)
            elif data_format == 'Decimal':
                data_str = str(data_byte)
            elif data_format == 'Hexadecimal':
                data_str = hex(data_byte)
            elif data_format == 'ASCII':
                data_str = chr(data_byte)
            else:
                data_str = hex(data_byte)
            label_text = f"D:{data_str}"
            self.create_cursor(group_idx, sample_idx, label_text)
        elif event == 'STOP':
```

```python
    def display_decoded_message(self, decoded_data: Dict) -> None:
            elif data_format == 'Hexadecimal':
                data_str = hex(data_byte)
            elif data_format == 'ASCII':
                data_str = chr(data_byte)
            else:
                data_str = hex(data_byte)
            label_text = f"D:{data_str}"
            self.create_cursor(group_idx, sample_idx, label_text)
        elif event == 'STOP':
            # Create cursor for STOP condition
            self.create_cursor(group_idx, sample_idx, 'Stop')

            # Build message string
            message = decoded_data.get('message', [])
            message_str = ""
            for item in message:
                if item['type'] == 'Address':
                    addr: int = item['data']
                    rw_bit: Optional[int] = item.get('rw')
                    if data_format == 'Binary':
                        addr_str = bin(addr)
                    elif data_format == 'Decimal':
                        addr_str = str(addr)
                    elif data_format == 'Hexadecimal':
                        addr_str = hex(addr)
                    elif data_format == 'ASCII':
                        addr_str = chr(addr)
                    else:
                        addr_str = hex(addr)
                    if rw_bit is not None:
                        rw_str = 'Read' if rw_bit else 'Write'
                        message_str += f"Address: {addr_str} ({rw_str})\n"
                    else:
                        message_str += f"Address: {addr_str}\n"
                elif item['type'] == 'Data':
                    data_byte: int = item['data']
                    if data_format == 'Binary':
                        data_str = bin(data_byte)
                    elif data_format == 'Decimal':
                        data_str = str(data_byte)
                    elif data_format == 'Hexadecimal':
                        data_str = hex(data_byte)
                    elif data_format == 'ASCII':
                        data_str = chr(data_byte)
                    else:
                        data_str = hex(data_byte)
                    message_str += f"Data: {data_str}\n"
                elif item['type'] == 'ACK':
                    ack: int = item['data']
                    ack_str: str = 'ACK' if ack == 0 else 'NACK'
                    message_str += f"{ack_str}\n"

            message_str += "-" * 20 + "\n"

            # Append the message to the group's messages
            self.decoded_messages_per_group[group_idx].append(message_str)
```

Figure 6662:I2C.py – display decoded message

```python
    def create_cursor(self, group_idx: int, sample_idx: int, label_text: str) -> None:
        """
        Creates a visual cursor on the plot at the specified sample index with a label.

        Args:
            group_idx (int): The index of the I2C group (0-based).
            sample_idx (int): The sample index where the cursor should be placed.
            label_text (str): The text label to display alongside the cursor.
        """
        # Get base level for this group
        base_level = (4 - group_idx - 1) * 4  # Adjust as needed
        # Cursor color (keeping your tweaks)
        cursor_color = '#00F5FF'  # Use your preferred color
        # Create a vertical line segment between SDA and SCL levels
        y1 = base_level + 1
        y2 = base_level + 2
        x = 0  # Initial x position, will be updated in update_plot
        # Create line data
        line = pg.PlotDataItem([x, x], [y1, y2], pen=pg.mkPen(color=cursor_color, width=2))
        self.plot.addItem(line)
        # Add a label
        label = pg.TextItem(text=label_text, anchor=(0.1, 0.5), color=cursor_color)
        font = QFont("Arial", 12)
        label.setFont(font)
        self.plot.addItem(label)
        # Store the line, label, and sample index
        self.group_cursors[group_idx].append({
            'line': line,
            'label': label,
            'sample_idx': sample_idx,
            'base_level': base_level,
            'y1': y1,
            'y2': y2
        })

    def clear_decoded_text(self) -> None:
        """
        Clears all decoded text boxes and messages per group.
        """
        for idx in range(4):
            self.decoded_messages_per_group[idx].clear()
        # Cursors are already cleared in clear_data_buffers
```

Figure 6763:I2C.py – create cursor, clear_decoded_text

```python
    def closeEvent(self, event: Qt.QEvent) -> None:
        """
        Handles the close event of the I2CDisplay widget. Ensures that the worker thread is
        properly stopped before closing.

        Args:
            event (Qt.QEvent): The close event triggered when the widget is being closed.
        """
        self.worker.stop_worker()
        self.worker.quit()
        self.worker.wait()
        event.accept()

    def open_configuration_dialog(self, group_idx: int) -> None:
        """
        Opens the configuration dialog for a specific I2C group, allowing the user to update settings.

        Args:
            group_idx (int): The index of the I2C group to configure (0-based).
        """
        current_config = self.group_configs[group_idx]
        dialog = I2CConfigDialog(current_config, parent=self)
        if dialog.exec():
            new_config = dialog.get_configuration()
            self.group_configs[group_idx] = new_config
            print(f"Configuration for group {group_idx + 1} updated: {new_config}")
            # Update labels on the button to reflect new channel assignments
            sda_channel = new_config['data_channel']
            scl_channel = new_config['clock_channel']
            label = f"I2C {group_idx + 1}\nCh{sda_channel}:SDA\nCh{scl_channel}:SCL"
            self.channel_buttons[group_idx].setText(label)
            # Update trigger mode buttons
            self.sda_trigger_mode_buttons[group_idx].setText(f"SDA - {self.current_trigger_modes[sda_channel - 1]}")
            self.scl_trigger_mode_buttons[group_idx].setText(f"SCL - {self.current_trigger_modes[scl_channel - 1]}")
            # Update curves visibility
            is_checked = self.i2c_group_enabled[group_idx]
            sda_curve = self.group_curves[group_idx]['sda_curve']
            scl_curve = self.group_curves[group_idx]['scl_curve']
            sda_curve.setVisible(is_checked)
            scl_curve.setVisible(is_checked)
            # Clear data buffers
            self.clear_data_buffers()
            # Update worker's group configurations
            self.worker.group_configs = self.group_configs
```

Figure 6864: I2C.py – close Event & open_config_dialog

```python
"""
SPI.py

This module defines classes and functionalities related to handling SPI communication,
data processing, and graphical display for a Logic Analyzer application. It includes:

- SerialWorker: A QThread subclass that manages serial data reading, SPI decoding, and triggering mechanisms.
- FixedYViewBox: A custom PyQtGraph ViewBox that restricts scaling and translation on the Y-axis.
- EditableButton: A QPushButton subclass that allows for context menu operations like renaming and resetting.
- SPIChannelButton: An EditableButton subclass specific to SPI channels, with additional signals for configuration.
- SPIConfigDialog: A QDialog subclass that provides a user interface for configuring SPI channel settings.
- SPIDisplay: A QWidget subclass that provides the main interface for displaying and interacting with SPI data,
  including plotting, control buttons, and trigger configurations.

Dependencies:
- sys, serial, math, time, numpy, pyqtgraph
- PyQt6.QtWidgets, PyQt6.QtGui, PyQt6.QtCore
- collections.deque
- InterfaceCommands (custom module)
- aesthetic (custom module)
"""

import sys
import serial
import math
import time
from typing import List, Dict, Optional, Any

import numpy as np
import pyqtgraph as pg
from PyQt6.QtWidgets import (
    QWidget,
    QVBoxLayout,
    QHBoxLayout,
    QGridLayout,
    QInputDialog,
    QMenu,
    QPushButton,
    QLabel,
    QLineEdit,
    QComboBox,
    QDialog,
    QRadioButton,
    QButtonGroup,
    QSizePolicy,
)
from PyQt6.QtGui import QIcon, QIntValidator, QFont
from PyQt6.QtCore import QTimer, QThread, pyqtSignal, Qt, QPoint
from collections import deque

from InterfaceCommands import (
    get_trigger_edge_command,
    get_trigger_pins_command,
)
from aesthetic import get_icon


class SerialWorker(QThread):
    """
    SerialWorker handles SPI serial communication in a separate thread. It reads incoming data from
    the serial port, decodes SPI messages, processes trigger conditions for multiple SPI groups,
    and emits signals when data or decoded messages are ready for processing.

    Attributes:
        data_ready (pyqtSignal): Signal emitted when new raw data is ready. Carries an integer value and sample index.
        decoded_message_ready (pyqtSignal): Signal emitted when a decoded SPI message is ready. Carries a dictionary with message details.
        is_running (bool): Flag indicating whether the worker is active.
        channels (int): Number of channels to monitor for triggers.
        group_configs (List[Dict]): Configuration settings for each SPI group.
        trigger_modes (List[str]): List of trigger modes for each channel.
        states (List[str]): Current state of the state machine for each SPI group.
        current_bits_mosi (List[str]): Current bits collected on MOSI for each SPI group.
        current_bits_miso (List[str]): Current bits collected on MISO for each SPI group.
        last_clk_values (List[int]): Last sampled CLK values for edge detection.
        last_ss_values (List[int]): Last sampled SS values for edge detection.
        sample_idx (int): Global sample index counter.
    """

    data_ready = pyqtSignal(int, int)  # For raw data values and sample indices
    decoded_message_ready = pyqtSignal(dict)  # For decoded messages

    def __init__(
        self,
        port: str,
        baudrate: int,
        channels: int = 8,
        group_configs: Optional[List[Dict[str, Any]]] = None,
    ) -> None:
        """
        Initializes the SerialWorker thread with the specified serial port parameters and SPI group configurations.

        Args:
            port (str): The serial port to connect to (e.g., 'COM3', '/dev/ttyUSB0').
            baudrate (int): The baud rate for serial communication.
            channels (int, optional): The number of channels to monitor for triggers. Defaults to 8.
            group_configs (List[Dict[str, Any]], optional): Configuration settings for each SPI group. Defaults to None.
        """
        super().__init__()
        self.is_running: bool = True
        self.channels: int = channels
        self.group_configs: List[Dict[str, Any]] = group_configs if group_configs else [{} for _ in range(2)]
        self.trigger_modes: List[str] = ['No Trigger'] * self.channels
        self.sample_idx: int = 0  # Initialize sample index

        # Initialize SPI decoding variables for each group
        self.states: List[str] = ['IDLE'] * len(self.group_configs)
        self.current_bits_mosi: List[str] = [''] * len(self.group_configs)
        self.current_bits_miso: List[str] = [''] * len(self.group_configs)
        self.last_clk_values: List[int] = [0] * len(self.group_configs)
        self.last_ss_values: List[int] = [1] * len(self.group_configs)  # Assuming active low SS

        try:
            self.serial = serial.Serial(port, baudrate, timeout=0.1)
        except serial.SerialException as e:
            print(f"Failed to open serial port: {str(e)}")
            self.is_running = False
```

Figure 69: SPI.py – Serial Worker - init

```python
    def set_trigger_mode(self, channel_idx: int, mode: str) -> None:
        """
        Sets the trigger mode for a specific channel.

        Args:
            channel_idx (int): The index of the channel (0-based).
            mode (str): The trigger mode to set (e.g., 'No Trigger', 'Rising Edge', 'Falling Edge').
        """
        if 0 <= channel_idx < self.channels:
            self.trigger_modes[channel_idx] = mode
        else:
            print(f"Channel index {channel_idx} out of range.")

    def run(self) -> None:
        """
        The main loop of the worker thread. Continuously reads data from the serial port,
        processes SPI decoding, and emits data_ready and decoded_message_ready signals when appropriate.
        """
        while self.is_running:
            if self.serial.in_waiting:
                raw_data = self.serial.read(self.serial.in_waiting).splitlines()
                for line in raw_data:
                    try:
                        data_value = int(line.strip())
                        self.data_ready.emit(data_value, self.sample_idx)  # Emit data_value and sample_idx
                        self.decode_spi(data_value, self.sample_idx)
                        self.sample_idx += 1  # Increment sample index
                    except ValueError:
                        print(f"Invalid data received: {line.strip()}")
                        continue

```

```python
    def reset_decoding_states(self) -> None:
        """
        Resets the SPI decoding state machines for all groups, clearing buffers and states.
        """
        self.states = ['IDLE'] * len(self.group_configs)
        self.current_bits_mosi = [''] * len(self.group_configs)
        self.current_bits_miso = [''] * len(self.group_configs)
        self.last_clk_values = [0] * len(self.group_configs)
        self.last_ss_values = [1] * len(self.group_configs)
        self.sample_idx = 0  # Reset sample index

    def stop_worker(self) -> None:
        """
        Stops the worker thread by setting the running flag to False and closing the serial port.
        """
        self.is_running = False
        if self.serial.is_open:
            self.serial.close()
```

Figure 70: SPI.py65 – Serial Worker – Reset decode states and stop worker

```python
class SPIConfigDialog(QDialog):
    """
    SPIConfigDialog provides a user interface for configuring SPI group settings, including
    SS channel, CLK channel, MOSI channel, MISO channel, data bits, first bit order, SS active level, and data format.
    """

    def __init__(self, current_config: Dict[str, Any], parent: Optional[QWidget] = None) -> None:
        """
        Initializes the SPIConfigDialog with the current configuration.

        Args:
            current_config (Dict[str, Any]): The current configuration settings for the SPI group.
            parent (QWidget, optional): The parent widget. Defaults to None.
        """
        super().__init__(parent)
        self.setWindowTitle("SPI Configuration")
        self.current_config: Dict[str, Any] = current_config  # Dictionary to hold current configurations

        self.init_ui()
```

Figure 7166: SPI.py – SPI Config Dialog – init

```python
    def init_ui(self) -> None:
        """
        Sets up the user interface components of the configuration dialog.
        """
        layout = QVBoxLayout()

        # SS Channel Selection
        ss_layout = QHBoxLayout()
        ss_label = QLabel("Slave Select (SS) Channel:")
        self.ss_combo = QComboBox()
        self.ss_combo.addItems([f"Channel {i+1}" for i in range(8)])
        ss_channel = self.current_config.get('ss_channel', 1)
        self.ss_combo.setCurrentIndex(ss_channel - 1)
        ss_layout.addWidget(ss_label)
        ss_layout.addWidget(self.ss_combo)
        layout.addLayout(ss_layout)

        # SS Active Level
        ss_active_layout = QHBoxLayout()
        ss_active_label = QLabel("SS Active Level:")
        self.ss_active_group = QButtonGroup(self)
        self.ss_active_low = QRadioButton("Low")
        self.ss_active_high = QRadioButton("High")
        self.ss_active_group.addButton(self.ss_active_low)
        self.ss_active_group.addButton(self.ss_active_high)
        ss_active_layout.addWidget(ss_active_label)
        ss_active_layout.addWidget(self.ss_active_low)
        ss_active_layout.addWidget(self.ss_active_high)
        layout.addLayout(ss_active_layout)

        ss_active = self.current_config.get('ss_active', 'Low')
        if ss_active.lower() == 'low':
            self.ss_active_low.setChecked(True)
        else:
            self.ss_active_high.setChecked(True)

        # Clock Channel Selection
        clock_layout = QHBoxLayout()
        clock_label = QLabel("Clock (CLK) Channel:")
        self.clock_combo = QComboBox()
        self.clock_combo.addItems([f"Channel {i+1}" for i in range(8)])
        clk_channel = self.current_config.get('clock_channel', 2)
        self.clock_combo.setCurrentIndex(clk_channel - 1)
        clock_layout.addWidget(clock_label)
        clock_layout.addWidget(self.clock_combo)
        layout.addLayout(clock_layout)

        # MOSI Channel Selection
        mosi_layout = QHBoxLayout()
        mosi_label = QLabel("Master Out Slave In (MOSI) Channel:")
        self.mosi_combo = QComboBox()
        self.mosi_combo.addItems([f"Channel {i+1}" for i in range(8)])
        mosi_channel = self.current_config.get('mosi_channel', 3)
        self.mosi_combo.setCurrentIndex(mosi_channel - 1)
        mosi_layout.addWidget(mosi_label)
        mosi_layout.addWidget(self.mosi_combo)
        layout.addLayout(mosi_layout)

        # MISO Channel Selection
        miso_layout = QHBoxLayout()
        miso_label = QLabel("Master In Slave Out (MISO) Channel:")
        self.miso_combo = QComboBox()
        self.miso_combo.addItems([f"Channel {i+1}" for i in range(8)])
        miso_channel = self.current_config.get('miso_channel', 4)
        self.miso_combo.setCurrentIndex(miso_channel - 1)
        miso_layout.addWidget(miso_label)
        miso_layout.addWidget(self.miso_combo)
        layout.addLayout(miso_layout)

        # Bits Selection
```


```python
    def init_ui(self) -> None:
        mosi_layout = QHBoxLayout()
        mosi_label = QLabel("Master Out Slave In (MOSI) Channel:")
        self.mosi_combo = QComboBox()
        self.mosi_combo.addItems([f"Channel {i+1}" for i in range(8)])
        mosi_channel = self.current_config.get('mosi_channel', 3)
        self.mosi_combo.setCurrentIndex(mosi_channel - 1)
        mosi_layout.addWidget(mosi_label)
        mosi_layout.addWidget(self.mosi_combo)
        layout.addLayout(mosi_layout)

        # MISO Channel Selection
        miso_layout = QHBoxLayout()
        miso_label = QLabel("Master In Slave Out (MISO) Channel:")
        self.miso_combo = QComboBox()
        self.miso_combo.addItems([f"Channel {i+1}" for i in range(8)])
        miso_channel = self.current_config.get('miso_channel', 4)
        self.miso_combo.setCurrentIndex(miso_channel - 1)
        miso_layout.addWidget(miso_label)
        miso_layout.addWidget(self.miso_combo)
        layout.addLayout(miso_layout)

        # Bits Selection
```


```python
        bits_layout = QHBoxLayout()
        bits_label = QLabel("Data Bits:")
        self.bits_input = QLineEdit()
        self.bits_input.setValidator(QIntValidator(1, 32))
        self.bits_input.setText(str(self.current_config.get('bits', 8)))
        bits_layout.addWidget(bits_label)
        bits_layout.addWidget(self.bits_input)
        layout.addLayout(bits_layout)

        # First Bit Selection
        first_bit_layout = QHBoxLayout()
        first_bit_label = QLabel("First Bit Order:")
        self.first_bit_group = QButtonGroup(self)
        self.first_msb = QRadioButton("MSB")
        self.first_lsb = QRadioButton("LSB")
        self.first_bit_group.addButton(self.first_msb)
        self.first_bit_group.addButton(self.first_lsb)
        first_bit_layout.addWidget(first_bit_label)
        first_bit_layout.addWidget(self.first_msb)
        first_bit_layout.addWidget(self.first_lsb)
        layout.addLayout(first_bit_layout)

        first_bit = self.current_config.get('first_bit', 'MSB')
        if first_bit.upper() == 'MSB':
            self.first_msb.setChecked(True)
        else:
            self.first_lsb.setChecked(True)

        # Data Format Selection
        format_layout = QHBoxLayout()
        format_label = QLabel("Data Format:")
        self.format_combo = QComboBox()
        self.format_combo.addItems(["Binary", "Decimal", "Hexadecimal", "ASCII"])
        self.format_combo.setCurrentText(self.current_config.get('data_format', 'Hexadecimal'))
        format_layout.addWidget(format_label)
        format_layout.addWidget(self.format_combo)
        layout.addLayout(format_layout)

        # Buttons
        button_layout = QHBoxLayout()
        ok_button = QPushButton("OK")
        cancel_button = QPushButton("Cancel")
        ok_button.clicked.connect(self.accept)
        cancel_button.clicked.connect(self.reject)
        button_layout.addWidget(ok_button)
        button_layout.addWidget(cancel_button)
        layout.addLayout(button_layout)
```

Figure 7267: SPI.py – SPI Config Dialog – init_ui

```python
    def get_configuration(self) -> Dict[str, Any]:
        """
        Retrieves the updated configuration settings from the dialog.

        Returns:
            Dict[str, Any]: A dictionary containing the updated SPI group configuration.
        """
        return {
            'ss_channel': self.ss_combo.currentIndex() + 1,
            'ss_active': 'Low' if self.ss_active_low.isChecked() else 'High',
            'clock_channel': self.clock_combo.currentIndex() + 1,
            'mosi_channel': self.mosi_combo.currentIndex() + 1,
            'miso_channel': self.miso_combo.currentIndex() + 1,
            'bits': int(self.bits_input.text()),
            'first_bit': 'MSB' if self.first_msb.isChecked() else 'LSB',
            'data_format': self.format_combo.currentText(),
        }
```

Figure 7368: SPI.py – SPI Config Dialog – get configuration

```python
class SPIDisplay(QWidget):
    """
    SPIDisplay provides the main interface for displaying and interacting with SPI data.
    It includes graphical plots, control buttons, and configurations for multiple SPI groups.

    Attributes:
        period (int): The period for sample timing.
        num_samples (int): Number of samples to capture.
        port (str): Serial port for communication.
        baudrate (int): Baud rate for serial communication.
        channels (int): Number of channels for the logic analyzer.
        bufferSize (int): Size of the data buffer.
        data_buffer (List[deque]): Data buffers for each channel.
        sample_indices (deque): Sample indices buffer.
        total_samples (int): Total number of samples captured.
        is_single_capture (bool): Flag indicating if a single capture is active.
        current_trigger_modes (List[str]): Current trigger modes for each channel.
        trigger_mode_options (List[str]): Available trigger mode options.
        sample_rate (int): Sampling rate in Hz.
        group_configs (List[Dict]): Configuration settings for each SPI group.
        default_group_configs (List[Dict]): Default configuration settings for each SPI group.
        spi_group_enabled (List[bool]): Flags indicating whether each SPI group is enabled.
        decoded_messages_per_group (Dict[int, List[str]]): Decoded messages for each SPI group.
        group_cursors (List[List[Dict[str, Any]]]): Cursors for each SPI group.
        timer (QTimer): Timer for updating the plot.
        is_reading (bool): Flag indicating if data reading is active.
        worker (SerialWorker): Worker thread handling serial communication.
        group_curves (List[Dict[str, pg.PlotDataItem]]): Plot curves for SS, CLK, MOSI, and MISO of each group.
        colors (List[str]): List of colors for plotting each group.
        channel_buttons (List[SPIChannelButton]): Buttons to toggle SPI group visibility and configuration.
        ss_trigger_mode_buttons (List[QPushButton]): Buttons to toggle trigger modes for SS of each group.
        clk_trigger_mode_buttons (List[QPushButton]): Buttons to toggle trigger modes for CLK of each group.
        sample_rate_input (QLineEdit): Input field for sample rate.
        num_samples_input (QLineEdit): Input field for number of samples.
        toggle_button (QPushButton): Button to start/stop data acquisition.
        single_button (QPushButton): Button to initiate a single data capture.
    """
```

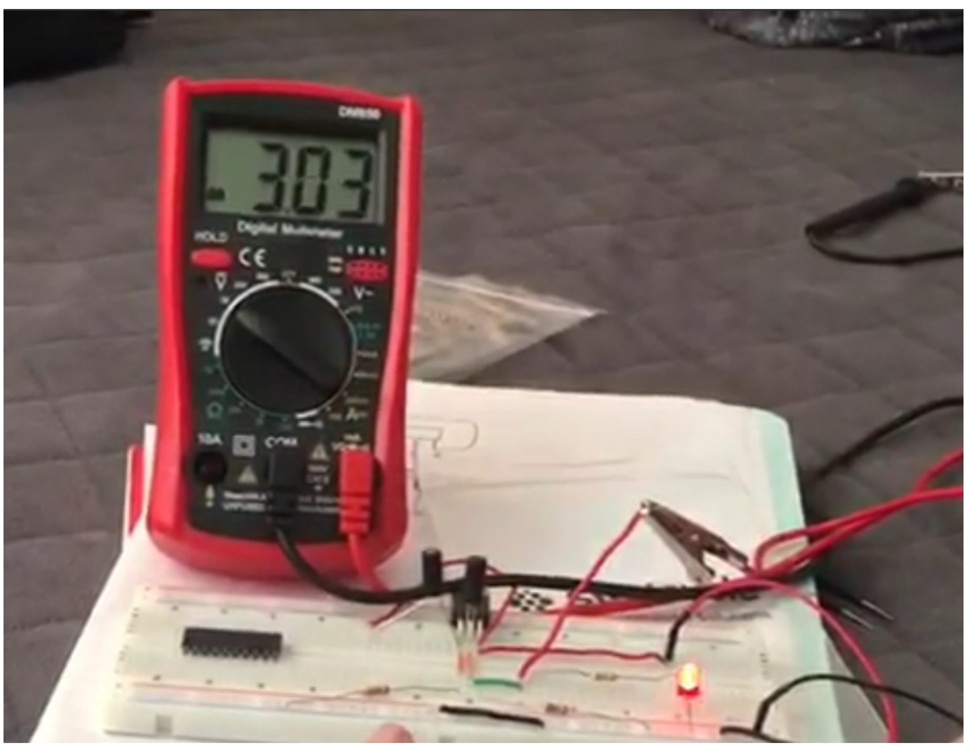
Figure 7469: SPI.py – SPI Display – DocString

6.2.3 PCB Testing

A printed circuit board (PCB) serves as both a voltage and data conduit, facilitating communication among different circuit components. It acts as a transmission hub that allows seamless data transfer between Microcontroller and the PC, while consistently supplying to all the peripheral components connected to it. PCB testing is essential to validate the data integrity and its validity. PCB test plan for the PCB includes:

Test 1: Verifying the circuit Logic output using Voltmeter.

This test aims to confirm the accuracy and functionality of the circuit by sampling its output using a digital Voltmeter. The data obtained from the voltmeter was analyzed to validate logic level high accuracy.



Test 2: Comparing output signal deviation using a known signal.

The test involves conducting functionality by applying a known input signal into the PCB. During the test the PCB circuit output will be analyzed verifying the output aligns with the expected result.

Figure 70: (Sending a known signal to Transceiver)

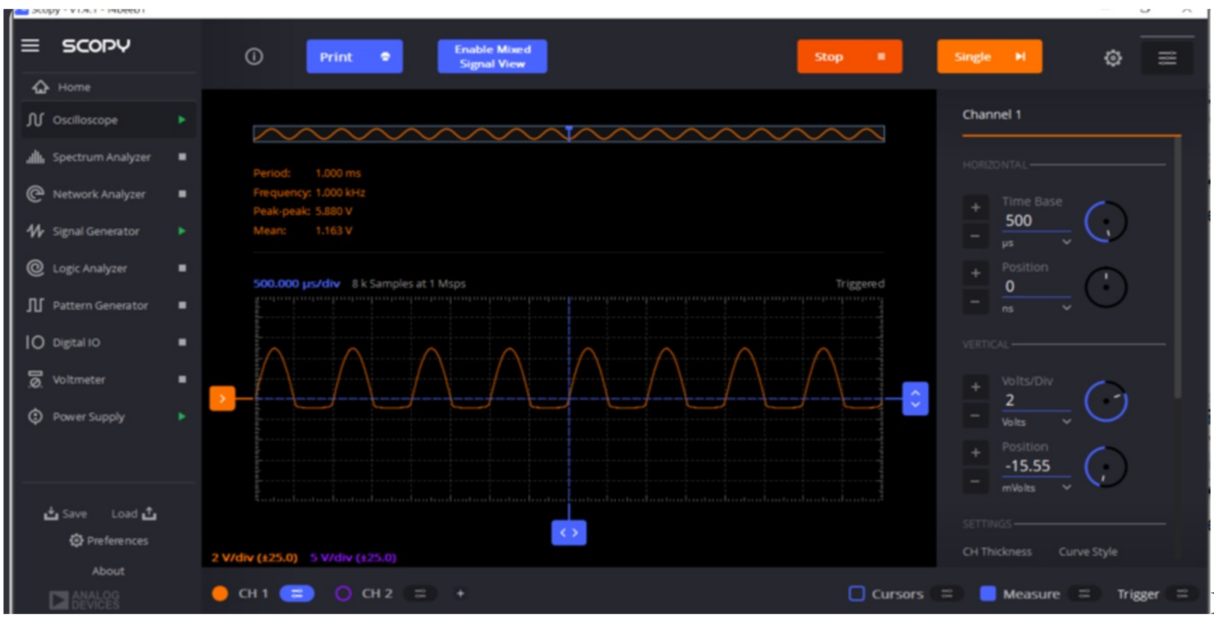


Figure 71: Received output signal through the Bus transceiver

7. Project Success Evaluation:

7.1: Overall Project Evaluation:

Overall, this project can be considered a success. GUI team successfully created an interface that is both user friendly and complements interfaces already in use by students today like Digilent Waveforms, or Scopy. This interface connects with MCU via USB communication protocol to communicate data from MCU to GUI and vice versa. From here GUI can process the data coming from MCU and plot the logic signal on the screens. Utilizing the data GUI can successfully decode I2C and SPI protocol. Unable to process UART signal decoding because our UART signal was not generated properly from other MCU.

7.2: Other issues:

- **Reason for the Project**: With the onset of the COVID-19 pandemic, demand for computer chips increased while production or supply decreased. The workforce and students alike were sent home to complete work from home. For electrical engineering students, their curriculum consists of a series of hardware lab classes where state of the art bench equipment is required to simulate and test hardware. However, due to the online nature of classes, the on-campus lab rooms were unavailable for use, forcing students to buy all-in-one USB devices to complete their experiments. Devices like the Analog Discovery 2 by Digilent or the ADALM2000 were recommended by ECE professors for use in these experiments. Currently the AD2 costs around $299, and the ADALM costs around $236; unaffordable to the average college student. This project sought to develop a cheaper alternative to these devices as a part of a three-team/project effort. In May of 2024, a group of students successfully developed a two-channel oscilloscope based off the STM32-F303RE MCU, the same one we are using. In December 2023, a group of students successfully developed AWG based off the same board. Our project solved the need for Logic analyzer component of the bigger device. Since all three components are based on the same MCU, there will be a group that combines our three projects, making a more affordable laboratory tool at around a $50 price range: significantly cheaper than the other alternatives on the market today.

- **Use Cases of the Project:** The main use cases of the project are undergraduate electrical and computer engineering students that are required to complete coursework. In addition, it can market to hobbyists and academic researchers where access to moderate electrical engineering bench equipment is needed.

- **Final Design Maintenance:** By creating a design case where the end user only has access to the USB connector, channel outputs, and external tap-in to the power supply; we eliminate any maintenance that will need to be done on the side of the hardware. The device will require minimal maintenance and if any is needed, they will be the replacement of the connectors from physical wear and tear. The GUI will periodically require updates of python packages, as updates to the GUI would be released via GitHub.
- **Life Cycle of Final Design:** All parts and components in this project were sourced

from common vendors like Digi key and Mouser, and replacements can be easily found online. Our Interface code will be open source and can be found easily on GitHub. Anyone can download and easily create our product. When it comes to the disposal of our device, it would be considered E-Waste and can be sent to any recycling center where other devices containing computer chips are sent. This project can be constantly improved and thus does not truly have an end of lifetime.

8: Administrative Section

8.1: Project Progress:

8.1.1: Front End:

The Front-End schematic was designed using KiCad and manufactured through JLPCB. The initial phase of the Front-End development involved researching suitable components that met the project's requirements, while also considering the original Front-End design from the AWG group. After completing the first schematic, we promptly had it manufactured to serve as a test board. Upon identifying issues with the Front-End, we decided to simulate the circuitry using a breadboard to validate our findings. Once the issues with the bus transceiver were confirmed and a proper replacement was identified, we proceeded with the development of the final prototype.

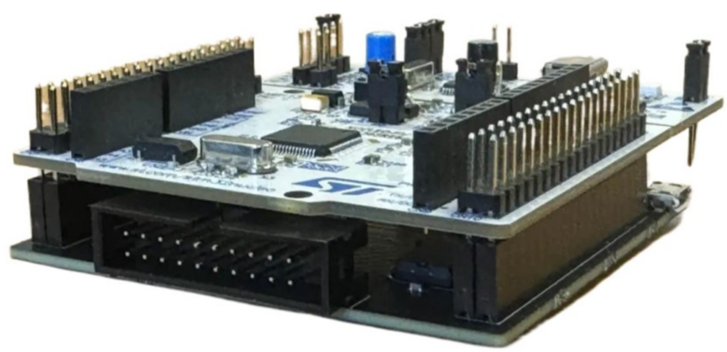


Figure 72: Final Prototype

8.1.2: MCU:

The MCU was configured using the STM32Cube Integrated Development Environment. During the summer, we took the time to familiarize ourselves with the MCU and its development environment, gaining an understanding of key components such as timers, RAM, USB communication, and interrupt service routines. We tested the USB protocol configuration and timers to facilitate data sampling from the MCU to the GUI. Additionally, we experimented with graphing logic signals from the MSP430, sampling the data through the STM32, and displaying it on the GUI. Once we successfully displayed signals on the GUI, we implemented additional functionalities, including command reception and determining the number of samples required after a trigger event. These features were thoroughly tested using basic logic signals from the MSP430, yielding exceptional results. After achieving these milestones, we transitioned to the Raspberry Pi to send signal to our MCU to decode properly.

8.1.3: GUI:

GUI was developed using Python. The first semester of Senior Design was dedicated to learning how to develop a GUI using PyQt6, where we would be able to have buttons that we can interact with and send UART messages between the MCU and GUI. The second semester began with us figuring out how to plot square waves on a graph using PyQtgraph. Once we were able to have the contents of the buffer displayed as square waves, we moved on to decoding the different communication protocols. The first one we were able to do was I2C, which was followed by SPI. Once we were able to successfully decode said signals, we needed to find a way to display the decoded text. We ended up using the cursor functionality within PyQtgraph where the label would be the decoded text. We couldn't get the timing down for the UART decoding since it is asynchronous.

8.2: Project Challenges:

8.2.1: Front End:

Upon completing the initial design of the Front End, we encountered an issue with the digital logic signal output. During the summer and at the beginning of ECE 493, we revisited the Front-End schematic, specifically the bus transceiver output, to ensure it produced the correct digital logic signals. After selecting the appropriate bus transceivers, we finalized the design by optimizing the placement of capacitors and resistors, removing any extraneous components.

Additionally, we also encountered the USB-PC connectivity issue, where the USB was not recognized by the PC. To address the issue of noise interference during data transfer we added a capacitor for noise cancellation and signal stabilization. Pull- up resistor was also adjusted to ensure the quality of the signal ensuring reliable USB connection. With these adjustments, the Front-End was successfully completed.

8.2.2: MCU:

Throughout the project's lifespan, we encountered multiple challenges, with understanding the MCU being the most significant hurdle, as we had no prior experience working with the STM32 in our academic curriculum. We had to familiarize ourselves with its code structure, timers, interrupt service routines (ISRs), and USB transmission and reception functions. The primary issue was establishing a reliable USB connection; the MCU failed to connect properly to the PC, repeatedly displaying driver-related errors. Another major challenge was configuring the timer's speed and determining the correct period and pre-scaler values to ensure accurate data sampling. Additionally, achieving a seamless USB connection through the front-end board proved problematic. We could not resolve this issue entirely and had to resort to workarounds, such as pressing the reset button or switching the JP5 jumper from U5V to E5V. Addressing this problem would require additional circuit modifications.

8.2.3: GUI:

PyQtgraph doesn't have an obvious to place text on the graphing window. This made displaying the decoding logic text on the graphing window difficult. There was a cursor functionality available within the PyQt library which would let us place a cursor anywhere on the graph. The important thing about this was that the cursor had a label which gave us a way to show text on the graph. The second major challenge had to do with UART decoding. Since this was an asynchronous communication protocol, the decoding had to be done without a clock. I wasn't able to successfully calculate the correct sampling rate given a baud rate.

8.3: Man, Hour Devoted to the project

- Sultan Alghamdi - MCU/GUI Code - 160 Hours
- Shahroz Shahbaz. - MCU/GUI Code - 171 Hours
- Furat Alhafez - 3D Design /RSCH - 160 Hours
- Sam Nepal - PCB Schematics - 180 Hours
- Julian Nigg - GUI/MCU Code - 170 Hours
- Thomas Senai - PCB Design - 167 Hours

Over the course of the project, our team collectively dedicated approximately 1008 hours, distributed among various members and different aspects of the project. This total includes regular team meetings and the documentation of our progress. A significant portion of the time was allocated to MCU programming, GUI design, and PCB construction. Specifically, we spent around 320 hours on the MCU development, 347 hours on the front-end design, and 341 hours on creating the logic analyzer interface (GUI).

8.4: Funds Spent

8.4.1: Front End – Version 1

Part Detail	Top Designator	Qty	Source	Ext. Price
IRF9530NSTRLPBF Extended C157638	Q1	2	JLCPCB	$1.48
CL10B223KB8NNNC Basic C21122	C2	2	JLCPCB	$0.01
KT-0603R Basic C2286	D1	2	JLCPCB	$0.01
0603WAF1004T5E Basic C22935	R1	2	JLCPCB	$0.00
10118194-0001LF Extended C132563	J1	5	JLCPCB	$0.92
0603WAF2200T5E Basic C22962	R2	2	JLCPCB	$0.00
0603WAF1201T5E Basic C22765	R5	2	JLCPCB	$0.00
FCC0603B203K500CT Extended C5137632	C1	20	JLCPCB	$0.06
74LCX245MTC Extended C719775	U1,U2	4	JLCPCB	$2.77

Figure 73: Cost of Parts for First PCB

8.4.2: Front End – Version 2

Part Detail	Top Designator	Qty	Source	Ext. Price
CC0603KRX7R9BB104 Basic C14663	C1,C2	4	JLCPCB	$0.01
KT-0603R Basic C2286	D2	2	JLCPCB	$0.01
0603WAF1004T5E Basic C22935	R1	2	JLCPCB	$0.00
10118194-0001LF Extended C132563	J1	5	JLCPCB	$0.91
0603WAF1001T5E Basic C21190	R3	2	JLCPCB	$0.00
0603WAF1201T5E Basic C22765	R5	2	JLCPCB	$0.00

Figure 744: Cost of Parts Second PCB

8.4.3: Front End – Version 3

Part Detail	Top Designator	Qty	Source	Ext. Price
CC0603KRX7R9BB104 Basic C14663	C1,C2,C4,C5	8	JLCPCB	$0.02
74HC245D,653 Extended C5625	U1,U3	4	JLCPCB	$0.85
0603WAF220JT5E Basic C23345	R2,R4	4	JLCPCB	$0.00
KT-0603R Basic C2286	D2	2	JLCPCB	$0.01
0603WAF1004T5E Basic C22935	R1,R10,R11,R12,R13,R14,R15,R16,R17,...	34	JLCPCB	$0.03
10118194-0001LF Extended C132563	J1	5	JLCPCB	$0.92
CL10A105KB8NNNC Basic C15849	C3	2	JLCPCB	$0.01
0603WAF1501T5E Basic C22843	R5	2	JLCPCB	$0.00
IRF9530NS-VB Extended C4355054	Q2	2	JLCPCB	$1.18
0603WAF1001T5E Basic C21190	R3	2	JLCPCB	$0.00

Figure 75: Cost of Parts for Final PCB

Part	Part Number	Quantity	Cost per Unit ($)	Total Cost ($)
20 Pin Connector Header	**30320-5002HB**	5	0.88	4.40
Female Pin Header 1x19 Pins for ESP32 Module	B0CFDYMRK2	20	0.49	9.99
8 Pin Single Row **Straight Female Header**	B07J5B9LT5	50	0.18	8.99

Figure 76: Additional Components

Note: For Figure 76, the last row of components was not utilized in the final Front End design.

8.4.4: MCU

Line Number	Mouser Part Number Customer Part Number Manufacturer Part Number Description	Requested Delivery Date(s)	Estimated Shipment Date(s)	Quantity	Unit Price (USD)	Extended Price (USD)
1	449-LFXTAL029665REEL [1] LFXTAL029665Reel 8MHz 16pF -10C 60C	FEB 27, 2024	FEB 27, 2024	5	0.580	2.90
2	80-CBR06C200J5GAUTO [1] CBR06C200J5GACAUTO 50Volt 20pF C0G 0.05	FEB 27, 2024	FEB 27, 2024	10	0.226	2.26
3	511-NUCLEO-F303RE [1] NUCLEO-F303RE STM32 Nucleo-64 deve	FEB 27, 2024	FEB 27, 2024	3	10.980	32.94
4	512-74LCX245MTC [1] 74LCX245MTC Bidirectional Trans	FEB 27, 2024	FEB 27, 2024	5	0.450	2.25
5	640-USB3080-30-00-A [1] USB3080-30-00-A Micro B Skt, Bottom-	FEB 27, 2024	FEB 27, 2024	3	0.710	2.13
	[1] RoHS: Compliant					
Shipping Notes			Merchandise Total (USD)			$42.48
			Shipping			$7.99
			Estimated Tax			$2.55

Figure 77: Total money spends for purchasing MCU and parts

8.4.5: Total Fund Spent

Total Spent on Project	$242.51
PCB	$140.79
MCU	$53.02
Additional Components	$48.07

8.4.6: Cost Per Unit

Project Requirements	$50
PCB + Parts	$13.64
MCU + Crystal Oscillator + 2 capacitor	$17.67
Device Cost	$31.31

8.5: Individual Team Member Contributions

8.5.1: Front End

- **Thomas Senai:** Created the schematic and PCB design within Kicad and soldered components onto the Front End.

- **Sam Nepal:** Selected the appropriate hardware component and created circuit blueprints needed for final PCB design. Developed a PCB prototype model and performed through testing to ensure circuit functionality meets the specified requirements.

8.5.2: MCU

- **Shahroz Shahbaz**: Created the setup for Py-serial USB communication between PC and MCU, using virtual comport. Also, setup the functions and ISR need to fill up the buffer and sample the data

- **Sultan Alghamdi**: Written code to test the functionality of USB communication. Implemented Received command code and trigger checking functionality.

Both were able to successfully write code that is needed to capture, sample, trigger check and transmission of data to the PC for visualization.

8.5.3: GUI

- **Julian Nigg**: Designed the interface, set up communication between the MCU and GUI, and did the signal decoding.

- **Sultan Alghamdi**: Set up the 3-byte commands which are being sent to the MCU so that the MCU knows when to start and stop sampling, which channels are active, and the speed at which to sample. converted the user's numeric choice of samples after the trigger condition is met by adjusting the prescalar and period of timer 16 through USB commands to the MCU.

- **Shahroz Shahbaz**: Wrote a basic GUI code to test out the graphing of some random logic signals.

8.5.4: 3D Design

- **Furat Alhafez**: Designed the final 3D-printed case for the logic analyzer, ensuring it securely housed the hardware components while incorporating ventilation and connectivity cutouts. Additionally, contributed to group research efforts to integrate all components of the project cohesively.

9. Lesson Learned

9.1: Additional Knowledge and Skills Learned

9.1.1: Front End

When developing the schematic for the front end, it was a valuable learning experience to understand how to manually create footprints for the Nucleo board. This was crucial, as the entire front-end structure depended on the accuracy of the footprint.

9.1.2: MCU

The STM32FR303RE uses the STM32CubeIDE application to code in C language or assembly language. It provides some helpful interactive graphical interfaces to view or change the various pin layout, timers, and other components on the board. The debugger features live expressions which were used to test various functionalities of the MCU. The debugger is a helpful tool that can be used when the GUI has not been developed to help visualize and debug the buffer. However, there are some limitations as the buffer size increases, it becomes more difficult to see irregularities. A GUI can be a very helpful tool in cleaning out any remaining bugs with the MCU. It can help visualize the change in sampling frequency through an increase or decrease of the width of the signal, the trigger functionality can be checked by plotting the buffer when the trigger is detected, and any noise or spikes in the buffer can be more easily spotted with a large buffer size.

9.1.3: GUI

The time dedicated to learning how to make a good-looking GUI could have been saved to by using a tool such as QT Design Studio, as opposed to coding the whole thing. The second thing; this project should have been developed using C instead of Python. The program runs incredibly slow on laptops (except for the M-Series MacBook Pro and Desktops), especially when all 8 channels are active. Synchronous communication protocols are relatively simple to implement since we can rely on the clock to make sure the data is accurate. This is not the case for UART were getting the timing down was really challenging.

9.2: Teaming Experience

At the start of our project, our team met twice a week. Early in the week, we held a meeting without our faculty supervisor to review our progress, and later in the week, we met to conduct research and work on the project. Our faculty supervisor provided guidance, offering solutions whenever we encountered challenges. In the second half of the project, we transitioned to having one official meeting with our faculty supervisor each week, while scheduling additional meetings as needed to focus on completing our work.

9.2.1: Project Sub-Teams

To ensure the success of our project, we divided our team into three groups based on individual skills and expertise. One group focused on the MCU, another on developing the GUI, and the third on designing the PCB. While each group had its primary responsibilities, team members were flexible and assisted in other areas as needed, depending on the project's requirements for that week.

9.2.2: Team Communication/Dynamics

Effective communication and team collaboration was a topmost priority for the project success. Regular meetings were conducted where each sub-team provided an update about their progress, shared the obstacles faced, aligned their work with the other teams, and documented any changes made. Additionally, collaborative tools like Microsoft SharePoint, google Docs, GitHub and other project management platforms were used for remote tasking, setting deadlines and providing real time feedback to each team member. While each project sub-team was focused on their specific area, the uses of the collaborative tool provided flexibility and support to each team when needed. This dynamic approach helped in project unity, allowing team members to address the challenges and complete the project efficiently.

9.2.3: Project Management/Schedule

To keep the team and project on track, we used several communication and project management tools. Our primary platform was a team Discord server, organized with separate channels for each project area to facilitate asynchronous discussions. The server also allowed for sharing images and data, which were essential for documentation. We utilized GitHub to manage all source code for the GUI, MCU, and the KiCad schematics for PCB designs. SharePoint was employed for collaborative work on deliverables such as presentations and documents. Finally, we held weekly face-to-face meetings in the Engineering building with Dr. Kaps. These meetings provided an opportunity to update him on our progress, receive constructive feedback, address technical challenges, and resolve any team dynamics issues.

10: References

1. *ADALM2000 Evaluation Board | Analog Devices*. https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/adalm2000.html#eb-overview. Accessed 8 Dec. 2024.
2. “Analog Discovery 2: 100MS/s USB Oscilloscope, Logic Analyzer and Variable Power Supply.” *Digilent*, https://digilent.com/shop/analog-discovery-2-100ms-s-usb-oscilloscope-logic-analyzer-and-variable-power-supply/. Accessed 8 Dec. 2024.
3. Duong, Phu, Luong, Duy, Alsaei, Jasem A. J. M, Phan, Bao, Nguyen, Giang & Nguyen, Thu Viet Minh. “Affordable USB Oscilloscope Final Project Report.” 5 May 2023.
4. “Graphical User Interface (GUI) Fundamentals.” *Mailchimp*, https://mailchimp.com/resources/graphical-user-interface/. Accessed 8 Dec. 2024.
5. “I2C vs SPI vs UART – Introduction and Comparison of Their Similarities and Differences.” *Total Phase Blog*, 1 Dec. 2021, https://www.totalphase.com/blog/2021/12/i2c-vs-spi-vs-uart-introduction-and-comparison-similarities-differences/.
6. “PyQt6 Tutorial 2024, Create Python GUIs with Qt.” *Python GUIs*, 6 Jan. 2021, https://www.pythonguis.com/pyqt6-tutorial/.
7. “Saleae Logic 8.” *Saleae, Inc.*, https://www.saleae.com/products/saleae-logic-8. Accessed 8 Dec. 2024.
8. *STM32CubeIDE - Integrated Development Environment for STM32 - STMicroelectronics*. https://www.st.com/en/development-tools/stm32cubeide.html. Accessed 8 Dec. 2024.
9. *USB Logic Analyzer - 24MHz/8-Channel - TOL-18627 - SparkFun Electronics*. https://www.sparkfun.com/products/18627. Accessed 8 Dec. 2024.
10. “What Is a Microcontroller? | Definition from TechTarget.” *Search IoT*, https://www.techtarget.com/iotagenda/definition/microcontroller. Accessed 8 Dec. 2024.