University of Michigan

Image-Guided Medical Robotics Lab

ROB590 Directed Study

LABORATORY REPORT

Developing a digital interface to a ScannerMAX galvanometer controller

---

Student Name	**Student ID**
Kyoungmo Koo	08125025

Directed by:

Professor Mark Draelos

Submission Date : 01/01/2024

1 Introduction

In the field of Optical Coherence Tomography (OCT), precision and reliability in scanning are paramount. The Galvanometer Scanner stands as the most widely implemented scanner in this domain, noted for its accuracy and efficiency. This laboratory report details an experiment conducted using the ScannerMAX galvanometer controller, chosen for its advanced features and compatibility with diverse control inputs. Traditionally, galvanometer scanners are controlled via analog signals, with the servo driver's analog command input orchestrating the movement. However, such analog systems are susceptible to noise and disturbances, which can arise from various sources, including communication channels.

To enhance the robustness of the system and mitigate the influence of noise, this experiment pioneers the use of the servo driver's digital channel. The FB4 protocol, a digital communication standard comprising Clock, FrameSync, and Data lines, serves as the backbone for this digital interface. The FB4 protocol's design parallels the Serial Audio Interface (SAI) protocol, commonly employed for audio data exchange. Initial attempts to utilize the Serial Peripheral Interface (SPI) were abandoned in favor of FB4 due to protocol incompatibilities, which will be further explored in section 2 of this report.

The transition to digital control necessitated a shift from the conventional Arduino microprocessor to the more capable NUCLEO-L476RG, powered by an STM32 board. This change was dictated by the Arduino's limitations in handling the complex requirements of the SAI protocol. The STM32 interface facilitated several crucial adjustments to the system. Among these were the activation of SAI channel blocks and the precise tuning of internal clock frequencies, enabling the system to support communication speeds up to 192 kHz.

Continuing from the foundational steps of enhancing the OCT scanning system with a robust digital communication protocol, the experimental setup involved a circuit integration of the microprocessor and the servo driver. This setup enabled communication with the servo driver via the SAI protocol, which was pivotal for the system's real-time scanning capability. Concurrently, a communication channel utilizing the Universal Asynchronous Receiver-Transmitter (UART) was established between the laptop and the STM32 board, ensuring a reliable interface for command and control.

To achieve seamless and continuous data exchange, an interrupt-driven approach was employed. The use of interrupts allowed for real-time data handling without the need for constant polling by the microprocessor. To address the inherent limitations of a single-core microprocessor, Direct Memory Access (DMA) channels were implemented. These channels function independently of the core processor, thus facilitating parallel data transmission and reception without overburdening the CPU.

The precise synchronization of these communication protocols is critical, especially in the context of real-time scanning where timing is essential. The implementation of a timing diagram ensured the meticulous coordination of all activities, including memory allocation and the sequential operation of callback functions triggered by interrupts. These functions were strategically deployed to manage the activities without memory overflow.

A notable challenge encountered during the experiment was the high-speed data transmission demands of the UART channel. The system had to be optimized to handle these speeds efficiently, which is a focal point of analysis in this report.

In summary, this report will delve into the intricacies of the experimental setup in section 2, elucidate the methodology in section 3, and present the resulting data in section 4. A comprehensive discussion on the outcomes and their implications will follow in section 5, with the final conclusions drawn in section 6, culminating the findings of this experiment.

2 Experimental Setup

1. NUCLEO-L476RG (STM32 board) Configuration

   (a) Parameter settings

   Figure 1 and 2 show the parameter settings of SAI protocol and UART channel.

   In configuration, two SAI blocks are employed and synchronized. The SAI Block A operates as the master controller of the SAI channel, while Block B functions as the slave. The slots for both blocks are enabled, with the data size set to 32 bits. The number of slots is configured to 1 to accommodate the 16-bit data for both the X and Y positions, and the remaining parameters are retained at their default values.

   For the UART configuration, the global interrupt is enabled, permitting interrupt calls from the main function to manage the interrupt routines.

   The DMA settings are distinct for the UART and SAI. The SAI blocks are set to "circular" mode, facilitating continuous data transmission and reception. In contrast, the UART's data transmission is set to "normal" mode to allow transmissions at specific times. The receiving channel of the UART, however, is configured to "circular" mode to maintain constant data reception.

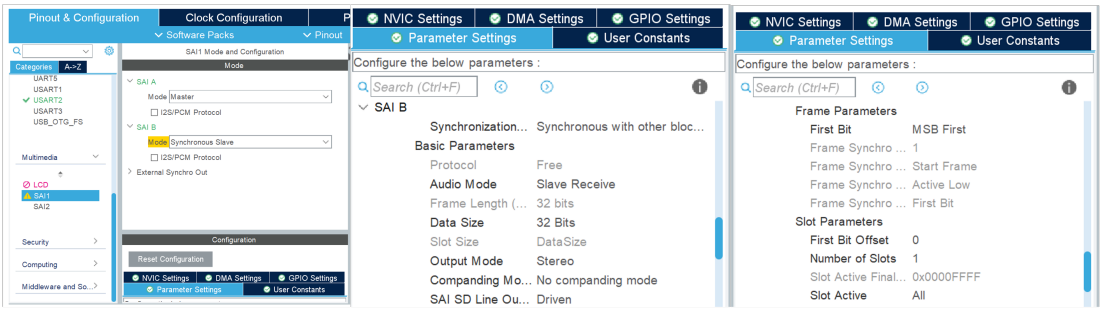

Figure 1: Parameter configuration for SAI protocol

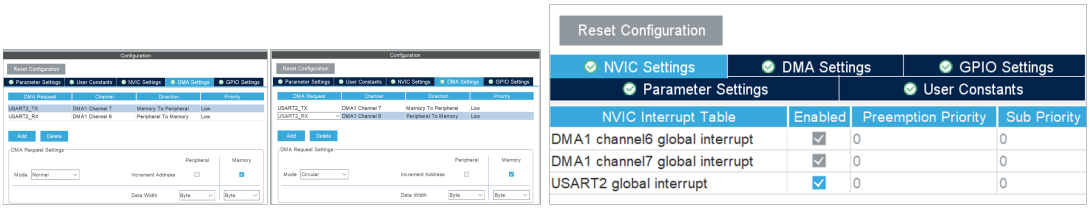

Figure 2: Parameter configuration for UART Channel

   (b) Peripheral pins

   The peripheral pins of the STM32 board are depicted in fig. 3. The FrameSync, Clock, and Data lines for the SAI protocol have been appropriately assigned, facilitating the establishment of the digital circuit. In addition, the data lines for the UART channel have been defined. However, for this experiment, data transmission and reception are conducted via the micro USB port on the STM32 board, thus the aforementioned peripheral pins are not utilized. This strategy appeared to be the problem of the experiment, and will be discussed in details in section 5. All pins have been configured to their default settings by the STM32CubeIDE.

   (c) Clock

   The final frequency transmitted to the SAI pin is determined through a process of multiplication and division, utilizing the PLL (Phase-Locked Loop) of the SAI to

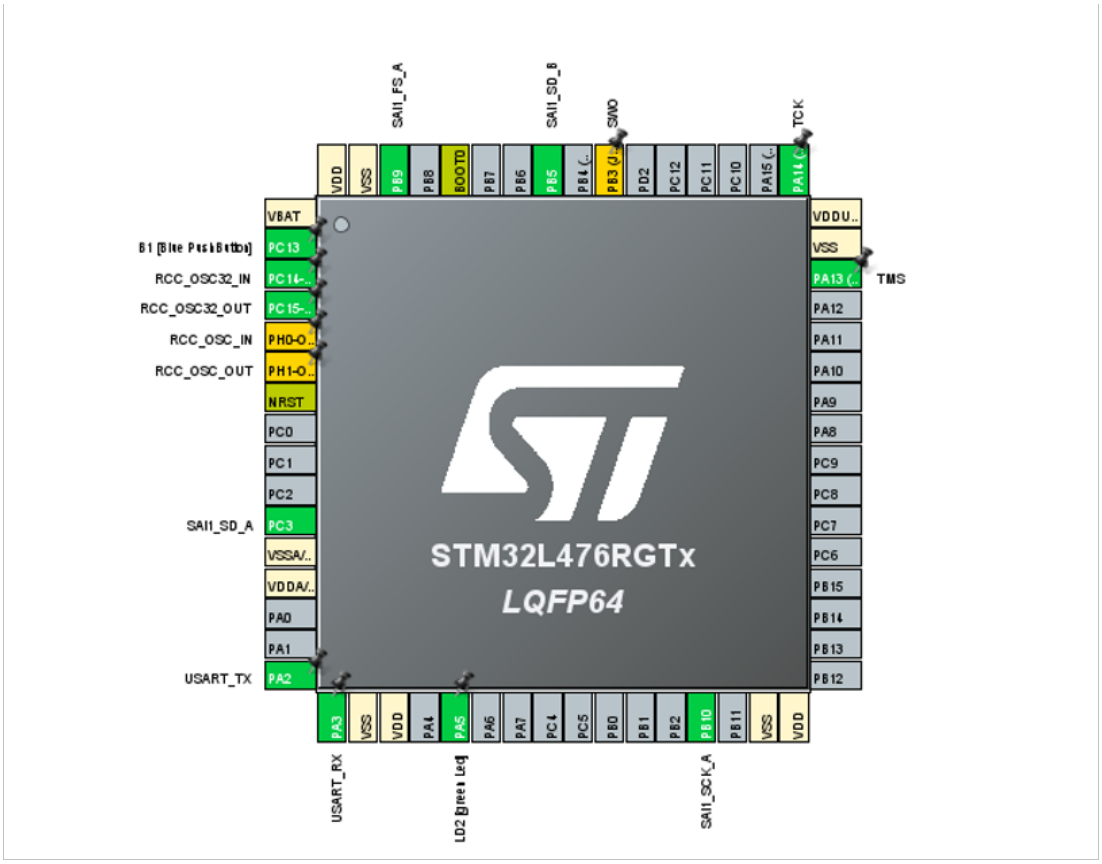

Figure 3: Peripheral pins of nucleo64-L476RG

achieve the desired adjustments. The maximum frequency capability of the PLL SAI is computed using the following equation 1 and 2. Here, the HSI RC stands for the High-Speed Internal Resistor-Capacitor Oscillator, which serves as the source for the Output Clock Frequency. Users have the capability to fine-tune this frequency by altering the Multiplication Factor (MF) and the Division Factor (DF) accordingly. The Output Frame Frequency, in turn, is calculated by dividing the Output Clock Frequency by 256. It is noteworthy that the Output Clock Frequency(Internal) differs from the frequency utilized by the SAI block. This discrepancy arises because, for the purposes of our experiment, only a single slot is active within the SAI block configuration. Figure 4 shows the clock configuration for peripheral SAI.

$$\text{Output Clock Frequency} = \frac{\text{HSI RC}}{2} * \frac{MF}{DF} \tag{1}$$

$$\text{Output Frame Frequency(Internal)} = \frac{\text{Output Clock Frequency}}{256} \tag{2}$$

2. Servo Driver

In the servo driver system, the digital command input section facilitates digital communication between the STM32 board and the servo driver. The FB4 protocol, which enables bidirectional communication including position feedback from the servo driver, is activated through a firmware update of the MachDSP software. MachDSP serves as the user interface for the driver and must be updated to its latest version to utilize the FB4 protocol effectively. Specifically, pin 3 of the servo driver is designated for transmitting position feedback from the perspective of the servo driver. Pin 4, on the other hand, is assigned to receive scan pattern inputs. Additionally, the FrameSync and Clock signals are received through pins 5 and 6, respectively. This digital circuitry configuration enables bidirectional communication between the Serial Audio Interface (SAI) transmission blocks of the STM32 boards and the digital command input section of the servo driver.

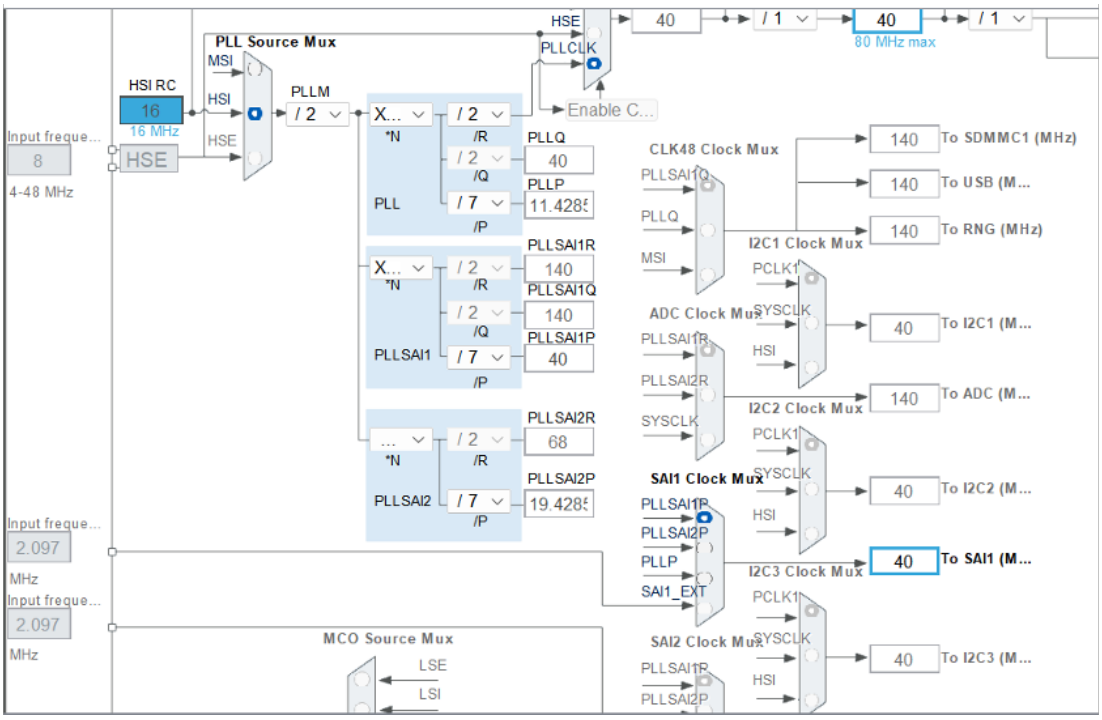

Figure 4: Clock configuration for peripheral SAI.

Communication flows in two directions: from the STM32 to the servo driver, it transmits scan patterns; and from the servo driver to the STM32, it sends position feedback.

3. MachDSP & PuTTY (Debuggers)

   MachDSP and PuTTY are used for debugging the code of STM32 board. The figures regarding the results from MachDSP will be discussed in the section 5.

   The MachDSP software displays a graphical representation of the scan pattern input from the laptop, which is instrumental in determining the presence of noise within the scan pattern. For instance, initial tests confirmed the incompatibility of the SPI protocol with the FB4 standard, whereas the SAI protocol was found to be compatible. Additionally, the frame frequency transmitted from the STM32 board to the servo driver can be monitored. Within the MachDSP interface, the inputs for the X and Y axes are available for separate examination, allowing for a detailed analysis of each component's signal integrity.

   PuTTY serves as the debugging tool for examining the STM32 code via a serial port channel. By aligning the baud rate with that specified in the STM32 code, it becomes possible to capture signals sent through the UART or USART channels — analogous to using the serial monitor in Visual Studio Code. This setup enabled us to monitor the position feedback signals from the servo driver through the STM32 board. Additionally, we could ascertain the maximum data rate that could be transmitted smoothly without overloading the system. For experiments necessitating the upload of a text file containing the scan pattern, TeraTerm was the preferred utility.

4. Arduino

   Initially, the Arduino ESP32 Nano was employed as the microprocessor for the digital communication circuit. While the Arduino platform offered greater ease of use, its capabilities were found to be limited for our requirements. In the prototyping phase, the Arduino was utilized with the SPI protocol. However, the complexity significantly increased when attempting to implement the SAI protocol, DMA channels, and interrupt functions. Due to these challenges, the Nucleo 64-L476RG board from the STM32 series was chosen as a more suitable alternative to the Arduino for this project.

3 Methodology

1. Communication Diagram

   Figure 5 shows the communication diagram between computer, STM32 board, servo driver, and MachDSP software.

   (a) Computer - STM32 board

   The computer, utilizing Pyserial library of Python, generates a scan pattern (scan position) and transmits it to the STM32 board. In return, the STM32 board sends position feedback to the computer, enabling the user to monitor the motor's position accurately. This bidirectional communication is facilitated through the UART interface.

   (b) STM32 board - Servo Driver

   The scan position, inputted into the STM32 board, is subsequently relayed to the Servo Driver. In turn, the position feedback, as measured by the Servo Driver, is transmitted back to the STM32 board. This two-way communication is facilitated through the SAI interface.

   (c) Servo Driver - MachDSP software

   The MachDSP software, interfacing with the servo driver, presents a command display to the user. This feature enables users to debug and verify whether commands are correctly transmitted to the servo driver. Additionally, MachDSP allows users to adjust parameters for PID (Proportional-Integral-Derivative) control, facilitating the optimization of motor control.

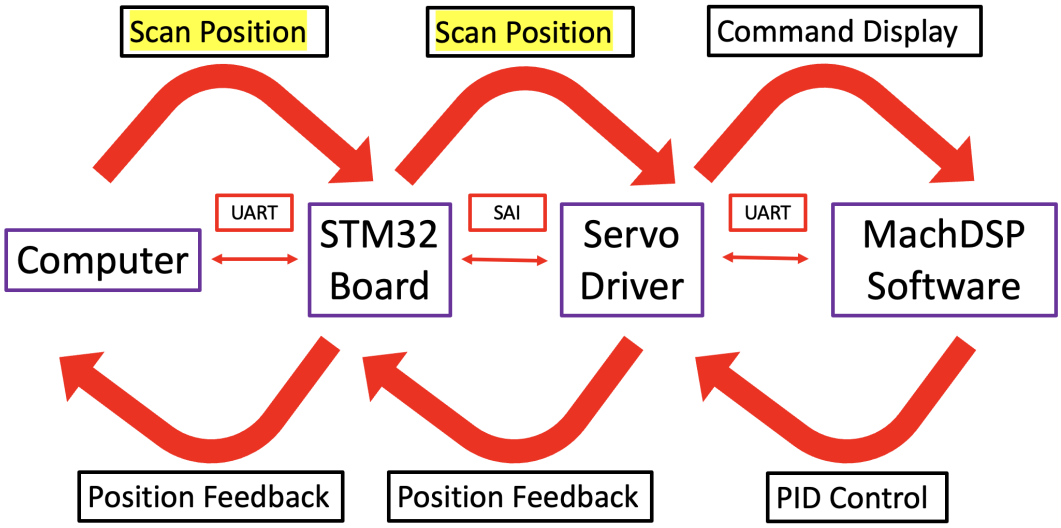

Figure 5: Communication diagram of the experiment

2. Timing Diagram

   Figure 6 illustrates the timing diagram involving the computer, STM32 board, servo driver, and MachDSP software. Given the data transmission limitations over UART, particularly with the ST-Link version we are using, which has an internal clock frequency of 4 MHz, we opted for an initial frame frequency of 8 kHz for the prototype. This approach allows for potential frequency increases in subsequent iterations.

For our experiments, we employed a baud rate of 0.8 Mbps. Based on the baud rate definition, the estimated time required for transmitting a buffer size of 500 uint32_t is approximately 20 ms. The transmission and reception of each SAI block segment take about 62.5 ms. This duration ensures that the timing diagram remains coherent. Furthermore, half of the buffer, which is not being transmitted during a specific segment, is fully updated before its transmission. Thus, we can seamlessly send the updated portion of the buffer in the next segment without any timing issues.

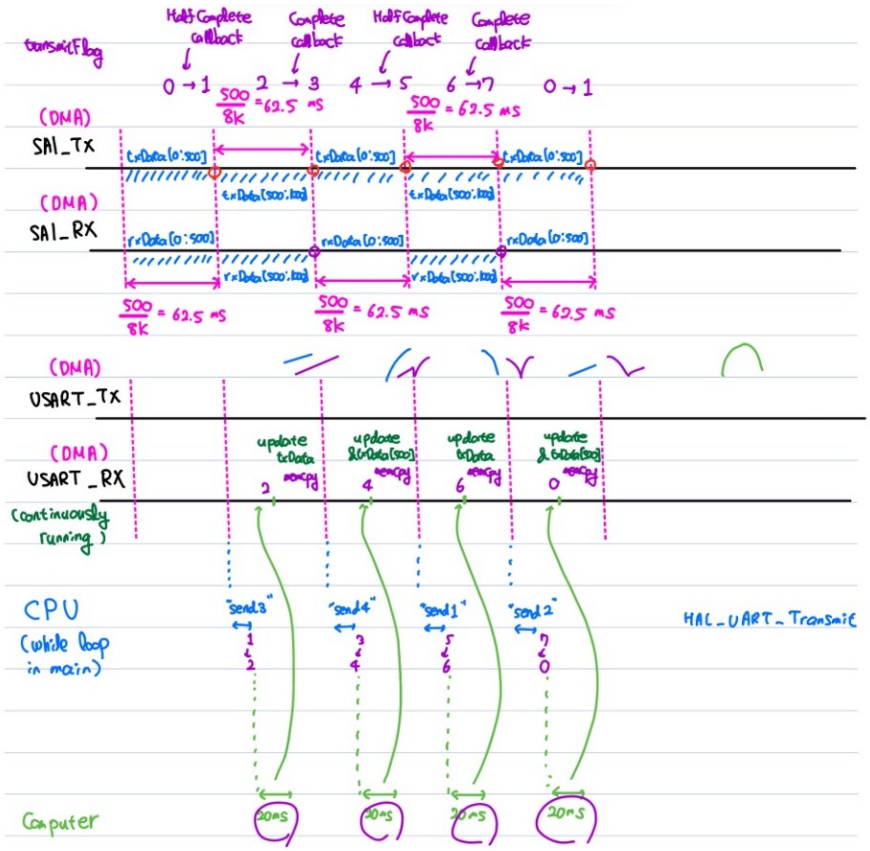

Figure 6: Timing diagram of the experiment

Timing diagram operates as follows:

(a) SAI TX / SAI RX

In the setup described, the first half of the SAI_TX buffer is transmitted to the servo driver using SAI Block A, while simultaneously, the first half of the SAI_RX buffer(buffer used for storing position feedback) is received from the servo driver via SAI Block B. These operations are conducted independently through DMA (Direct Memory Access) channels. The duration for these processes is calculated as 500/8000 = 62.5 milliseconds. Upon successful transmission and reception, the TransmitFlag variable is updated from 0 to 1. Subsequently, a message with the content "1" is sent to the computer.

(b) State change & callback function called

Following the successful transmission of the message, the computer sends a new scan pattern to update the first half of the SAI_TX buffer via the UART channel. If the previously sent pattern was a sinusoidal pattern, a ramping pattern is sent; if it was a ramping pattern, a sinusoidal pattern is sent in reverse. Upon receiving new scan pattern via UART channel, the TransmitFlag variable is updated from 1 to 2. The data received via the UART channel is then copied into the buffer using the memcpy function. The time taken by the memcpy operation within the STM32 board is negligible when compared to the duration required for data transmission over the

UART channel.

(c) SAI TX / SAI RX

Similar to scenario 2a, the TransmitFlag variable is now updated to 3, and a message "2" is sent to the computer. Concurrently, the SAI_RX complete callback function is invoked, initiating the transmission of the rxData buffer via the SPI DMA channel. The choice of SPI over USART for this operation is due to issues that will be elaborated in the section 5. But, this transmission strategy, involving the entire buffer, poses challenges because data updates and transmissions in the front part of the buffer occur simultaneously. This overlap can lead to the transmission of unexpected values. To mitigate this issue, for the new timing diagram in section 5, fig. 9, the TX part of the operation is divided in half, ensuring a more controlled and sequential data handling process.

(d) State change & callback function called

After the message is transmitted, the same procedure as in step 2b is executed.

These steps, 2a through 2d, are performed repetitively. To ensure accurate system performance, two distinct scan patterns are utilized: a sinusoidal pattern stored in the TxSine[1000] buffer and a ramping pattern in the TxStep[1000] buffer. Alternating between these patterns is crucial, as relying on a single pattern would not adequately verify the system's ability to update correctly.
Both the SAI_TX and SAI_RX channels are set to operate in circular DMA mode. This configuration allows for continuous data transmission and reception using the respective buffers. Similarly, USART_RX is also configured as circular DMA to constantly monitor incoming scan patterns from the computer. In contrast, SPI_TX is set as normal DMA, triggered only at specific times by a callback function, reflecting its requirement for operation only at certain intervals.

3. Codes

Codes for STM32 board is written in C language. The code for laptop is written in Python language, especially using Pyserial language for communication via UART channel. The code of STM32 board is modified in main.c and stm32l4xx_hal_sai.c from default code of STM32 CUBEIDE.

(a) main.c

- **Modification of Callback Functions:** The callback functions are adapted to update and transfer the `TransmitFlag` variable according to the current state and the specific callback function invoked.
- **Handling HAL_SAI_RxCpltCallback:** When `HAL_SAI_RxCpltCallback` is called (the RX complete callback for SAI), data is transmitted through the SPI DMA channel.
- **Handling HAL_UART_RxCpltCallback:** Upon triggering of `HAL_UART_RxCpltCallback` (indicating receipt of a new scan pattern via the UART channel), the data is updated in accordance with the `TransmitFlag` variable.
- **Continuous Circular DMA Operations:** The functions `SAI_TransmitReceive` and `HAL_UART_Receive_DMA` are executed for ongoing circular DMA activities for both SAI and UART.
- **While Loop for TransmitFlag Management:** A while loop is employed to modify the `TransmitFlag`, updating its status. It also sends a message to the computer, conveying the current state of the `TransmitFlag`.

(b) stm32l4xx_hal_sai.c

`SAI_TransmitReceive`, which calls `HAL_SAI_Transmit_DMA` and `HAL_SAI_Receive_DMA` is written.

(c) PySerial

Write binary data for sine position and step position into respective buffers. Next, reset the input buffer before receiving a message from the STM32 board. After receiving the data, decode it to determine the current state of the TransmitFlag. Finally, transmit appropriate binary data through the serial port.

Details of code can be found in GitHub Repository for STM32 Project

4. Results

Figure 7 displays the MachDSP screen, illustrating the raw command input received from the STM32 board. This screen indicates the alternating appearance of both sine scan and ramping scan patterns, demonstrating the effectiveness of our strategy in using a variable TransmitFlag to manage communication between the STM32 board and the Servo Driver. Furthermore, the timing diagram remains orderly, with each activity being completed prior to its respective deadline.

Figure 7: Sine scan pattern and ramping scan pattern of X position appearing alternatively on MachDSP software

Figure 8 presents the clock and data signals from the SPI channels, transmitting the position feedback of the scanner. The SPI's functionality has been confirmed to operate seamlessly within its designated clock frequency via the DMA channel. The empty segments in channels D4 and D5 indicate that the SPI transmission is significantly faster compared to the SAI transmission.

Figure 7 and 8 proves that the following three communications are being done well in frame frequency of 8 kHz.

(a) UART : Laptop to STM32 board

(b) SAI : Servo Driver (bidirectional) STM32 board

(c) SPI : STM32 board to Oscilloscope

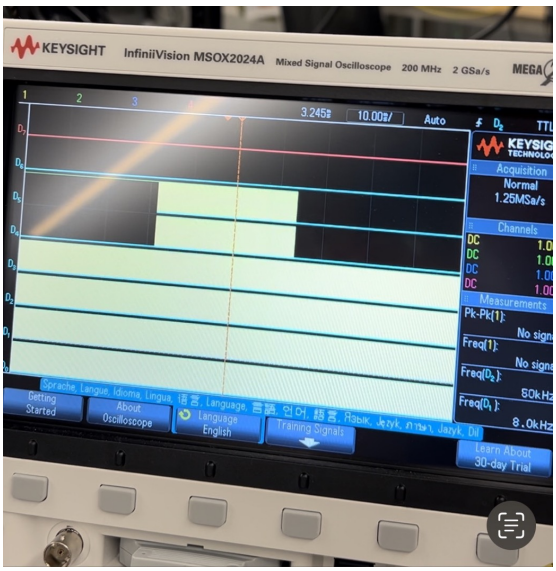

Figure 8: Channel D4 and D5 shows the clock and data signals from SPI channels, which is transmitting position feedback of scanner.

5. Discussion

In this section, how to improve the current experiment will be discussed in details.

(a) Modification of timing diagram

Figure 9 presents the updated timing diagram for our forthcoming experiment, which operates at a frame frequency of 96 kHz. We've managed to reduce the transmission and reception time for the SAI protocol to just 5.2 ms, calculated as 500/96000.
We have set the baud rate at 10 Mbit/s, the maximum achievable rate for a UART channel. This adjustment significantly reduces the time required to transmit half of the buffer (one chunk) to only 1.6 ms, computed as 500/10000000. In this setup, we've transitioned from using SPI to UART for transmitting the servo driver's position feedback data. However, this change introduces a challenge: both transmission and reception must now occur via the UART channel.

Given that the USB 2.0 cable, which is integral to our experiment, does not support full-duplex UART, we have planned for sequential transmission and reception. The USART_TX_DMA will be initiated only after the completion of USART_RX_DMA, signaled by an activated callback function.

As depicted in fig. 9, the total time for both transmitting and receiving one chunk via UART is approximately 3.2 ms (1.6 ms * 2). This duration is about 60 % less than the 5.2 ms required to update one chunk of the TX and RX buffer. Therefore, the timing diagram is expected to remain stable and free from disruptions.

(b) Usage of UART peripherals

In our current experiment, we set the baud rate at 0.8 Mbit/s, with the system unable to exceed 2 Mbit/s. This limitation stems from the specifications of the Nucleo 64-L476RG board used in the experiment. Specifically, this board incorporates an ST-Link V2 interface, which has a maximum operating clock frequency of 4 MHz. This clock frequency inherently restricts the maximum achievable baud rate, posing a significant challenge for data transmission in our experimental setup.

For the future experiment, we will implement the use of an FTDI USB-to-UART Serial Adapter Cable for connecting UART peripheral to the computer.

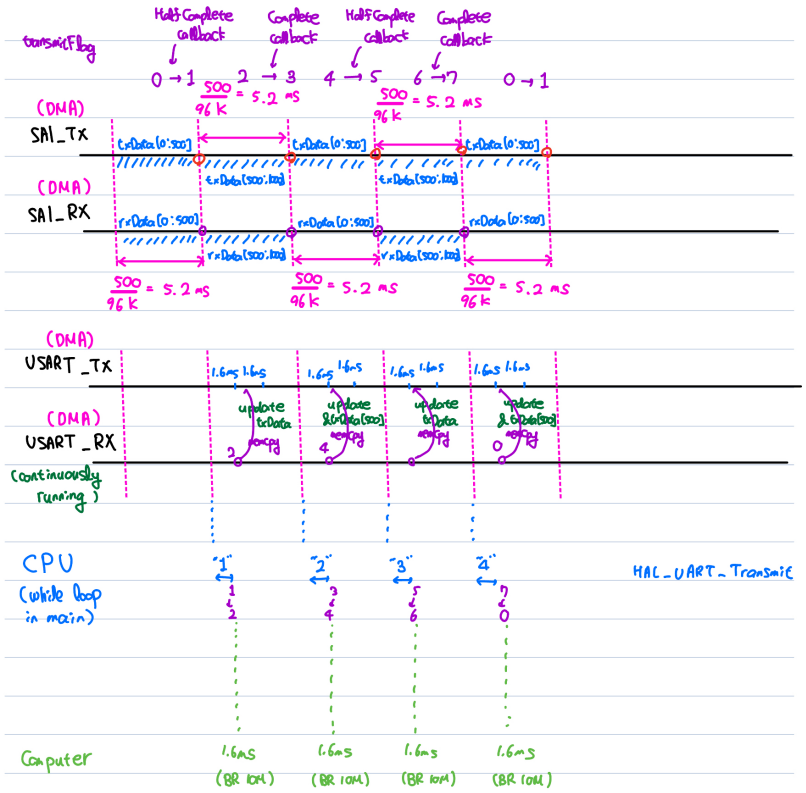

Figure 9: New timing diagram for the future experiment.

6. Conclusion

This report details our experimental setup, which includes a STM32 board, a Servo Driver, and a Laptop. We delve into the programming aspects of this setup: the STM32 board is programmed in C, while the laptop utilizes Python. Additionally, the report features a comprehensive timing diagram. This diagram not only illustrates the transmission and reception (TX/RX) across different DMA channels but also outlines the strategies employed for interrupts and callback functions. These strategies are crucial for ensuring seamless and simultaneous real-time operations during scanning.

Our results indicate successful communication: scan patterns sent from the laptop's serial port are flawlessly transmitted to the Servo Driver. Furthermore, position feedback from the Servo Driver is observable on an oscilloscope, confirming its transmission via the Servo Driver's SPI channel. However, the experiment faces limitations due to the maximum baud rate constraints of the ST-Link and UART channel. These limitations are identified as a primary challenge in our current setup. To address this, we explore the possibility of using FTDI and UART peripherals as a workaround to bypass the ST-Link limitations. This potential solution is discussed in detail in the section 5 of the report.

Additionally, the transition to a digital interface in the hardware implementation is expected to markedly diminish the noise typically associated with the analog components of existing scanning systems. Exploring control systems dedicated to optimizing scan patterns in conjunction with this digital interface represents an additional, promising area for future research.