**THE UNIVERSITY OF DANANG**
**UNIVERSITY OF SCIENCE AND TECHNOLOGY**
**Faculty of Advanced Science and Technology**

**LABORATORY REPORT**

# DEVICE NETWORKS

**Instructor** : Nguyen Huynh Nhat Thuong

**Class** : 21ECE

**Group members:** Luong Nhu Quynh

Vo Van Buu

Nguyen Thi Tam

Tran Hoang Minh

Da Nang, 6th May 2025

## Table of Contents

**1. Project requirements:** ........ 2

**2. Hardware** ........ 2

**2.1. Block diagram** ........ 2

**2.2. Wiring diagram** ........ 3

**2.3. Real diagram photo** ........ 3

**3. Software** ........ 4

**3.1. Microcontroller configuration** ........ 4

**3.2. Flowchart** ........ 11

**3.3. Code** ........ 11

**4. Results** ........ 14

**4.1. Testing plan** ........ 14

**4.2. Results** ........ 15

## 1. Project requirements:

This report details a laboratory exercise conducted as part of the Device Network course, focusing on the practical implementation of the Two-way communication between one Master (A) and two Slaves (B and C) using the SPI protocol

The Master (A) is equipped with two push buttons. Pressing each button will send a specific message to the corresponding Slave:

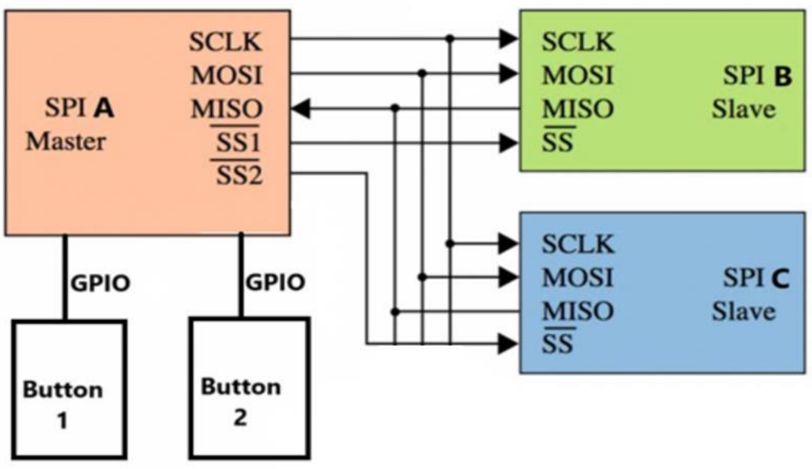
- Button 1: Sends "A hello B" to Slave B.
- Button 2: Sends "A hello C" to Slave C.

Upon receiving the message, the respective Slave will respond back to the Master:

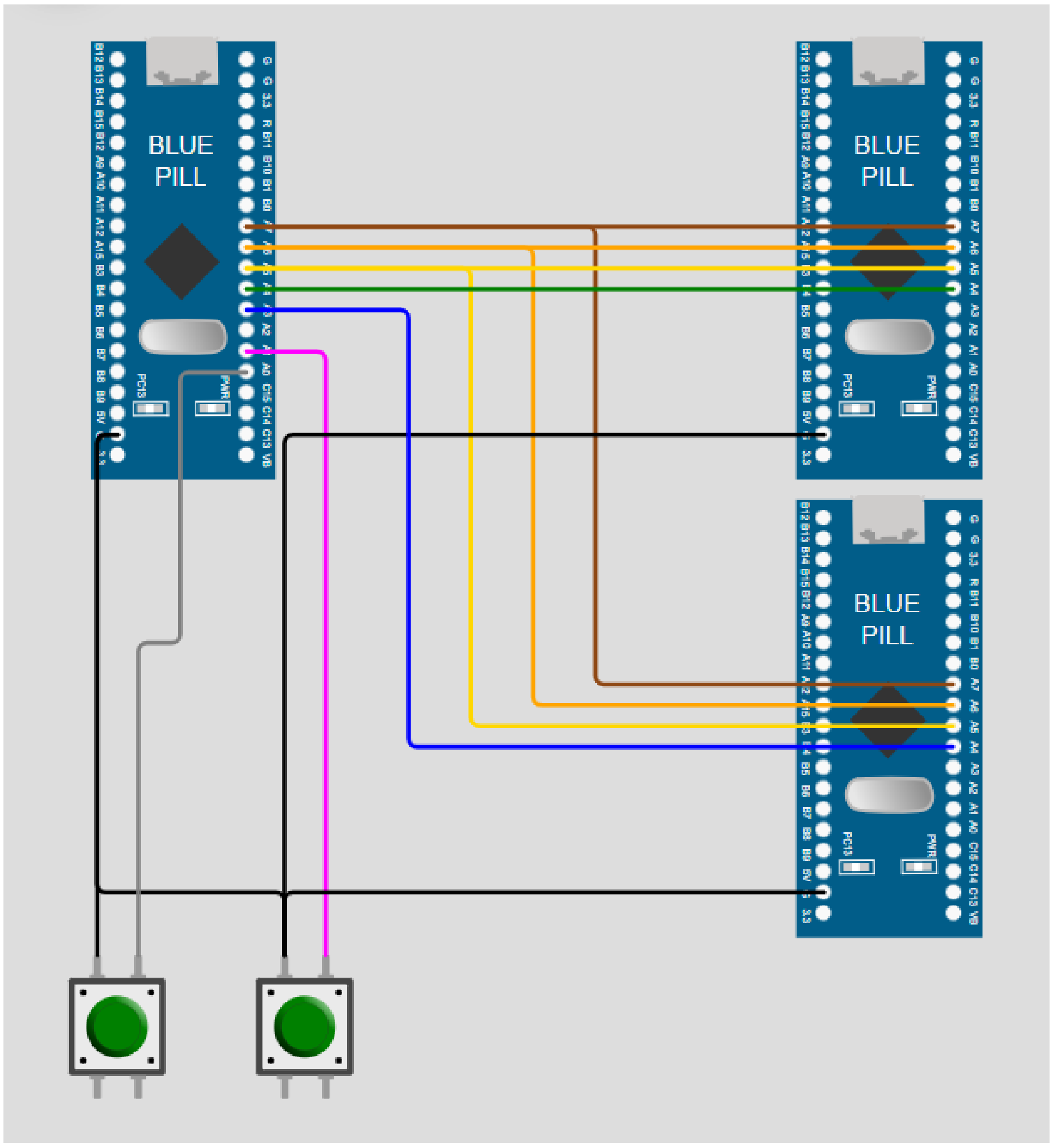
- Slave B responds with "B hello A".
- Slave C responds with "C hello A".

## 2. Hardware

### 2.1. Block diagram

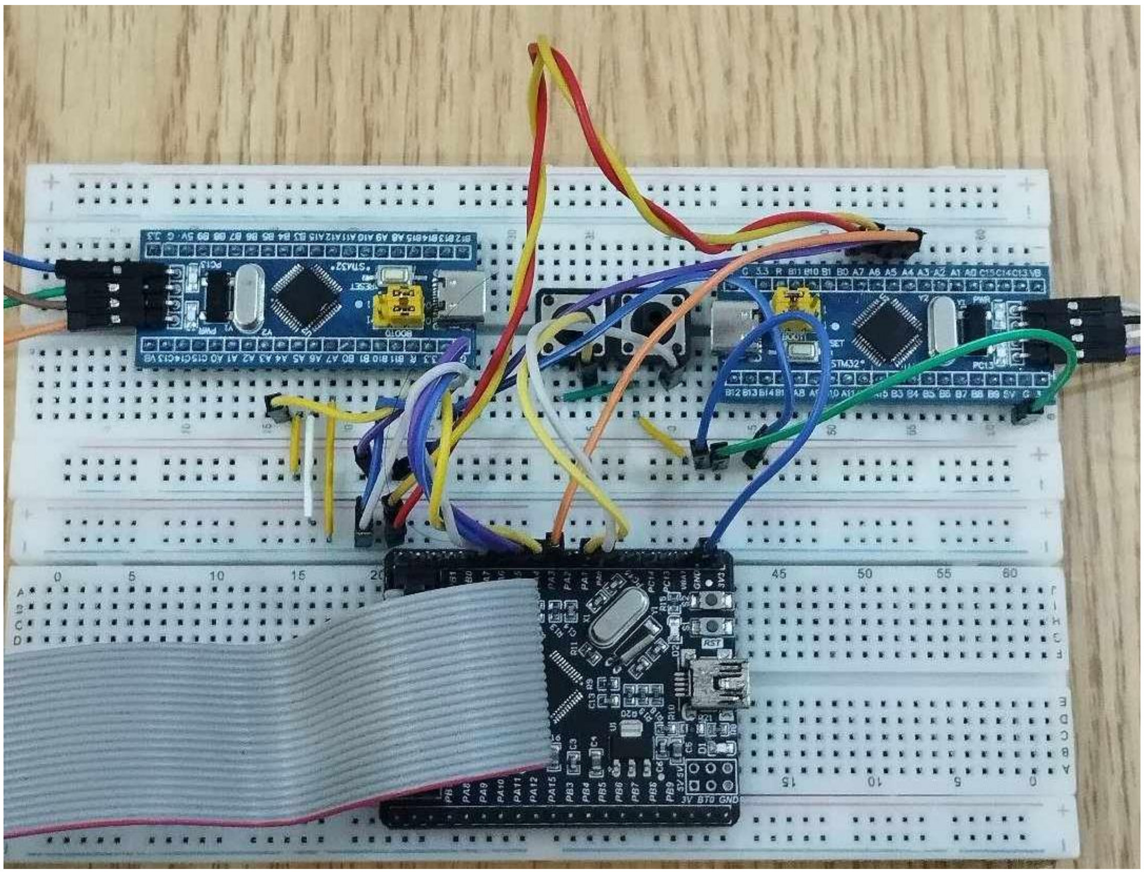


## 2.2. Wiring diagram



## 2.3. Real diagram photo

The communication is implemented using the SPI protocol, ensuring two-way data exchange between the Master and the Slaves.

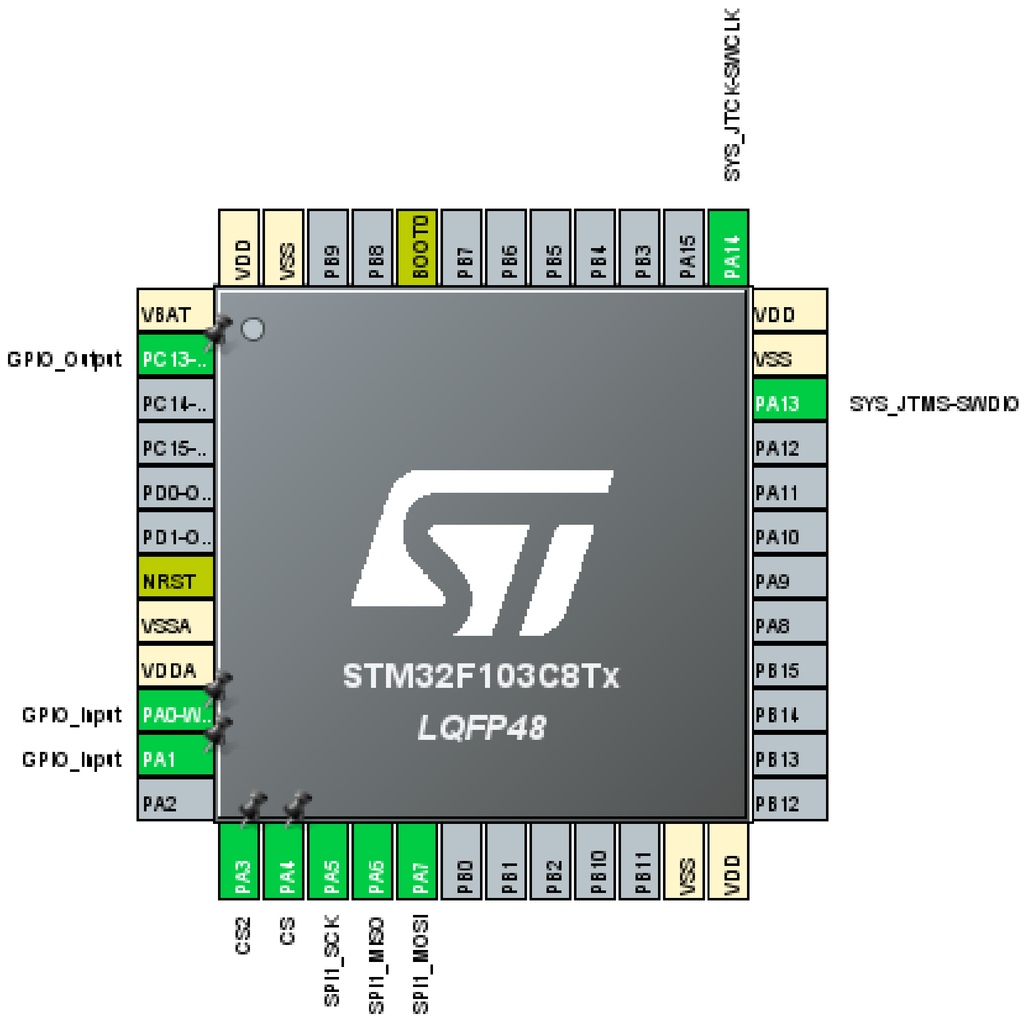


# 3. Software

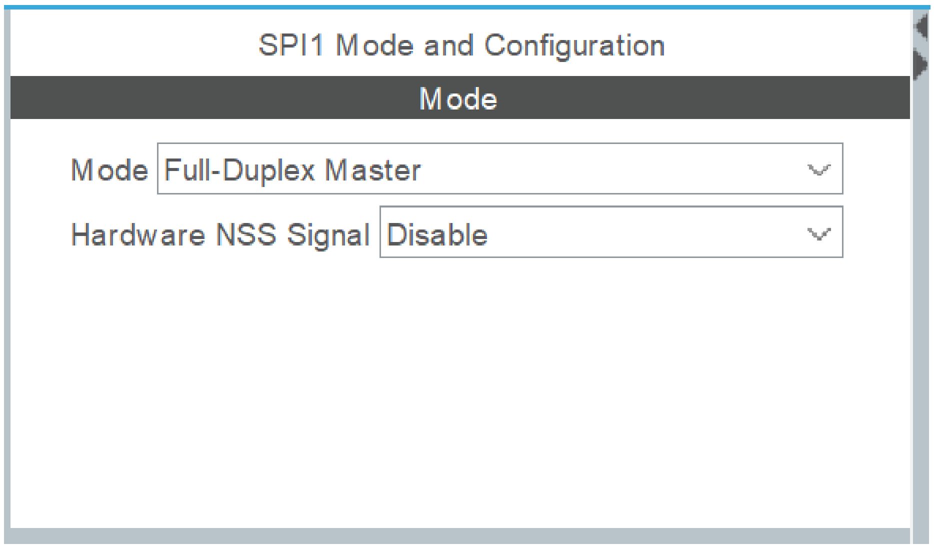
## 3.1. Microcontroller configuration

**Master A:**

Master A is configured to operate in SPI Master mode, using two push buttons connected to GPIO pins (PA0 and PA1) for initiating communication. The SPI interface (SPI1) is set up with a clock speed suitable for reliable communication, and two chip select (CS) lines (CS_Pin and CS2_Pin) are used to select Slave B or Slave C. The Master pulls the CS line low to initiate communication and high to end it.



Enable SPI1 from “Connectivity” menu and choose “Full-Duplex Master”.


SPI1 Mode and Configuration

Mode

Mode: Full-Duplex Master

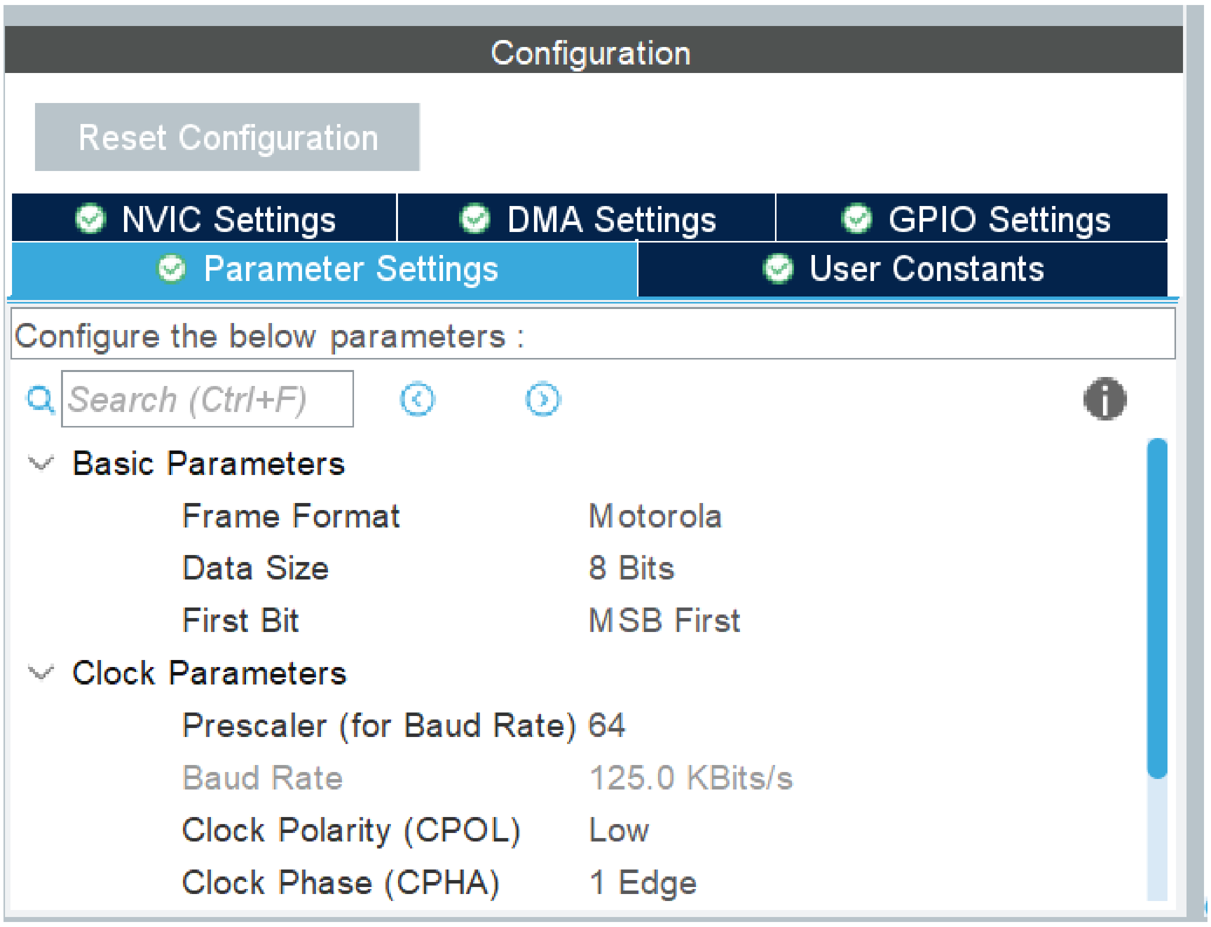
Hardware NSS Signal: Disable

In STM32CubeMX, we choose these settings:
Basic parameters:

- **Frame Format: Motorola.** This is the standard SPI frame format. It supports Clock Polarity (CPOL) and Clock Phase (CPHA), which are used to sync data correctly.
- **Data Size: 8 Bits.** Each SPI transfer will send/receive 8 bits (1 byte) of data. This is typical for most SPI devices.
- **First Bit: MSB First.** Data is transmitted starting with the Most Significant Bit (bit 7 first, down to bit 0). Most SPI peripherals use this order.

Clock parameters:

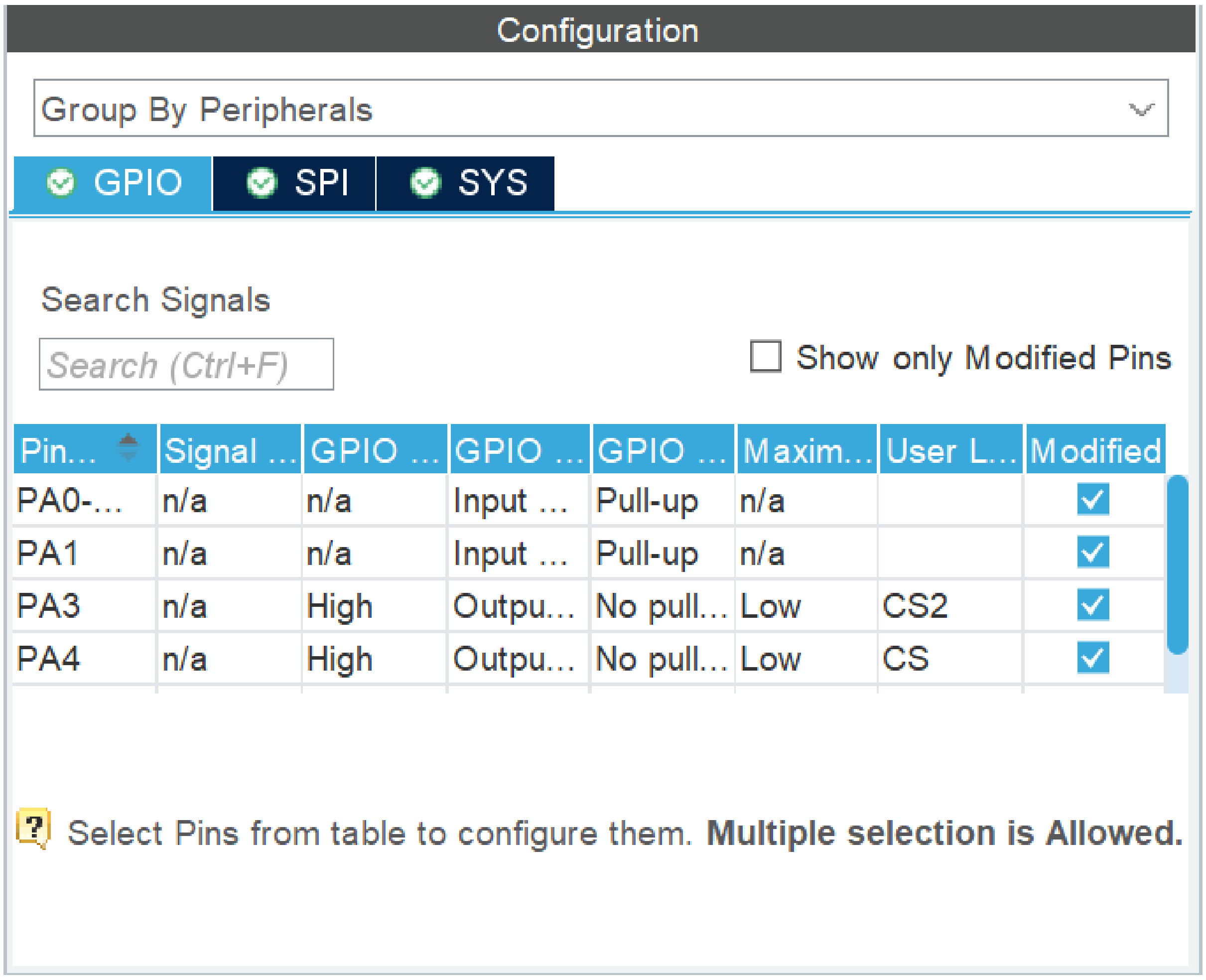
- **Prescaler (for Baud Rate): 64.** This divides the main system clock to lower the SPI clock. For example, if the system clock is 8 MHz:
  > SPI Clock = 8 MHz / 64 = 125 KHz
- **Baud Rate: 125.0 KBits/s.** This is the effective data rate. Lower speeds are useful for testing or communicating with slower peripherals (e.g., sensors, displays, or another MCU).
- **Clock Polarity (CPOL): Low.** SPI clock idles at low level (0) when not transmitting.
- **Clock Phase (CPHA): 1 Edge.** Data is captured on the first edge of the clock signal (rising edge if CPOL=Low).



| | |
|---|---|
| Basic Parameters | |
| Frame Format | Motorola |
| Data Size | 8 Bits |
| First Bit | MSB First |
| Clock Parameters | |
| Prescaler (for Baud Rate) | 64 |
| Baud Rate | 125.0 KBits/s |
| Clock Polarity (CPOL) | Low |
| Clock Phase (CPHA) | 1 Edge |

In that case, we need to go to the “GPIO Settings” tab and

1. Set PA0 and PA1 to Input mode and pull their setting to pull-up.
2. Set PA3 and PA4 as Output mode with No pull (default is fine).

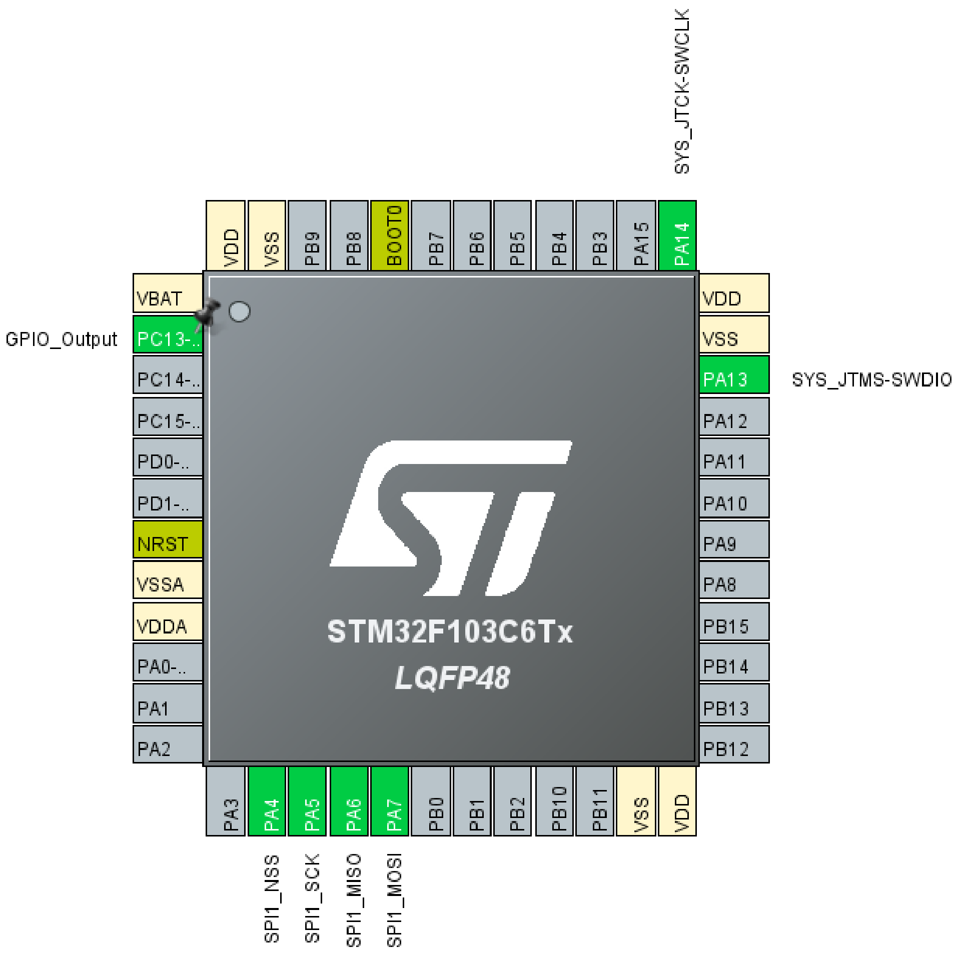


| Pin... | Signal ... | GPIO ... | GPIO ... | GPIO ... | Maxim... | User L... | Modified |
|---|---|---|---|---|---|---|---|
| PA0-... | n/a | n/a | Input ... | Pull-up | n/a | | ✓ |
| PA1 | n/a | n/a | Input ... | Pull-up | n/a | | ✓ |
| PA3 | n/a | High | Outpu... | No pull... | Low | CS2 | ✓ |
| PA4 | n/a | High | Outpu... | No pull... | Low | CS | ✓ |

Select Pins from table to configure them. **Multiple selection is Allowed.**

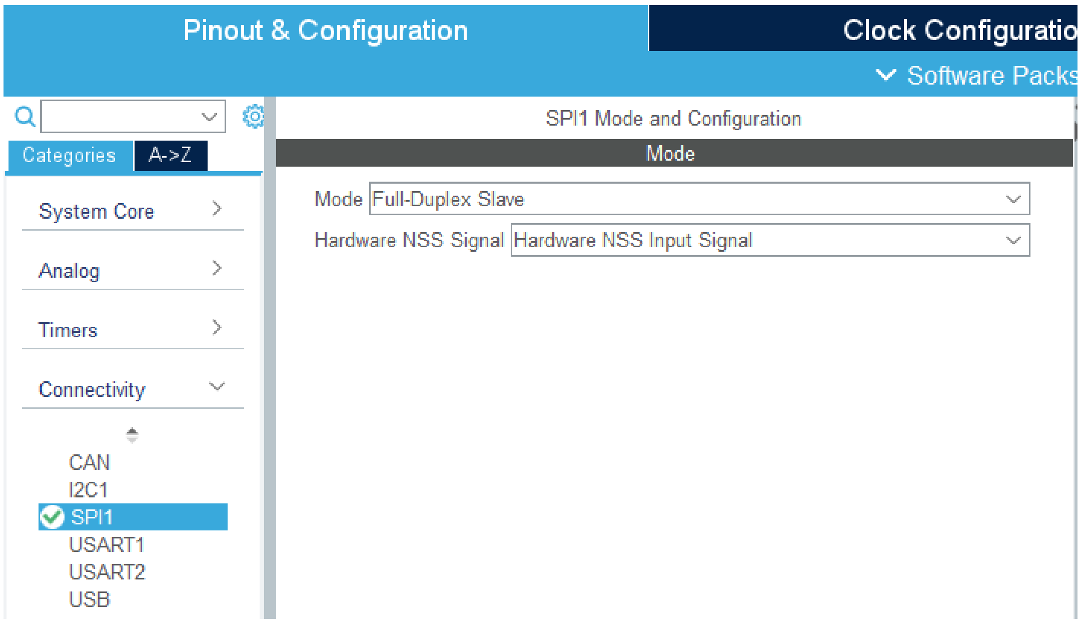
**Slave B and C:**

Slaves B and C are configured in SPI Slave mode, each with its own CS line connected to the Master. They receive data via SPI1 and respond with predefined messages (“B hello A” or “C hello A”). The slaves are programmed to echo back the received message or send their response when selected by the Master.



Enable SPI1 from the "Connectivity" menu and choose "Full-Duplex Slave" and "Hardware NSS Input Signal". In this setting, the NSS pin acts as a hardware-controlled input:

- When NSS is low, the SPI slave is enabled and ready to communicate.
- When NSS is high, SPI communication is disabled.



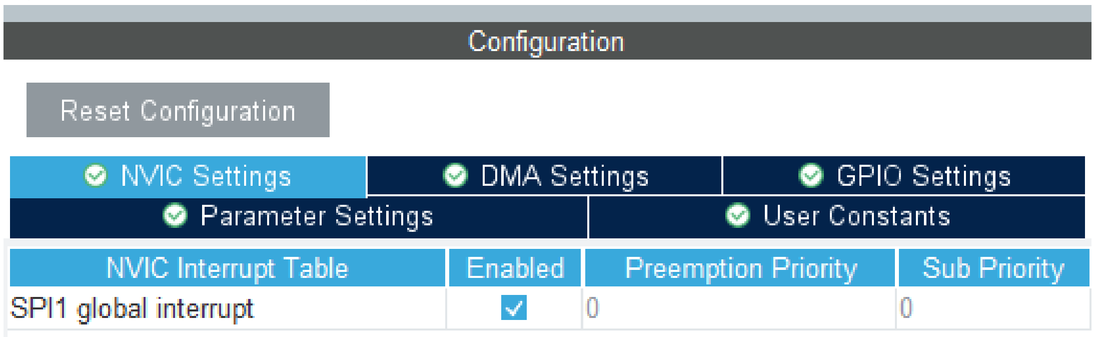
SPI1 Mode and Configuration

Mode

Mode: Full-Duplex Slave

Hardware NSS Signal: Hardware NSS Input Signal

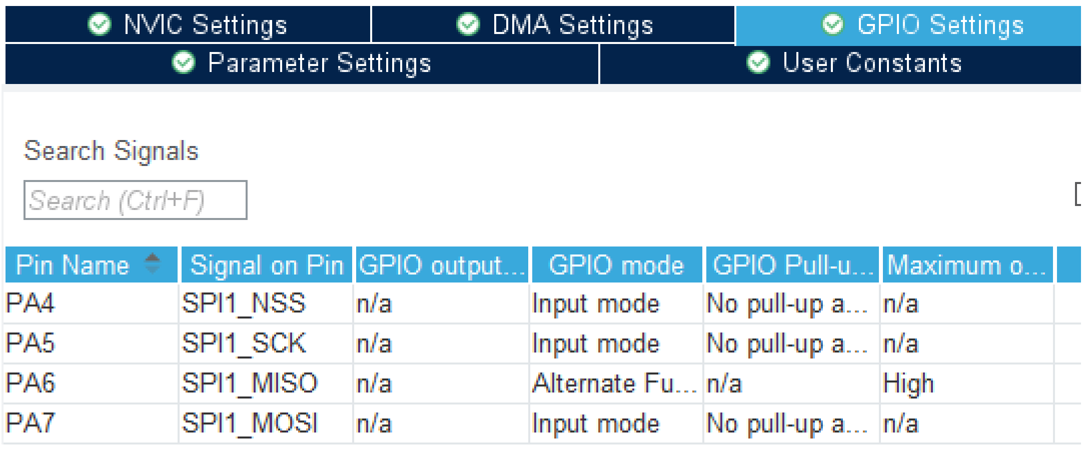
We keep the “Preemption Priority” and “Sub Priorities” set to their default values 0.
The SPI1 interrupt handler will be triggered when an SPI event happens. The lowest number means highest priority. This interrupt can preempt others with lower priority (higher number). Sub Priority: 0 is used to resolve conflicts when two interrupts with the same preemption priority occur. Lower sub-priority will be served first.



| NVIC Interrupt Table | Enabled | Preemption Priority | Sub Priority |
|---|---|---|---|
| SPI1 global interrupt | ✓ | 0 | 0 |

In that case, we need to go to the “GPIO Settings” tab and

- **PA4 (NSS)**: Used by the master to enable the slave. Set to Input mode. No internal pull-up/down needed if externally controlled.
- **PA5 (SCK)**: Clock line from master. Also Input mode. The slave listens for clock pulses.
- **PA6 (MISO)**: Slave sends data here. Set as Alternate Function, not input, because it's an output pin from the slave's perspective.
- **PA7 (MOSI)**: Master sends data. Set as Input mode, so the slave can receive it.



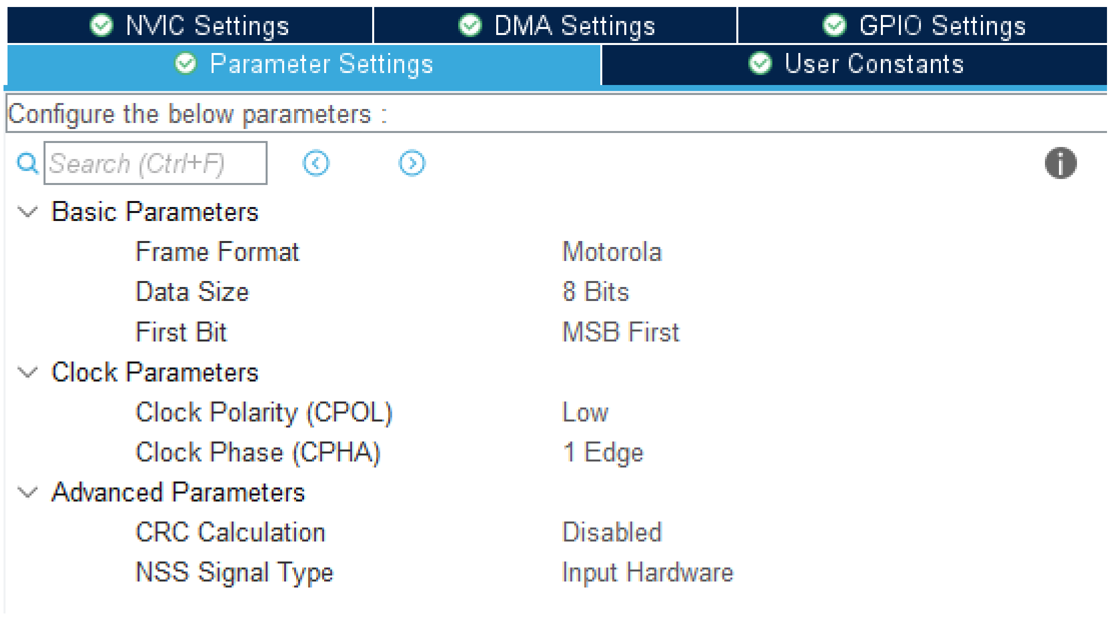
| Pin Name | Signal on Pin | GPIO output… | GPIO mode | GPIO Pull-u… | Maximum o… |
|---|---|---|---|---|---|
| PA4 | SPI1_NSS | n/a | Input mode | No pull-up a… | n/a |
| PA5 | SPI1_SCK | n/a | Input mode | No pull-up a… | n/a |
| PA6 | SPI1_MISO | n/a | Alternate Fu… | n/a | High |
| PA7 | SPI1_MOSI | n/a | Input mode | No pull-up a… | n/a |

Basic parameters:

- **Frame Format**: **Motorola.** This is the standard SPI format with NSS, SCK, MISO, and MOSI.
- **First Bit**: **MSB First** The most significant bit is transmitted first.

Basic parameters:

- **Clock Polarity (CPOL)**: **Low.** Clock is low when idle (CPOL = 0).
- **Clock Phase (CPHA)**: **1 Edge.** Data is captured on the first clock edge (rising edge, since CPOL = 0).



| | |
|---|---|
| Basic Parameters | |
| Frame Format | Motorola |
| Data Size | 8 Bits |
| First Bit | MSB First |
| Clock Parameters | |
| Clock Polarity (CPOL) | Low |
| Clock Phase (CPHA) | 1 Edge |
| Advanced Parameters | |
| CRC Calculation | Disabled |
| NSS Signal Type | Input Hardware |

## 3.2. Flowchart

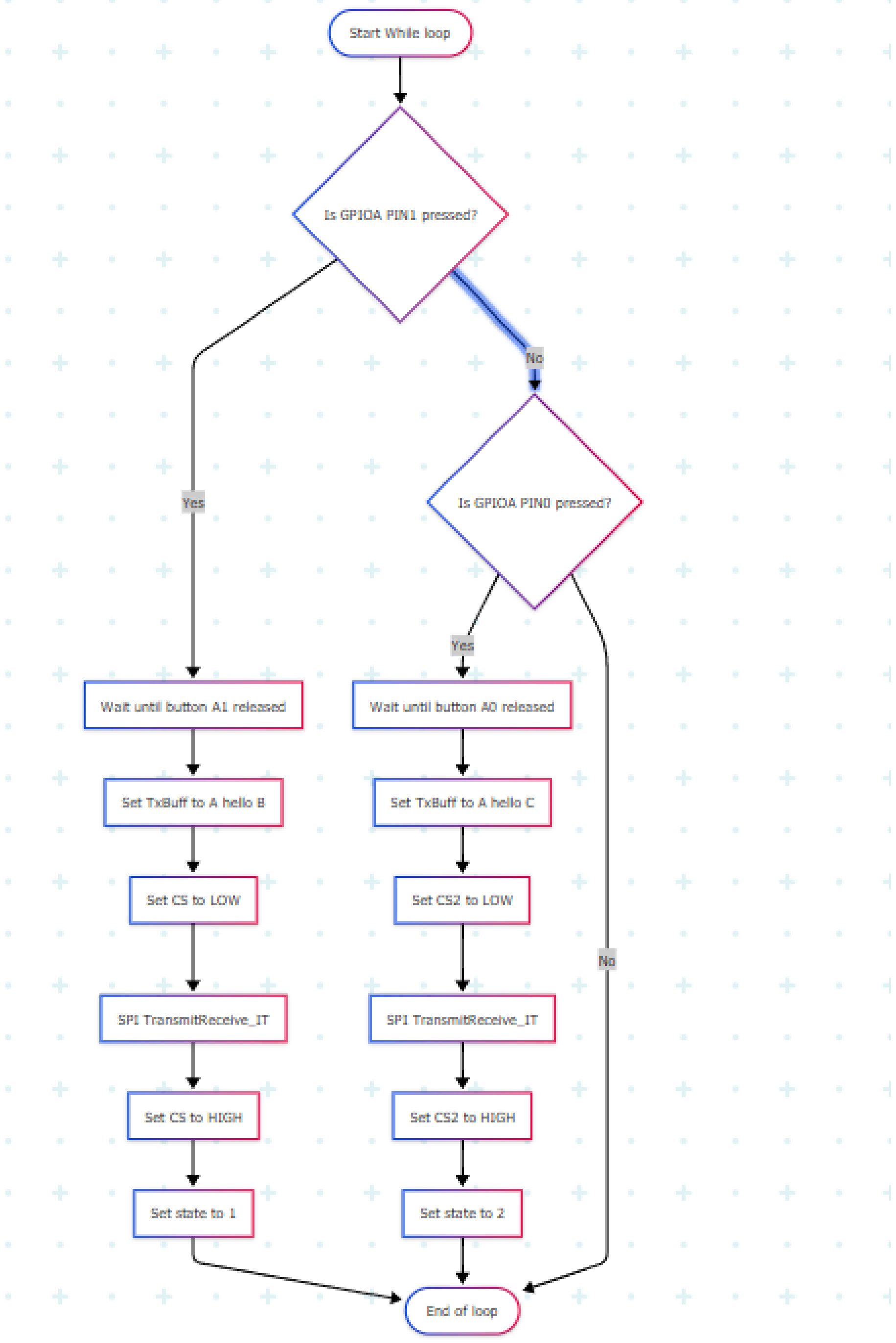


Figure: The operating flowchart of systems.

## 3.3. Code

**Master A:**

The code is added in between /* USER CODE BEGIN Includes */ and /* USER CODE END Includes */. Insert this code in between:

```
#include <stdio.h>
#define sizeofBuff 12
uint8_t u8_SPI1_TxBuff[sizeofBuff] = "Master send";
uint8_t u8_SPI1_RxBuff[sizeofBuff];
uint8_t u8_SPI2_RxBuff[sizeofBuff];
uint8_t state = 1;
```

The code is added in between /* USER CODE BEGIN WHILE*/ and /* USER CODE END WHILE*/. Insert this code in between:

```
        if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == 0) //Khi bấm nút A1
            {
             while(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == 0) {}
             sprintf((char*)u8_SPI1_TxBuff, "A hello B\n");
             HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_RESET);
             HAL_SPI_TransmitReceive(&hspi1, u8_SPI1_TxBuff, u8_SPI1_RxBuff,
sizeofBuff, 100);
             HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_SET);
             state = 1;
            }
            else if(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0) == 0)//Khi bấm nút A0
            {
                 while(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == 0) {}
                 sprintf((char*)u8_SPI1_TxBuff, "A hello C\n");
                 HAL_GPIO_WritePin(CS2_GPIO_Port, CS2_Pin,
GPIO_PIN_RESET);
                 HAL_SPI_TransmitReceive(&hspi1, u8_SPI1_TxBuff,
u8_SPI1_RxBuff, sizeofBuff, 100);
                 HAL_GPIO_WritePin(CS2_GPIO_Port, CS2_Pin, GPIO_PIN_SET);
                 state =2;
            }
```

**Slave B:**

The code is added in between /* USER CODE BEGIN Includes */ and /* USER CODE END Includes */. Insert this code in between:

```
#include <stdio.h>
#define sizeofBuff 12
uint8_t u8_SPI1_TxBuff[sizeofBuff] = "B hello A\n";
uint8_t u8_SPI1_RxBuff[sizeofBuff];
uint8_t Rx_slave[]={0};
uint8_t Count;
```

The code is added in between /* USER CODE BEGIN 0*/ and /* USER CODE END 0*/. Insert this code in between:

```
void HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)
{
 /* Prevent unused argument(s) compilation warning */
 UNUSED(hspi);
	if(hspi->Instance == SPI1)
	{
		HAL_SPI_TransmitReceive_IT(&hspi1, u8_SPI1_TxBuff,
u8_SPI1_RxBuff, sizeofBuff);
	}
 /* NOTE : This function should not be modified, when the callback is needed,
	the HAL_SPI_RxCpltCallback should be implemented in the user file
  */
}
```

The code is added in between /* USER CODE BEGIN 2*/ and /* USER CODE END 2*/. Insert this code in between:

HAL_SPI_TransmitReceive_IT(&hspi1, u8_SPI1_TxBuff, u8_SPI1_RxBuff, sizeofBuff);

**Slave C:**

Code for the Slave C quit the same as Slave B, we just need to change the u8_SPI1_TxBuff

The code is added in between /* USER CODE BEGIN Includes */ and /* USER CODE END Includes */. Insert this code in between:

```
#include <stdio.h>
#define sizeofBuff 12
uint8_t u8_SPI1_TxBuff[sizeofBuff] = "C hello A\n";
uint8_t u8_SPI1_RxBuff[sizeofBuff];
uint8_t Rx_slave[]={0};
uint8_t Count;
```

The code is added in between /* USER CODE BEGIN 0*/ and /* USER CODE END 0*/. Insert this code in between:

```
void HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)
{
 /* Prevent unused argument(s) compilation warning */
 UNUSED(hspi);
	if(hspi->Instance == SPI1)
	{
		HAL_SPI_TransmitReceive_IT(&hspi1, u8_SPI1_TxBuff,
u8_SPI1_RxBuff, sizeofBuff);
	}
 /* NOTE : This function should not be modified, when the callback is needed,
	the HAL_SPI_RxCpltCallback should be implemented in the user file
  */
}
```

The code is added in between /* USER CODE BEGIN 2*/ and /* USER CODE END 2*/.
Insert this code in between:

```
 HAL_SPI_TransmitReceive_IT(&hspi1, u8_SPI1_TxBuff, u8_SPI1_RxBuff, sizeofBuff);
```

# 4. Results

## 4.1. Testing plan

**Button Functionality Test (Master Side)**

- Verify that pressing Button 1 sends the correct message "A hello B" to Slave B.
- Verify that pressing Button 2 sends the correct message "A hello C" to Slave C.

**Slave Response Test**

- Ensure Slave B only responds with "B hello A" when it receives "A hello B".
- Ensure Slave C only responds with "C hello A" when it receives "A hello C".

**SPI Data Integrity Check**

- Use logic analyzer to monitor SPI communication between Master and Slaves.
- Confirm that no corrupted or incomplete messages are exchanged.

**Sequential Communication Test**

- Press the buttons on Master one after another.
- Confirm that the Master handles each communication in turn without conflict.

**Idle State Test**

- Power on the system without pressing any buttons.
- Ensure that no SPI communication occurs until a button is pressed.

**Error Handling Test**

- Simulate unexpected or malformed messages (if possible).
- Verify that Slaves ignore unknown messages and do not respond incorrectly.

**Simultaneous Press Test**

- Press both buttons at the same time (if physically possible).
- Observe if the Master handles the collision or queueing of messages properly.

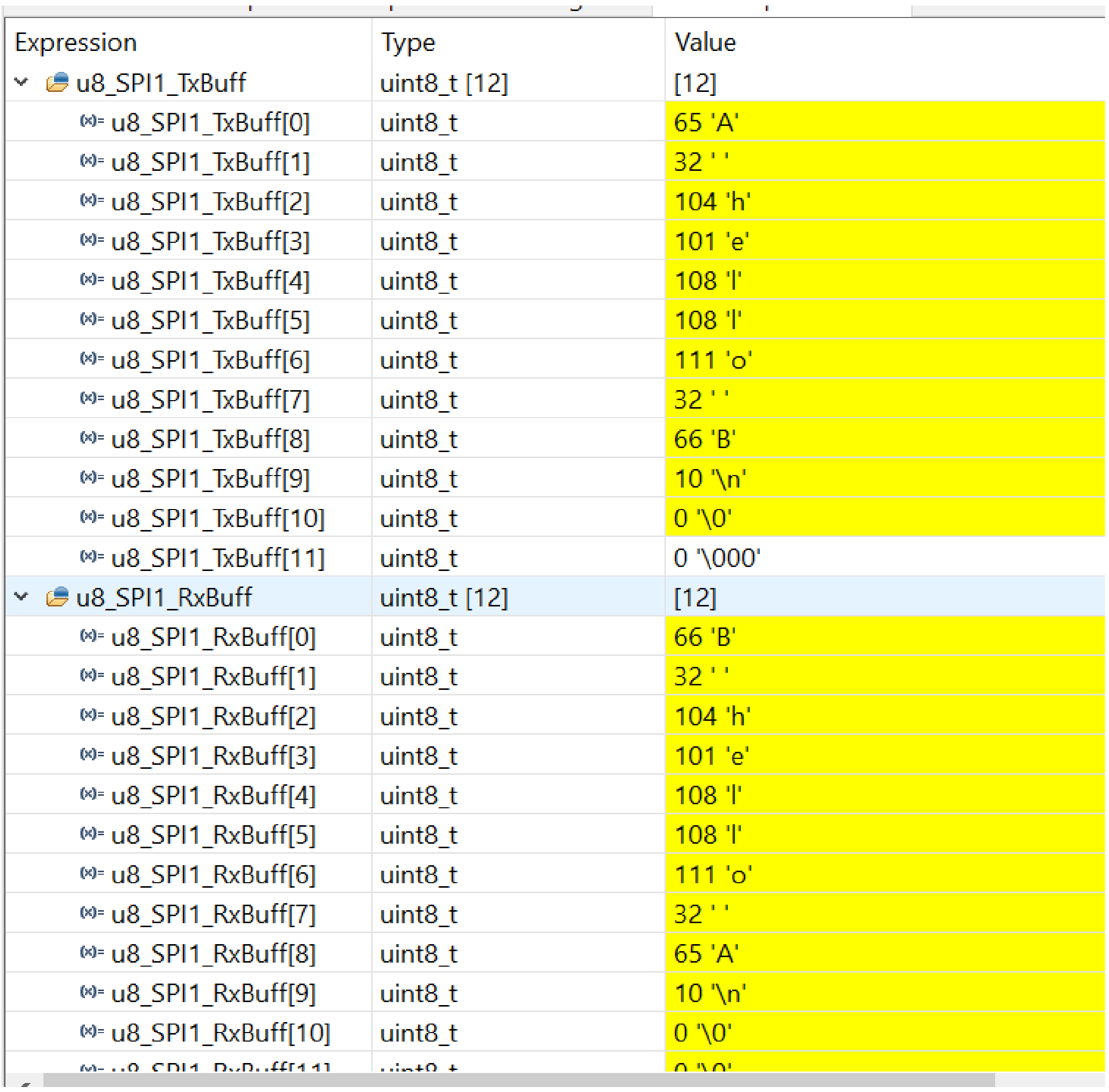
**Response Timing Test**

- Measure the time between sending the message and receiving the response.
- Confirm the communication is completed within an acceptable time frame (e.g., under 100ms).

## 4.2. Results

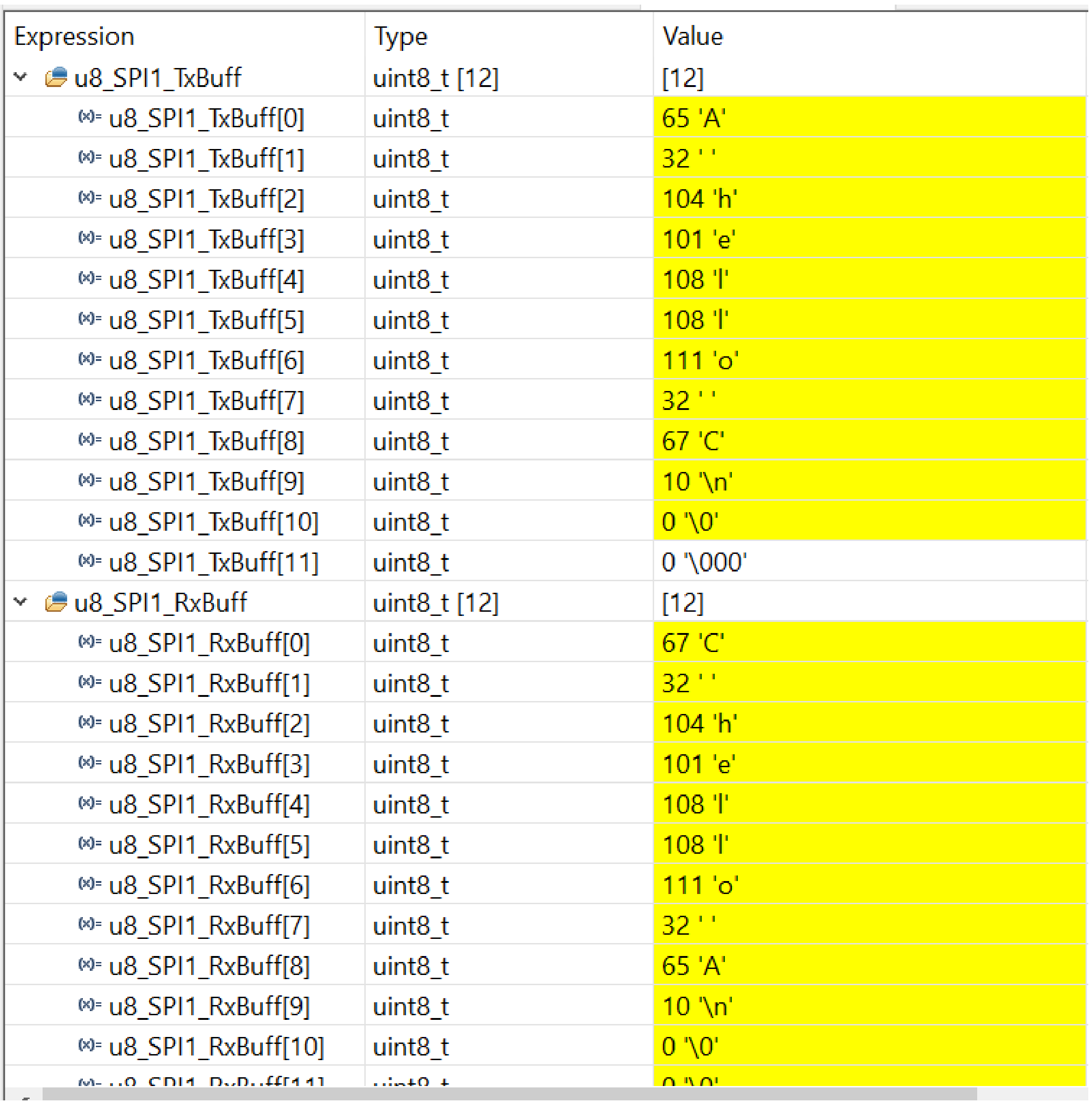
**Master A:**

Master A successfully sends "A hello B" to Slave B when button 1 (PA1) is pressed and "A hello C" to Slave C when button 2 (PA0) is pressed. The Master receives and stores the responses ("B hello A" or "C hello A") in its receive buffer, verified via debugging or serial output.



| Expression | Type | Value |
|---|---|---|
| u8_SPI1_TxBuff | uint8_t [12] | [12] |
| u8_SPI1_TxBuff[0] | uint8_t | 65 'A' |
| u8_SPI1_TxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_TxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_TxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_TxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[8] | uint8_t | 66 'B' |
| u8_SPI1_TxBuff[9] | uint8_t | 10 '\n' |
| u8_SPI1_TxBuff[10] | uint8_t | 0 '\0' |
| u8_SPI1_TxBuff[11] | uint8_t | 0 '\000' |
| u8_SPI1_RxBuff | uint8_t [12] | [12] |
| u8_SPI1_RxBuff[0] | uint8_t | 66 'B' |
| u8_SPI1_RxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_RxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_RxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_RxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[8] | uint8_t | 65 'A' |
| u8_SPI1_RxBuff[9] | uint8_t | 10 '\n' |
| u8_SPI1_RxBuff[10] | uint8_t | 0 '\0' |
| u8_SPI1_RxBuff[11] | uint8_t | 0 '\0' |

Figure: The debug screen of Master A after pushing button 1.

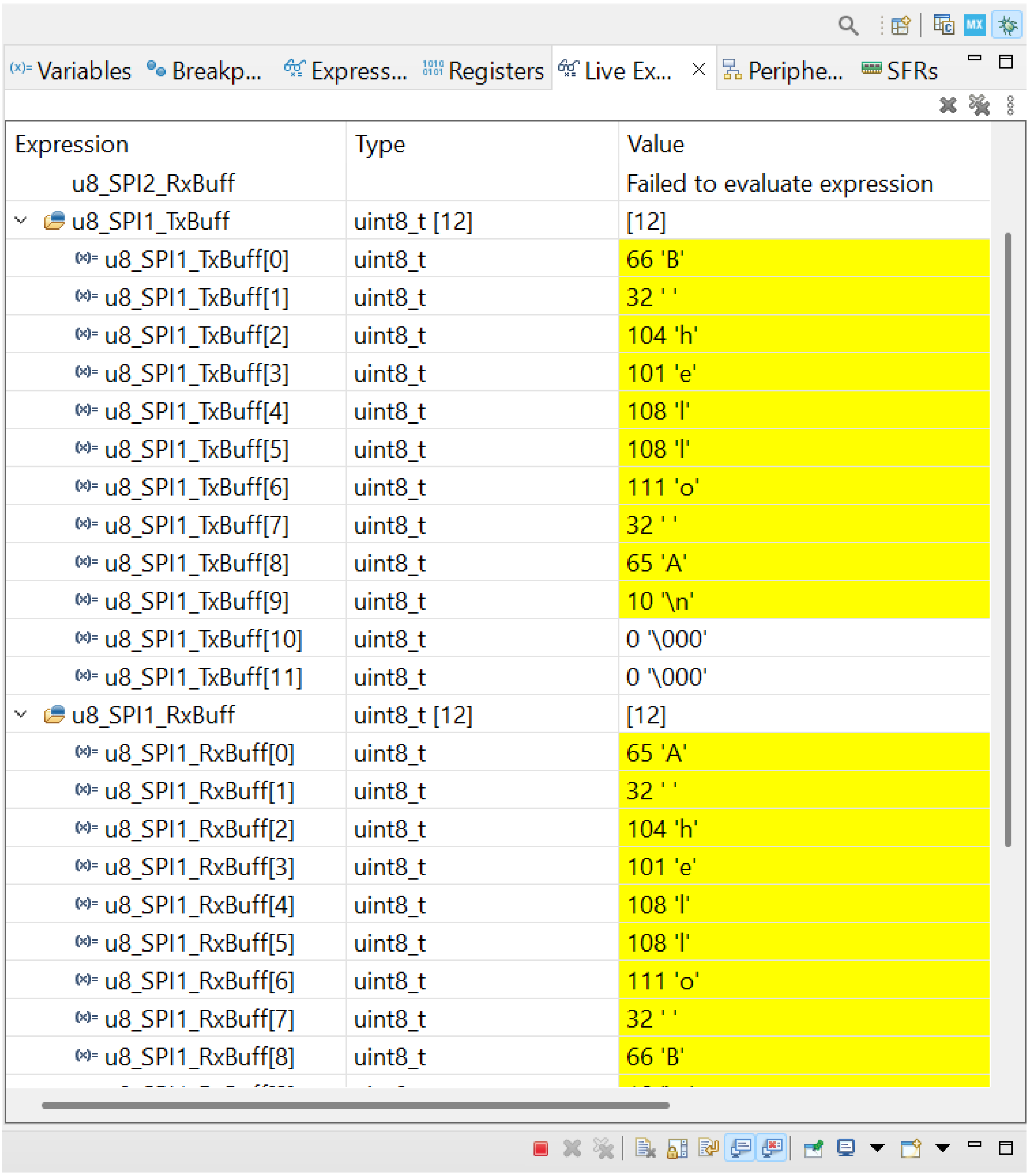


| Expression | Type | Value |
|---|---|---|
| u8_SPI1_TxBuff | uint8_t [12] | [12] |
| u8_SPI1_TxBuff[0] | uint8_t | 65 'A' |
| u8_SPI1_TxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_TxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_TxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_TxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[8] | uint8_t | 67 'C' |
| u8_SPI1_TxBuff[9] | uint8_t | 10 '\n' |
| u8_SPI1_TxBuff[10] | uint8_t | 0 '\0' |
| u8_SPI1_TxBuff[11] | uint8_t | 0 '\000' |
| u8_SPI1_RxBuff | uint8_t [12] | [12] |
| u8_SPI1_RxBuff[0] | uint8_t | 67 'C' |
| u8_SPI1_RxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_RxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_RxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_RxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[8] | uint8_t | 65 'A' |
| u8_SPI1_RxBuff[9] | uint8_t | 10 '\n' |
| u8_SPI1_RxBuff[10] | uint8_t | 0 '\0' |
| u8_SPI1_RxBuff[11] | uint8_t | 0 '\0' |

Figure: The debug screen of Master A after pushing button 2.

**Slave B:**

Slave B receives “A hello B” when its CS line is active and responds with “B hello A”. The communication is stable, with no data corruption observed.

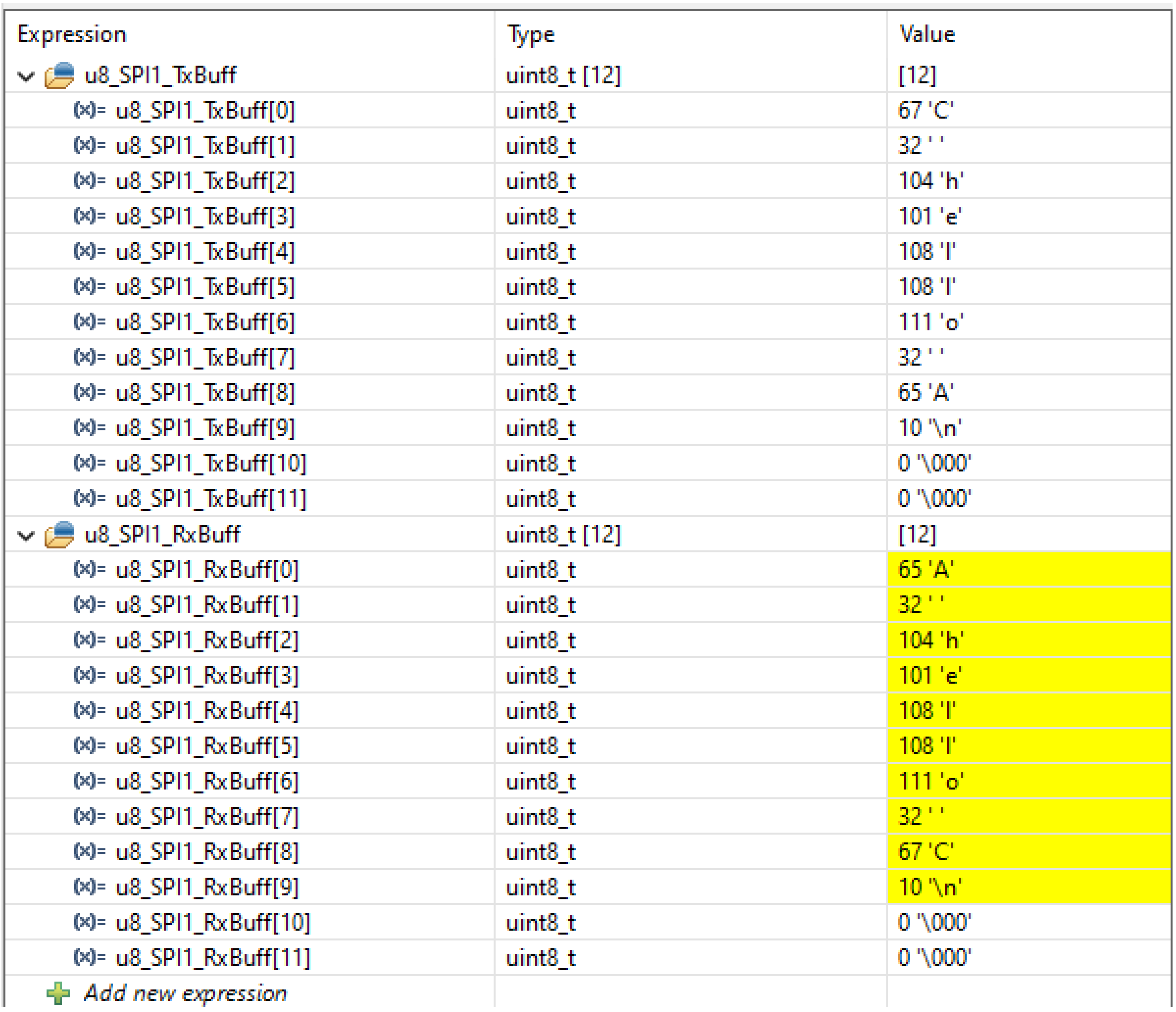


| Expression | Type | Value |
|---|---|---|
| u8_SPI2_RxBuff | | Failed to evaluate expression |
| u8_SPI1_TxBuff | uint8_t [12] | [12] |
| u8_SPI1_TxBuff[0] | uint8_t | 66 'B' |
| u8_SPI1_TxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_TxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_TxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_TxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_TxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_TxBuff[8] | uint8_t | 65 'A' |
| u8_SPI1_TxBuff[9] | uint8_t | 10 '\n' |
| u8_SPI1_TxBuff[10] | uint8_t | 0 '\000' |
| u8_SPI1_TxBuff[11] | uint8_t | 0 '\000' |
| u8_SPI1_RxBuff | uint8_t [12] | [12] |
| u8_SPI1_RxBuff[0] | uint8_t | 65 'A' |
| u8_SPI1_RxBuff[1] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[2] | uint8_t | 104 'h' |
| u8_SPI1_RxBuff[3] | uint8_t | 101 'e' |
| u8_SPI1_RxBuff[4] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[5] | uint8_t | 108 'l' |
| u8_SPI1_RxBuff[6] | uint8_t | 111 'o' |
| u8_SPI1_RxBuff[7] | uint8_t | 32 ' ' |
| u8_SPI1_RxBuff[8] | uint8_t | 66 'B' |

Figure: The debug screen of Slave B  after pushing button 1. .

**Slave C:**

Slave C receives “A hello C” when its CS line is active and responds with “C hello A”. The response is correctly received by the Master.



| Expression | Type | Value |
|---|---|---|
| u8_SPI1_TxBuff | uint8_t [12] | [12] |
| (x)= u8_SPI1_TxBuff[0] | uint8_t | 67 'C' |
| (x)= u8_SPI1_TxBuff[1] | uint8_t | 32 ' ' |
| (x)= u8_SPI1_TxBuff[2] | uint8_t | 104 'h' |
| (x)= u8_SPI1_TxBuff[3] | uint8_t | 101 'e' |
| (x)= u8_SPI1_TxBuff[4] | uint8_t | 108 'l' |
| (x)= u8_SPI1_TxBuff[5] | uint8_t | 108 'l' |
| (x)= u8_SPI1_TxBuff[6] | uint8_t | 111 'o' |
| (x)= u8_SPI1_TxBuff[7] | uint8_t | 32 ' ' |
| (x)= u8_SPI1_TxBuff[8] | uint8_t | 65 'A' |
| (x)= u8_SPI1_TxBuff[9] | uint8_t | 10 '\n' |
| (x)= u8_SPI1_TxBuff[10] | uint8_t | 0 '\000' |
| (x)= u8_SPI1_TxBuff[11] | uint8_t | 0 '\000' |
| u8_SPI1_RxBuff | uint8_t [12] | [12] |
| (x)= u8_SPI1_RxBuff[0] | uint8_t | 65 'A' |
| (x)= u8_SPI1_RxBuff[1] | uint8_t | 32 ' ' |
| (x)= u8_SPI1_RxBuff[2] | uint8_t | 104 'h' |
| (x)= u8_SPI1_RxBuff[3] | uint8_t | 101 'e' |
| (x)= u8_SPI1_RxBuff[4] | uint8_t | 108 'l' |
| (x)= u8_SPI1_RxBuff[5] | uint8_t | 108 'l' |
| (x)= u8_SPI1_RxBuff[6] | uint8_t | 111 'o' |
| (x)= u8_SPI1_RxBuff[7] | uint8_t | 32 ' ' |
| (x)= u8_SPI1_RxBuff[8] | uint8_t | 67 'C' |
| (x)= u8_SPI1_RxBuff[9] | uint8_t | 10 '\n' |
| (x)= u8_SPI1_RxBuff[10] | uint8_t | 0 '\000' |
| (x)= u8_SPI1_RxBuff[11] | uint8_t | 0 '\000' |
| Add new expression | | |

Figure: The debug screen of Slave C  after pushing button 2.