Title page: EEDG/CE 6302 Lab Report 10
CE6302, Embedded Systems, 302 Laboratory - 83204
Lab Topic: MSP432: I2C Communication between MSP432 board and a Slave device
Xiongtao Zhang, xxz240008
Lab partner name: Yuyang Hsieh
Group Number 4
Instructor name: Tooraj Nikoubin
TA name: Seyed Saeed
Date: 11/4/2024

1. Objective:

Purpose: The purpose of lab 10 is to establish I2C communication between the MSP432 and the Adafruit PCF8523 RTC module, while configuring the I2C module on the MSP430 board.The I2C protocol is a synchronous, multi-controller/multi-target (historically termed as multi-master/multi-slave), single-ended, serial communication bus, which is widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

Methods used: I2C uses two lines: serial data line (SDA) and serial clock line (SCL). Both are bidirectional and pulled up with resistors. Communication begins with the START signal, then master device sending an address frame, a read/write bit, and waiting ack from slave, and sending data frame(s); each data frame needs an ack from slave(when writing) or master (when reading); the communication ends with a STOP signal.

Results: In the writing part of the lab, the MSP430(master) sent the address frame, write bit, and 3 bytes data frame to PCF8523 RTC(slave);The Interrupt Service Routine (ISR) was used for sending data through TXBUF on MSP430; Analog Discovery was used to validate the bits transited on SDA and the clock wave on SCL; After analyzing these waves, we validated that all the bits were sent as desired, and the timing was correct. In the reading part of the lab, first a writing operation was sent by the master, and then performed reading from the slave. After analyzing through waveforms, we validated that all the writing and reading operations were transmitted as desired, and the timing was correct.

Hardware used: MSP430 launchpad, Adafruit PCF8523 RTC, Analog Discovery.

Software used: Code Composer Studio (CCS), Waveforms

Major Conclusions: By configuring I2C parameters, setting clock source and adjusting the BRW value, a MSP430 board was set as I2C master, and a I2C communication between MSP430 board and PCF8523 RTC was established. By analyzing the waveforms on the SDA and SCL lines, the I2C communication was validated and performed as desired.

2. Introduction:

2.1 Hardware and Software Background Information
Code Composer Studio is an integrated development environment (IDE) for TI's microcontrollers and processors. It comprises a suite of tools used to develop and debug embedded applications. Code Composer Studio includes an optimizing C/C++ compiler, source code editor, project build environment,

debugger, profiler and many other features. Code Composer Studio combines the advantages of the Eclipse® and Theia frameworks with advanced capabilities from TI resulting in a compelling feature-rich environment. The cloud-based Code Composer Studio enables development in the cloud without the need to download and install large amounts of software.
The PCF8523 is a CMOS1 Real-Time Clock (RTC) and calendar optimized for low power consumption. Data is transferred serially via the I2C-bus with a maximum data rate of 1000 kbit/s.Alarm and timer functions are available with the possibility to generate a wake-up signal on an interrupt pin. An offset register allows fine-tuning of the clock. The PCF8523 has a backup battery switch-over circuit, which detects power failures and automatically switches to the battery supply when a power failure occurs

2.2 Purpose of the Experiment
The purpose of lab 10 is to learn the operation of I2C protocol in a single master, single slave configuration. The configuration to set the MSP430 board as a I2C master and the operations of writing to slave and reading from slave through MSP430(master) were demonstrated. The basic working principle and configuring processes of I2C were also apprehend through the lab and the practical operation experience was also obtained.

2.3 Summary of the Experiment
In lab 10, an I2C communication was established between MSP430 and PCF8523 RTC, several writing(sending 3 bytes data) and reading(writing one byte to slave and then reading from slave) operations were performed; and by analyzing the waveforms on the SDA and SCL lines, the communications were verified.

2.4 Findings of the experiment
The I2C communication between the MSP430 and the PCF8523 RTC was successfully established and verified. Both writing and reading operations were performed correctly, with the expected data being transmitted and received. The oscilloscope waveforms on the SDA and SCL lines confirmed proper I2C signaling. Start and stop conditions, as well as data and acknowledgment (ACK) bits, were clearly observed. The timing diagrams matched the expected I2C protocol behavior, confirming correct communication sequences.

3. Explanation:

3.1 Experiment procedure

To establish I2C communication between the MSP432 LaunchPad and the Adafruit RTC, the MSP432 is connected to a computer via a micro-USB cable, functioning as the Master device. After compiling the I2C Master code for the MSP432, P1.6 (SDA) of the MSP432 is connected to the SDA pin on the RTC, and P1.7 (SCK) is connected to the SCK pin on the RTC. The RTC is powered by connecting its Vdd and Gnd pins to the corresponding Vdd (5V) and Gnd of the MSP432. In Code Composer Studio (CCS), a new project is created for the Master device with the name "I2C_Master_Write_3Bytes," selecting the MSP432P401R board. Another CCS window is opened to create a second project for reading data from a specific register location on the RTC, using the same board selection. The Waveforms application is installed to monitor the signals generated during I2C communication.

In the code, EUSCI_B0 is configured in I2C master mode, setting SMCLK as the clock source and adjusting the BRW to 60, while specifying the correct Slave address for the Adafruit RTC. An automatic

stop condition is set up, and the transmission start flag is activated to initiate data transfer. Within the main loop, flag polling is used to distinguish between writing to the RTC's register and reading from it. An Interrupt Service Routine (ISR) is written to handle the transmission of the register address and reading of data from the RTC, utilizing the Interrupt Vector to manage both tasks effectively. The STPIFG (Stop Condition Interrupt Flag) is checked to determine when the I2C transmission is complete. Once the code is ready, the respective projects are run in CCS, and the I2C communication is observed using the Waveforms application. In Waveforms, the Logic Analyzer option is selected, SDA and SCK channels are added, the I2C protocol is chosen for observation, and the Run option is selected to monitor the output of the data transmission and reception.

3.2 Experiment Images for Part 1

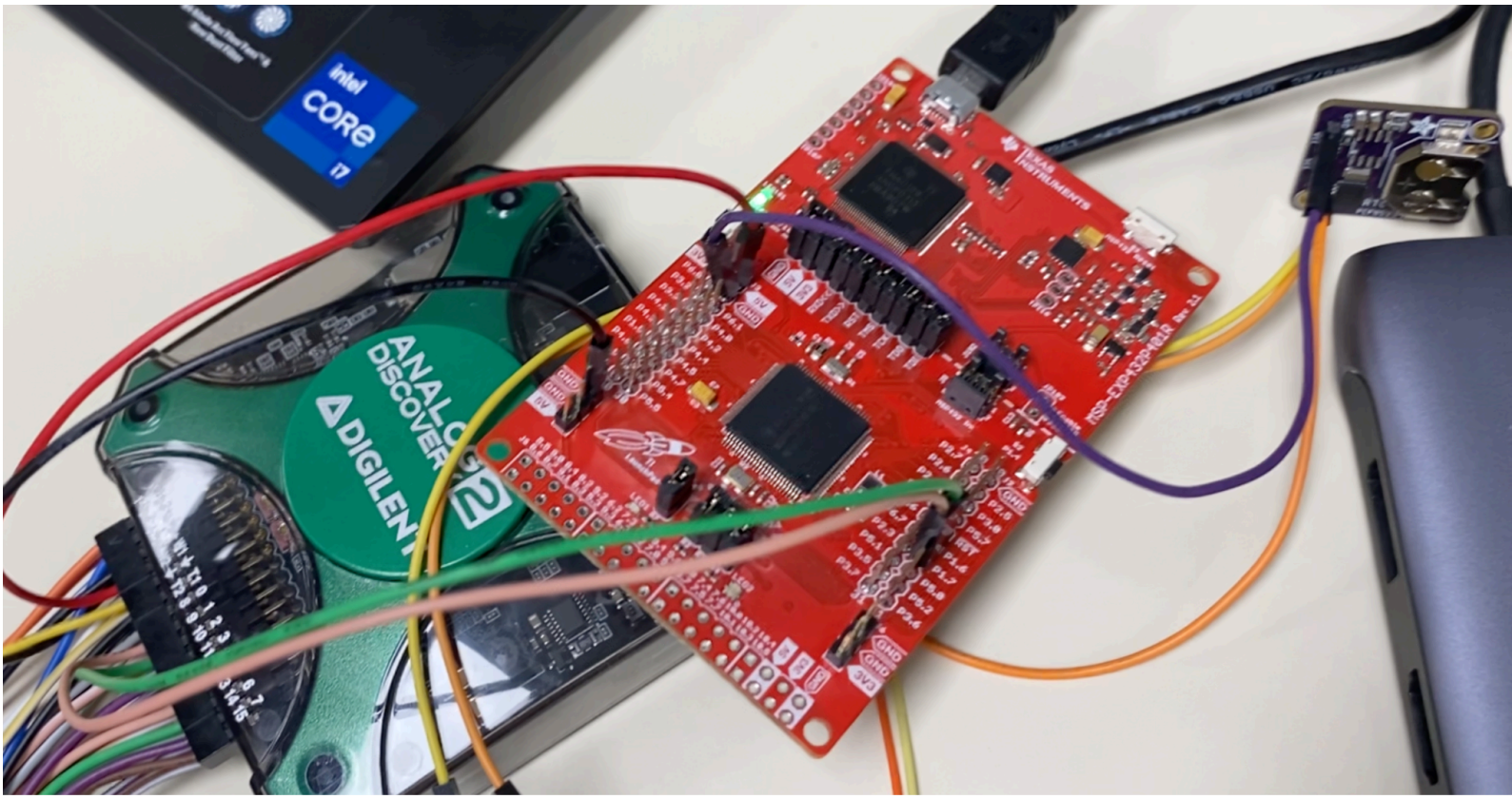


Image 3.2-1: I2C connection established between MSP432 and the Adafruit RTC

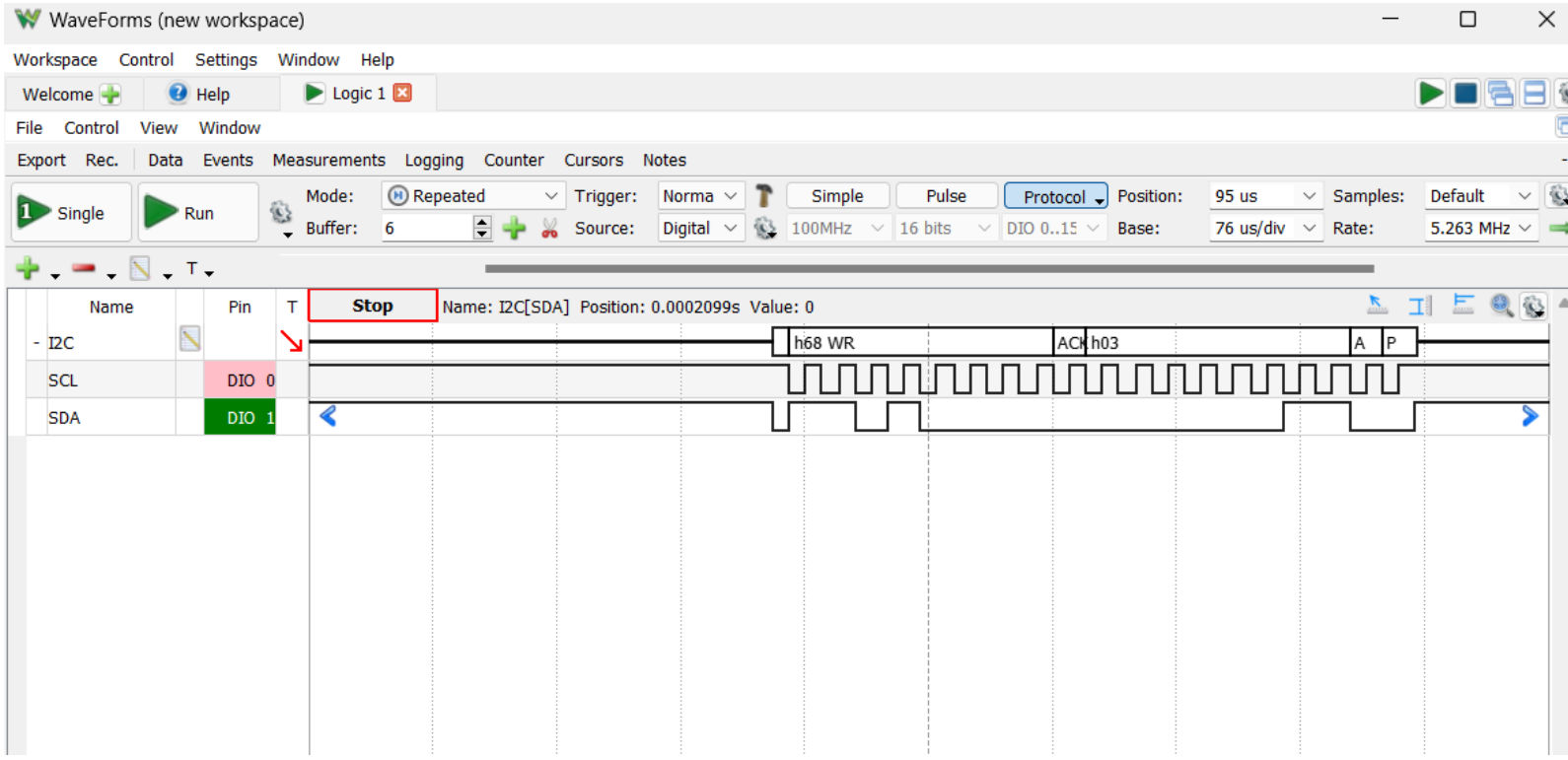


Image 3.2-2: 0x03 to TXBUF

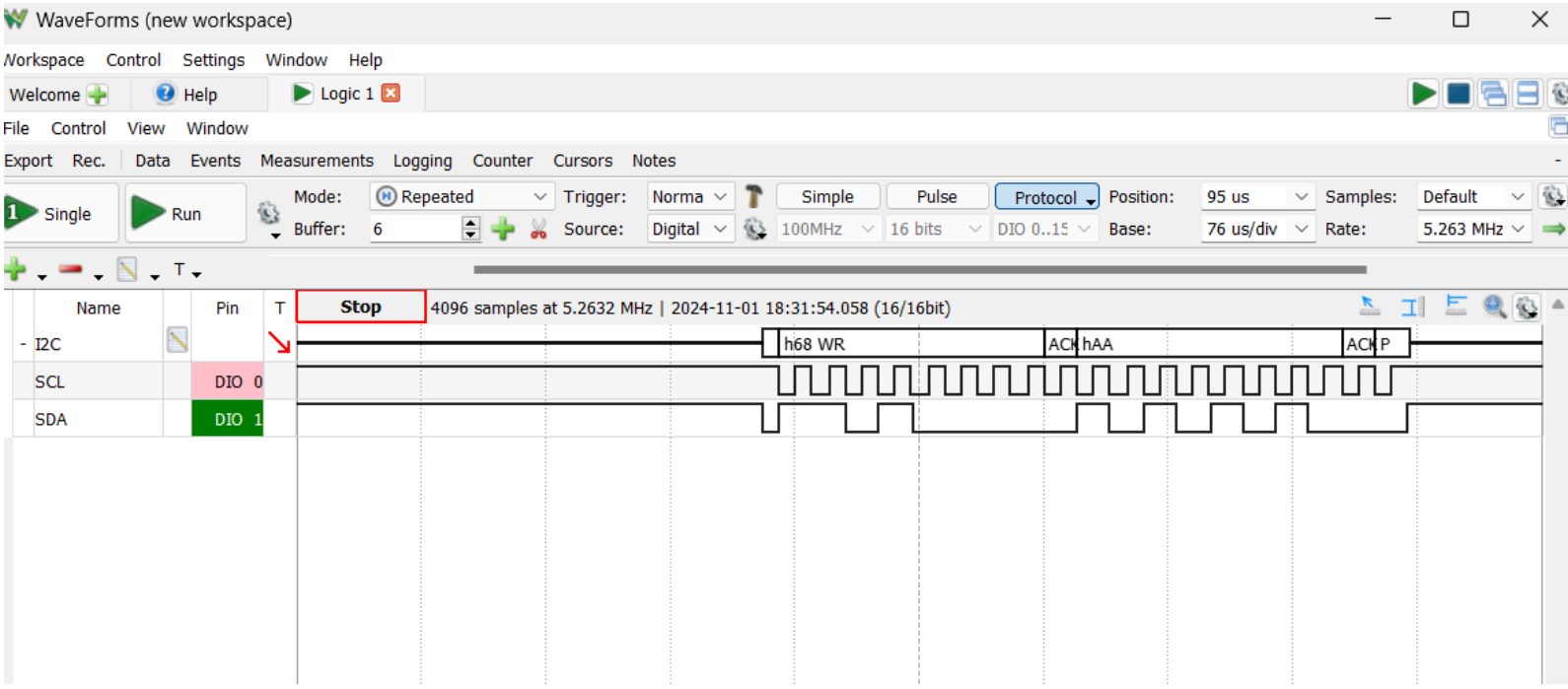


Image 3.2-3: 0xAA to TXBUF

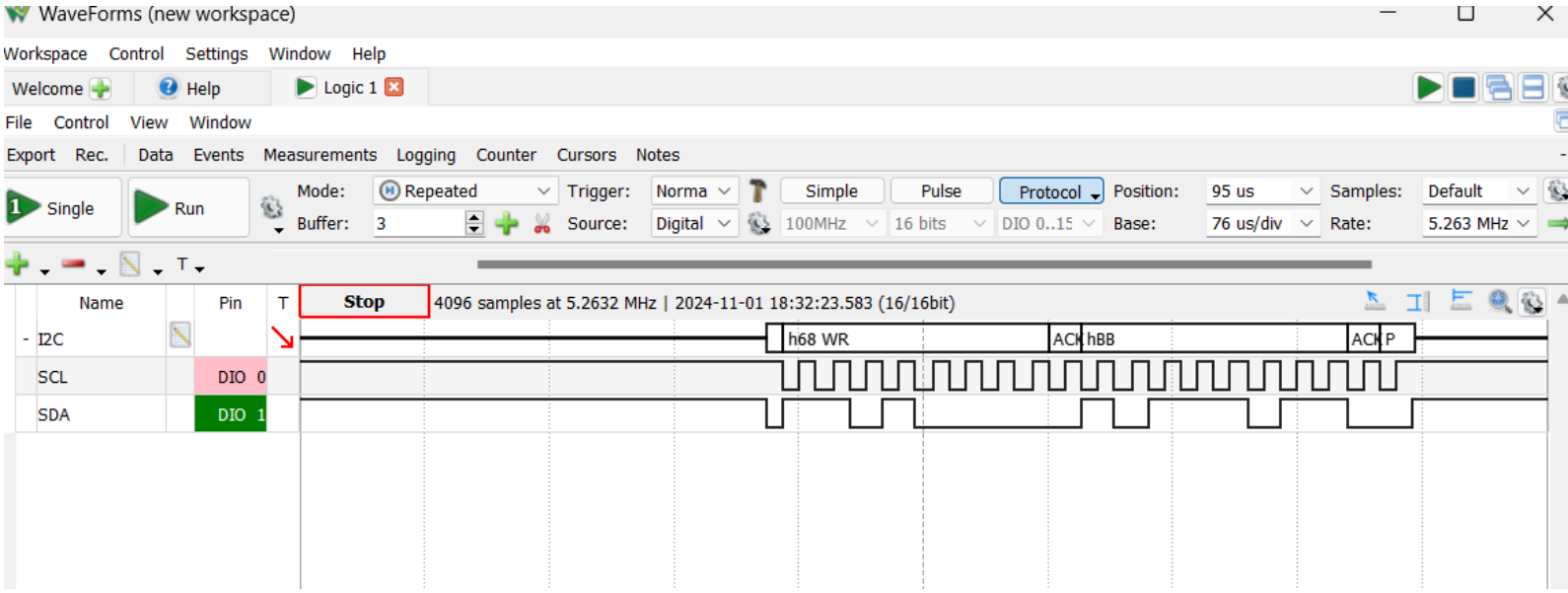


Image 3.2-4: 0xBB to TXBUF

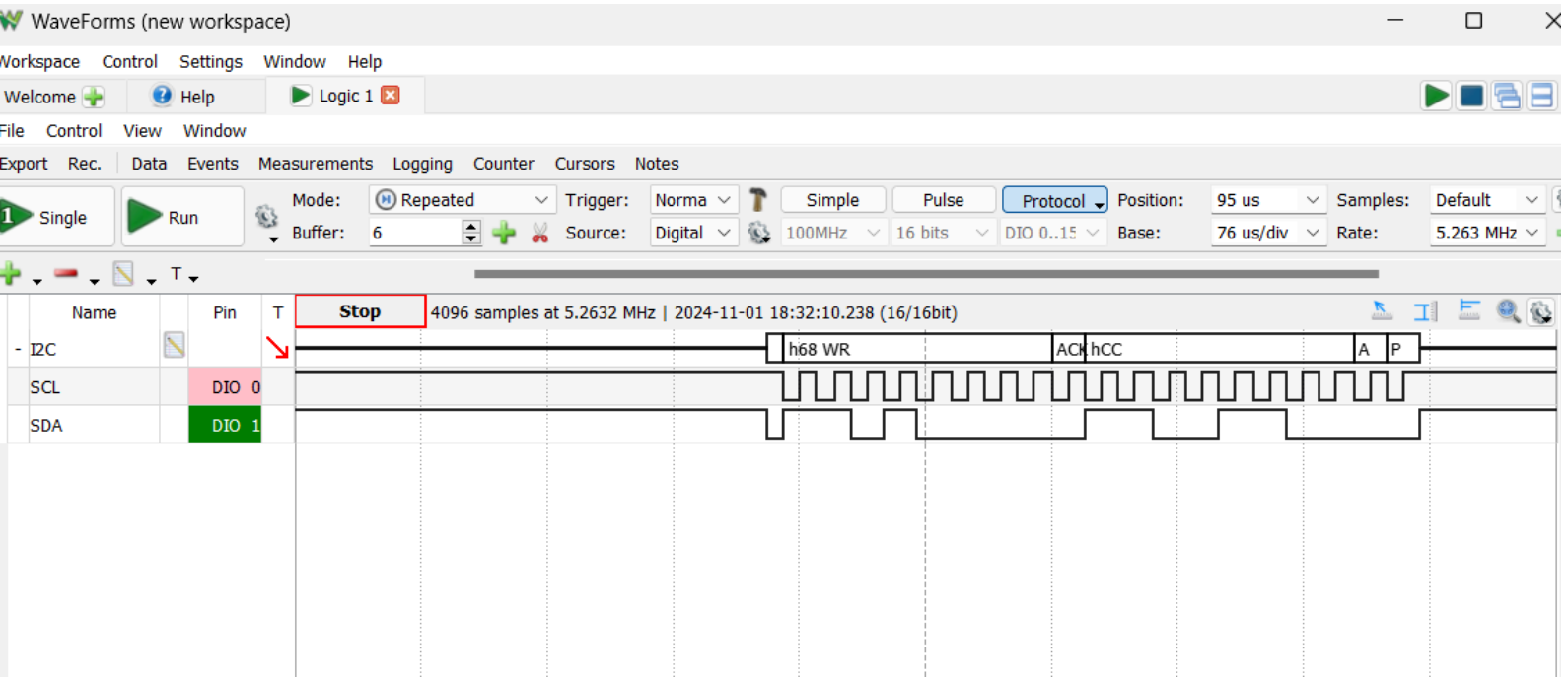


Image 3.2-5: 0xCC to TXBUF

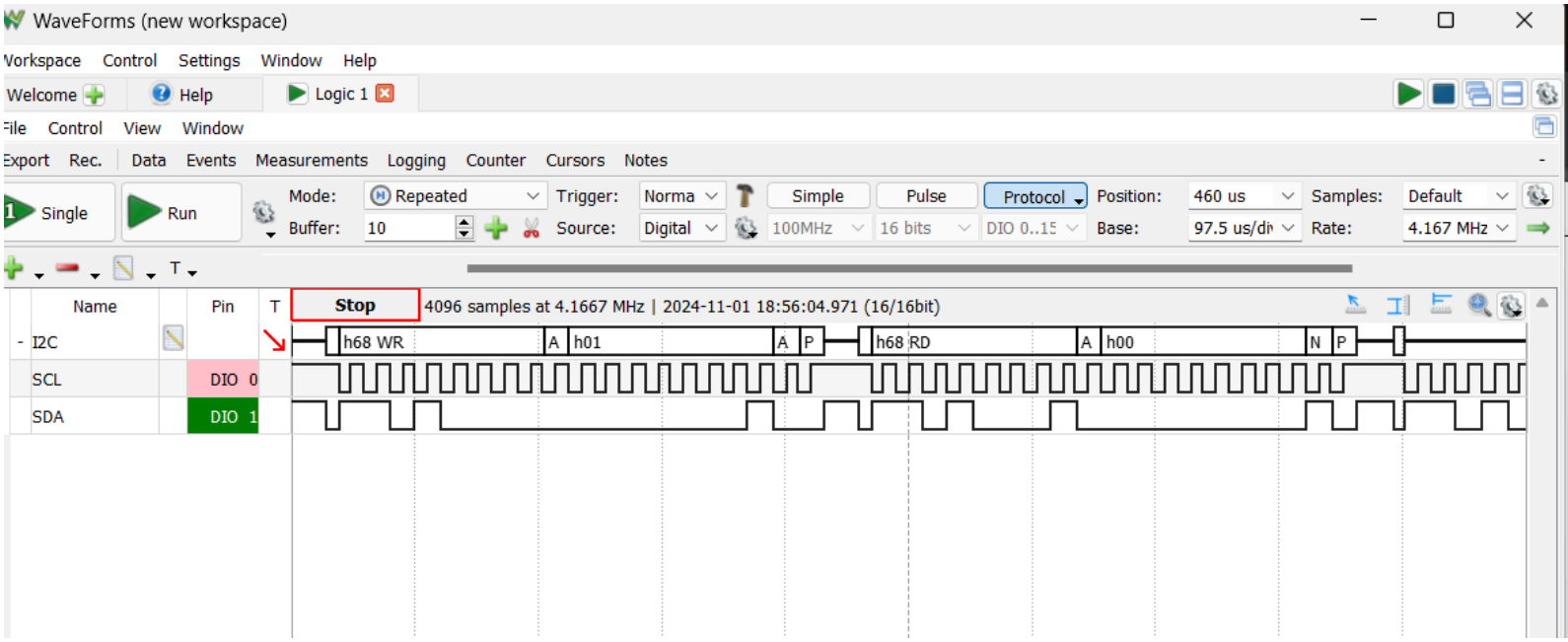


Image 3.2-6: send the register address and read the data (0x01) returned by the RTC

## 3.3 Code

```c
#include "msp.h"
#include <stdio.h>
#include <stdint.h>

uint8_t Packet[] = {0x03, 0xAA, 0xBB, 0xCC}; // Data packet: register address + 3 data bytes
uint8_t Data_Cnt = 0; // Counter for the number of bytes sent

void main(void) {
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // Stop the WatchDog Timer

    // Configure EUSCI_B0 for I2C communication
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;        // Put eUSCI_B0 in reset state for configuration
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_UCSSEL_2;     // Select SMCLK (6 MHz) as clock source
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_MODE_3;       // Set to I2C mode
    EUSCI_B0->BRW = 60;                            // Set baud rate (SCL = 100 KHz)
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_MST;          // Set as I2C master mode
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;           // Set to transmit mode (TX)

    EUSCI_B0->I2CSA = 0x68;                        // Set the I2C slave address
    EUSCI_B0->CTLW1 |= EUSCI_B_CTLW1_ASTP_2;       // Enable auto-stop mode
    EUSCI_B0->TBCNT = 1;                           // Set byte count for auto-stop (1 byte)

    // Configure GPIO pins P1.6 (SCL) and P1.7 (SDA) for I2C
    P1->SEL0 |= BIT6 | BIT7;
    P1->SEL1 &= ~(BIT6 | BIT7);

    // Take EUSCI_B0 out of reset mode
    EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_SWRST;

```

```
    // Enable TX interrupt for EUSCI_B0 and global interrupts
    EUSCI_B0->IE |= EUSCI_B_IE_TXIE0;              // Enable TX interrupt
    NVIC_EnableIRQ(EUSCIB0_IRQn);                  // Enable EUSCI_B0 interrupt in NVIC
    __enable_irq();                                // Enable global interrupts

    while (1) {
        int i = 0;

        // Initiate I2C transmission by setting the START condition
        EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;

        // Optional delay before the next transmission
        for (i = 0; i < 10000; i++) {}
    }
}

// EUSCI_B0 Interrupt Service Routine (ISR) for I2C transmission
void EUSCIB0_IRQHandler(void) {
    int i = 0;

    // Check if TX interrupt flag is set
    if (1) {  // EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG0
        if (Data_Cnt < sizeof(Packet)) {           // Check if there are more bytes to send
            EUSCI_B0->TXBUF = Packet[Data_Cnt++];  // Load next byte from Packet array into TX buffer

            // delay for stability
            for (i = 0; i < 10000; i++) {}
        }

         // Reset Data_Cnt for next transmission cycle once all bytes are sent
         if (Data_Cnt == sizeof(Packet)) {
             Data_Cnt = 0;
         }
     }
}

```

Image 3.3-1: code for I2C Write

```c
#include "msp.h"
char Data_In = 0; // Variable to store received data

void main(void) {
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD; // Stop the watchdog timer

    // Configure EUSCI_B0 for I2C communication
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;     // Put eUSCI_B0 in reset mode
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_UCSSEL_3;  // Select SMCLK as clock source
    EUSCI_B0->BRW = 60;                         // Set baud rate (SMCLK divided by 60)
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_MODE_3;    // Set mode to I2C
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_MST;       // Set as I2C master mode

    EUSCI_B0->I2CSA = 0x68;                     // Set the I2C slave address
    EUSCI_B0->CTLW1 |= EUSCI_B_CTLW1_ASTP_2;    // Enable auto-stop mode
    EUSCI_B0->TBCNT = 1;                        // Set byte count for transmission (1 byte)

    // Configure GPIO pins P1.6 and P1.7 for I2C functionality (SCL and SDA)
    P1->SEL0 |= BIT6 | BIT7;
    P1->SEL1 &= ~(BIT6 | BIT7);

    // Take EUSCI_B0 out of reset mode
    EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_SWRST;

    // Enable TX and RX interrupts
    EUSCI_B0->IE |= EUSCI_B_IE_TXIE0; // Enable TX interrupt
    EUSCI_B0->IE |= EUSCI_B_IE_RXIE0; // Enable RX interrupt
    __enable_irq();                   // Enable global interrupts
    NVIC_EnableIRQ(EUSCIB0_IRQn);     // Enable EUSCI_B0 interrupt in NVIC

    while(1) {
        // Transmit data to the slave
        EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;     // Set to transmit mode
        EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;  // Generate a START condition

        // Wait until stop condition is sent (polling for stop flag)
        while((EUSCI_B0->IFG & EUSCI_B_IFG_STPIFG) == 0);
        EUSCI_B0->IFG &= ~EUSCI_B_IFG_STPIFG;    // Clear the stop flag

        // Receive data from the slave
        EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_TR;    // Set to receive mode
        EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;  // Generate a repeated START condition for reading

        // Wait until stop condition is sent (polling for stop flag)
        while((EUSCI_B0->IFG & EUSCI_B_IFG_STPIFG) == 0);
        EUSCI_B0->IFG &= ~EUSCI_B_IFG_STPIFG;    // Clear the stop flag
    }
}
```

```
49
50 // EUSCI_B0 Interrupt Service Routine (ISR)
51 void EUSCIB0_IRQHandler(void) {
52     // Switch on the interrupt vector to handle TX and RX interrupts
53     switch(EUSCI_B0->IV) {
54     case 0x16: // RX interrupt flag
55         Data_In = EUSCI_B0->RXBUF; // Read received data from RX buffer
56         break;
57     case 0x18: // TX interrupt flag
58         EUSCI_B0->TXBUF = 0x03;    // Write data (0x03) to TX buffer for transmission
59         break;
60     default:
61         break;
62     }
63 }
```

Image 3.3-2: code for I2C Read

4. Discussions and Conclusions:

4.1 Comparison of your experimental results with the research/theory of the concept with reasoning.

The experimental results from this lab align closely with the theoretical expectations of I2C communication and the behavior of the MSP432 in master mode. The I2C protocol theory suggests that data transmission between a master and a slave device should occur in a synchronized manner, with the master generating clock pulses and controlling data flow through START and STOP conditions. In practice, this was observed when the MSP432 successfully sent and received data from the Adafruit RTC through the expected SDA and SCL lines. The observed waveforms on the logic analyzer closely matched the timing diagrams presented in I2C protocol documentation, showing distinct START and STOP conditions and proper data acknowledgment.

According to theory, setting the correct baud rate and enabling automatic STOP conditions should facilitate smooth communication without data loss. This was confirmed experimentally when configuring the EUSCI_B0 module's baud rate to 100kHz (using SMCLK with a divider) produced stable data transmissions. Interrupt-based communication, as implemented in the lab, is theoretically more efficient for handling real-time data transfer, minimizing processor idling compared to polling. Experimentally, using an ISR for data transmission proved effective, with the MSP432 managing each byte of data promptly and correctly resetting the index once the packet was fully sent, as anticipated.

4.2 Learnings from the experiment

This lab provided practical experience in I2C communication, embedded programming, and signal monitoring using the MSP432 microcontroller as an I2C master to communicate with an Adafruit RTC. Key concepts covered included configuring the EUSCI_B0 module for I2C master mode, setting up clock sources and baud rates, and managing data flow using flag polling and transmission start flags.

Implementing an interrupt service routine (ISR) to handle data transmission and reception showcased the efficiency of interrupt-driven programming. Monitoring I2C signals with the Waveforms application added a visual component, reinforcing understanding of the protocol's timing and structure.

One of the biggest challenges during the lab was realizing that the Waveforms application needed to be unplugged and reset to display the output accurately. This issue likely occurred due to the way Waveforms interfaces with the Analog Discovery hardware, which can sometimes hold previous configurations in memory or become misaligned with the current signals. Resetting the connection ensured that the software correctly reinitialized and detected real-time data, allowing for accurate signal observation. This troubleshooting step highlighted the importance of managing hardware-software synchronization, particularly in setups requiring live data visualization.

5. References:

Wikipedia, https://en.wikipedia.org/wiki/I%C2%B2C
Texas Instruments, https://www.ti.com/tool/CCSTUDIO
NXP, https://www.nxp.com/docs/en/data-sheet/PCF8523.pdf