Project 8: Serial Communications, Part 2 – I²C

Michael Steven Salvatierra Ramírez

Date: 10/16/2025

EE 3123 Embedded Systems

1. Introduction

This project implements I²C communication between an MSP430G2553 (master) and an AT24C16 EEPROM (slave) to perform addressable reads/writes over a two-wire bus. The workflow was organized into five steps: issuing a START, sending the 7-bit device address with R/W, transmitting the start memory address, transferring data from the slave to the master, and finishing with a STOP. The MSP430's USCI_B0 was configured for I²C, managed ACK/NACK handshakes, and decoded three bytes to confirm correct addressing and data flow. All bus activity, START/STOP timing, address frames, ACKs, and data bytes was captured and verified on the oscilloscope, providing signal-level validation of the protocol sequence.

2. Methods

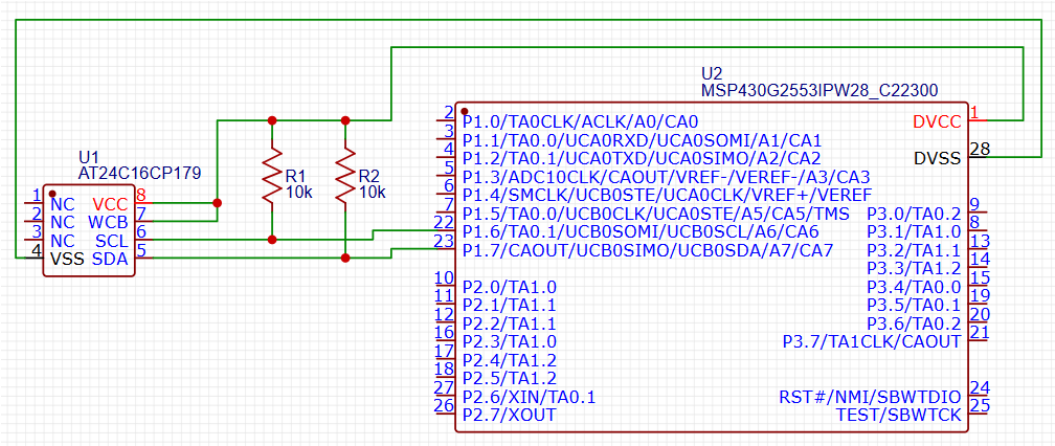

Figure 2.1 Schematic for the I²C circuit.

Step 1 — Send device address (0x50) and read flag

This program configures the MSP430G2553's USCI_B0 to act as an I²C master and send only the device-address phase to an AT24C16 EEPROM, then issues a STOP. In i2c_init(), P1.6/P1.7 are routed to I²C (SCL/SDA), USCI_B0 is held in reset, set to master, I²C, synchronous mode, and clocked from SMCLK. With SMCLK assumed at 1 MHz, UCB0BR1:BR0 = 0:10 divides by 10 to target ≈100 kHz SCL.

The 7-bit slave address is loaded as 0x50 (AT24C16), and the module is released from reset. The helper i2c_start_write_then_wait_addr_ack() puts the bus in transmitter mode (UCTR) and generates a START (UCTXSTT) to send the 7-bit address + write bit; the USCI clears UCTXSTT when the address phase completes (ACK/NACK resolved). step1_start_addr() calls that helper, then immediately requests a STOP and waits until the STOP condition completes. main() disables the watchdog, initializes I²C, performs "Step 1", and idles forever.

```
#include <msp430.h>
#include <stdint.h>

#define I2C_DEV7_AT24C16 0x50 //AT24C16, page 000 -> 0b1010_000 = 0x50

void i2c_init(void) {
    // Select I2C function on P1.6/1.7
    P1SEL  |= BIT6 | BIT7;
    P1SEL2 |= BIT6 | BIT7;

    UCB0CTL1 = UCSWRST;                     // hold USCI_B0 in reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;  // Master, I2C, synchronous
    UCB0CTL1 = UCSWRST + UCSSEL_2;          // SMCLK

    UCB0BR0 = 10; //100 kHz SCL
    UCB0BR1 = 0;

    UCB0I2CSA = I2C_DEV7_AT24C16;           // 7-bit slave address
    UCB0CTL1 &= ~UCSWRST;                   // release for operation
}

int i2c_start_write_then_wait_addr_ack(void) {
    // Wait until address phase finishes or a NACK shows up
    while (UCB0CTL1 & UCTXSTT);
    UCB0CTL1 |= UCTR + UCTXSTT;         // Transmitter mode + START
    return 0;
}

void step1_start_addr(void) {
    i2c_start_write_then_wait_addr_ack();  // START + DeviceAddress(W)
    UCB0CTL1 |= UCTXSTP;                    // STOP to close it out
    while (UCB0CTL1 & UCTXSTP) ;
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    i2c_init();
    // --- STEP 1: START + Device Address (Write) + ACK
    step1_start_addr();

    while (1);
}
```

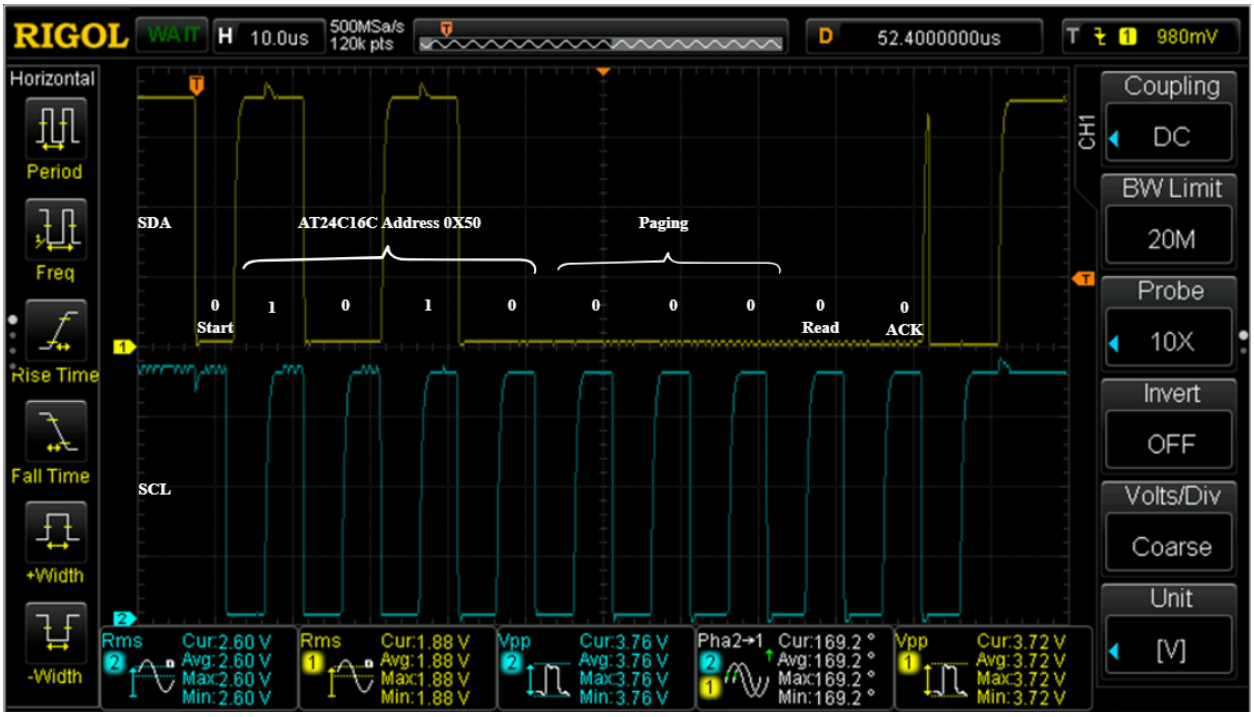

Figure 2.2. I²C waveform showing the START condition, device address, read and ACK

Step 2 — Send the EEPROM word address (0xE0).

step2_send_word_addr(0xE0) performs a single-byte write to set the AT24C16's internal address pointer. It begins a write transaction (UCTR | UCTXSTT), so USCI_B0 sends the 7-bit slave address 0x50 with the write bit and waits for the ACK. Then i2c_send_byte waits for UCB0TXIFG, loads UCB0TXBUF with 0xE0, and waits again for UCB0TXIFG to re-assert, indicating the byte has shifted out and was ACKed by the EEPROM. Finally, it issues UCTXSTP and waits until the STOP completes. On the AT24C16, this byte provides the low 8 bits of the memory address; the upper address bits (block select) are encoded in the device (0x50 → A2..A0 = 000), so 0xE0 targets offset 0xE0 within that block. This step doesn't read or write data yet, it just positions the EEPROM's internal pointer for subsequent operations.

```
#include <msp430.h>
#include <stdint.h>

#define I2C_DEV7_AT24C16 0x50 //AT24C16, page 000 -> 0b1010_000 = 0x50

void i2c_init(void) {
    // Select I2C function on P1.6/1.7
    P1SEL  |= BIT6 | BIT7;
    P1SEL2 |= BIT6 | BIT7;

    UCB0CTL1 = UCSWRST;                     // hold USCI_B0 in reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;  // Master, I2C, synchronous
    UCB0CTL1 = UCSWRST + UCSSEL_2;          // SMCLK

    UCB0BR0 = 10; //100 kHz SCL
    UCB0BR1 = 0;
```

```
    UCB0I2CSA = I2C_DEV7_AT24C16;          // 7-bit slave address
    UCB0CTL1 &= ~UCSWRST;                  // release for operation
}

int i2c_start_write_then_wait_addr_ack(void) {
    // Wait until address phase finishes or a NACK shows up
    while (UCB0CTL1 & UCTXSTT);
    UCB0CTL1 |= UCTR + UCTXSTT;        // Transmitter mode + START
    return 0;
}

int i2c_send_byte(uint8_t b) {
    while (!(IFG2 & UCB0TXIFG));
    UCB0TXBUF = b;
    // wait for the byte shift (TXIFG set again for next byte)
    while (!(IFG2 & UCB0TXIFG));
    return 0;
}

void step1_start_addr(void) {
    i2c_start_write_then_wait_addr_ack();  // START + DeviceAddress(W)
    UCB0CTL1 |= UCTXSTP;                   // STOP to close it out
    while (UCB0CTL1 & UCTXSTP) ;
}

void step2_send_word_addr(uint8_t word) {
    if (i2c_start_write_then_wait_addr_ack() == 0
         && i2c_send_byte(word) == 0)
    {
        UCB0CTL1 |= UCTXSTP;
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    i2c_init();
    // --- STEP 1: START + Device Address (Write) + ACK
    step1_start_addr();
    // --- STEP 2: + Word Address byte (0xE0)
    step2_send_word_addr(0xE0);

    while (1);
}
```

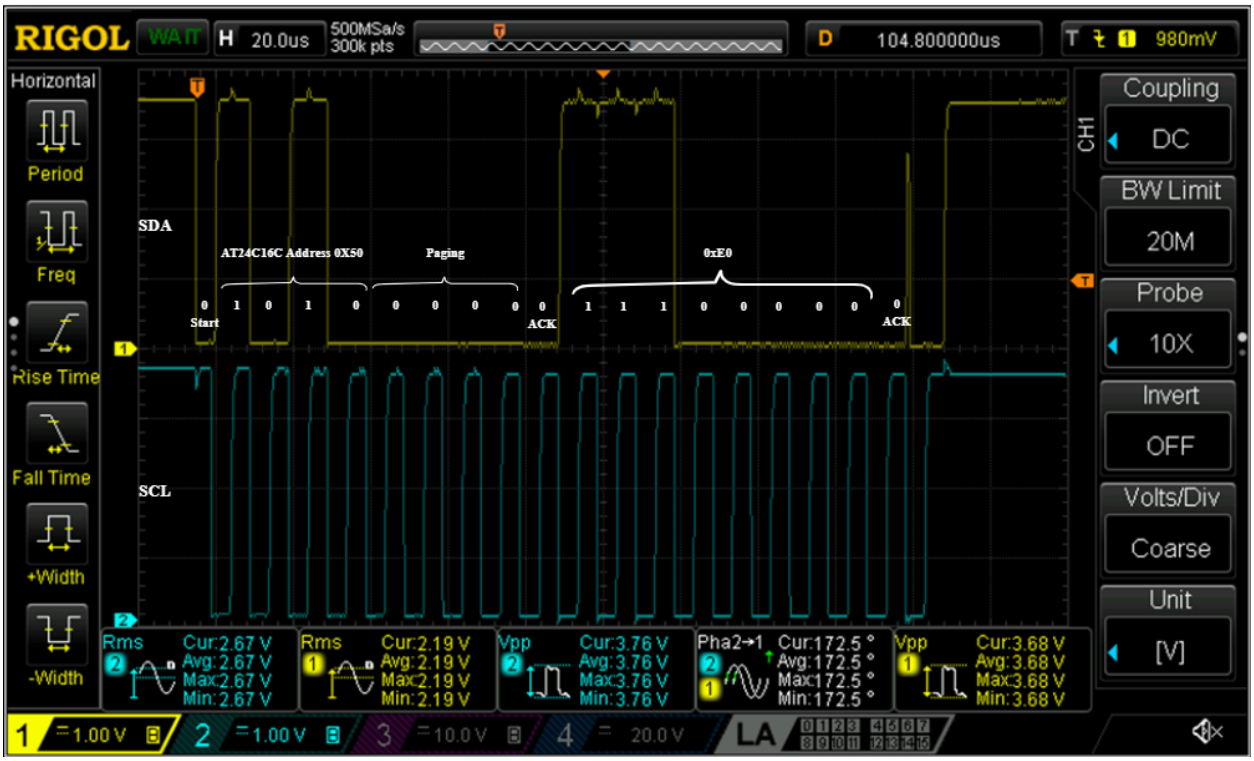

Figure 2.3. I²C waveform showing the START, device address (0x50 + Write), word address (0xE0), ACK, and STOP from the EEPROM.

Step 3 — Repeated-START for a read (address pointer already set).

step3_rs_addr_read() first does a brief write phase to load the EEPROM's internal word address (i2c_start_write_then_wait_addr_ack(); i2c_send_byte(0xE0)). Without issuing a STOP, it then switches USCI_B0 to receiver mode (UCTR=0) and asserts a repeated START (UCTXSTT), which re-addresses the same slave (0x50) but now with the read bit. When the address phase completes (USCI clears UCTXSTT), the bus is in a legal read transaction at address 0xE0. For this demo, no data byte is actually fetched—the code issues UCTXSTP immediately and waits for it to complete—so the sequence exercises the classic Write word-addr → Repeated-START → Read pattern without transferring payload. In a real read, you would: (1) wait for UCB0RXIFG, (2) read from UCB0RXBUF, and (3) for the final byte, set UCTXSTP *before* reading it so the USCI generates NACK+STOP on that last byte. Optionally check UCNACKIFG to detect a NACK at either address phase.

```
#include <msp430.h>
#include <stdint.h>

#define I2C_DEV7_AT24C16 0x50 //AT24C16, page 000 -> 0b1010_000 = 0x50

void i2c_init(void) {
    // Select I2C function on P1.6/1.7
    P1SEL  |= BIT6 | BIT7;
    P1SEL2 |= BIT6 | BIT7;

    UCB0CTL1 = UCSWRST;                     // hold USCI_B0 in reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;  // Master, I2C, synchronous
```

```
    UCB0CTL1 = UCSWRST + UCSSEL_2;         // SMCLK

    UCB0BR0 = 10; //100 kHz SCL
    UCB0BR1 = 0;

    UCB0I2CSA = I2C_DEV7_AT24C16;          // 7-bit slave address
    UCB0CTL1 &= ~UCSWRST;                  // release for operation
}

int i2c_start_write_then_wait_addr_ack(void) {
    // Wait until address phase finishes or a NACK shows up
    while (UCB0CTL1 & UCTXSTT);
    UCB0CTL1 |= UCTR + UCTXSTT;        // Transmitter mode + START
    return 0;
}

int i2c_send_byte(uint8_t b) {
    while (!(IFG2 & UCB0TXIFG));
    UCB0TXBUF = b;
    // wait for the byte shift (TXIFG set again for next byte)
    while (!(IFG2 & UCB0TXIFG));
    return 0;
}

int i2c_repeated_start_read_then_wait_addr_ack(void) {
    UCB0CTL1 &= ~UCTR;    // Receiver mode
    UCB0CTL1 |= UCTXSTT;  // Repeated START
    // Wait for address to complete (START cleared)
    while (UCB0CTL1 & UCTXSTT);
    return 0;
}

void step1_start_addr(void) {
    i2c_start_write_then_wait_addr_ack();  // START + DeviceAddress(W)
    UCB0CTL1 |= UCTXSTP;                   // STOP to close it out
    while (UCB0CTL1 & UCTXSTP) ;
}

void step2_send_word_addr(uint8_t word) {
    if (i2c_start_write_then_wait_addr_ack() == 0
         && i2c_send_byte(word) == 0))
    {
        UCB0CTL1 |= UCTXSTP;
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

static void step3_rs_addr_read(void) {
    if (i2c_start_write_then_wait_addr_ack() == 0) {
        i2c_send_byte(0xE0);               // word address
        // repeated START + Addr(R)
        i2c_repeated_start_read_then_wait_addr_ack();
        UCB0CTL1 |= UCTXSTP;
        while (UCB0CTL1 & UCTXSTP) ;
```

```
    }
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    i2c_init();
    // --- STEP 1: START + Device Address (Write) + ACK
    // step1_start_addr();
    // --- STEP 2: + Word Address byte (0xE0)
    // step2_send_word_addr(0xE0);
    // --- STEP 3: Repeated START + Device Address (Read)
    step3_rs_addr_read();
    while (1);
}
```

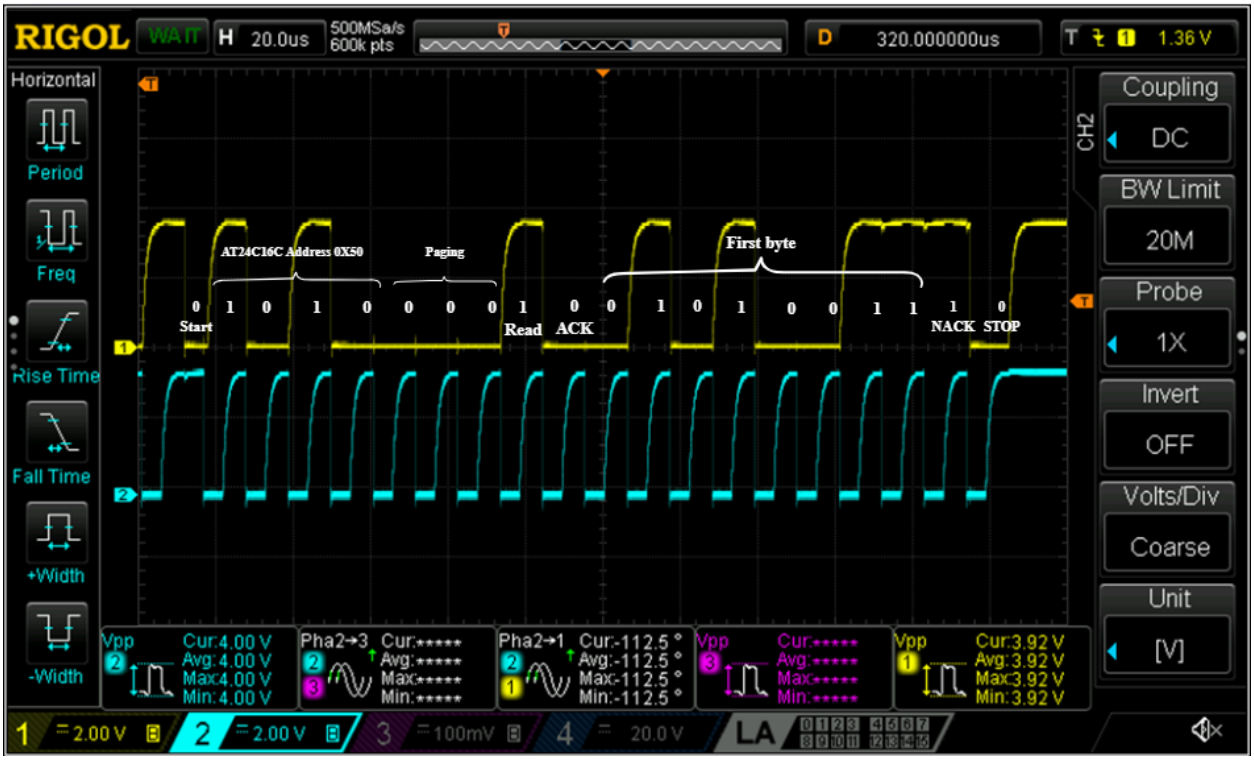

Figure 2.4. I^{2}C waveform showing the repeated START condition, device address (0x50 + Read), ACK from the EEPROM, followed by the first data byte, and ending with the NACK and STOP condition generated by the MSP430.

Figure 2.5. Full I^{2}C sequence up to Step 3.

Step 4 — Read the next byte after Step 3.

Since Step 3 already retrieved the first data byte at 0xE0 (via repeated-START + Read and a single-byte NACK+STOP), Step 4 proceeds to fetch the following byte at 0xE1. Functionally, this is a sequential read: the EEPROM's internal pointer auto-incremented to 0xE1 after the 0xE0 read, so the transaction now targets that address. In practice you can either (a) issue a fresh random read starting at 0xE1 (START → DevAddr(W) → 0xE1 → repeated START → DevAddr(R) → read one byte with NACK+STOP), or (b) if you had left the bus open in Step 3, simply read one more byte and then NACK+STOP. Upon return, the newly read byte corresponds to 0xE1 (i.e., the "second" byte following the one obtained in Step 3).

```
#include <msp430.h>
#include <stdint.h>

#define I2C_DEV7_AT24C16 0x50 //AT24C16, page 000 -> 0b1010_000 = 0x50

void i2c_init(void) {
    // Select I2C function on P1.6/1.7
    P1SEL  |= BIT6 | BIT7;
    P1SEL2 |= BIT6 | BIT7;

    UCB0CTL1 = UCSWRST;                     // hold USCI_B0 in reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;   // Master, I2C, synchronous
    UCB0CTL1 = UCSWRST + UCSSEL_2;          // SMCLK

    UCB0BR0 = 10; //100 kHz SCL
```

```
    UCB0BR1 = 0;

    UCB0I2CSA = I2C_DEV7_AT24C16;          // 7-bit slave address
    UCB0CTL1 &= ~UCSWRST;                  // release for operation
}

int i2c_start_write_then_wait_addr_ack(void) {
    // Wait until address phase finishes or a NACK shows up
    while (UCB0CTL1 & UCTXSTT);
    UCB0CTL1 |= UCTR + UCTXSTT;        // Transmitter mode + START
    return 0;
}

int i2c_send_byte(uint8_t b) {
    while (!(IFG2 & UCB0TXIFG));
    UCB0TXBUF = b;
    // wait for the byte shift (TXIFG set again for next byte)
    while (!(IFG2 & UCB0TXIFG));
    return 0;
}

int i2c_repeated_start_read_then_wait_addr_ack(void) {
    UCB0CTL1 &= ~UCTR;    // Receiver mode
    UCB0CTL1 |= UCTXSTT;  // Repeated START
    // Wait for address to complete (START cleared)
    while (UCB0CTL1 & UCTXSTT);
    return 0;
}

uint8_t i2c_rx_one_with_stop(void) {
    UCB0CTL1 |= UCTXSTP;     // NACK last byte + STOP after it arrives
    while (!(IFG2 & UCB0RXIFG)) ;  // wait for the byte
    return UCB0RXBUF;
}

uint8_t i2c_rx_next(void) {
    while (!(IFG2 & UCB0RXIFG)) ;
    return UCB0RXBUF;
}

static uint8_t i2c_rx_last_with_stop(void) {
    //Generate NACK+STOP BEFORE reading the last byte
    UCB0CTL1 |= UCTXSTP;
    while (!(IFG2 & UCB0RXIFG)) ;
    return UCB0RXBUF;
}

int eeprom_read_n(uint8_t word_addr, uint8_t *buf, uint8_t n) {
    //Phase 1: START + Write + WordAddr
    if (i2c_start_write_then_wait_addr_ack() != 0) return -1;
    if (i2c_send_byte(word_addr) != 0) return -2;
    // Phase 2: REPEATED START + Read
    if (i2c_repeated_start_read_then_wait_addr_ack() != 0) return -3;
```

```
    // Read N-1 bytes with ACK (USCI auto-ACKs intermediate bytes)
    int i = 0;
    for (i = 0; i < n - 1; ++i) {
        buf[i] = i2c_rx_next();
    }
    // Last byte: NACK + STOP
    buf[n - 1] = i2c_rx_last_with_stop();
    while (UCB0CTL1 & UCTXSTP) ; // ensure STOP sent

    return 0;
}

void step1_start_addr(void) {
    i2c_start_write_then_wait_addr_ack();  // START + DeviceAddress(W)
    UCB0CTL1 |= UCTXSTP;                   // STOP to close it out
    while (UCB0CTL1 & UCTXSTP) ;
}

void step2_send_word_addr(uint8_t word) {
    if (i2c_start_write_then_wait_addr_ack() == 0 &&
    i2c_send_byte(word) == 0)
    {
        UCB0CTL1 |= UCTXSTP;
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

void step3_rs_addr_read(void) {
    if (i2c_start_write_then_wait_addr_ack() == 0) {
        i2c_send_byte(0xE0);               // prime a word address
        // repeated START + Addr(R)
        i2c_repeated_start_read_then_wait_addr_ack();
        UCB0CTL1 |= UCTXSTP;              // close without reading
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

void step4_read_second_byte(void) {
    uint8_t buf[3];
    eeprom_read_n(0xE0, buf, 2); // read 0xE0, 0xE1
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    i2c_init();
    // --- STEP 1: START + Device Address (Write) + ACK
    // step1_start_addr();
    // --- STEP 2: + Word Address byte (0xE0)
    // step2_send_word_addr(0xE0);
    // --- STEP 3: Repeated START + Device Address (Read) + first byte
    // step3_rs_addr_read();
    // --- STEP 4: Read next byte
    step4_read_second_byte();
    while (1);
```

}

Figure 2.6. I^{2}C waveform showing the second data byte read from the EEPROM during Step 4, where the master sends an ACK after receiving the byte and immediately issues a STOP condition, returning the bus to the idle state.

Figure 2.6. Full I^2C sequence up to Step 4.

Step 5 — Reading three consecutive bytes (0xE0, 0xE1, 0xE2).

In Step 5, the function step5_read_second_byte() calls eeprom_read_n(0xE0, buf, 3) to perform a sequential read of three bytes starting at memory address 0xE0. This routine sends a START condition, the device address with the write bit, and the word address (0xE0) to set the EEPROM's internal pointer. Then, a repeated START followed by the device address with the read bit initiates the read phase. The first two bytes (from 0xE0 and 0xE1) are automatically ACKed by the MSP430's USCI module, keeping the bus active for more data. Before the last byte (from 0xE2), the master asserts UCTXSTP, causing a NACK + STOP once that final byte is received. This cleanly ends the transaction, leaving buf[0], buf[1], and buf[2] containing the values from EEPROM addresses 0xE0, 0xE1, and 0xE2, respectively.

```
#include <msp430.h>
#include <stdint.h>

#define I2C_DEV7_AT24C16 0x50 //AT24C16, page 000 -> 0b1010_000 = 0x50


void i2c_init(void) {
    // Select I2C function on P1.6/1.7
    P1SEL  |= BIT6 | BIT7;
    P1SEL2 |= BIT6 | BIT7;

    UCB0CTL1 = UCSWRST;                     // hold USCI_B0 in reset
    UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;  // Master, I2C, synchronous
    UCB0CTL1 = UCSWRST + UCSSEL_2;          // SMCLK

    UCB0BR0 = 10;
    UCB0BR1 = 0;

    UCB0I2CSA = I2C_DEV7_AT24C16;           // 7-bit slave address
    UCB0CTL1 &= ~UCSWRST;                   // release for operation
}

int i2c_start_write_then_wait_addr_ack(void) {
    // Wait until address phase finishes or a NACK shows up
    while (UCB0CTL1 & UCTXSTT);
    UCB0CTL1 |= UCTR + UCTXSTT;         // Transmitter mode + START
    return 0;
}

int i2c_send_byte(uint8_t b) {
    while (!(IFG2 & UCB0TXIFG));
    UCB0TXBUF = b;
    // wait for the byte shift (TXIFG set again for next byte)
    while (!(IFG2 & UCB0TXIFG));
    return 0;
}
```

```
int i2c_repeated_start_read_then_wait_addr_ack(void) {
    UCB0CTL1 &= ~UCTR;    // Receiver mode
    UCB0CTL1 |= UCTXSTT;  // Repeated START
    // Wait for address to complete (START cleared)
    while (UCB0CTL1 & UCTXSTT);
    return 0;
}

uint8_t i2c_rx_one_with_stop(void) {
    UCB0CTL1 |= UCTXSTP;     // NACK last byte + STOP after it arrives
    while (!(IFG2 & UCB0RXIFG)) ;        // wait for the byte
    return UCB0RXBUF;
}

uint8_t i2c_rx_next(void) {
    while (!(IFG2 & UCB0RXIFG)) ;
    return UCB0RXBUF;
}

static uint8_t i2c_rx_last_with_stop(void) {
    //Generate NACK+STOP BEFORE reading the last byte
    UCB0CTL1 |= UCTXSTP;
    while (!(IFG2 & UCB0RXIFG)) ;
    return UCB0RXBUF;
}

int eeprom_read_n(uint8_t word_addr, uint8_t *buf, uint8_t n) {
    //Phase 1: START + Write + WordAddr
    if (i2c_start_write_then_wait_addr_ack() != 0) return -1;
    if (i2c_send_byte(word_addr) != 0) return -2;
    // Phase 2: REPEATED START + Read
    if (i2c_repeated_start_read_then_wait_addr_ack() != 0) return -3;

    // Read N-1 bytes with ACK (USCI auto-ACKs intermediate bytes)
    int i = 0;
    for (i = 0; i < n - 1; ++i) {
        buf[i] = i2c_rx_next();
    }
    // Last byte: NACK + STOP
    buf[n - 1] = i2c_rx_last_with_stop();
    while (UCB0CTL1 & UCTXSTP) ; // ensure STOP sent

    return 0;
}

void step1_start_addr(void) {
    i2c_start_write_then_wait_addr_ack();  // START + DeviceAddress(W)
    UCB0CTL1 |= UCTXSTP;                   // STOP to close it out
    while (UCB0CTL1 & UCTXSTP) ;
}
```

```
void step2_send_word_addr(uint8_t word) {
    if (i2c_start_write_then_wait_addr_ack() == 0 &&
        i2c_send_byte(word) == 0)
    {
        UCB0CTL1 |= UCTXSTP;
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

void step3_rs_addr_read(void) {
    if (i2c_start_write_then_wait_addr_ack() == 0) {
        i2c_send_byte(0xE0);               // prime a word address
       // repeated START + Addr(R)
        i2c_repeated_start_read_then_wait_addr_ack();
        UCB0CTL1 |= UCTXSTP;         // close without reading
        while (UCB0CTL1 & UCTXSTP) ;
    }
}

void step4_read_second_byte(void) {
    uint8_t buf[3];
    eeprom_read_n(0xE0, buf, 2); // read 0xE0, 0xE1
}

void step5_read_second_byte(void) {
    uint8_t buf[3];
    eeprom_read_n(0xE0, buf, 3); // read 0xE0, 0xE1, 0XE2
}



int main(void) {
    WDTCTL = WDTPW | WDTHOLD;
    i2c_init();
    // --- STEP 1: START + Device Address (Write) + ACK
    // step1_start_addr();
    // --- STEP 2: + Word Address byte (0xE0)
    // step2_send_word_addr(0xE0);
    // --- STEP 3: Repeated START + Device Address (Read) + first byte
    // step3_rs_addr_read();
    // --- STEP 4: Read next byte
    step4_read_second_byte();
    // --- STEP 5: Read next byte
    // step5_read_second_byte();
    while (1);
}
```

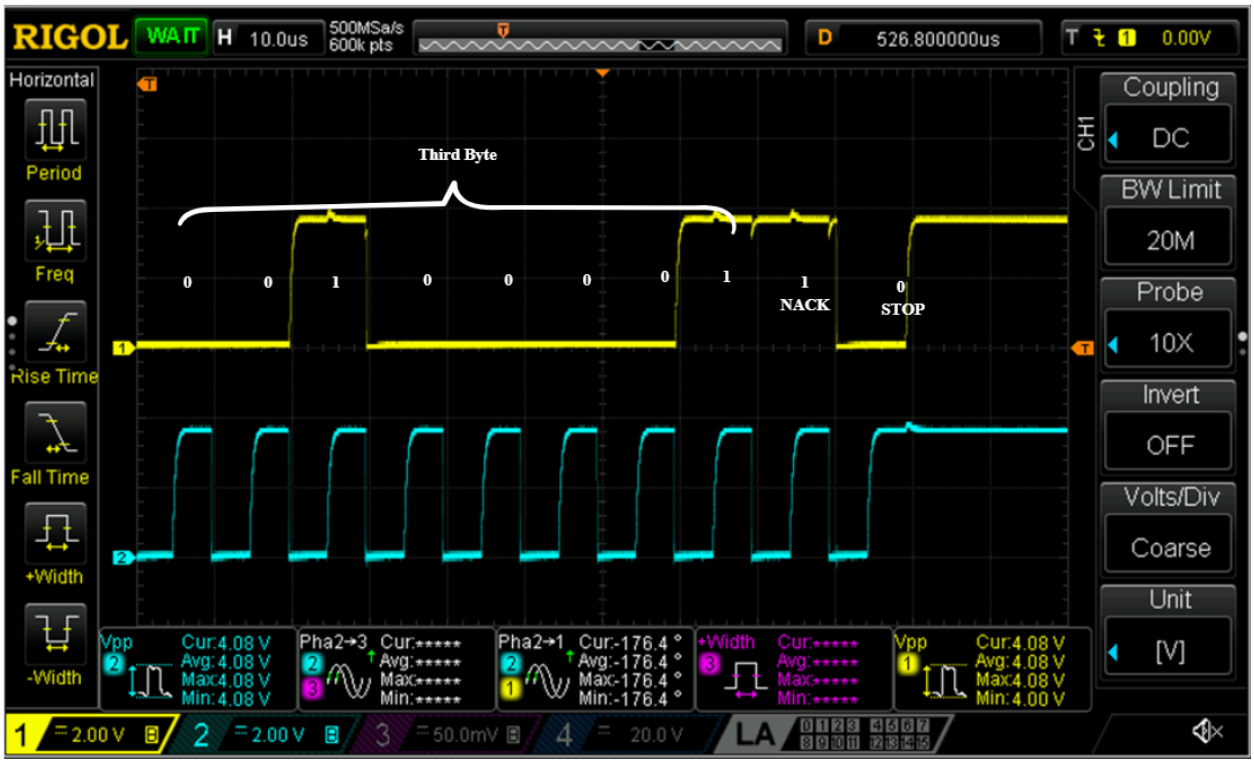

Figure 2.7. I^{2}C waveform showing the sequential read of three bytes from the EEPROM during Step 5, starting at address 0xE0. The master sends a START, followed by ACKs after the first two bytes, and finally issues a NACK + STOP after the third byte, returning the bus to the idle state.

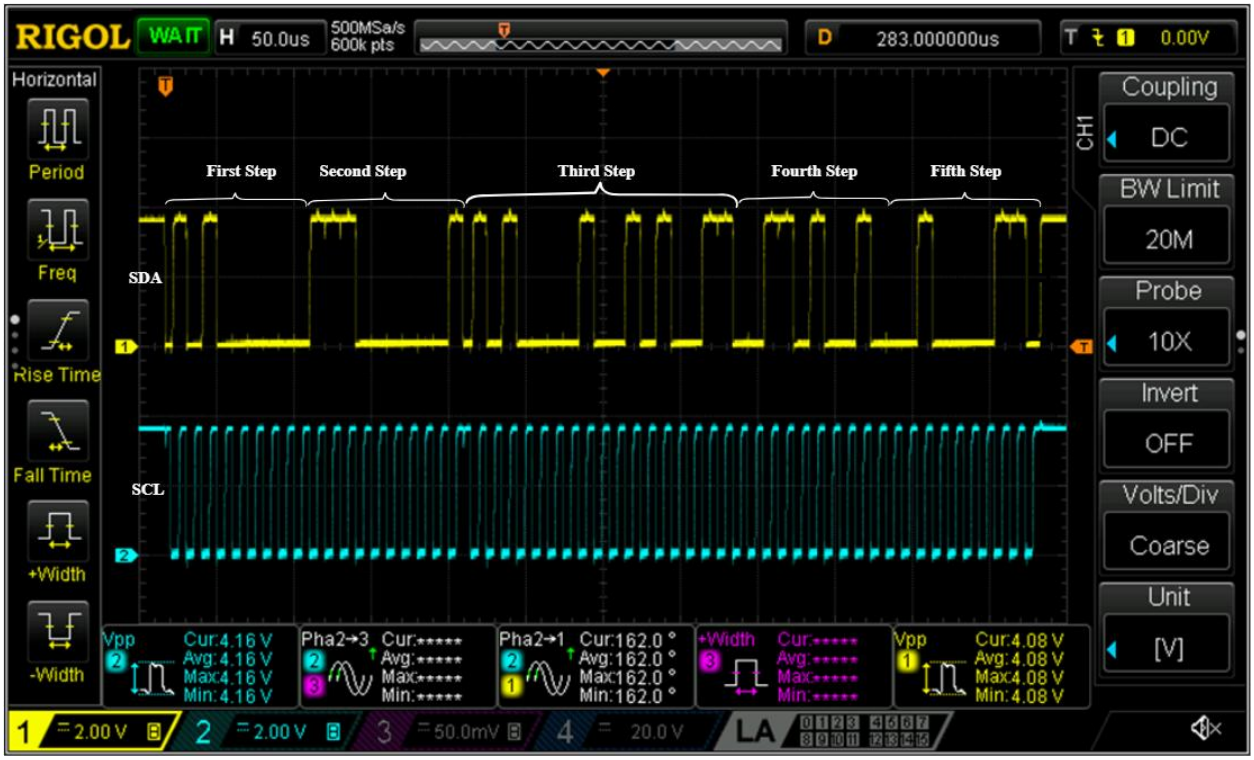

Figure 2.8. Full I^{2}C sequence up to Step

3. Results

Throughout the five-step I^2C communication sequence between the MSP430G2553 and the AT24C16 EEPROM, each transmission stage was successfully observed and verified using the oscilloscope.

In Step 1, the microcontroller generated a proper START condition, sent the device address (0x50 + Write), received an ACK, and completed the frame with a STOP condition.

In Step 2, the EEPROM's word address (0xE0) was transmitted, setting the internal pointer for subsequent reads, again showing a valid ACK and STOP on the oscilloscope.

Step 3 demonstrated a repeated START sequence followed by the device address with the Read bit, and the first data byte was successfully received. The waveform confirmed the repeated START and the ACK from the slave device.

In Step 4, the second data byte (0xE1) was read through a current address read, where the master sent an ACK and then issued a STOP condition, leaving the bus idle.

Finally, in Step 5, the EEPROM returned three consecutive bytes (0xE0, 0xE1, and 0xE2) in a sequential read operation. The MSP430 automatically ACKed the first two bytes and correctly ended the transmission with a NACK + STOP after the last byte. The resulting waveforms validated each phase of the I^2C protocol, start, addressing, ACK/NACK responses, data transfer, and stop, confirming reliable communication and correct EEPROM operation.

Having successfully retrieved the three secret bytes from the EEPROM, their values were decoded into the ASCII characters 'S', 'i', and '!', revealing the hidden message "Si!" and confirming that the data transmission and decoding process worked correctly.

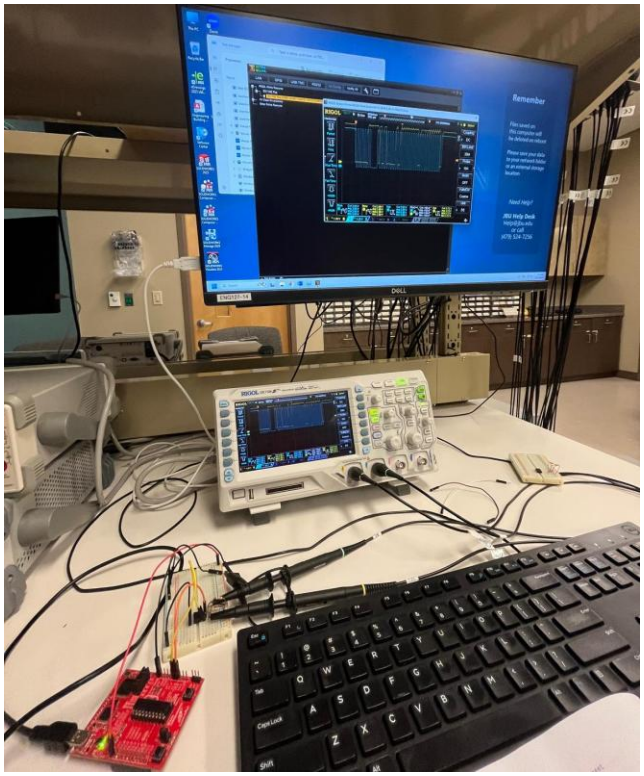

Figure 3.1. Setup for the project.

4. Conclusion

The I^2C link between the MSP430G2553 and the AT24C16 was implemented and verified on the oscilloscope, with correct START/STOP timing, address phases, ACK/NACK handling, and sequential reads via the EEPROM's auto-increment. This project was challenging I had to learn the oscilloscope, interpret bus timing, and adopt practical debugging (e.g., resetting the board when the lines latched low). The example codes Grace provided (12.19 and 12.21) were especially helpful as references. Crucially, implementing the work step-by-step with separate logic made it easier to backtrack when something failed and resulted in cleaner, reusable, and scalable code. Even though this has been the toughest project so far, this is my favorite as well.

5. References

Unsalan, C., & Gurhan, H. D. (2013). *Programmable Microcontrollers with Applications: MSP430 LaunchPad with CCS and Grace*. McGraw-Hill. (Examples 12.19 and 12.21 referenced.)