2017

Project 1 Report

ECE 461

ZACH HANNUM

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Contents

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Blocking Code Implementation

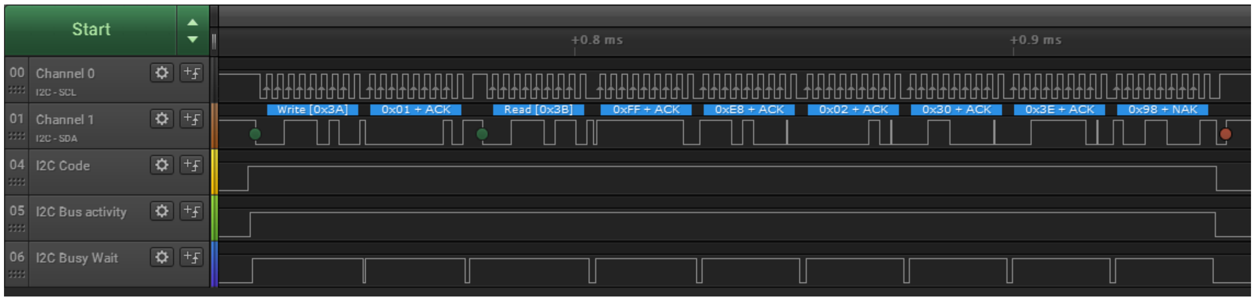

Figure 1: Blocking Code I²C message on logic analyzer

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Finite State Machine Implementation

Input and Output Data

Instead of being called once, like the blocking code, `i2c_read_bytes_fsm()` is called many times in order to go through each state and eventually finish reading all 6 bytes. To pass data to the FSM, the original parameters used in the blocking i^{2}c code were kept; more specifically the device address (`dev_adx`), the register address (`reg_adx`), the data array pointer, and the data count. Due to the simplicity of this project, no sophisticated handshaking was needed for the i^{2}c communications. The FSM simply returns 1 when communications are finished, allowing the main program to continue. All four of the parameters are constant throughout the entire program. The FSM as is would not work if the program needed to use i^{2}c communications to any other device. To accommodate multiple devices, an idle state and more complex handshaking would be added so that the program could only send a new data request (to a different device or otherwise) when the FSM is in the idle (ready) state.

Control Flow Diagram

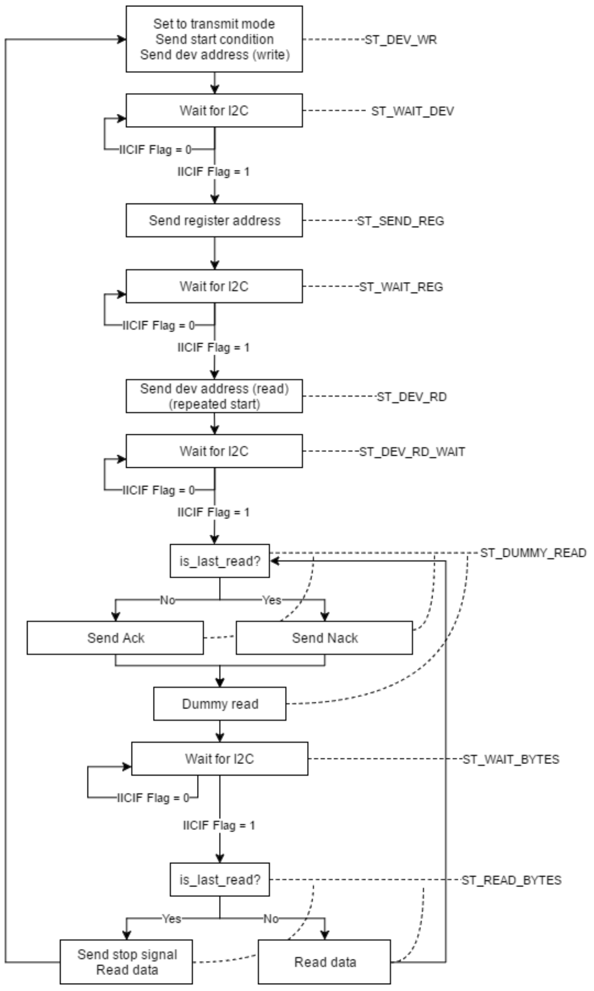

Figure 2: FSM I^{2}C Control Flow diagram

State Machine Diagram

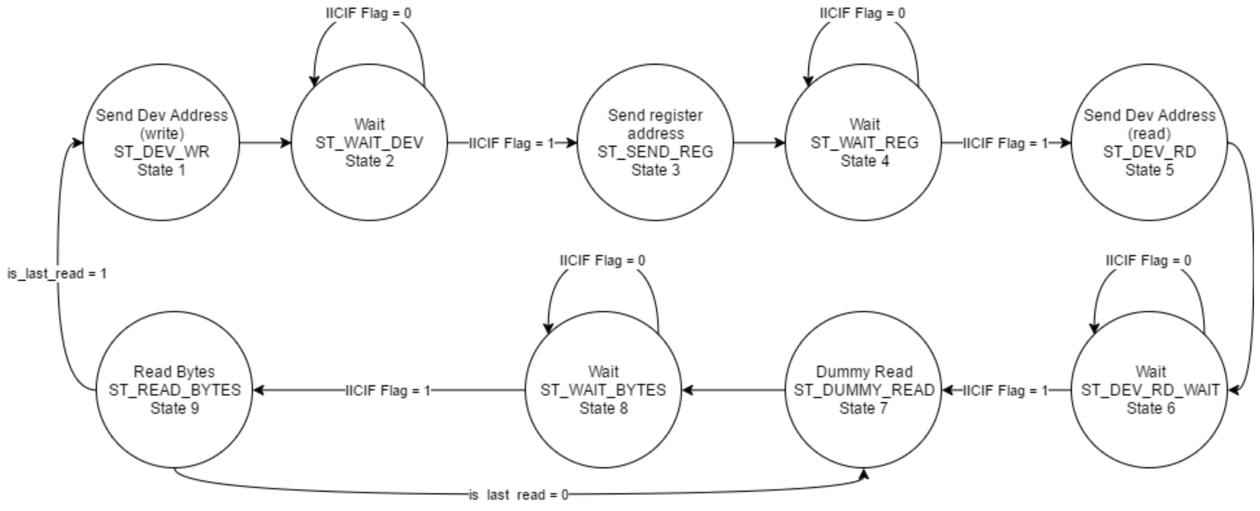
Send Dev Address (write) ST_DEV_WR State 1
Wait ST_WAIT_DEV State 2
IICIF Flag = 0
IICIF Flag = 1
Send register address ST_SEND_REG State 3
Wait ST_WAIT_REG State 4
IICIF Flag = 0
IICIF Flag = 1
Send Dev Address (read) ST_DEV_RD State 5
Wait ST_DEV_RD_WAIT State 6
IICIF Flag = 0
IICIF Flag = 1
Dummy Read ST_DUMMY_READ State 7
Wait ST_WAIT_BYTES State 8
IICIF Flag = 0
IICIF Flag = 1
Read Bytes ST_READ_BYTES State 9
is_last_read = 1
is_last_read = 0

Figure 3: FSM I^{2}C State Diagram

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Logic Analyzer

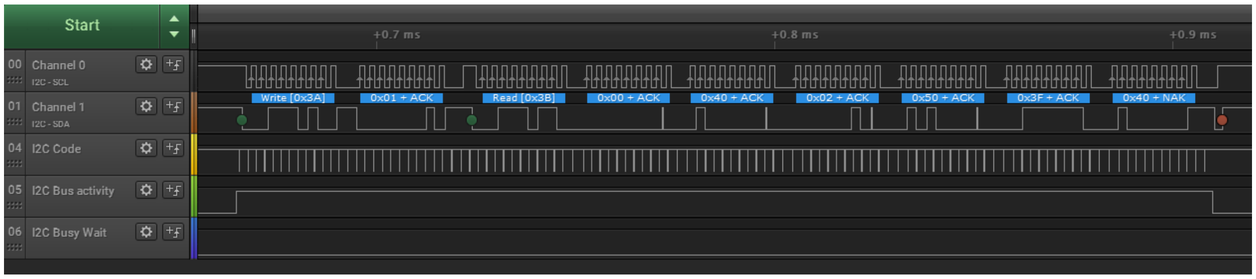

Figure 4: FSM I²C message with no delay

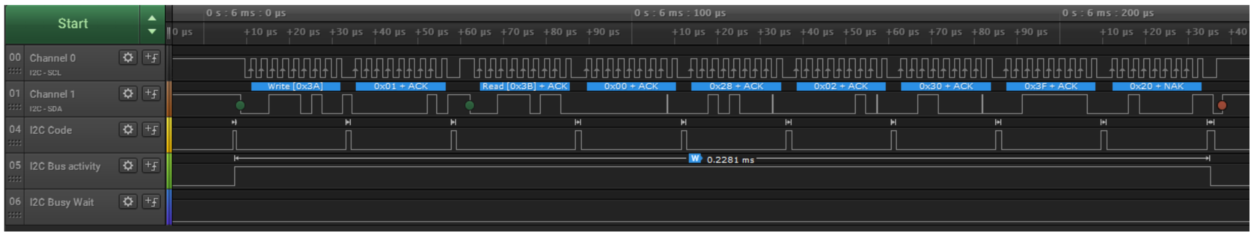

Figure 5: FSM I²C message with ~17 µs delay

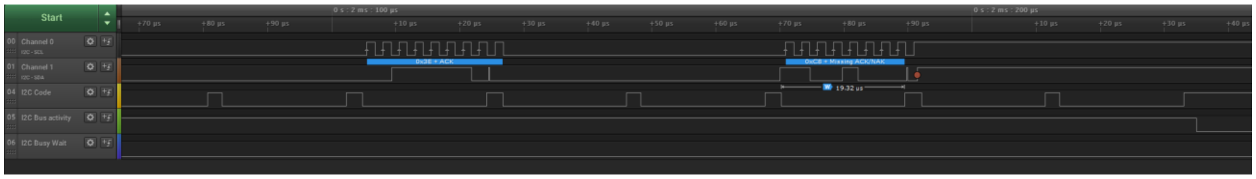

Figure 6: FSM I²C message with ~19 µs delay - Missing NAK

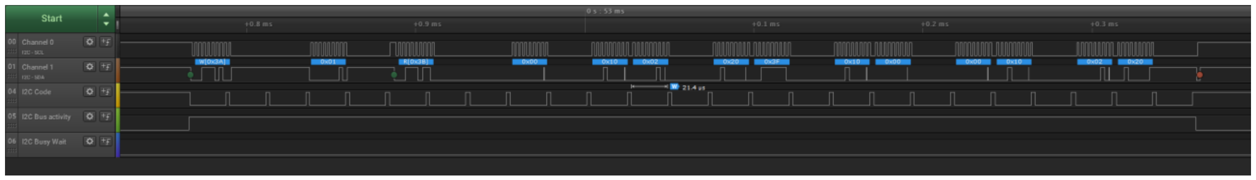

Figure 7: FSM I²C message with ~21 µs delay - Reading 11 bytes (broken)

As seen in Figures 5-7, polling the I²C State Machine at ~17 µs intervals is the largest delay without causing issues with the communications. This corresponds to `ShortDelay(8)` in code. Increasing the delay to `ShortDelay(9)` or ~19 µs, The I²C message is missing the NAK at the end of reading all 6 bytes. Interestingly though, the program still seems to function correctly. At `ShortDelay(9)` or ~21 µs, the program completely breaks, reading out 11 bytes of data. The `data[]` array changes on every iteration, causing the LED to flash white constantly.

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Interrupt Service Routine Implementation

Input and Output Data

Initially, the I^{2}C ISR is triggered by a start function called by the main program that sends the device address. After that, the remainder of I^{2}C communications is handled within the ISR. Like the FSM, since the device address and register address do not change for this project, they have been hard coded into the ISR, using MMA_ADDR and REG_XHI. To make the ISR more adaptable, global variables for the device address and register address could be used, along with global ready bits that the main program and the ISR could set. One for the main program to send a new request, and another for the ISR to set when it is ready to accept new requests. In this project, a global variable is set by the ISR (`i2c_is_complete`) when all 6 bytes have been read and the stop signal is sent. Additionally, a global data array (`isr_data[]`) is used to read the bytes. After the ISR completes, the main program copies the data to the local `data[]` array.

2.2.2016
Zach Hannum
zphannum@ncsu.edu

Logic Analyzer

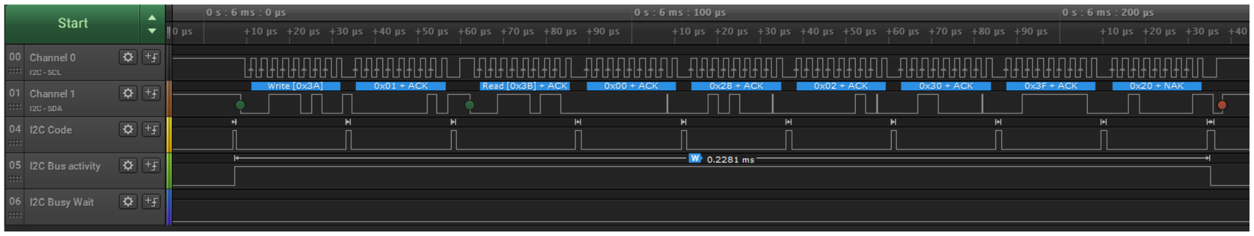

Figure 8: ISR I²C Full Message

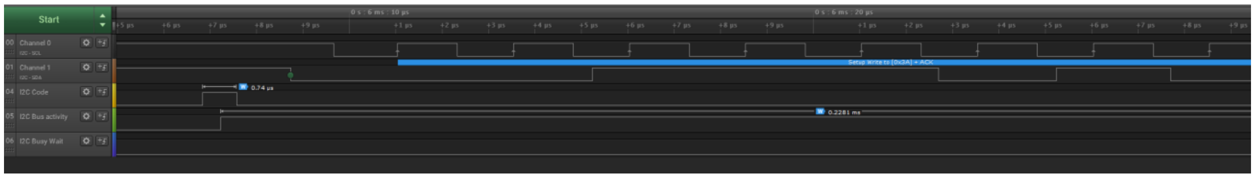

Figure 9: ISR I²C Message detail 1

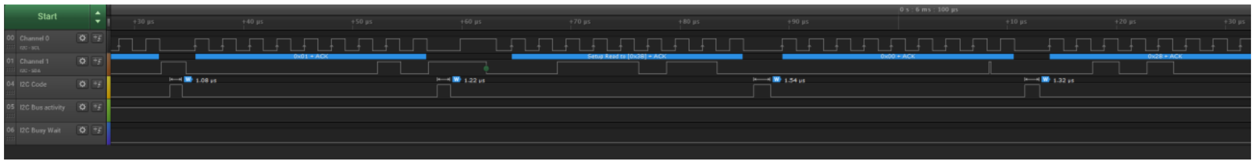

Figure 10: ISR I²C Message detail 2

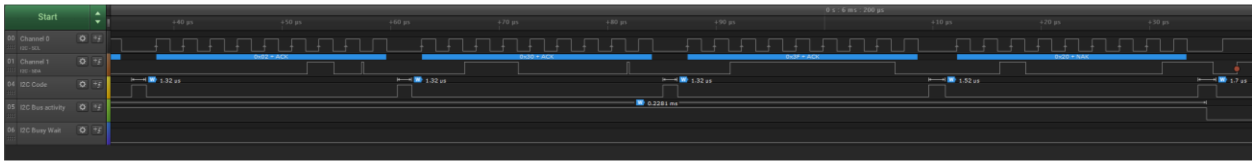

Figure 11: ISR I²C Message detail 3

As seen the detailed I²C message figures (9-11), the total processor time executing the I²C code is

0.74 µs + 1.08 µs + 1.22 µs + 1.54 µs + 1.32 µs + 1.32 µs + 1.32 µs + 1.32 µs + 1.52 µs + 1.7 µs = **13.08 µs**

Figure 8 shows that the total duration of the message on the bus is **228.1 µs**. So the processor only needed to run ~6% of the time that the message was on the bus, a significant decrease from the 100% of the time the original blocking code would run.