ECE 372 Project 2
(Project Report)

By Abdullah Almarzouq

March 12,2020

Table of contents

Introduction:

The project report includes a detailed analysis on how to initialize I2C driver on BeagleBone Black to control a DC motor controller board (Adafruit Feather Motor Controller). However, instead of controlling a DC motor, we will control a stepper motor. The whole process of controlling the stepper motor through I2C can be generalized as the following: The created program will use I2C (as master) to a send commands (bytes) to the PCA9685 chip on the motor controller which will then generate the PWMs signals, as inputs to the two H-Bridges, on the TB6612FNG chip on the motor controller, after that the stepper motor should move to step through the desired steps.

Moreover, the project has two parts. The first part goes over the initialization process of the I2C driver on the BeagleBone Black, how to control the I2C to generate and send the desired signals/bytes and checking the SCL and SDA signals to confirm that the desired bytes have been sent. Whereas, the second part includes the detailed process on how to initialize the PCA9685 chip and how to send the desired commands/bytes to control the stepper motor.

PART I:

The first, as mentioned in the introduction, goes over the initialization of the I2C driver, generating the desired signals and checking the SCL and SDA signals to confirm that the desired signals are being sent correctly.

Main Tasks:

- Initialize I2C on BeagleBone Black board to correctly generate the desired clock frequency signal and the required data signals on the I2C bus.

High Level Algorithm:

- Change the mode on the conf_spi0_cs0 register (Pin 17) to connect I2C_SCL signal to pin17
- Change the mode on the conf_spi0_d1 register (Pin 18) to connect I2C_SDA signal to pin18
- Turn on clock of I2C1
- Disable the I2C1 signals
- Write to I2C1_PSC register to scale the frequency from 48MHz to 12MHz
- Change to 400Kbps SCL for Fast/Standard mode
- Configure Slave address
- Enable I2C as master transmitter
- Poll to check if bus is free:
    - If bus is not free, then keep Polling
- Else if bus is free, then:
    - Initialize byte count
    - Set start and stop conditions
    - Send the desired byte/s

- Jump back to Polling loop

Low Level Algorithm:

MAINLINE:

- Write 0x3A to 0x44E10958 conf_spi0_d1 register to connect I2C_SDA to pin 18
- Write 0x3A to 0x44E1095C conf_spi0_cs0 register to connect I2C_SCL to pin 17
- Write 0x2 to 0x44E00048 CM_PER_I2C1_CLKCTRL register to turn on I2C1 clock
- Write 0x0000 to 0x4802A0A4 I2C1_CON to disable I2C1
- Write 0x3 to 0x4802A0B0 I2C1_PSC to scale the frequency from 48MHz to 12MHz
- Write 0x8 to 0x4802A0B4 I2C1_SCLL to change SCLL to 400 Kbps
- Write 0xA to 0x4802A0B8 I2C1_SCLH to change SCLH to 400 Kbps
- Write 0x60 to 0x4802A0AC I2C1_SA to configure slave address
- Write 0x8600 to 4802A0A4 I2C1_CON to enable I2C1 as master transmitter
- Jump to Polling loop

POLL:

- Read value in I2C1_IRQSTATUS_RAW register at 0x4802A024
- Test with 0x1000 to check if Bus Busy bit is set to 1 (busy)
- If BB bit is 1 then keep polling. Else continue to SEND branch

SEND:

- Write 0x1 to 0x4802A098 I2C1_CNT to initialize the number of bytes to be sent (1 byte)
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0xFF to 0x4802A09C I2C1_DATA to send the byte through I2C bus
- Jump back to Polling loop

Experimental Results:

After initialization the I2C driver to generate the correct SCL and SDA signals, both signals have been tested using a scope and the results are as shown below.

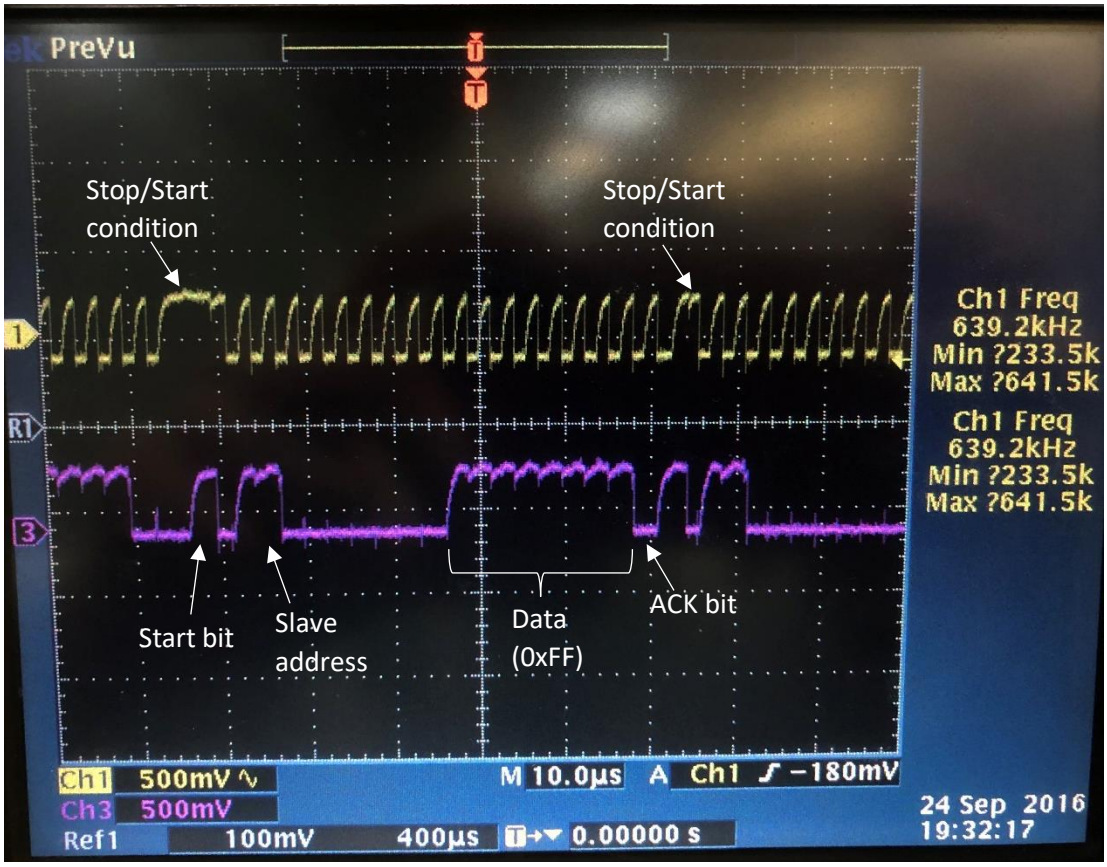

Figure 1: Testing the SCL and SDA signals

As it can be seen from the figure above, the yellow wave represents the SCL signal, while the purple wave represents the SDA signal. We can also see that once start bit goes low on the SDA signal, the start condition on the SCL goes high to initiate byte transfer. After the start bit, we can see the slave address (0x60) on the SDA signal, then after that we can see the byte that has been sent (oxFF) and finally the ACK bit (low) and the stop condition. Last but not least, the frequency of the signal, can be seen on the right, is 639.2kHz which is fairly close the 400KHz we were supposed to see.

LOG:

2/28/20:

- The values needed to change the modes on the pin Mux have been found.
- All the needed addresses have been. Addresses such as the I2C1 base address, the I2C registers offsets and the control model to turn on the I2C1 clock and such.
- The value for the frequency scaling has been calculated (PSC=3) and the values for SCLL = 8 & SCLH = 10 have been found by calculations.
- The high level algorithm has been written.

3/2/20:

- The low level algorithm has been written.
- The assembly code has also been written.
- The program has been built and the desired values are being stored to the desired address and registers.
- However, when the program is running, the oscilloscope reading that is being shown on the scope is not the correct/desired reading. Which means that the program is not running correctly. Or that the algorithm itself is not written as it should.

3/3/20:

- Today I managed to get the I2C to work and the right/desired reading on the scope for both SCL and SDA signals.
- The first thing I did was changing the value stored in conf_spi0_cs0 and conf_spi0_d1 from 0xA to 0x3A to change to mode 2 and select pullup and enable receiver.
- After that, I disabled I2C1 by writing 0x0000 to I2C_CON register right before the initialization of the I2C driver.
- Then I enabled I2C1 by writing 0x8600 to I2C_CON right after the initialization but before the polling loop.
- Then when the program jumps to the polling loop, it checks if the Bus Busy bit in I2C_IRQSTATUS_RAW is asserted or not instead of checking XRDY bit.
- Finally, the program jumps to SEND branch if the but is free to send the desired byte. The send branch initializes DCOUNT in I2C_CNT register, writes 0x8603 to I2C_CON to start sending and asserting the stop condition after it has done sending. Then, the branch writes a byte (0xFF) to I2C_DATA to start the transmission and finally the program jumps back to the polling loop.

PART II:

The second part, as mentioned in the introduction, includes a detailed process on how to initialize the PCA9685 chip on the Adafruit Feather board to control the stepper motor. The program in the first will be used to control the PCA9685 chip by sending the desired signals through the I2C1 bus.

Main Tasks:

- Initialize the PCA9685 chip to send the correct PWM signals to the control the stepper motor. The registers on the PCA9685 chip are going to be controlled and modified by sending the correct signals through the I2C1 bus which was initialized and tested in the first part.

High Level Algorithm:

- Initialize PCA9685:
    - Perform software reset
    - Put PCA9685 in sleep mode and change the Pre_Scale
    - Restart PCA9685
    - Set Totem Pole structure & non-inverted bits in mode2 register
    - Set PWM2 and PWM7 high
    - Set ALL_LED_OFF_H to zero
- Set counter for 200 steps

Repeat:

- Set PWM3-6 in the following sequence to make the stepper motor step in clockwise rotation:

PWM4	PWM3	PWM5	PWM6
H	L	H	L
H	L	L	H
L	H	L	H
L	H	H	L

- Decrement counter after each step

Repeat until counter is zero

*Note: Set a delay loop for 5000 between each command. Also, set a delay loop of 400,000 between each step.

Low Level Algorithm:

MAINLINE:

- Write 0x00000000 to 0x4802A0AC I2C_SA to configure slave address to perform software reset
- Write 0x8600 to 0x4802A0A4 I2C_CON to enable I2C1
- Poll BB bit as before, write 0x1 to 0x4802A098 I2C_CNT to send one byte
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x06 to 0x4802A09C to I2C_DATA to perform a software reset
- Delay for 5000 then poll BB bit
- Write 0x0000 to 0x4802A0A4 I2C_CON to disable I2C
- Write 0xE0 to 0x4802A0AC to configure ALLCALL slave address
- Write 0x8600 to 0x4802A0A4 to enable I2C
- Poll BB bit, write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x00 to 0x4802A09C to I2C_DATA to access Mode1 register
- Write 0x11 to 0x4802A09C to I2C_DATA to put PCA9685 in sleep mode
- Delay for 5000, poll BB bit.
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0xFE to 0x4802A09C to I2C_DATA to access Pre_Scale register
- Write 0x05 to 0x4802A09C to I2C_DATA to change to 1KHz
- Delay for 5000, poll BB bit.
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x00 to 0x4802A09C to I2C_DATA to access Mode1 register
- Write 0x01 to 0x4802A09C to I2C_DATA to assert sleep bit to zero
- Delay for 5000, poll BB bit.
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x00 to 0x4802A09C to I2C_DATA to access Mode1 register
- Write 0x81 to 0x4802A09C to I2C_DATA to restart PCA9685
- Delay for 5000, poll BB bit.
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x01 to 0x4802A09C to I2C_DATA to access Mode2 register
- Write 0x04 to 0x4802A09C to I2C_DATA to set totem pole structure and non-inverted
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x0F to 0x4802A09C to I2C_DATA to access LED2_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x23 to 0x4802A09C to I2C_DATA to access LED7_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH

- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0xFD to 0x4802A09C to I2C_DATA to access ALL_LED_OFF_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000
- Initialize counter for 200 steps

STEP1:

- Poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x17 to 0x4802A09C to I2C_DATA to access LED4_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x13 to 0x4802A09C to I2C_DATA to access LED3_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1B to 0x4802A09C to I2C_DATA to access LED5_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1F to 0x4802A09C to I2C_DATA to access LED6_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Decrement Steps counter (if zero then stop)
- Delay for 400,000

STEP2:

- Poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x17 to 0x4802A09C to I2C_DATA to access LED4_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x13 to 0x4802A09C to I2C_DATA to access LED3_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions

- Write 0x1B to 0x4802A09C to I2C_DATA to access LED5_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1F to 0x4802A09C to I2C_DATA to access LED6_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Decrement Steps counter (if zero then stop)
- Delay for 400,000

STEP3:

- Poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x17 to 0x4802A09C to I2C_DATA to access LED4_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x13 to 0x4802A09C to I2C_DATA to access LED3_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1B to 0x4802A09C to I2C_DATA to access LED5_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1F to 0x4802A09C to I2C_DATA to access LED6_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Decrement Steps counter (if zero then stop)
- Delay for 400,000

STEP4:

- Poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x17 to 0x4802A09C to I2C_DATA to access LED4_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x13 to 0x4802A09C to I2C_DATA to access LED3_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit

- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1B to 0x4802A09C to I2C_DATA to access LED5_ON_H
- Write 0x10 to 0x4802A09C to I2C_DATA to set it to HIGH
- Delay for 5000, poll BB bit
- Write 0x2 to 0x4802A098 I2C_CNT to send two bytes
- Write 0x8603 to 0x4802A0A4 I2C1_CON to set start and stop conditions
- Write 0x1F to 0x4802A09C to I2C_DATA to access LED6_ON_H
- Write 0x00 to 0x4802A09C to I2C_DATA to set it to LOW
- Decrement Steps counter (if zero then stop)
- Delay for 400,000
- Jump back to STEP1

STOP: NOP
NOP

LOG:

3/5/20:

- I went over the schematics of the PCA9685 and the Toshiba chi which are on the Adafruit board.
- From the schematics and the datasheet of the PCA9685 I figured out the needed addresses and registers for both the initialization of the PCA9685 and PWMs outputs which are inputs on the Toshiba chip.
- I wrote the high level algorithm.
- After obtaining all the values that are going to be saved on each register, I started writing the low level algorithm.

3/6/20:

- I developed the assembly code for the stepper motor program and I also managed to build it and fix all the typos and such.
- After running and testing the program, it turned out that it doesn't work for some reason.
- Even when I try to test the SDA and/or the SCL signals I'm not getting any signals on the scope. Even though the program is running without any errors and it doesn't get stuck in any loop.
- The issue could be due to a fault in the initialization process.

3/7/20:

- Today the stepper motor program worked! The issue was in using the LEDn_OFF_H signals. Instead to fix the issue, I started using the LEDn_ON_H signals only to either turn it fully on or fully off. I also zeroed ALL_LED_OFF_H during the initialization process.
- The only issue after that was that the motor does a full rotation with a counter of 52 instead of 200.
- It turned out that the program was written in a way so that the counter gets decremented after four steps instead of after each step. Therefore, I added branches for each step and after each step the counter gets decremented. Now a full rotation executes in 200 steps.
- I also re-calculated the PreScale value using 25MHz instead of 12MHz in the following equation:

$$Pre\ Scale = \left(\frac{25MHz}{4096\ \times 1kHz}\right) - 1 = 5.1\ \cong 5$$

- I also added a button feature. When the button is pushed, an IRQ interrupt happens which lets the motor step 200 steps to perform a full rotation.

I development and wrote both programs (in PART I & PART II) by myself with no help from anyone except the instructor and/or the TA. I did not give any help to anyone else and I did not copy anything from online sources.

Signature: Abdullah Almarzouq