ME461 Semester Homework #3
I2C Serial Port, Interfacing the DAN777 and the BQ32000

Check off #1: Demonstrate DAN777 Communication Working
Due Wednesday October 25th, 1:00 PM

Check off #2: Demonstrate DAN777 And BQ32000 Both Communicating
Due Wednesday November 1st, 1:00 PM

Submission at Box Due November 1st, 12:00 Midnight

Make sure to read this entire document before getting started so you know all the parts of the project and can plan accordingly. Also if you would like, I have copied to the class Box folder four recorded lectures that I made Spring 2021 for my SE423 class. You can find them in the "I2CLectures_SE423_Spring21" folder. I will be lecturing on the I2C serial port this semester but you may find these additional lectures helpful.

The goal of this project is to teach you how to use the I2C serial port to communicate with external sensors and other peripherals. You will be programming the F28379D to use its I2C serial port to communicate with the "DAN777" chip and the BQ32000 Real-Time Clock chip. In this project, you will learn more how to use the I2C serial port compared to learning how to setup the registers of the F28379D's I2C peripheral like many of the labs have done for other peripherals. Below you are given the initialization code for the F28379D's I2C peripheral and some example functions using the I2C serial port. Following these examples, you will create "write" and "read" functions for both the DAN777 and BQ32000.

Comparing SPI to I2C there are a number of differences you need to keep in mind when programming I2C. The biggest difference is that I2C has only one data line so it can be set to read data for a certain number of bytes or set to write a certain number of bytes. This makes I2C slower along with its maximum recommended bit rate of 400000 bits/second where SPI can go 20Mbits/second or higher. Additionally I2C does not use a chip select (or slave select) pin, but instead transmits a slave address to select the slave device. All this is done with the three wires, SCL, SDA and GND connecting the I2C master (F28379D) with many possible I2C slave devices.

Because the I2C serial port is usually used for slower devices, we will not be using the I2C interrupts. Instead, you will be "polling" the status of the I2C serial port to know when it is ready to send more data or has received new data. If you would like to learn more about the I2C serial port and use its interrupt capabilities, talk to me about using I2C in your class final project.

After studying the given examples and the datasheet of the DAN777 and BQ32000, create separate functions for reading and writing the DAN777 and BQ32000. Because these are "polling" functions they will need to be called inside main()'s while(1) loop.

The TMS320F28379D I2C users guide can be found at:
http://coecsl.ece.illinois.edu/ME461/labs/I2CCondensed_TechRef.pdf
The DAN777 Datasheet can be found at:
http://coecsl.ece.illinois.edu/me461/Labs/DAN777_Datasheet.pdf
The BQ32000 Datasheet with some ME461 additions can be found at:
http://coecsl.ece.illinois.edu/me461/Labs/BQ32000Condensed.pdf

Four wires you need to solder.

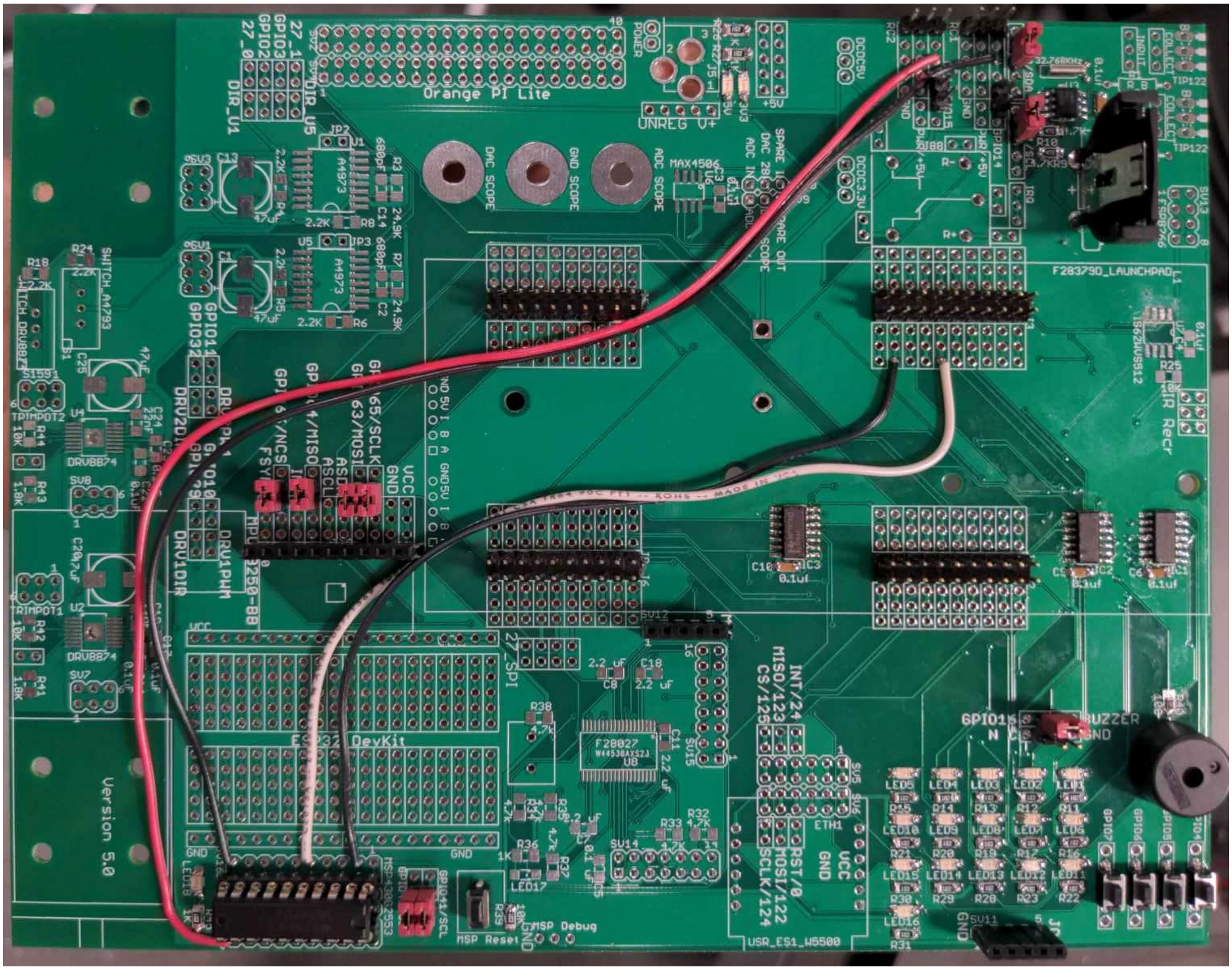
You will need to solder four wires to your green board connecting the Joystick voltages to the DAN777 ADCIN1 and ADCIN2 pins and connecting the DAN777 RCServo output pins to the 3 pin RCServo headers of your green board.

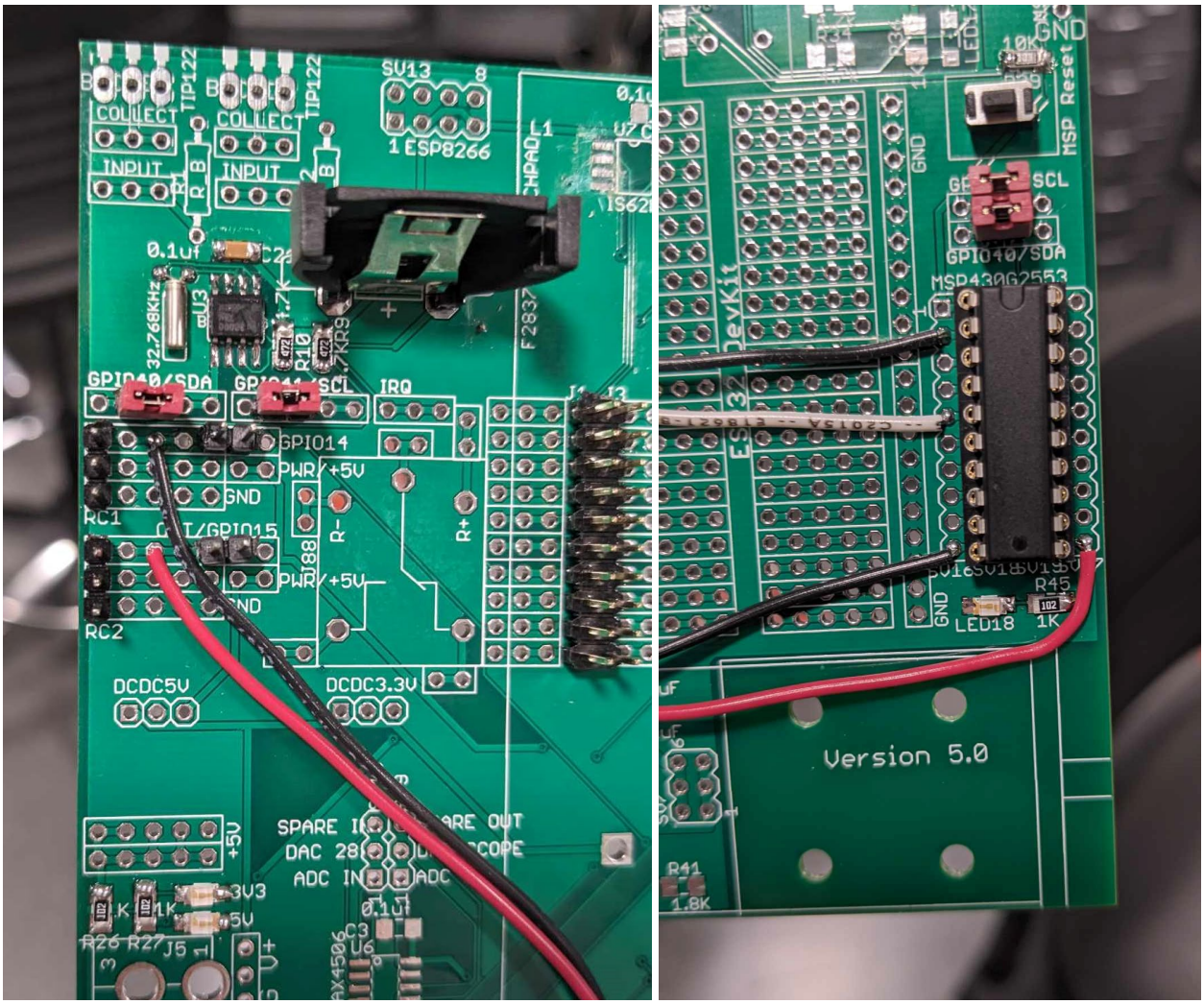
The DAN777 ADCIN1 pin is brought out at pin 2. Connect it to F28379D Launchpad pin 29 which is already wired to the joystick's X axis. Shorter black wire in below picture.

The DAN777 ADCIN2 pin is brought out at pin 5. Connect it to F28379D Launchpad pin 26 which is already wired to the joystick's Y axis. White wire in below picture.

The DAN777 RC1 pin is brought out at pin 10. Connect it to the higher of the three pin headers and the pin that is connected through a jumper to GPIO14. Make sure to remove this jumper so only RC1 is connected to the 3 pin header. This is the longer black wire in the below picture.

The DAN777 RC2 pin is brought out at pin 12. Connect it to the lower of the three pin headers and the pin that is connected through a jumper to GPIO15. Make sure to remove this jumper so only RC2 is connected to the 3 pin header. This is the red wire in the below picture.





Function Declarations

Copy and paste the following function and global variable declarations towards the top of your C file. The following sections have some code you will need to copy and paste directly and some example code you will have to adjust for the specific chips we are using in this homework.

```
void I2CB_Init(void);
int16_t WriteDAN777RCServo(uint16_t RC1, uint16_t RC2);
int16_t ReadDAN777ADC(uint16_t *ADC1, uint16_t *ADC2);
int16_t WriteBQ32000(uint16_t second,uint16_t minute,uint16_t hour,uint16_t day,uint16_t date,uint16_t
month,uint16_t year);
int16_t ReadBQ32000(uint16_t *second,uint16_t *minute,uint16_t *hour,uint16_t *day,uint16_t *date,uint16_t
*month,uint16_t *year);
int16_t I2C_CheckIfTX(uint16_t timeout);
int16_t I2C_CheckIfRX(uint16_t timeout);

uint16_t RunI2C = 0; // Flag variable to indicate when to run I2C commands
int16_t I2C_OK = 0;
int32_t num_WriteCHIPXYZ_Errors = 0;
int32_t num_ReadCHIPXYZ_Errors = 0;
```

I2C initialization function

Copy and paste the `I2C_CheckIfTX, I2C_CheckIfRX, and I2CB_Init()` functions into your C file. Call `I2CB_Init()` after the `init_serial` functions in `main()` and before the while(1) continuous loop (not inside the loop). `I2C_CheckIfTX and I2C_CheckIfRX` are helper functions used for error handling. Read the comments for an idea of what is happening in these functions. Note that this code uses EPWM7 as a timer that will timeout in the event of an I2C error. The I2C specification referenced in the comments can be found here: https://espace.cern.ch/CMS-MPA/SiteAssets/SitePages/Documents/I2C_bus_specifications_V2_0.pdf

```
/* Functions to check if I2C is ready to transfer or receive.
 * Here we utilize EPWM7 to keep track of time. If I2C is
 * not ready, wait 100ms then try again. */
int16_t I2C_CheckIfTX(uint16_t timeout) {
    int16_t Xrdy = 0;
    EPwm7Regs.TBCTR = 0x0; // Clear counter
    EPwm7Regs.TBCTL.bit.CTRMODE = 0; // unfreeze, and enter up count mode
    while(!Xrdy) {
        if (EPwm7Regs.TBCTR > timeout) {  // if it has been 100ms
            EPwm7Regs.TBCTL.bit.CTRMODE = 3; // freeze counter
            return -1;
        }
        Xrdy = I2cbRegs.I2CSTR.bit.XRDY;
    }
    return 0;
}


int16_t I2C_CheckIfRX(uint16_t timeout) {
    int16_t Rrdy = 0;
    EPwm7Regs.TBCTR = 0x0; // Clear counter
    EPwm7Regs.TBCTL.bit.CTRMODE = 0; //unfreeze, and enter up count mode
    while(!Rrdy) {
        if (EPwm7Regs.TBCTR > timeout) {  // if we have been in this function for 100ms
            EPwm7Regs.TBCTL.bit.CTRMODE = 3; // freeze counter
            return -1;
        }
        Rrdy = I2cbRegs.I2CSTR.bit.RRDY;
    }
    return 0;
}

void I2CB_Init(void) {
    // Setting up EPWM 7 to use as a timer for error handling
    EALLOW;
    EPwm7Regs.ETSEL.bit.SOCAEN = 0;     // Disable SOC on A group
    EPwm7Regs.TBCTL.bit.CTRMODE = 3;    // Freeze counter
    EPwm7Regs.TBCTR = 0x0;              // Clear counter
    EPwm7Regs.TBPHS.bit.TBPHS = 0x0000; // Phase is 0
    EPwm7Regs.TBCTL.bit.PHSEN = 0;      // Disable phase loading
    EPwm7Regs.TBCTL.bit.CLKDIV = 7;     // divide by 1  50Mhz Clock
    EPwm7Regs.TBPRD = 0xFFFF;           // PRD not used for timer
    // Notice here we are not using the PWM signal, so CMPA/CMPB are not set
    EDIS;

    /* If an I2C device is holding the SDA Line Low you need to tell the device
     * to reinitialize by clocking 9 clocks to the device to reset it. */
    GpioDataRegs.GPBSET.bit.GPIO41 = 1; // Here make sure SCL clk is high
    GPIO_SetupPinMux(41, GPIO_MUX_CPU1, 0);
    GPIO_SetupPinOptions(41, GPIO_OUTPUT, GPIO_PUSHPULL);

    // Set SDA as GPIO input pin for now to check if SDA is being held low.
```

```
    GPIO_SetupPinMux(40, GPIO_MUX_CPU1, 0);
    GPIO_SetupPinOptions(40, GPIO_INPUT, GPIO_PULLUP);

    /* Check if SDA is low.  If it is manually set the SCL pin high and low to
     * create nine clock periods. For more reading see the I2C specification linked
     * in this homework document and search for "nine clock pulses" */
    if (GpioDataRegs.GPBDAT.bit.GPIO40 == 0) {  // If SDA low
        // Pulse CLK 9 Times if SDA pin Low
        for (int i = 0; i<9; i++) {
            GpioDataRegs.GPBSET.bit.GPIO41 = 1;
            DELAY_US(30);
            GpioDataRegs.GPBCLEAR.bit.GPIO41 = 1;
            DELAY_US(30);
        }
    }

    // Now setup GPIO40 as SDAB and GPIO41 and SCLB
    EALLOW;
    /* Enable internal pull-up for the selected I2C pins */
    GpioCtrlRegs.GPBPUD.bit.GPIO40 = 1;
    GpioCtrlRegs.GPBPUD.bit.GPIO41 = 1;

    /* Set qualification for the selected I2C pins */
    GpioCtrlRegs.GPBQSEL1.bit.GPIO40 = 3;
    GpioCtrlRegs.GPBQSEL1.bit.GPIO41 = 3;

    /* Configure which of the possible GPIO pins will be I2C_B pins using GPIO regs*/
    GpioCtrlRegs.GPBGMUX1.bit.GPIO40 = 1;
    GpioCtrlRegs.GPBMUX1.bit.GPIO40 = 2;

    GpioCtrlRegs.GPBGMUX1.bit.GPIO41 = 1;
    GpioCtrlRegs.GPBMUX1.bit.GPIO41 = 2;
    EDIS;

    // At breakpoint, allow I2C to continue operating
    I2cbRegs.I2CMDR.bit.FREE = 1;

    // Initialize I2C
    I2cbRegs.I2CMDR.bit.IRS = 0;

    // 200MHz / 20 = 10MHz
    I2cbRegs.I2CPSC.all = 19;

    // 10MHz/40 = 250KHz
    I2cbRegs.I2CCLKL = 15*3;  //psc > 2 so d = 5  See Usersguide
    I2cbRegs.I2CCLKH = 15*3;  //psc > 2 so d = 5  See Usersguide

    I2cbRegs.I2CIER.all = 0x00;

    I2cbRegs.I2CMDR.bit.IRS = 1;
    DELAY_US(2000);
}
```

Example of an I2C write function

This is for a fictitious chip named CHIPXYZ. It has 16 registers inside the chip. Here the function is writing to registers 4, 5, 6, and 7. Registers 4 and 5 are each 8bits and make up a 16bit command to CHIPXYZ. Register 4 is the LSB (least significant byte) and Register 5 is the MSB (most significant byte). In the same way registers 6 and 7 make up another command to CHIPXYZ and register 6 is LSB and register 7 is MSB. CHIPXYZ's slave address is **0x3A**.

```
// Write 2 16-bit commands (LSB then MSB) to I2C Slave CHIPXYZ starting at CHIPXYZ's register 4
uint16_t WriteTwo16BitValuesToCHIPXYZ(uint16_t Cmd16bit_1, uint16_t Cmd16bit_2) {
    uint16_t Cmd1LSB = 0;
```

```
    uint16_t Cmd1MSB = 0;
    uint16_t Cmd2LSB = 0;
    uint16_t Cmd2MSB = 0;
    int16_t I2C_Xready = 0;
    Cmd1LSB = Cmd16bit_1 & 0xFF;        //Bottom 8 bits of command
    Cmd1MSB = (Cmd16bit_1 >> 8) & 0xFF; //Top 8 bits of command
    Cmd2LSB = Cmd16bit_2 & 0xFF;        //Bottom 8 bits of command
    Cmd2MSB = (Cmd16bit_2 >> 8) & 0xFF; //Top 8 bits of command

    // Allow time for I2C to finish up previous commands.
    DELAY_US(200);

    if (I2cbRegs.I2CSTR.bit.BB == 1) { // Check if I2C busy. If it is, it's better
        return 2;                      // to exit and try again next sample.
    }                                  // This should not happen too often.

    I2C_Xready = I2C_CheckIfTX(39062); // Poll until I2C is ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }

    I2cbRegs.I2CSAR.all = 0x3A;   // Set I2C address to that of ChipXYZ's
    I2cbRegs.I2CCNT     = 5;      // Number of values to send plus start register: 4 + 1
    I2cbRegs.I2CDXR.all = 4;      // First need to transfer the register value to start writing data
    I2cbRegs.I2CMDR.all = 0x6E20; // I2C in master mode (MST), I2C is in transmit mode (TRX) with start
and stop

    I2C_Xready = I2C_CheckIfTX(39062);   // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }
    I2cbRegs.I2CDXR.all = Cmd1LSB;       // Write Command 1 LSB
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Xready = I2C_CheckIfTX(39062);   // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }
    I2cbRegs.I2CDXR.all = Cmd1MSB;       // Write Command 1 MSB
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Xready = I2C_CheckIfTX(39062);   // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }
    I2cbRegs.I2CDXR.all = Cmd2LSB;       // Write Command 2 LSB
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Xready = I2C_CheckIfTX(39062);   // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }
    I2cbRegs.I2CDXR.all = Cmd2MSB;       // Write Command 2 MSB

    // Since I2CCNT = 0 at this point, a stop condition will be issued

    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }
    return 0;
}
```

Example of an I2C read function. This is for a fictitious chip named CHIPXYZ. It has 16 registers inside the chip. Here the function is reading registers 10, 11, 12 and 13. Registers 10 and 11 are each 8bits and make up a 16bit sensor reading from CHIPXYZ. Register 10 is the LSB (least significant byte) and Register 11 is the MSB (most significant byte). In the same way registers 12 and 13 make up another 16bit sensor reading from CHIPXYZ and register 12 is LSB and register 13 is MSB. CHIPXYZ's slave address is **0x3A**. Notice how the function first writes the start register with one I2C start condition and then re-issues a second start condition to switch the I2C communication to a read command.

```
/* Read Two 16 Bit values from I2C Slave CHIPXYZ starting at CHIPXYZ's register 10.
 * Notice the Rvalue1 and Rvalue2 passed as pointers (passed by reference). So pass
 * address of the uint16_t variable when using this function. For example:
 *     uint16_t Rval1 = 0;
 *     uint16_t Rval2 = 0;
 *     err = ReadTwo16BitValuesFromCHIPXYZ(&Rval1,&Rval2);
 * This allows Rval1 and Rval2 to be changed inside the function and return the
 * values read inside the function. */
uint16_t ReadTwo16BitValuesFromCHIPXYZ(uint16_t *Rvalue1,uint16_t *Rvalue2) {
    uint16_t Val1LSB = 0;
    uint16_t Val1MSB = 0;
    uint16_t Val2LSB = 0;
    uint16_t Val2MSB = 0;
    int16_t I2C_Xready = 0;
    int16_t I2C_Rready = 0;

    // Allow time for I2C to finish up previous commands.
    DELAY_US(200);

    if (I2cbRegs.I2CSTR.bit.BB == 1) { // Check if I2C busy. If it is, it's better
        return 2;                      // to exit and try again next sample.
    }                                  // This should not happen too often.

    I2C_Xready = I2C_CheckIfTX(39062); // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }

    I2cbRegs.I2CSAR.all = 0x3A;   // I2C address of ChipXYZ
    I2cbRegs.I2CCNT     = 1;      // Just sending address to start reading from
    I2cbRegs.I2CDXR.all = 10;     // Start reading at this register location
    I2cbRegs.I2CMDR.all = 0x6620; // I2C in master mode (MST), I2C is in transmit mode (TRX) with start

    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Xready = I2C_CheckIfTX(39062); // Poll until I2C ready to transmit
    if (I2C_Xready == -1) {
        return 4;
    }

    // Reissuing another start command to begin reading the values we want
    I2cbRegs.I2CSAR.all = 0x3A;   // I2C address of ChipXYZ
    I2cbRegs.I2CCNT     = 4;      // Receive count
    I2cbRegs.I2CMDR.all = 0x6C20; // I2C in master mode (MST), TRX=0 (receive mode) with start & stop

    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Rready = I2C_CheckIfRX(39062);   //Poll until I2C has received 8-bit value
    if (I2C_Rready == -1) {
```

```
        return -1;
    }
    Val1LSB = I2cbRegs.I2CDRR.all;       // Read CHIPXYZ
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Rready = I2C_CheckIfRX(39062);   //Poll until I2C has received 8-bit value
    if (I2C_Rready == -1) {
        return -1;
    }
    Val1MSB = I2cbRegs.I2CDRR.all;       // Read CHIPXYZ
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Rready = I2C_CheckIfRX(39062);   //Poll until I2C has received 8-bit value
    if (I2C_Rready == -1) {
        return -1;
    }
    Val2LSB = I2cbRegs.I2CDRR.all;       // Read CHIPXYZ
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    I2C_Rready = I2C_CheckIfRX(39062);   //Poll until I2C has received 8-bit value
    if (I2C_Rready == -1) {
        return -1;
    }
    Val2MSB = I2cbRegs.I2CDRR.all;       // Read CHIPXYZ
    if (I2cbRegs.I2CSTR.bit.NACK == 1) { // Check for No Acknowledgement
        return 3; // This should not happen
    }

    // Since I2CCNT = 0 at this point, a stop condition will be issued

    *Rvalue1 = (Val1MSB << 8) | (Val1LSB & 0xFF);
    *Rvalue2 = (Val2MSB << 8) | (Val2LSB & 0xFF);
    return 0;
}
```

Example read/write

Look at the code below. This is the while(1) loop of your main() function. It is an example to show you how to properly call the I2C functions. The variable `RunI2C` is used as a flag variable in a similar manner to `UARTPrint`. You will modify the below code to work with your defined DAN777 read/write functions, as well as the BQ32000 read function.

```
while(1)
{
    if (UARTPrint == 1 ) {
        serial_printf(&SerialA,"CMDXYZ1: %d, CMDXYZ2: %d, RETval1: %d, RETval2: %d\r\n", CMDXYZ1, CMDXYZ2,
RETval1, RETval2);
        UARTPrint = 0;
    }

    if (RunI2C == 1)  {
        RunI2C = 0;

        // Write to CHIPXYZ
        I2C_OK = WriteTwo16BitValuesToCHIPXYZ(CMDXYZ1, CMDXYZ2);
        num_WriteCHIPXYZ_Errors = 0;
        while(I2C_OK != 0) {
            num_WriteCHIPXYZ_Errors++;
```

```
            if (num_WriteCHIPXYZ_Errors > 2) {
                serial_printf(&SerialA,"WriteTwo16BitValuesToCHIPXYZ Error: %d\r\n",I2C_OK);
                I2C_OK = 0;
            } else {
                I2CB_Init();
                DELAY_US(100000);
                I2C_OK = WriteTwo16BitValuesToCHIPXYZ(CMDXYZ1, CMDXYZ2);
            }
        }

        // Read CHIPXYZ
        I2C_OK = ReadTwo16BitValuesFromCHIPXYZ(&RETval1, &RETval2);
        num_ReadCHIPXYZ_Errors = 0;
        while(I2C_OK != 0) {
            num_ReadCHIPXYZ_Errors++;
            if (num_ReadCHIPXYZ_Errors > 2) {
                serial_printf(&SerialA,"ReadTwo16BitValuesFromCHIPXYZ Error: %d\r\n",I2C_OK);
                I2C_OK = 0;
            } else {
                I2CB_Init();
                DELAY_US(100000);
                I2C_OK = ReadTwo16BitValuesFromCHIPXYZ(&RETval1, &RETval2);
            }
        }
    }
}
```

Items to complete first checkoff due Wednesday October 25th, 1:00 PM. Worth 20% of HW Grade:

Demo/checkoff your DAN777 code calling the WriteDAN777RCServo and ReadDAN777ADC functions, one right after the other, inside main() and its while(1) loop. In one of the CPU timer interrupt functions, set the "RunI2C" flag variable to 1 every 20ms. This will flag your code in the while(1) loop to run the DAN777 I2C functions every 20ms. Print to Teraterm, every 100 ms, the two DAN777 ADC readings. In addition, command the two RC servo motors to continuously increase to approximately 90 degrees and then decrease to approximately -90 degrees just as we did in Lab 3. Remember that you will need 4 AA batteries in your battery pack to provide power to the RCServos. Don't forget to turn on the battery packs ON switch. The final part of the checkoff will be displaying signals on the oscilloscope. Show the SDA and SCL I2C signals along with the RC servo PWM signals at the same time. Take a picture of the signals at two different duty cycles for your future submission.

Items to complete second checkoff due Wednesday November 1st, 1:00 PM. Worth 20% of HW Grade:

Demo/checkoff both your DAN777 and BQ32000 code working together calling the three functions WriteDAN777RCServo, ReadDAN777ADC and ReadBQ32000 one right after the other. Call these three functions in a row every 20ms. Print to Teraterm, every 100 ms, the two DAN777 ADC readings and the current date and time in the format "day mm/dd/yy hour:minute:second." i.e. "Wednesday 03/24/21 23:16:34". In addition, command two the RC servo motors to continuously increase to approximately 90 degrees and then decrease to approximately -90 degrees just as we did in Lab 3. Remember that you will need 4 AA batteries in your battery pack to provide power to the RCServos. Don't forget to turn on the battery packs ON switch. The final part of the checkoff will be displaying the I2C signals on the oscilloscope.

In addition, demonstrate that your `WriteBQ32000()` works to set a new day, date, and time. You only need to call `WriteBQ32000()` once before the while(1) loop in main().

Box Submission due November 1st, 11:59PM

1. Submit to your Box folder:
    a. Your commented main C file. Should include the I2C functions you wrote.
    b. Tera Term screen shot showing the printout of your program working.
    c. Two pictures (with your phone) of the Oscilloscope displaying different PWM duty cycles produced by your I2C code transmitting new duty cycle commands. At your check off, you will demonstrate the PWM duty cycle continuously varying. Here you will just take two pictures.