10/07/23

July 10, 2023 9:04 AM

Time: 8h

Start 8:00

Research PIC18 XC8 i2c examples.

I couldn't really find anything. Everything just says use the MCC and that's it. But that doesn't work either.

I did find this:
https://onlinedocs.microchip.com/oxy/GUID-DB72E815-2CD8-46B2-8FA5-176501CEA038-en-US-9/GUID-5C58BBAE-7FF1-43CA-8032-16B009A4F9F8.html

Which talks about detecting erroneous start and stop conditions.

My main issues is just that the start condition is always set. But maybe I can try and implement this and see if it helps.

Programming issues

Last week there was an issue where I was no longer able to program the PIC. Me and Nick messed with it and found that the error has to do with using the PI as the power supply as opposed to the Arduino shield.

We switched back to the shield and it appears to be working now

Timer Interrupts

```
#include <xc.h>
#include <stdbool.h>

#define _XTAL_FREQ 4000000

bool flag = 0;

void setup(){
  OSCFRQ = 0b0010;
  OSCCON1bits.NOSC = 0b110;

  /*Config Output pin for test*/
  ANSELBbits.ANSELB6 = 0;
  TRISBbits.TRISB6 = 0;   //Set B6 to output
  LATBbits.LATB6 = 0;

  ANSELAbits.ANSELA0 = 0;
  TRISAbits.TRISA0 = 0;
  RA0PPS = 0x23;

  /*Config Timer0*/
  T0CON0bits.EN = 0;     //Disable timer
  T0CON0bits.MD16 = 0;    //Set timer to 16bit mode
  T0CON0bits.OUTPS = 0b0000;  //Set post scaler to 16

  T0CON1bits.CS = 0b010;     //Set clock source to Fosc/4
  T0CON1bits.ASYNC = 0;      //Clock is synchronized to Fosc/4
  T0CON1bits.CKPS = 0b0000;   //Prescaler set to 1

  T0CON0bits.EN = 1;        //Enable timer

  /*Config Interrupt*/
  INTCON0bits.GIE = 1;    //Enable interrupts
  INTCON0bits.IPEN = 1;   //Enable Interrupt Priorities

  PIR3bits.TMR0IF = 0;    //Clear interrupt flag
  PIE3bits.TMR0IE = 1;    //Enable timer0 interrupts
}

void __interrupt(irq(TMR0)) ISR(void){
  PIR3bits.TMR0IF = 0;
```

```
  //flag  = !flag;
  LATBbits.LATB6 = 1;
  for(int i = 0; i < 2; i++){
    NOP();
  }
  LATBbits.LATB6 = 0;
}

void loop(){
  //LATBbits.LATB6 = flag;
}


void main(void){
  setup();
  while(1){
    loop();
  }
  return;
}
```

This script is working where I get a consistent 3.906kHz trigger on the 8bit timer mode.

8bit timer rolls over every 256 cycles

The timer should be counting at 1Mhz
1000000/256=3906.25

Which means that this is working completely properly.

Let me try switching into 16 bit mode and seeing if everything still makes sense.

1000000/65536=15.2588
I should get about 15.26Hz

I am getting 15.24Hz, which is very close, however the strange triggers are occuring again. This was not occuring in 8bit mode.

If I use an 8bit timer, can I get a 1 second delay?

$\frac{1MHz}{256} = 3906.25\,Hz = 256\times10^{-6}\,s = 256\,\mu s$

1ms / count

$\sim 4\times10^{3}$ prescale required

$4096 \cdot 256\times10^{-6} = 1.0485\ldots$

Prescale 4096

$\frac{1}{4096} = 244.14$

256 - 244 = 12 preset

For 1ms

Prescale 4 ⇒

$\frac{4}{Fosc} \cdot Prescale \cdot Cnt = Period$

$\frac{4}{4\times10^{6}} \cdot 4 \cdot Cnt = Period$

$4 \times 10^{-6} \cdot Cnt = Period$

$$Cnt = \frac{1 \times 10^{-3}}{4 \times 10^{-6}} = 250$$

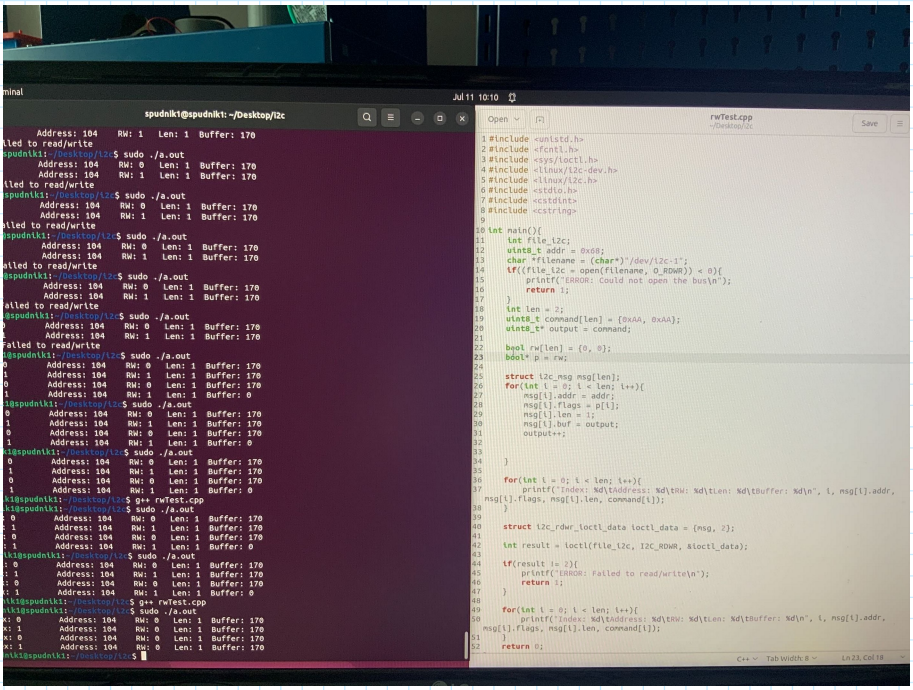
Stop: 4:00

11/07/23

July 11, 2023 8:36 AM

Time: 7h

Start 8:30



To do

- I2C on PIC
- I2C on PI for IMU
- UART on PIC

I played around with a test script and got a generic method of declaring the message structure that should allow it to be dynamic in the library. Currently though it is limited to 1 byte transactions.

The addresses on the board are 1byte and the data is read from a high and low register individually so that should also be 1byte each.

I added this into my library

```
int i2c::writeReadBus(uint8_t address, int length, uint8_t *command, bool *R_W){
    /*
    writeReadBus(uint8_t address, int length, uint8_t *command, bool *R_W) - This method is to write to a client on the I2C bus
        Parameters:
            address     -   7bit address of the client to read data from
            length      -   Number of commands
            command     -   Pointer to array holding the write commands and acting as a buffer to store the read commands
            R_W         -   Pointer to an array holding a 1 (read) or 0 (write) to determine transaction

        Return:
            0           -   The transmission was successful
            ~0          -   An error occured
    */

    struct i2c_msg msg[length];     //Initialize structure for messages

    /*Format each of the messages in the msg structure*/
    for(int i = 0; i < length; i++){
        msg[i].addr = address;  //Set the client address
        msg[i].flags = R_W[i];  //Set read (1) or write (0)
        msg[i].len = 1;         //Set length of message
        msg[i].buf = command;   //Provide the read/write buffer
        command++;              //Increment the pointer to the buffer
    }

    struct i2c_rdwr_ioctl_data ioctl_data = {msg, length};  //Format the messages into the read/write structure

    /*Perform transaction*/
    if(ioctl(file_i2c, I2C_RDWR, &ioctl_data) != length){
        printf("ERROR:\tFailed to complete read/write command");
        return 1;
    }
    return 0;
}
```

This appears to be working properly. I cant really check the read right now as I have nothing to read, but once I get the IMU library set up I can do that.

IMU library

Now that I have this, I think I can start making the library for the IMU -- First I am going to go through the datasheet and see what I may need.

8.2 USER_CTRL

This could be useful in a setup routine

Name: USER_CTRL
Address: 3 (03h)
Type: USR0
Bank: 0
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7	DMP_EN	1 – Enables DMP features. 0 – DMP features are disabled after the current processing round has completed.
6	FIFO_EN	1 – Enable FIFO operation mode. 0 – Disable FIFO access from serial interface. To disable FIFO writes by DMA, use FIFO_EN register. To disable possible FIFO writes from DMP, disable the DMP.
5	I2C_MST_EN	1 – Enable the I²C Master I/F module; pins ES_DA and ES_SCL are isolated from pins SDA/SDI and SCL/ SCLK. 0 – Disable I²C Master I/F module; pins ES_DA and ES_SCL are logically driven by pins SDA/SDI and SCL/ SCLK.
4	I2C_IF_DIS	1 – Reset I²C Slave module and put the serial interface in SPI mode only.
3	DMP_RST	1 – Reset DMP module. Reset is asynchronous. This bit auto clears after one clock cycle of the internal 20 MHz clock.
2	SRAM_RST	1 – Reset SRAM module. Reset is asynchronous. This bit auto clears after one clock cycle of the internal 20 MHz clock.
1	I2C_MST_RST	1 – Reset I²C Master module. Reset is asynchronous. This bit auto clears after one clock cycle of the internal 20 MHz clock. **NOTE**: This bit should only be set when the I²C master has hung. If this bit is set during an active I²C master transaction, the I²C slave will hang, which will require the host to reset the slave.
0	-	Reserved.

8.4 PWR_MGMT_1

Could be useful to disable when not being used

Name: PWR_MGMT_1
Address: 6 (06h)
Type: USR0
Bank: 0
Serial IF: R/W
Reset Value: 0x41

BIT	NAME	FUNCTION
7	DEVICE_RESET	1 – Reset the internal registers and restores the default settings. Write a 1 to set the reset, the bit will auto clear.
6	SLEEP	When set, the chip is set to sleep mode (in sleep mode all analog is powered off). Clearing the bit wakes the chip from sleep mode.
5	LP_EN	The LP_EN only affects the digital circuitry, it helps to reduce the digital current when sensors are in LP mode. Please note that the sensors themselves are set in LP mode by the LP_CONFIG register settings. Sensors in LP mode, and use of LP_EN bit together help to reduce overall current. The bit settings are: 1: Turn on low power feature. 0: Turn off low power feature. LP_EN has no effect when the sensors are in low-noise mode.
4	-	Reserved.
3	TEMP_DIS	When set to 1, this bit disables the temperature sensor.
2:0	CLKSEL[2:0]	**Code: Clock Source** 0: Internal 20 MHz oscillator 1-5: Auto selects the best available clock source – PLL if ready, else use the Internal oscillator 6: Internal 20 MHz oscillator 7: Stops the clock and keeps timing generator in reset **NOTE**: CLKSEL[2:0] should be set to 1~5 to achieve full gyroscope performance.

8.5 PWR_MGMT_2

Name: PWR_MGMT_2
Address: 7 (07h)
Type: USR0
Bank: 0
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:6	-	Reserved.
5:3	DISABLE_ACCEL	Only the following values are applicable: 111 – Accelerometer (all axes) disabled. 000 – Accelerometer (all axes) on.
2:0	DISABLE_GYRO	Only the following values are applicable: 111 – Gyroscope (all axes) disabled. 000 – Gyroscope (all axes) on.

Output register for the accelerometer

8.18 ACCEL_XOUT_H

Name: ACCEL_XOUT_H
Address: 45 (2Dh)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_XOUT_H[7:0]	High Byte of Accelerometer X-axis data.

8.19 ACCEL_XOUT_L

Name: ACCEL_XOUT_L
Address: 46 (2Eh)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_XOUT_L[7:0]	Low Byte of Accelerometer X-axis data. To convert the output of the accelerometer to acceleration measurement use the formula below: X_acceleration = ACCEL_XOUT/Accel_Sensitivity

8.20 ACCEL_YOUT_H

Name: ACCEL_YOUT_H
Address: 47 (2Fh)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_YOUT_H[7:0]	High Byte of Accelerometer Y-axis data.

8.21 ACCEL_YOUT_L

Name: ACCEL_YOUT_L
Address: 48 (30h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_YOUT_L[7:0]	Low Byte of Accelerometer Y-axis data. To convert the output of the accelerometer to acceleration measurement use the formula below: Y_acceleration = ACCEL_YOUT/Accel_Sensitivity

8.22 ACCEL_ZOUT_H

Name: ACCEL_ZOUT_H
Address: 49 (31h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_ZOUT_H[7:0]	High Byte of Accelerometer Z-axis data.

8.23 ACCEL_ZOUT_L

Name: ACCEL_ZOUT_L
Address: 50 (32h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_ZOUT_L[7:0]	Low Byte of Accelerometer Z-axis data. To convert the output of the accelerometer to acceleration measurement use the formula below: Z_acceleration = ACCEL_ZOUT/Accel_Sensitivity

Output registers for the gyro

8.24 GYRO_XOUT_H

Name: GYRO_XOUT_H
Address: 51 (33h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_XOUT_H[7:0]	High Byte of Gyroscope X-axis data.

8.25 GYRO_XOUT_L

Name: GYRO_XOUT_L
Address: 52 (34h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_XOUT_L[7:0]	Low Byte of Gyroscope X-axis data. To convert the output of the gyroscope to angular rate measurement use the formula below: X_angular_rate = GYRO_XOUT/Gyro_Sensitivity

8.26 GYRO_YOUT_H

Name: GYRO_YOUT_H
Address: 53 (35h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_YOUT_H[7:0]	High Byte of Gyroscope Y-axis data.

8.27 GYRO_YOUT_L

Name: GYRO_YOUT_L
Address: 54 (36h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_YOUT_L[7:0]	Low Byte of Gyroscope Y-axis data. To convert the output of the gyroscope to angular rate measurement use the formula below: Y_angular_rate = GYRO_YOUT/Gyro_Sensitivity

8.28 GYRO_ZOUT_H

Name: GYRO_ZOUT_H
Address: 55 (37h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_ZOUT_H[7:0]	High Byte of Gyroscope Z-axis data.

8.29 GYRO_ZOUT_L

Name: GYRO_ZOUT_L
Address: 56 (38h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_ZOUT_L[7:0]	Low Byte of Gyroscope Z-axis data. To convert the output of the gyroscope to angular rate measurement use the formula below: Z_angular_rate = GYRO_ZOUT/Gyro_Sensitivity

8.30 TEMP_OUT_H

Temperature output -- might not be useful but maybe

Name: TEMP_OUT_H
Address: 57 (39h)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	TEMP_OUT_H[7:0]	High Byte of Temp sensor data.

8.31 TEMP_OUT_L

Name: TEMP_OUT_L
Address: 58 (3Ah)
Type: USR0
Bank: 0
Serial IF: R
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	TEMP_OUT_L[7:0]	Low Byte of Temp sensor data. To convert the output of the temperature sensor to degrees C use the following formula: TEMP_degC = ((TEMP_OUT – RoomTemp_Offset)/Temp_Sensitivity) + 21degC

Setting up/configuring the gyro -- could be good to give some user control over this as opposed to presetting it.

10.1 GYRO_SMPLRT_DIV

Name: GYRO_SMPLRT_DIV
Address: 0 (00h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	GYRO_SMPLRT_DIV[7:0]	Gyro sample rate divider. Divides the internal sample rate to generate the sample rate that controls sensor data output rate, FIFO sample rate, and DMP sequence rate. **NOTE:** This register is only effective when FCHOICE = 1'b1 (FCHOICE_B register bit is 1'b0), and (0 < DLPF_CFG < 7). ODR is computed as follows: 1.1 kHz/(1+GYRO_SMPLRT_DIV[7:0])

10.2 GYRO_CONFIG_1

Name: GYRO_CONFIG_1
Address: 1 (01h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x01

BIT	NAME	FUNCTION
7:6	-	Reserved.
5:3	GYRO_DLPFCFG[2:0]	Gyro low pass filter configuration as shown in Table 16.
2:1	GYRO_FS_SEL[1:0]	Gyro Full Scale Select: 00 = ±250 dps 01= ±500 dps 10 = ±1000 dps 11 = ±2000 dps
0	GYRO_FCHOICE	0 – Bypass gyro DLPF. 1 – Enable gyro DLPF.

The gyroscope DLPF is configured by GYRO_DLPFCFG, when GYRO_FCHOICE = 1. The gyroscope data is filtered according to the value of GYRO_DLPFCFG and GYRO_FCHOICE as shown in Table 16.

10.3 GYRO_CONFIG_2

Name: GYRO_CONFIG_2
Address: 2 (02h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:6	-	Reserved.
5	XGYRO_CTEN	X Gyro self-test enable.
4	YGYRO_CTEN	Y Gyro self-test enable.
3	ZGYRO_CTEN	Z Gyro self-test enable.
2:0	GYRO_AVGCFG[2:0]	Averaging filter configuration settings for low-power mode. 0: 1x averaging. 1: 2x averaging. 2: 4x averaging. 3: 8x averaging. 4: 16x averaging. 5: 32x averaging. 6: 64x averaging. 7: 128x averaging.

Gyroscope offset values

10.4 XG_OFFS_USRH

Name: XG_OFFS_USRH
Address: 3 (03h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	X_OFFS_USER[15:8]	Upper byte of X gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.5 XG_OFFS_USRL

Name: XG_OFFS_USRL
Address: 4 (04h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	X_OFFS_USER[7:0]	Lower byte of X gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.6 YG_OFFS_USRH

Name: YG_OFFS_USRH
Address: 5 (05h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	Y_OFFS_USER[15:8]	Upper byte of Y gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.7 YG_OFFS_USRL

Name: YG_OFFS_USRL
Address: 6 (06h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	Y_OFFS_USER[7:0]	Lower byte of Y gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.8 ZG_OFFS_USRH

Name: ZG_OFFS_USRH
Address: 7 (07h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	Z_OFFS_USER[15:8]	Upper byte of Z gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.9 ZG_OFFS_USRL

Name: ZG_OFFS_USRL
Address: 8 (08h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	Z_OFFS_USER[7:0]	Lower byte of Z gyro offset cancellation. Step size: 0.0305 dps/LSB.

10.11 ACCEL_SMPLRT_DIV_1

Accelerometer config

Name: ACCEL_SMPLRT_DIV_1
Address: 16 (10h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:4	-	Reserved.
3:0	ACCEL_SMPLRT_DIV[11:8]	MSB for ACCEL sample rate div.

10.12 ACCEL_SMPLRT_DIV_2

Name: ACCEL_SMPLRT_DIV_2
Address: 17 (11h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:0	ACCEL_SMPLRT_DIV[7:0]	LSB for ACCEL sample rate div. ODR is computed as follows: 1.125 kHz/(1+ACCEL_SMPLRT_DIV[11:0])

10.15 ACCEL_CONFIG

Name: ACCEL_CONFIG
Address: 20 (14h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x01

BIT	NAME	FUNCTION
7:6	-	Reserved.
5:3	ACCEL_DLPFCFG[2:0]	Accelerometer low pass filter configuration as shown in Table 18.
2:1	ACCEL_FS_SEL[1:0]	Accelerometer Full Scale Select: 00: ±2g 01: ±4g 10: ±8g 11: ±16g
0	ACCEL_FCHOICE	0: Bypass accel DLPF. 1: Enable accel DLPF.

10.16 ACCEL_CONFIG_2

Name: ACCEL_CONFIG_2
Address: 21 (15h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
7:5	-	Reserved.
4	AX_ST_EN_REG	X Accel self-test enable.
3	AY_ST_EN_REG	Y Accel self-test enable.
2	AZ_ST_EN_REG	Z Accel self-test enable.
1:0	DEC3_CFG[1:0]	Controls the number of samples averaged in the accelerometer decimator: 0: Average 1 or 4 samples depending on ACCEL_FCHOICE (see Table 19). 1: Average 8 samples. 2: Average 16 samples. 3: Average 32 samples.

10.18 TEMP_CONFIG

Temp config

Name: TEMP_CONFIG
Address: 83 (53h)
Type: USR2
Bank: 2
Serial IF: R/W
Reset Value: 0x00

BIT	NAME	FUNCTION
2:0	TEMP_DLPFCFG[2:0]	Low pass filter configuration for temperature sensor as shown in the table below:

TEMP_DLPFCFG<2:0>	TEMP SENSOR	
	NBW (HZ)	RATE (KHZ)
0	7932.0	9
1	217.9	1.125
2	123.5	1.125
3	65.9	1.125
4	34.1	1.125
5	17.3	1.125
6	8.8	Rate (kHz)
7	7932.0	9

13.3 HXL TO HZH: MEASUREMENT DATA

Magnet readings

ADDR	REGISTER NAME	D7	D6	D5	D4	D3	D2	D1	D0
READ-ONLY REGISTER									
11H	HXL	HX7	HX6	HX5	HX4	HX3	HX2	HX1	HX0
12H	HXH	HX15	HX14	HX13	HX12	HX11	HX10	HX9	HX8
13H	HYL	HY7	HY6	HY5	HY4	HY3	HY2	HY1	HY0
14H	HYH	HY15	HY14	HY13	HY12	HY11	HY10	HY9	HY8
15H	HZL	HZ7	HZ6	HZ5	HZ4	HZ3	HZ2	HZ1	HZ0
16H	HZH	HZ15	HZ14	HZ13	HZ12	HZ11	HZ10	HZ9	HZ8
Reset		0	0	0	0	0	0	0	0

13.2 ST1: STATUS 1

Magnet status register

ADDR	REGISTER NAME	D7	D6	D5	D4	D3	D2	D1	D0
READ-ONLY REGISTER									
10H	ST1	0	0	0	0	0	0	DOR	DRDY
Reset		0	0	0	0	0	0	0	0

DRDY: Data Ready

"0": Normal

"1": Data is ready

DRDY bit turns to "1" when data is ready in Single measurement mode, Continuous measurement mode 1, 2, 3, 4 or Self-test mode. It returns to "0" when any one of ST2 register or measurement data register (HXL to TMPS) is read.

DOR: Data Overrun

"0": Normal

"1": Data overrun

DOR bit turns to "1" when data has been skipped in Continuous measurement mode 1, 2, 3, 4. It returns to "0" when any one of ST2 register or measurement data register (HXL to TMPS) is read.

13.5 CNTL2: CONTROL 2

Magnet control reg

ADDR	REGISTER NAME	D7	D6	D5	D4	D3	D2	D1	D0
READ/WRITE REGISTER									
31H	CNTL2	0	0	0	MPDE4	MODE3	MODE2	MODE1	MODE0
Reset		0	0	0	0	0	0	0	0

MODE[4:0] bits: Operation mode setting

"00000": Power-down mode

"00001": Single measurement mode

"00010": Continuous measurement mode 1

"00100": Continuous measurement mode 2

"00110": Continuous measurement mode 3

"01000": Continuous measurement mode 4

"10000": Self-test mode

Other code settings are prohibited

When each mode is set, AK09916 transits to the set mode.

12:30 back

13.6 CNTL3: CONTROL 3

Was setting up the code to have seperate configuration functions with all the settings as parameters

ADDR = 32H	REGISTER NAME	D7	D6	D5	D4	D3	D2	D1	D0
		READ/WRITE REGISTER							
	CNTL3	0	0	0	0	0	0	0	SRST
	Reset	0	0	0	0	0	0	0	0

This is pretty messy and would require all parameters to be updated when changing a setting

My thought was to give a default to each parameter, so you would only have to pass the ones that aren't the default value

SRST: Soft reset

but this would mean they could only be called once or it would reset

for example

"0": Normal

"1": Reset

Void Foo(int a=0, int b=0);

When "1" is set, all registers are initialized. After reset, SRST bit turns to "0" automatically.

Foo(a=1); → a=1, b=0

This seems like everything that should be relevant -- from my understanding at least.

Foo(b=1); → a=0, b=1

a reset

12:00 lunch

I think it may be better to have all of these as variables to the class, then I can have one generic update(); function that writes everything to the IMU

Updating everything at once may be inefficient for the bus but should not need to occur very often

saving everything as a variable should be easier as the resetting issue is abided

To organize things, I may try to figure out how to make a config structure so all the variables are contained

0101 & ~(0110)

0101
& 1001
0001

12/07/23

Time: 7.5h

July 12, 2023 11:02 AM

Start: 8:00

I have been working on my IMU library all morning

I eventually got it finished to a point where I feel like I can test it.

I put it on the PI and had to deal with a ton of compilation errors, as expected.

Using the I2Cdetect command, I can see that there are two devices on the bus -- The imu at address 0x68 and the magnetometer at address 0x0C. This is good cause this is how I was understadning it worked.

When I run my test script to read the registers, I get a ton of runtime errors and nearly all of the communications are failing.

After adding in some print statements to debug, I found that the USER_CTRL setup communication was the only one that didn't fail. Looking into the code again, I see that this was the only command that had a length of two, using the format ADDRW, REGADDR, ADDRW, DATA

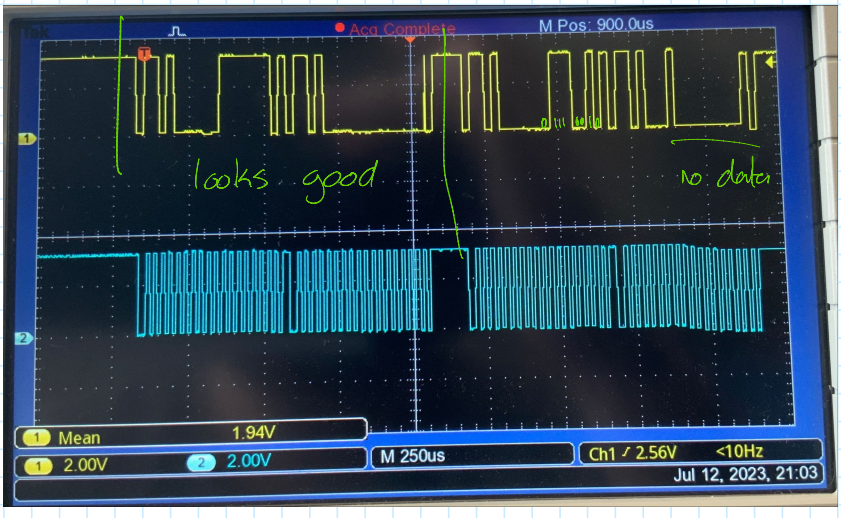
In other commands, I kept this going, for example:
ADDRW, REGADDR1 ADDRW, DATA, ADDRW, REGADDR2, ADDRW, DATA, ADDRW, REGADDR3, ADDRW, DATA

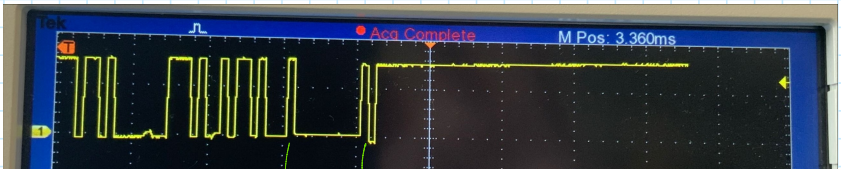
This fails.

I think I need to split everything into 2 command transactions which stop conditions in between.

I did this in the PWR_MGMT setup as it was only 4 commands. Splitting it into 2 and 2 remove the communicaiton failure. I am going to need to change this for all funcitons and try again.

I changed everything and now I am not getting any compiler or runtime errors. However I am still not getting the desired output. Everything is reading 0. Not sure if that is from the IMU or just my method of managing data on the PI.





Comands should be

0x68 client address + W(0)
0x7F change bank reg
0x68 address + W(0)
0x00 bank

0x68 client address + W(0)
0x39 temp ##
0x68 + R(1)
XXXX data

0x68 W(0)
0x3A
0x68 R(1)
XXXX data

Lunch 12:00-12:30

All of the communication seems to make sense, it is just that the IMU is not writing any data to the bus on the read commands… I am not sure if that means I am formatting the commands wrong, or if there is just no data in the registers to read. The data sheet says that the registers can be read at any time. I checked the default config values I set and I didn't accidentally disable the sensors by accident.

I am realizing that I never name a magnetometer config function and I need to set the mode to get it to run, so that is one thing that is definitely wrong but that should not be affecting the other things also reading 0.

13.4 ST2: STATUS 2

ADDR	REGISTER NAME	D7	D6	D5	D4	D3	D2	D1	D0
READ-ONLY REGISTER									
18H	ST2	0	RSV30	RSV29	RSV28	HOFL	0	0	0
Reset		0	0	0	0	0	0	0	0

ST2[6:4] bits: Reserved register for AKM.

HOFL: Magnetic sensor overflow

"0": Normal

"1": Magnetic sensor overflow occurred

In Single measurement mode, Continuous measurement mode 1, 2, 3, 4, and Self-test mode, magnetic sensor may overflow even though measurement data register is not saturated. In this case, measurement data is not correct and HOFL bit turns to "1". When measurement data register is updated, HOFL bit is updated.

ST2 register has a role as data reading end register, also. When any of measurement data register (HXL to TMPS) is read in Continuous measurement mode 1, 2, 3, 4, it means data reading start and taken as data reading until ST2 register is read. Therefore, when any of measurement data is read, be sure to read ST2 register at the end.

I am also supposed to read the STATUS 2 register of the mag after reading the sensor

I added a function to read the 'whoami' register. This register is read only and always returns the same value, 0xEA == 234.

I ran this as a test to see if I can communicate with the IMU at all.

Sure enough, I got 234 back from the device. Which means I think my method of reading the IMU is correct. I think maybe there is a problem where I need to tell the sensor to start writing to the registers somehow…

But I don't see that in the datasheet.

PDF

imu.h

imu.h

hardwareInterfacing\imu-lib\imu.h

```
#ifndef IMU_H
#define IMU_H

#include "../i2c-lib/i2c.h"
#include <stdio.h>

class imu {
    private:
        uint8_t addr;   //i2c address of the IMU
        i2c I2C;

    public:
        struct {
            bool DMP_EN = 0;
            bool FIFO_EN = 0;
            bool I2C_MST_EN = 0;
            bool I2C_IF_DIS = 0;
            bool DMP_RST = 0;
            bool SRAM_RST = 0;
            bool I2C_MST_RST = 0;
        } USER_CTRL;

        struct {
            bool DEVICE_RESET = 0;
            bool SLEEP = 0;
            bool LP_EN = 0;
            bool TEMP_DIS = 0;
            int CLKSEL = 1;
            bool DISABLE_ACCEL = 0;
            bool DISABLE_GYRO = 0;
        } PWR_MGMT;

        struct{
            uint8_t GYRO_SMPLRT_DIV = 0;
            uint8_t GYRO_DLPFCFG = 0;
            uint8_t GYRO_FS_SEL = 0;
            bool GYRO_FCHOICE = 0;
            bool XGYRO_CTEN = 0;
            bool YGYRO_CTEN = 0;
            bool ZGYRO_CTEN = 0;
            uint8_t GYRO_AVGCFG = 0;
        } GYRO_CONFIG;

        struct{
            uint16_t ACCEL_SMPLRT_DIV = 0;
            uint8_t ACCEL_DLPFCFG = 0;
            uint8_t ACCEL_FS_SEL = 0;
            bool ACCEL_FCHOICE = 0;
            bool AX_ST_EN_REG = 0;
            bool AY_ST_EN_REG = 0;
            bool AZ_ST_EN_REG = 0;
            uint8_t DEC3_CFG = 0;
```

```
            bool AZ_ST_EN_REG = 0;
            uint8_t DEC3_CFG = 0;
```

```

            bool ACCEL_INTEL_EN = 0;
            bool ACCEL_INTEL_MODE_INT = 1;
        } ACCEL_CONFIG;

        struct{
            uint8_t MODE = 0b00000010;
            bool SRST = 0;
        } MAG_CONFIG;

        imu(i2c& I2Cobj, uint8_t address = 0b1101000);

        void update_USER_CTRL();
        void update_PWR_MGMT();
        void update_GYRO_CONFIG();
        void update_ACCEL_CONFIG();
        void update_MAG_CONFIG();

        void setGyroOffset(uint16_t x, uint16_t y, uint16_t z);
        void setAccelOffset(uint16_t x, uint16_t y, uint16_t z);

        void readAccel(uint16_t *x, uint16_t *y, uint16_t *z);
        void readGyro(uint16_t *x, uint16_t *y, uint16_t *z);
        void readMag(uint16_t *x, uint16_t *y, uint16_t *z);
        void readTemp(uint16_t *t);

        void changeBank(uint8_t bank);

        uint8_t whoami();

};

#endif
```

imu.cpp

hardwareInterfacing\imu-lib\imu.cpp

```cpp
#include "imu.h"

imu::imu(i2c& I2Cobj, uint8_t address){
    addr = address;
    I2C = I2Cobj;

    update_USER_CTRL();
    update_PWR_MGMT();
    update_ACCEL_CONFIG();
    update_GYRO_CONFIG();
}

void imu::update_USER_CTRL(){
    changeBank(0x00);   //Change to bank 0
    uint8_t REG_ADDR = 0x03;

    uint8_t USER_CTRL_byte;
    USER_CTRL_byte |= USER_CTRL.DMP_EN << 7;
    USER_CTRL_byte |= USER_CTRL.FIFO_EN << 6;
    USER_CTRL_byte |= USER_CTRL.I2C_MST_EN << 5;
    USER_CTRL_byte |= USER_CTRL.I2C_IF_DIS << 4;
    USER_CTRL_byte |= USER_CTRL.DMP_RST << 3;
    USER_CTRL_byte |= USER_CTRL.SRAM_RST << 2;
    USER_CTRL_byte |= USER_CTRL.I2C_MST_RST << 1;

    uint8_t command[2] = {REG_ADDR, USER_CTRL_byte};
    bool rw[2] = {0, 0};

    I2C.writeReadBus(addr, 2, command, rw);
}

void imu::update_PWR_MGMT(){
    changeBank(0x00);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_PWR_MGMT_1 = 0x06;
    uint8_t REG_PWR_MGMT_2 = 0x07;

    uint8_t PWR_MGMT_1;
    PWR_MGMT_1 |= PWR_MGMT.DEVICE_RESET << 7;
    PWR_MGMT_1 |= PWR_MGMT.SLEEP << 6;
    PWR_MGMT_1 |= PWR_MGMT.LP_EN << 5;
    PWR_MGMT_1 |= PWR_MGMT.TEMP_DIS << 3;
    PWR_MGMT_1 |= (PWR_MGMT.CLKSEL & 0x03);

    uint8_t PWR_MGMT_2;
    if(PWR_MGMT.DISABLE_ACCEL){
```

```
45      PWR_MGMT_1 |= (PWR_MGMT.CLKSEL & 0x03);
46
47      uint8_t PWR_MGMT_2;
48      if(PWR_MGMT.DISABLE_ACCEL){
49          PWR_MGMT_2 |= 0b00111000;
50      } else {
51          PWR_MGMT_2 |= ~(0b00111000);
52      }
```

```
    if(PWR_MGMT.DISABLE_GYRO){
        PWR_MGMT_2 |= 0b00000111;
    } else {
        PWR_MGMT_2 |= ~(0b00000111);
    }

    rw[0] = 0;
    rw[1] = 0;

    command[0] = REG_PWR_MGMT_1;
    command[1] = PWR_MGMT_1;

    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_PWR_MGMT_2;
    command[1] = PWR_MGMT_2;

    I2C.writeReadBus(addr, 2, command, rw);
}

void imu::update_ACCEL_CONFIG(){
    changeBank(0x02);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_ACCEL_SMPLRT_DIV_1 = 0x10;
    uint8_t REG_ACCEL_SMPLRT_DIV_2 = 0x11;
    uint8_t REG_ACCEL_INTEL_CTRL = 0x12;
    uint8_t REG_ACCEL_CONFIG_1 = 0x14;
    uint8_t REG_ACCEL_CONFIG_2 = 0x15;

    uint8_t ACCEL_SMPLRT_DIV_1;
    uint8_t ACCEL_SMPLRT_DIV_2;
    uint8_t ACCEL_INTEL_CTRL;
    uint8_t ACCEL_CONFIG_1;
    uint8_t ACCEL_CONFIG_2;

    ACCEL_SMPLRT_DIV_1 = ACCEL_CONFIG.ACCEL_SMPLRT_DIV >> 8;

    ACCEL_SMPLRT_DIV_2 = ACCEL_CONFIG.ACCEL_SMPLRT_DIV & 0xFF;

    ACCEL_INTEL_CTRL |= ACCEL_CONFIG.ACCEL_INTEL_EN << 1;
    ACCEL_INTEL_CTRL |= ACCEL_CONFIG.ACCEL_INTEL_MODE_INT;

    ACCEL_CONFIG_1 |= (ACCEL_CONFIG.ACCEL_DLPFCFG & 0x07) << 3;
    ACCEL_CONFIG_1 |= (ACCEL_CONFIG.ACCEL_FS_SEL & 0x03) << 1;
    ACCEL_CONFIG_1 |= ACCEL_CONFIG.ACCEL_FCHOICE;

    ACCEL_CONFIG_2 |= ACCEL_CONFIG.AX_ST_EN_REG << 4;
    ACCEL_CONFIG_2 |= ACCEL_CONFIG.AY_ST_EN_REG << 3;
    ACCEL_CONFIG_2 |= ACCEL_CONFIG.AZ_ST_EN_REG << 2;
    ACCEL_CONFIG_2 |= ACCEL_CONFIG.DEC3_CFG & 0x03;

    rw[0] = 0;
    rw[1] = 0;

```

```
    command[0] = REG_ACCEL_SMPLRT_DIV_1;
    command[1] = ACCEL_SMPLRT_DIV_1;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ACCEL_SMPLRT_DIV_2;
    command[1] = ACCEL_SMPLRT_DIV_2;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ACCEL_INTEL_CTRL;
    command[1] = ACCEL_INTEL_CTRL;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ACCEL_CONFIG_1;
    command[1] = ACCEL_CONFIG_1;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ACCEL_CONFIG_2;
    command[1] = ACCEL_CONFIG_2;
    I2C.writeReadBus(addr, 2, command, rw);
}

void imu::update_GYRO_CONFIG(){
    changeBank(0x02);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_GYRO_SMPLRT_DIV = 0x00;
    uint8_t REG_GYRO_CONFIG_1 = 0x01;
    uint8_t REG_GYRO_CONFIG_2 = 0x02;

    uint8_t GYRO_SMPLRT_DIV;
    uint8_t GYRO_CONFIG_1;
    uint8_t GYRO_CONFIG_2;

    GYRO_SMPLRT_DIV = GYRO_CONFIG.GYRO_SMPLRT_DIV;

    GYRO_CONFIG_1 |= (GYRO_CONFIG.GYRO_DLPFCFG & 0x07) << 3;
    GYRO_CONFIG_1 |= (GYRO_CONFIG.GYRO_FS_SEL & 0x03) << 1;
    GYRO_CONFIG_1 |= GYRO_CONFIG.GYRO_FCHOICE;

    GYRO_CONFIG_2 |= GYRO_CONFIG.XGYRO_CTEN << 5;
    GYRO_CONFIG_2 |= GYRO_CONFIG.YGYRO_CTEN << 4;
    GYRO_CONFIG_2 |= GYRO_CONFIG.ZGYRO_CTEN << 3;
    GYRO_CONFIG_2 |= GYRO_CONFIG.GYRO_AVGCFG & 0x03;

    rw[0] = 0;
    rw[1] = 0;

    command[0] = REG_GYRO_SMPLRT_DIV;
    command[1] = GYRO_SMPLRT_DIV;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_GYRO_CONFIG_1;
    command[1] = GYRO_CONFIG_1;
    I2C.writeReadBus(addr, 2, command, rw);

```

```
    command[0] = REG_GYRO_CONFIG_2;
    command[1] = GYRO_CONFIG_2;
    I2C.writeReadBus(addr, 2, command, rw);
}

void imu::update_MAG_CONFIG(){
    uint8_t magAddr = 0x0C;
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_CNTL2 = 0x31;
    uint8_t REG_CNTL3 = 0x32;

    uint8_t CNTL2 = MAG_CONFIG.MODE & 0x1F;
    uint8_t CNTL3 = MAG_CONFIG.SRST;

    rw[0] = 0;
    rw[1] = 0;

    command[0] = REG_CNTL2;
    command[1] = CNTL2;
    I2C.writeReadBus(magAddr, 2, command, rw);

    command[0] = REG_CNTL3;
    command[1] = CNTL3;
    I2C.writeReadBus(magAddr, 2, command, rw);
}

void imu::setGyroOffset(uint16_t x, uint16_t y, uint16_t z){
    changeBank(0x02);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_XG_OFFS_USRH = 0x03;
    uint8_t REG_XG_OFFS_USRL = 0x04;
    uint8_t REG_YG_OFFS_USRH = 0x05;
    uint8_t REG_YG_OFFS_USRL = 0x06;
    uint8_t REG_ZG_OFFS_USRH = 0x07;
    uint8_t REG_ZG_OFFS_USRL = 0x08;

    uint8_t XG_OFFS_USRH = x << 8;
    uint8_t XG_OFFS_USRL = x & 0xFF;
    uint8_t YG_OFFS_USRH = y << 8;
    uint8_t YG_OFFS_USRL = y & 0xFF;
    uint8_t ZG_OFFS_USRH = z << 8;
    uint8_t ZG_OFFS_USRL = z & 0xFF;

    rw[0] = 0;
    rw[1] = 0;

    command[0] = REG_XG_OFFS_USRH;
    command[1] = XG_OFFS_USRH;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_XG_OFFS_USRL;
    command[1] = XG_OFFS_USRL;
```

```
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_YG_OFFS_USRH;
    command[1] = YG_OFFS_USRH;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_YG_OFFS_USRL;
    command[1] = YG_OFFS_USRL;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ZG_OFFS_USRH;
    command[1] = ZG_OFFS_USRH;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_ZG_OFFS_USRL;
    command[1] = ZG_OFFS_USRL;
    I2C.writeReadBus(addr, 2, command, rw);
}

void imu::setAccelOffset(uint16_t x, uint16_t y, uint16_t z){
    changeBank(0x02);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_XA_OFFS_USRH = 0x14;
    uint8_t REG_XA_OFFS_USRL = 0x15;
    uint8_t REG_YA_OFFS_USRH = 0x17;
    uint8_t REG_YA_OFFS_USRL = 0x18;
    uint8_t REG_ZA_OFFS_USRH = 0x1A;
    uint8_t REG_ZA_OFFS_USRL = 0x1B;

    uint8_t XA_OFFS_USRH = x << 8;
    uint8_t XA_OFFS_USRL = x & 0xFF;
    uint8_t YA_OFFS_USRH = y << 8;
    uint8_t YA_OFFS_USRL = y & 0xFF;
    uint8_t ZA_OFFS_USRH = z << 8;
    uint8_t ZA_OFFS_USRL = z & 0xFF;

    rw[0] = 0;
    rw[1] = 0;

    command[0] = REG_XA_OFFS_USRH;
    command[1] = XA_OFFS_USRH;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_XA_OFFS_USRL;
    command[1] = XA_OFFS_USRL;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_YA_OFFS_USRH;
    command[1] = YA_OFFS_USRH;
    I2C.writeReadBus(addr, 2, command, rw);

    command[0] = REG_YA_OFFS_USRL;
    command[1] = YA_OFFS_USRL;
    I2C.writeReadBus(addr, 2, command, rw);
```

```
277
278     command[0] = REG_ZA_OFFS_USRH;
279     command[1] = ZA_OFFS_USRH;
280     I2C.writeReadBus(addr, 2, command, rw);
281
282     command[0] = REG_ZA_OFFS_USRL;
283     command[1] = ZA_OFFS_USRL;
284     I2C.writeReadBus(addr, 2, command, rw);
285 }
286
287 void imu::readAccel(uint16_t *x, uint16_t *y, uint16_t *z){
288     changeBank(0x00);
289     uint8_t command[2];
290     bool rw[2];
291
292     uint8_t REG_ACCEL_XOUT_H = 0x2D;
293     uint8_t REG_ACCEL_XOUT_L = 0x2E;
294     uint8_t REG_ACCEL_YOUT_H = 0x2F;
295     uint8_t REG_ACCEL_YOUT_L = 0x30;
296     uint8_t REG_ACCEL_ZOUT_H = 0x31;
297     uint8_t REG_ACCEL_ZOUT_L = 0x32;
298
299     rw[0] = 0;
300     rw[1] = 1;
301
302     command[0] = REG_ACCEL_XOUT_H;
303     I2C.writeReadBus(addr, 2, command, rw);
304
305     *x = (uint16_t)command[1] << 8;
306
307     command[0] = REG_ACCEL_XOUT_L;
308     I2C.writeReadBus(addr, 2, command, rw);
309
310     *x = (uint16_t)command[1];
311
312     command[0] = REG_ACCEL_YOUT_H;
313     I2C.writeReadBus(addr, 2, command, rw);
314
315     *y = (uint16_t)command[1] << 8;
316
317     command[0] = REG_ACCEL_YOUT_L;
318     I2C.writeReadBus(addr, 2, command, rw);
319
320     *y = (uint16_t)command[1];
321
322     command[0] = REG_ACCEL_ZOUT_H;
323     I2C.writeReadBus(addr, 2, command, rw);
324
325     *z = (uint16_t)command[1] << 8;
326
327     command[0] = REG_ACCEL_ZOUT_L;
328     I2C.writeReadBus(addr, 2, command, rw);
329
330     *z = (uint16_t)command[1];
331 }
332
```

```
void imu::readGyro(uint16_t *x, uint16_t *y, uint16_t *z){
    changeBank(0x00);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_GYRO_XOUT_H = 0x33;
    uint8_t REG_GYRO_XOUT_L = 0x34;
    uint8_t REG_GYRO_YOUT_H = 0x35;
    uint8_t REG_GYRO_YOUT_L = 0x36;
    uint8_t REG_GYRO_ZOUT_H = 0x37;
    uint8_t REG_GYRO_ZOUT_L = 0x38;

    rw[0] = 0;
    rw[1] = 1;

    command[0] = REG_GYRO_XOUT_H;
    I2C.writeReadBus(addr, 2, command, rw);

    *x = (uint16_t)command[1] << 8;

    command[0] = REG_GYRO_XOUT_L;
    I2C.writeReadBus(addr, 2, command, rw);

    *x = (uint16_t)command[1];

    command[0] = REG_GYRO_YOUT_H;
    I2C.writeReadBus(addr, 2, command, rw);

    *y = (uint16_t)command[1] << 8;

    command[0] = REG_GYRO_YOUT_L;
    I2C.writeReadBus(addr, 2, command, rw);

    *y = (uint16_t)command[1];

    command[0] = REG_GYRO_ZOUT_H;
    I2C.writeReadBus(addr, 2, command, rw);

    *z = (uint16_t)command[1] << 8;

    command[0] = REG_GYRO_ZOUT_L;
    I2C.writeReadBus(addr, 2, command, rw);

    *z = (uint16_t)command[1];
}

void imu::readMag(uint16_t *x, uint16_t *y, uint16_t *z){
    uint8_t magAddr = 0x0C;
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_MAG_XOUT_H = 0x12;
    uint8_t REG_MAG_XOUT_L = 0x11;
    uint8_t REG_MAG_YOUT_H = 0x14;
    uint8_t REG_MAG_YOUT_L = 0x13;
    uint8_t REG_MAG_ZOUT_H = 0x16;
```

```
    uint8_t REG_MAG_ZOUT_L = 0x15;
    uint8_t REG_MAG_ST2 = 0x18;

    rw[0] = 0;
    rw[1] = 1;

    command[0] = REG_MAG_XOUT_H;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *x = (uint16_t)command[1] << 8;

    command[0] = REG_MAG_XOUT_L;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *x = (uint16_t)command[1];

    command[0] = REG_MAG_YOUT_H;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *y = (uint16_t)command[1] << 8;

    command[0] = REG_MAG_YOUT_L;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *y = (uint16_t)command[1];

    command[0] = REG_MAG_ZOUT_H;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *z = (uint16_t)command[1] << 8;

    command[0] = REG_MAG_ZOUT_L;
    I2C.writeReadBus(magAddr, 2, command, rw);

    *z = (uint16_t)command[1];

    command[0] = REG_MAG_ST2;
    I2C.writeReadBus(magAddr, 2, command, rw);
}

void imu::readTemp(uint16_t *t){
    changeBank(0x00);
    uint8_t command[2];
    bool rw[2];

    uint8_t REG_TEMP_OUT_H = 0x39;
    uint8_t REG_TEMP_OUT_L = 0x3A;

    rw[0] = 0;
    rw[1] = 1;

    command[0] = REG_TEMP_OUT_H;
    I2C.writeReadBus(addr, 2, command, rw);

    *t = (uint16_t)command[1] << 8;

```

```
445     command[0] = REG_TEMP_OUT_L;
446     I2C.writeReadBus(addr, 2, command, rw);
447 
448     *t = (uint16_t)command[1];
449 }
450 
451 void imu::changeBank(uint8_t bank){
452     uint8_t REG_ADDR = 0x7F;
453 
454     uint8_t command[2] = {REG_ADDR, bank << 4};  //Commands are the register address and then
    the value to assign to it
455     bool rw[2] = {0, 0};    //Both commands are a write operation
456 
457     I2C.writeReadBus(addr, 2, command, rw);
458 }
459 
460 uint8_t imu::whoami(){
461     changeBank(0x00);
462     uint8_t command[2];
463     bool rw[2];
464 
465     uint8_t REG_WHOAMI = 0x00;
466 
467     rw[0] = 0;
468     rw[1] = 1;
469 
470     command[0];
471     I2C.writeReadBus(addr, 2, command, rw);
472 
473     return command[1];
474 }
```

Leave 4:00

13/07/23

Time: 7.5h

Thursday, July 13, 2023 8:27 AM

Start 8:30AM

To write the internal ICM-20948 registers, the master transmits the start condition (S), followed by the I²C address and the write bit (0). At the 9th clock cycle (when the clock is high), the ICM-20948 acknowledges the transfer. Then the master puts the register address (RA) on the bus. After the ICM-20948 acknowledges the reception of the register address, the master puts the register data onto the bus. This is followed by the ACK signal, and data transfer may be concluded by the stop condition (P). To write multiple bytes after the last ACK signal, the master can continue outputting data rather than transmitting a stop signal. In this case, the ICM-20948 automatically increments the register address and loads the data to the appropriate register. The following figures show single and two-byte write sequences.

I messed this up — I was using the same setup as the read for the writes

Where is it
ADR+W → RegAdr → S → ADR+W → Data → P

instead I just need a simple write command

Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

To read the internal ICM-20948 registers, the master sends a start condition, followed by the I²C address and a write bit, and then the register address that is going to be read. Upon receiving the ACK signal from the ICM-20948, the master transmits a start signal followed by the slave address and read bit. As a result, the ICM-20948 sends an ACK signal and the data. The communication ends with a not acknowledge (NACK) signal and a stop bit from master. The NACK condition is defined such that the SDA line remains high at the 9th clock cycle. The following figures show single and two-byte read sequences.

Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

I changed all my write commands to be the normal write format instead of the read/write however it still doesn't work. I didn't think it would make a difference as it appears on all the config registers set to 0 I should still get readings. But it was definitely wrong and needed to be fixed. I may try and make a method that just reads back all the control registers to 1) check I am setting things as I expect during configuration and 2) check I can read other registers properly.

I made my script to read all the registers

I am not getting the output I expect — Not sure what is failing — Setting them or reading them

I might add print statements in my config to show the formatted byte before it is sent
It is possible I have an error in my bit-shifting and logic that is messing everything up.

I could also comment out the config and just read the default register values assuming they are 0

I added the print outs to check what the bytes being written to the registers are in the first place. I found that I had some errors where I never actually initialized byte to 0 before performing the or operations and that messed some things up. Also the power management 2 byte I had an | instead of the & which ended up disabling everything.

I changed some things and got it so that the bytes to be written to the registers match the ones that I read back after setting them.

And Now I am getting data from all of the sensors. However it does not appear that this data is very meaningful. All of the values are jumping around and don't appear to correlate to much.

Additionally I can see that the magnetometer does not change significantly when I hold a permanent magnet directly next to it.

There may be an error with way of saving the data. Cause I am noticing that it is never exceeding 255, even though it should be a 16bit number...

I may be able to see on the scope if the MSB is just 0.

I think I see data on the scope...

Which makes me think that something is wrong with the storage of the data.

I added a print after reading the H register of AX and there is data. Something is going wrong with the storage.

Leave 4:00

14/07/23

July 14, 2023 8:25 AM

Start 8:00

This morning I plan to go through my IMU library and document everything very well as it appears to be functional at this point.

I made it through documenting the cpp, h and example file — I think I will just need to make a read me file and I will be good to pach that away for a bit

I should also test the offset functions to make sure they work before deciding that it is ready

I created a readme file with some more documentation.

I added a line in the example script to test the offset of the Gyro and Accel.

Neither of them are working and I am not sure why. This is something that I will need to debug further but I do not have the brain power this afternoon. That is something that I will tackle on Monday

I started reading the UART section of the Datasheet. It seems very straight forward compared to the I2C. There is much less overhead with transactions -- it appears that data should just flow into and out of the buffers as it is available on the bus. Which should be less headaches compared to the I2C where there is the address matching in hardware before anything makes it to the visible registers.