17/07/23

7.5h

July 17, 2023 8:35 AM

Start 8:30

To-Do

- Fix the offset function of my IMU library
- Set up UART on the PIC
- Figure out the I2C on the PIC
    - Test the ADC and PWM scripts and make sure that my library to interface with them on the PI works
-

UART

34.2.1 UART Asynchronous Transmitter

The UART transmitter block diagram is shown in Figure 34-1. The heart of the transmitter is the serial Transmit Shift Register (TSR), which is not directly accessible by software. The TSR obtains its data from the transmit buffer, which is the UxTXB register.

34.2.1.1 Enabling the Transmitter

The UART transmitter is enabled for asynchronous operations by configuring the following control bits:

- TXEN = 1
- MODE = 0000 through 0011
- UxBRG = desired baud rate
- BRGS = desired baud rate multiplier
- RxyPPS = code for desired output pin
- ON = 1

All other UART control bits are assumed to be in their default state.

Setting the TXEN bit enables the transmitter circuitry of the UART. The MODE bits select the desired mode. Setting the ON bit enables the UART. When TXEN is set and the transmitter is not Idle, the TX pin is automatically configured as an output. When the transmitter is Idle, the TX pin drive is relinquished to the port TRIS control. If the TX pin is shared with an analog peripheral, the analog I/O function will be disabled by clearing the corresponding ANSEL bit.

Important: The UxTXIF Transmitter Interrupt flag is set when the TXEN Enable bit is set and the UxTXB register can accept data.

34.2.2 UART Asynchronous Receiver

The Asynchronous mode is typically used in RS-232 systems. The receiver block diagram is shown in Figure 34-2. The data is received on the RX pin and drives the data recovery block. The data recovery block is actually a high-speed shifter operating at 4 or 16 times the baud rate, whereas the serial Receive Shift Register (RSR) operates at the bit rate. When all bits of the character have been shifted in, they are immediately transferred to a two-character First-In First-Out (FIFO) memory. The FIFO buffering allows reception of two complete characters and the start of a third character before software must begin servicing the UART receiver. The FIFO registers and RSR are not directly accessible by software. Access to the received data is made via the UxRXB register.

34.2.2.1 Enabling the Receiver

The UART receiver is enabled for asynchronous operation by configuring the following control bits:

- RXEN = 1
- MODE = 0000 through 0011
- UxBRG = desired baud rate
- BRGS = desired baud rate multiplier
- RXPPS = code for desired input pin
- Input pin ANSEL bit = 0
- ON = 1

All other UART control bits are assumed to be in their default state.

I setup the UART and set it to transmit "Hello" in the main loop. There is data doing over the TX line and it is being received by Putty. But the data in putty is just a strange character. I have it set to UTF8 decoding which should read the ASCII correctly… I am going to check the data on the line to see if it represents what I expect.

'Hello' in ASCII
072 101 108 108 111

072	101	108	108	111
01001000	01100101	01101100	01101100	01101111

H = 00010010
e = 01010011
l = 00011011
l = 00011011
o = 11110110

I am noticing that the bit is reversed… I am wondering if that is causing the issue. It would be a weird problem but I can try to manually reverse the bytes before transmitting

https://stackoverflow.com/questions/2602823/in-c-c-whats-the-simplest-way-to-reverse-the-order-of-bits-in-a-byte

I am still getting the same weird character in PuTTY. But the data on the line seems right. There is a weird issue where there is another byte being transferred at some points. I am not sure what that is…

U+006F	o	6f	LATIN SMALL LETTER O

In UTF8, 6F is supposed to be a lowercase o.

The TX line is sending

I happened to accidentally take a picture of one of the bytes that was weird. I have no idea what that is, it doesn't even appear to have same BAUD rate..

IMU Offset

Previously I had tested the IMU offset function and was finding that it did not appear to be doing anything.

Today I changed the test script to output an average of 1000 readings (to get a relatively stable number) and then checked the resting values in X Y Z for the gyro

X: ~ -130
Y: ~ -55

Z: ~ -45

When I set the offset to max (0xFF), The values do change:
X: ~ 881
Y: ~ 965
Z: ~ 970

Meaning the offset is approximately:
881.0+130.0=1,011
965.0+55.0=1,020
970.0+45.0=1,015

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.

The datasheet does not specify whether the data is signed or unsigned. Considering I set the data to 0xFF, all bits are 1, meaning if there was a sign bit, it would be negative. Also, changing the offset to 0x7F (changing the MSb to 0) the value is still positive. This suggests that the offset can only be positive.

Given the datasheet, the resolution is 0.0305dps/lsb

0.0305*2^16=1,998.848

Meaning the offset is 0-2k dps.

I have the device in a range of +/- 250dps

Why is this not max offset?

Maybe I am overflowing it.

250/0.0305=8,196.7213

I am dumb -- this is 16bit, meaning the 0xff is actually not setting the most significant byte at all.
I should be sending 0xFFFF

I changed this and saw no difference… I check my code and I was bit shifting the wrong directions…

After fixing that issue, 0xFFFF and the offset appeared to do nothing…

I changed to 0x7FFFF and found that all offsets are maxing out the range, all readings are 32767

Since I am in +/-250 scale, and each offset value is an offset of 0.0305
250/0.0305=8,196.7213 = 0x2004
So 0x2000 should get me close to the max value with this range, but not be max all of the time

This gives a reading of ~32550 in each axis, which is basically what I expected.

So I am thinking that this is still only + offset

9.7 XA_OFFS_H

Name: XA_OFFS_H
Address: 20 (14h)
Type: USR1
Bank: 1
Serial IF: R/W
Reset Value: Trimmed on a per-part basis for optimal performance

BIT	NAME	FUNCTION
7:0	XA_OFFS[14:7]	Upper bits of the X accelerometer offset cancellation.

9.8 XA_OFFS_L

Name: XA_OFFS_L
Address: 21 (15h)
Type: USR1
Bank: 1
Serial IF: R/W
Reset Value: Trimmed on a per-part basis for optimal performance

BIT	NAME	FUNCTION
7:1	XA_OFFS[6:0]	Lower bits of the X accelerometer offset cancellation.
0	-	Reserved.

The accel offset is only 15bit and is not calibrated to a g value. I am wondering if this is again an unsigned offset and is just subracted from the reading. Cause the MSb is no longer needed if unsigned instead of signed

The base values for the accelerometer are:
X: 620
Y: -590
Z: 16765

When I change the offset to 0xFFFF (all 1's) the values are:
X: 992
Y: 835
Z: 3174

992-620=372
835+590=1,425
3174-16765=-13591

These offsets are completely inconsistent even though all were set to the same value….

I shifted the value of the offsets by 1 so that the LSb is not used. Unsurpsisingly this had no effect on the output at it was already being ignored and I set everything to 1's

Using an offset of 0x00FF (which is 255), the values are:
X: 7950
Y: 6350
Z: 25380

7950-620=7330
6350+590=6940
25380-16765=8615

This offset is much more consistent, but it is not clear how this is related to the set value…

Using 0x000F, I get:
X: 7959
Y: 6350
Z: 25380

These are the same as using the offset of 0x00FF. Even though this one is only a value of 15 as opposed to 255….

If I use 0x0FFF, I get approximately the same results as I do with 0xFFFF. This is strange behaviour.

The datasheet really does not specify what the value of the Accel offset correlates to, but it does not make sense the way it is currently behaving. I guess one thing to note is that there is a difference when the H register is set compared to when it is empty

18/07/23

Time: 7.5h

Tuesday, July 18, 2023 8:13 AM

Start 8:00

Data bits

The data bits are the user data or "useful" bits and come immediately after the start bit. There can be 5 to 9 user data bits, although 7 or 8 bits is most common. These data bits are usually transmitted with the least significant bit first.

Example:
If we want to send the capital letter "S" in 7-bit ASCII, the bit sequence is 1 0 1 0 0 1 1. We first reverse the order of the bits to put them in least significant bit order, that is 1 1 0 0 1 0 1, before sending them out. After the last data bit is sent, the stop bit is used to end the frame and the line returns to the idle state.

- 7-bit ASCII 'S' (0x52) = 1 0 1 0 0 1 1
- LSB order = 1 1 0 0 1 0 1

This website outlines the problem I was seeing

It looks like I was right to reverse the byte first

Another thing that I could do is just check the BAUD rate on the scope to make sure it is ~9600. Cause this could also explain why I am not able to read things properly. That would be a nice simple problem if I just messed up my math and have had the wrong BAUD rate the whole time.

At 9600bits/s, I should be getting 1.00000000/9600.000=0.00010417s/bit

Meaning if I check the width of one pulse on the scope, it should be ~104us

Looking at the image I took yesterday, one bit appears to be much closer to ~500us, which is definitely not in the +/- 10% spec

With that period, I am looking at
1/500e-6=2,000.0 for the BAUD rate. Not even close

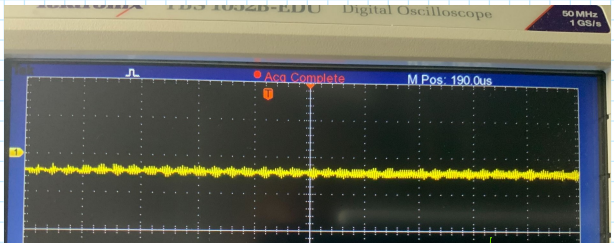
Bits 15:0 – BRG[15:0] Baud Rate Generator Value
The UART Baud Rate equals [Fosc*(1+(BRGS*3)]/[(16*(BRG-1))]

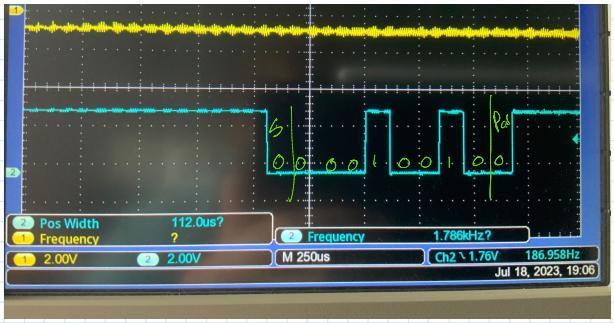
I calculated 27 as the BRG to get 9600 BAUD

BAUD	9.6154e+03
brg	27
brgs	0
fosc	4000000

Matlab is telling me that it should be approximately 9600…

Unless Fosc is not actually set to 4MHz. If it is Fosc/4 or if Fosc is actually set to 1MHz, I would get ~2400 BAUD which is much closer to what I am seeing on the scope.





I tried to write "H"
I got a consistent output of ";" in putty (semi colon)
The data on the line is 00010010
H is supposed to be 01001000 (reverse of what was seen)
; is supposed to be 0x3B → 00111011, which is not at all close

msg	unsigned char[6]	0x50D	"Hello\\0"	
msg[0]	unsigned char	0x50D	'H'; 0x48	01001000
msg[1]	unsigned char	0x50E	'e'; 0x65	01100101
msg[2]	unsigned char	0x50F	'l'; 0x6c	01101100
msg[3]	unsigned char	0x510	'l'; 0x6c	01101100
msg[4]	unsigned char	0x511	'o'; 0x6f	01101111
msg[5]	unsigned char	0x512	NUL; 0x0	00000000

This is my message variable, and I am transmitting each character in a for loop

The loop goes msg[0], msg[1], msg[2], msg[3],msg[4]
Which is clearly H e l l o
H

Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0x2E	00101110
U1TXB	SFR	0x2A3	0x48	01001000
data	unsigned char	0x50B	'H'; 0x48	01001000

e

Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0x1E	00011110
U1TXB	SFR	0x2A3	0x65	01100101
data	unsigned char	0x50B	'e'; 0x65	01100101

l

Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0x9E	10011110
U1TXB	SFR	0x2A3	0x65	01100101
data	unsigned char	0x50B	'l'; 0x6c	01101100

l

Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0x9E	10011110
U1TXB	SFR	0x2A3	0x65	01100101
data	unsigned char	0x50B	'l'; 0x6c	01101100

o

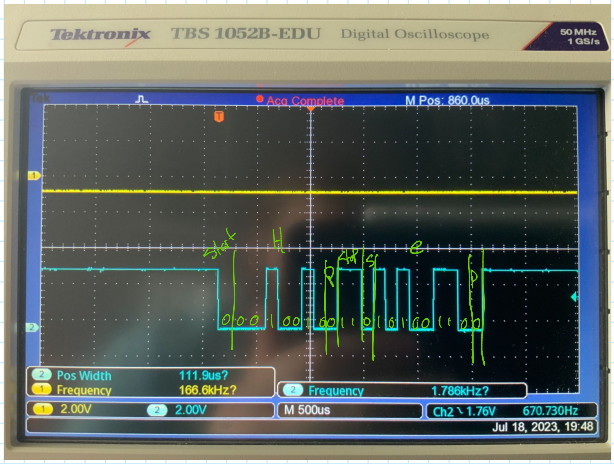
Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0x9E	10011110
U1TXB	SFR	0x2A3	0x65	01100101
data	unsigned char	0x50B	'o'; 0x6f	01101111

H

Name	Type	Address	Value	Binary
<Enter new watch>				
U1FIFO	SFR	0x2B0	0xAE	10101110
U1TXB	SFR	0x2A3	0x48	01001000
data	unsigned char	0x50B	'H'; 0x48	01001000

When I step through each time, I can see that the bits are being written. One thing I noticed is that the TXB register does not change after the second byte is loaded in

I know on the scope, it appears that only two bytes are being sent over the bus even though it is instructed to send 5. It apprears that something is going wrong and the full message is not being sent.



Only 'He' is being transmitted

The UFIFO register also changed at the beginning and then stayed constant.

34.20.7 UxFIFO

Name: UxFIFO
Address: 0x2B0,0x2C3,0x2D6

UART FIFO Status Register

Bit	7	6	5	4	3	2	1	0
	TXWRE	STPMD	TXBE	TXBF	RXIDL	XON	RXBE	RXBF
Access	R/W/S	R/W	R/W/S/C	R/S/C	R/S/C	S/C	R/W/S/C	R/S/C
Reset	0	0	1	0	1	1	1	0

Bit 7 – TXWRE Transmit Write Error Status (must be cleared by software)

Value	Condition	Description
1	MODE = LIN Host	UxP1L was written when a host process was active
1	MODE = LIN Client	UxTXB was written when UxP2 = 0 or more than UxP2 bytes have been written to UxTXB since last Break
1	MODE = Address detect	UxP1L was written before the previous data in UxP1L was transferred to TX shifter
1	MODE = All	A new byte was written to UxTXB when the output FIFO was full
0	MODE = All	No error

Bit 6 – STPMD Stop Bit Detection Mode

Value	Condition	Description
1	STP = 11	Assert UxRXIF at end of first Stop bit
1	STP ≠ 11	Assert UxRXIF at end of last Stop bit
0	STP = xx	Assert UxRXIF in middle of first Stop bit

Bit 5 – TXBE Transmit Buffer Empty Status

Value	Description
1	Transmit buffer is empty. Setting this bit will clear the transmit buffer and output shift register.
0	Transmit buffer is not empty. Software cannot clear this bit.

Bit 4 – TXBF Transmit Buffer Full Status

Value	Description
1	Transmit buffer is full
0	Transmit buffer is not full

Bit 3 – RXIDL Receive Pin Idle Status

Value	Description
1	Receive pin is in Idle state
0	UART is receiving Start, Stop, Data, Auto-baud, or Break

Bit 2 – XON Software Flow Control Transmit Enable Status

Value	Description
1	Transmitter is enabled
0	Transmitter is disabled

Bit 1 – RXBE Receive Buffer Empty Status

Value	Description
1	Receive buffer is empty. Setting this bit will clear the RX buffer(1).
0	Receive buffer is not empty. Software cannot clear this bit.

Bit 0 – RXBF Receive Buffer Full Status

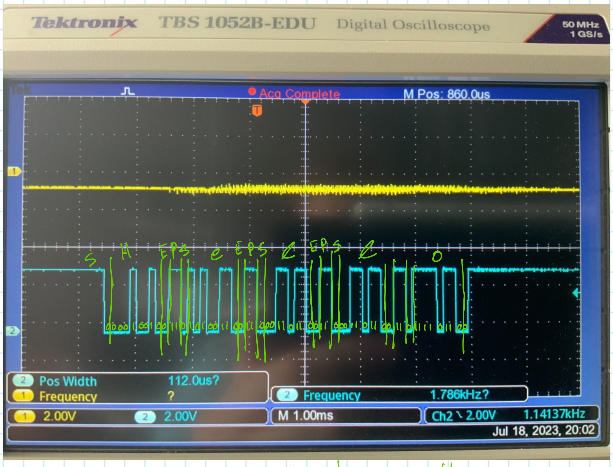
Value	Description
1	Receive buffer is full

Looking at the data, I found that the Error bit was being set, and the datasheet says that it is set when something is written to the FIFO while it is full. Clearly there was not enough time in my timeout for the register to output the data completely.

I increased the timeout to 1000 and found that every character was not making it into the buffer.

This is leaving a long delay between transmissions... approximately 15ms is spent between characters which does not

seem reasonable. Maybe instead of waiting for the buffer to be empty I can just wait until it is not full. Not sure if that will make a difference but I can try.



It appears to be writing hello properly

I programmed the Arduino Leonardo to send a H over the

19/07/23

Time: 7.5

Wednesday, July 19, 2023 8:04 AM

Start 8:00AM

To-do

Today my to do list includes

- Testing UART with the converter to see if it is working/setup properly on the PIC
- Look into the accelerometer offset on the IMU
- Go back to debugging the I2C

UART

I am now reading the RS232 voltage spec again and I am realizing I was wrong yesterday - It does work for 5-15V, but the other signal needs to be -5 - -15V. It is not 0-5V.

Nick emailed this morning and said he left the UART module on my desk so hopefully that has the + and - voltages on it.

From looking at the Arduino UART output yesterday, I am hopeful that the UART on the PIC is set up correctly as the output on the scope appears to be the exact same.

I2C

I went back to testing the I2C, I thought I was making progress cause I noticed that the PI was saying "failed to read" and not "failed to contact" so I thought I was getting an ACK from the PIC, but I am not.

Also a couple days ago I thought that I had made a mistake configuring the pins, I changed it and realized I was right the first time.

```
//Set pins to SDA and SCL output
RC0PPS = 0x22;  //Set pin C0 to SDA
RC1PPS = 0x21;  //Set pin C1 to SCL

//Set pins to SDA and SCL input
I2C1SDAPPS = 0b010000;  //Set C0 to SDA
I2C1SCLPPS = 0b010001; //Set C1 to SCL
```

This should be right.

Name	Type	Address	Value	Binary
<Enter new watch>				
I2C1ADB0	SFR	0x28E	0x00	00000000
I2C1ADB1	SFR	0x28F	0x00	00000000
I2C1ADR0	SFR	0x290	0xFF	11111111
I2C1ADR1	SFR	0x291	0xFE	11111110
I2C1ADR2	SFR	0x292	0xFF	11111111
I2C1ADR3	SFR	0x293	0xFE	11111110
I2C1BAUD	SFR	0x29D	0x00	00000000
I2C1BTO	SFR	0x29C	0xA3	10100011
I2C1BTOC	SFR	0x29F	0x06	00000110
I2C1CLK	SFR	0x29E	0x00	00000000
I2C1CNTH	SFR	0x28D	0x00	00000000
I2C1CNTL	SFR	0x28C	0x00	00000000
I2C1CON0	SFR	0x294	0x80	10000000
I2C1CON1	SFR	0x295	0x00	00000000
I2C1CON2	SFR	0x296	0x40	01000000
I2C1ERR	SFR	0x297	0x00	00000000
I2C1PIE	SFR	0x29B	0xDE	11011110
I2C1PIR	SFR	0x29A	0x04	00000100
I2C1RXB	SFR	0x28A	0x00	00000000
I2C1SCLPPS	SFR	0x271	0x11	00010001
I2C1SDAPPS	SFR	0x270	0x10	00010000
I2C1STAT0	SFR	0x298	0x80	10000000
I2C1STAT1	SFR	0x299	0x20	00100000
I2C1TXB	SFR	0x28B	0x00	00000000

This is a complete list of the I2C registers and their state in the idle state before any start or stop conditions are generated. This is just what the registers read when ever they are resting.

I am going to go through every bit and check the datasheet to make sure it makes sense what the values are.

REGISTER NAME	BITS 7	6	5	4	3	2	1	0	DESCRIPTION 7	6	5	4	3	2	1	0	LEGEND
I2C1ADB0	0	0	0	0	0	0	0	0	ADB - Received matching address buffer								ASSUMED CORRECT
I2C1ADB1	0	0	0	0	0	0	0	0	ADB - Unused in 7bit w/o mask								UNSURE
I2C1ADR0	1	1	1	1	1	1	1	1	ADR - 7 bit address, bit 0 is treated as X in this mode								ASSUMED INCORRECT
I2C1ADR1	1	1	1	1	1	1	1	0	ADR - 7 bit address 1								ASSUME IRRELEVANT
I2C1ADR2	1	1	1	1	1	1	1	1	ADR - 7 bit address 2, bit 0 is treated as X in this mode								UNUSED
I2C1ADR3	1	1	1	1	1	1	1	0	ADR - 7 bit address 2								
I2C1BAUD	0	0	0	0	0	0	0	0	BAUD - Set the I2C clock frequecy								
I2C1BTO	1	0	1	0	0	0	1	1	TOREC - Setting this will cause the I2C module to restart on timout event	TOBY32 - Time-Out prescaler	TOTIME -Set the timeout time						
I2C1BTOC	0	0	0	0	0	1	1	0						BTOC - Bus timeout clock source			
I2C1CLK	0	0	0	0	0	0	0	0					CLK - Select clock source for the I2C				
I2C1CNTH	0	0	0	0	0	0	0	0	CNT - CNT register which is decemented after each byte								
I2C1CNTL	0	0	0	0	0	0	0	0									
I2C1CON0	1	0	0	0	0	0	0	0	EN - This bit enables the I2C module	RSEN - Restart enable - only used in MODE=1XX	S - Host start - Only used in MODE=1XX	CSTR - Set to stretch the clock	MDR - Host Data Request - setting causes host to hold clock low	MODE - Sets the mode of the device - 000 is 7bit client w/o masking			
I2C1CON1	0	0	0	0	0	0	0	0	ACKCNT - When CNT reaches 0, send NACK (1) or ACK(0)	ACKDT - Send NACK when matching address or CNT != 0	ACKSTAT (transmission only) gets by hardware when ACK was not received for last trans	ACKT - Indicated the bus is in an ACK sequence - set by hardware on 8th falling edge and cleared on 9th rising	P - Host stop - initiate stop	RXO - Receive overflow status	TXU - Transmit Underflow status	CSD - Clock stretch disble	
I2C1CON2	0	1	0	0	0	0	0	0	ACNT - Auto load CNT with byte after address	GCEN - General Call Address Enable	FME - Fast Mode Enable	disable - received address is loaded into RXB	SDAHT - SDA Hold Time - Sets minimum hold time on SDA after falling edge of SCL		BFRET - Bus Free time selector		
I2C1ERR	0	0	0	0	0	0	0	0		BTOIF - Bus timeout error	BCLIF - Bus collision error	NACKIF - NACK detected on the bus (only when host or client is active)		BTOIE - Bus timeout interrupt enable	BCLIE - Collision interrupt enable	NACKIE - NACK interrupt enable	
I2C1PIE	1	1	0	1	1	1	1	1	CNTIE - Enable CNT interupt	ACKTIE - enable ACK status time interrupt		WRIE - Enable data write interrupt	ADRIE - Enable matching address interupt	PCIE - Enable stop interrupt	RSCIE - Enable restart interrupt	SCIE -Enable start interrupt	
I2C1PIR	0	0	0	0	0	1	0	0	CNTIF - Set when CNT reaches 0	ACKTIF - Set on the 9th falling edge of every byte when addressed as a client		WRIF - Data write interrupt flag - data byte detected	ADRIF - Address interrupt flag - set when matching address is received	PCIF - Stop condition interrupt	RSCIF - Restart condition detected	SCIF - Start condition flag	
I2C1RXB	0	0	0	0	0	0	0	0	RXB - Receive buffer - read only								
I2C1STAT0	1	0	0	0	0	0	0	0	BFRE - Bus free status	SMA - Client mode is active - set on the 8th falling edge after a matching address	MMA - Host Mode active	R - Read information - last matching address had a read condition	D - Indicated that the last byte received was data				
I2C1STAT1	0	0	1	0	0	0	0	0	TXWE - Transmit write error		TXBE - Transmit buffer empty		RXRE - Receive read error status	CLRBF - Set to clear the TX and RX buffers		RXBF - RX buffer full	
I2C1TXB	0	0	0	0	0	0	0	0	TXB - Transmit buffer - write only								

I found that:

- BTOC was set to a reserved value, so there is not clock source selected -- I will change to 011 for LFINTOSC
- CSD was not set -- setting this should disable clock stretches that will need to be cleared by software, should make things simpler for debugging
- PCIF is set -- I have known about this issue, not sure how to clear it
- I2C clock frequency was not set. I assumed that this is not needed in Client mode as the host generates the clock and I2C is synchronous, could set to 100kHz to be safe
- Unsure of the timeout setup and whether that will really matter in client mode
- CNT register says that it should be set when client and host mode are inactive -- meaning it should be set outside of the ISR, I was doing it inside -- I will just set the low register to max in the main loop for not to try and get something going
- FME - not sure if fast mode makes a difference when in client mode cause everything should be sync'd to the host clock
- SDAHT - Sets the min hold time of SDA after SCL -- I had this set at the maximum which was quite long, maybe turning that down could help -- I will change to 0b00 instead which is ~30us I believe -- Not sure if this is relevant for client

IT FINALLY DOES SOMETHING

I am able to get it to detect it is being called, ACK, and send back a byte of data

This is the current functional script -- much of the logic has been commented out as I was simplifying for debugging

main.c

PIC18\I2C.X\main.c

```
/*
 * File:   main.c
 * Author: damie
 *
 * Created on June 20, 2023, 1:00 PM
 */


#include <xc.h>

#define _XTAL_FREQ 4000000

void configI2C(){
    I2C1CON0bits.EN = 0;    //Disable I2c
    //SCL1 -- RC1
    //SDA1 -- RC0

    //Set all pins to digital not analog
    ANSELCbits.ANSELC0 = 0;
    ANSELCbits.ANSELC1 = 0;

    /*Configure pins*/
    //Set pins
    TRISCbits.TRISC0 = 0;
    TRISCbits.TRISC1 = 0;
    //Set pins to open drain
    ODCONCbits.ODCC0 = 1;
    ODCONCbits.ODCC1 = 1;

    //Set pins to SDA and SCL output
    RC0PPS = 0x22;  //Set pin C0 to SDA
    RC1PPS = 0x21;  //Set pin C1 to SCL

    //Set pins to SDA and SCL input
    I2C1SDAPPS = 0b010000;  //Set C0 to SDA
    I2C1SCLPPS = 0b010001; //Set C1 to SCL

    /*Config I2C*/
    I2C1CON0bits.MODE = 0b000; //Set to client 7bit w/o masking
    I2C1CON0bits.CSTR = 0;      //Enable clocking (not stretching clock)

    I2C1CON1bits.CSD = 1;       //Disable clock stretching

    I2C1CON2bits.ACNT = 0;      //First transmission after address will be loaded to the byte
count register
    I2C1CON2bits.GCEN = 1;      //Enable general address call -- will respond to address 0x00
    I2C1CON2bits.ABD = 0;
    I2C1CON2bits.SDAHT = 0b00;

    I2C1PIEbits.CNT1IE = 1;     //Enable interrupt when byte count reaches 0
    I2C1PIEbits.ACKTIE = 1;     //Enable interrupt for acknowledge clock stretching
    I2C1PIEbits.WRIE = 1;       //Enable interrupt for receiving data to process and
determine ACK or NACK
```

```
    I2C1PIEbits.ADRIE = 1;      //Enable interrupt for processing address
    I2C1PIEbits.PCIE = 0;       //Disable interrupt when stop condition detected
    I2C1PIEbits.RSCIE = 0;      //Disable interrupt when restart condition detected
    I2C1PIEbits.SCIE = 0;       //Disable interrupt when start condition detected

    //Clear stop, start and restart flags
    I2C1PIRbits.PCIF = 0;
    I2C1PIRbits.SCIF = 0;
    I2C1PIRbits.RSCIF = 0;

    I2C1BTOC = 0x03;            //Set clock source to LFINTOSC (32kHz)
    I2C1BTObits.TOREC = 1;      //If timeout, reset module
    I2C1BTObits.TOBY32 = 0;     //Enable 32 prescaling
    I2C1BTObits.TOTIME = 0x23;  //Set timeout time to 35ms (32kHz / 32 prescale * 35 = 35ms)

    I2C1ADR0 = 0xFF;   //Set 7bit address
    I2C1ADR1 = 0xFF;   //Set 7bit address

    I2C1CLK = 0b0011;   //Set the clock to MFINTOSC (500kHz)
    I2C1BAUD = 0x04;    //Freq = I2C1CLK/(BAUD+1) = 500kHz/(4+1) = 100kHz

    /*Config I2C interrupts*/
    INTCON0bits.GIE = 1;    //Enable interrupts
    INTCON0bits.IPEN = 1;   //Enable Interrupt Priorities


    IPR7bits.I2C1IP = 1;    //Set I2C interrupt to high prio
    PIR7bits.I2C1IF = 0;    //Clear interrupt bit
    PIE7bits.I2C1IE = 1;    //Enable I2C interrupt

    I2C1PIRbits.PCIF = 0;   //Clear stop flag

    I2C1CON0bits.EN = 1;    //Enable I2c

    __delay_ms(10);

    I2C1PIEbits.PCIE = 0;       //Enable interrupt when stop condition detected
    I2C1PIEbits.RSCIE = 1;      //Enable interrupt when restart condition detected
    I2C1PIEbits.SCIE = 1;       //Enable interrupt when start condition detected
}

void i2cStart(){
    /*
     *Address is received from the host
     * Address is compared to the address saved on this client
     * If there is a match, the SMA bit is set by hardware to activate client mode
     * Data bit D is cleared by hardware to indicate last byte was address
     */
    I2C1TXB = 0x0F;
    __delay_ms(10);
    NOP();
//
//    while(I2C1PIEbits.PCIE == 1 && I2C1PIRbits.PCIF == 0){
//        if(I2C1ADB0 == I2C1ADR0){
//        I2C1CON1bits.ACKDT = 0; //Set ACK
//        I2C1TXB = 0xAA;
```

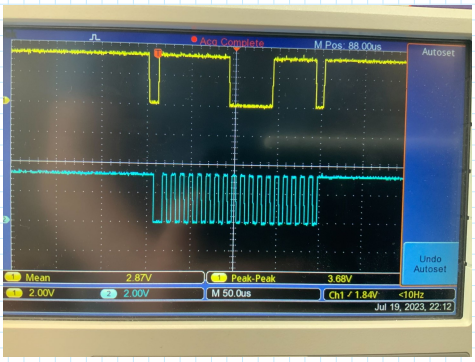
```
//        I2C1CON0bits.CSTR = 0;  //Release clock
//        }
//
//        /*
//         *ACK will be sent over SDA to acknowledge a matching address was received
//         *On 9th falling edge, ACKTIF will be set and clock will be held again.
//         *Reset CSTR and ACKTIF to release clock and resume
//         */
//
//
//        if(I2C1PIRbits.ACKTIF == 1 && I2C1CON0bits.CSTR == 1){
//            I2C1PIRbits.ACKTIF = 0;
//            I2C1CON0bits.CSTR = 0;
//        }
//
//        /*If ACK was generated, Host will now send byte to client
//         *When done, I2C1RXIF, I2C1IF, WRIF, D, and RXBF will be set
//         *Data is transfered into I2C1RXB
//         * Clock is stretched to allow ACK bit to be determined
//         * Because ACNT is enabled, first byte should be sent to the
//         */
//
//        if(I2C1CON0bits.CSTR == 1){
//            I2C1CON1bits.ACKDT = 0; //Set ACK
//            I2C1CON0bits.CSTR = 0; //Release clock
//        }
//
//        /*If the last matching address received ended with a READ command*/
//        if(I2C1STAT0bits.R == 1){
//            if(I2C1STAT1bits.TXBE == 0){  //If there is something in the buffer
//                I2C1STAT1bits.CLRBF = 1;    //Clear the buffer
//            }
//            //Send all data for ADC channels
//            for(int i = 0; i < 12; i++){
//                while(I2C1STAT1bits.TXBE == 0); //Wait for buffer to be emptied
//                I2C1TXB = 0x00;     //Load LsB of adc to the buffer
//                //while(I2C1CON1bits.ACKSTAT == 1);   //Wait for host to acknowledge
//
//                while(I2C1STAT1bits.TXBE == 0); //Wait for buffer to be emptied
//                I2C1TXB = 0x00;     //Load MsB to of adc the buffer
//                //while(I2C1CON1bits.ACKSTAT == 1);   //Wait for host to acknowledge
//            }
//        }
//
//        /*If the last matching address received ended with a WRITE command*/
//        if(I2C1STAT0bits.R == 0){
//            while(I2C1STAT1bits.RXBF == 0); //Wait for buffer to fill
//            I2C1CON1bits.ACKDT = 1;     //ACK
//            uint8_t addrH = I2C1RXB;    //Read receiver buffer
//            I2C1STAT1bits.CLRBF = 1;    //Clear buffer
//
//            while(I2C1STAT1bits.RXBF == 0); //Wait for buffer to fill
//            I2C1CON1bits.ACKDT = 1;     //ACK
//            uint8_t addrL = I2C1RXB;    //Read receiver buffer
//            I2C1STAT1bits.CLRBF = 1;    //Clear buffer
//
```

```
//            while(I2C1STAT1bits.RXBF == 0); //Wait for buffer to fill
//            I2C1CON1bits.ACKDT = 1;     //ACK
//            uint8_t duty_cycle = I2C1RXB;    //Read receiver buffer
//            I2C1STAT1bits.CLRBF = 1;    //Clear buffer
//
//            //PWMchangeDuty(addrH << 8 | addrL, duty_cycle);
//
//        }
//
//        if(I2C1PIRbits.CNTIF == 1){
//            //End of communication
//            I2C1CON1bits.ACKCNT = 1;    //Mark end of communication
//        }
//    }
}

void __interrupt(irq(I2C1)) ISR(void){
    if(I2C1PIRbits.SCIF == 1){
        i2cStart();
    }
    return;
}


void loop(){
    NOP();
    I2C1CNTL = 0x08;
}

void main(void) {
    configI2C();
    while(1){
        loop();
    }
    return;
}
```




```
spudnik1@spudnik1:~/Desktop/i2c$ sudo ./a.out
Data read:
15
SUCCESS: Ending script
spudnik1@spudnik1:~/Desktop/i2c$
```

Now I need to figure out exactly how I want it to act

First I want to check and see if it recognized that it was given a read condition.

```
    if(I2C1STAT0bits.R == 1){
        I2C1TXB = 0x0F;
    }
    //I2C1TXB = 0xAA;
    __delay_ms(10);
    NOP();
//
//    while(I2C1PIEbits.PCIE == 1 && I2C1PIRbits.PCIF == 0){
//        if(I2C1ADB0 == I2C1ADR0){
//        I2C1CON1bits.ACKDT = 0; //Set ACK
```

i2cStart

Output | Watches | Variables | Call Stack

Name	Type	Address	Value	Binary
<Enter new watch>				
I2C1ADB0	SFR	0x28E	0xFF	11111111
I2C1CNTL	SFR	0x28C	0x07	00000111
I2C1CON0	SFR	0x294	0x80	10000000
I2C1CON1	SFR	0x295	0x23	00100011
I2C1CON2	SFR	0x296	0x40	01000000
I2C1ERR	SFR	0x297	0x10	00010000
I2C1PIE	SFR	0x29B	0xDB	11011011
I2C1PIR	SFR	0x29A	0x4D	01001101
I2C1RXB	SFR	0x28A	0x00	00000000
I2C1STAT0	SFR	0x298	0x98	10011000
I2C1STAT1	SFR	0x299	0x20	00100000
I2C1TXB	SFR	0x28B	0x00	00000000

This is not writing the data properly, even though the R flag is being set. I am thinking that the if statement is too fast and the R flag is not being set until later.

I might try and add a small delay and see if I can get it to work. I fill likely need to use the clock stretching in production.

```
    for(int i = 0; i < 10000; i++){
        if(I2C1STAT0bits.R == 1){
            int temp = i;
            NOP();
        }
        __delay_us(1);
    }
    NOP();
```

This is stopping at i=1 suggesting there is effectively no time delay for the R bit to be set.

I can not figure out the timing on this, I am thinking it will be easier to just try to go into the clock stretching right away.

Bit 4 – CSTR Client Clock Stretching[3]

Value	Condition	Description
1		Clock is held low (clock stretching)
0		Enable clocking, SCL control is released
	SMA = 1 and RXBF = 1[6]:	Set by hardware on 7th falling SCL edge User must read I2CxRXB and clear CSTR to release SCL
	SMA = 1 and TXBE = 1 and I2CxCNT != 0:	Set by hardware on 8th falling SCL edge User must write to I2CxTXB and clear CSTR to release SCL
	when ADRIE = 1[4]:	Set by hardware on 8th falling edge of matching received address User must clear CSTR to release SCL
	SMA = 1 and WRIE = 1:	Set by hardware on 8th falling SCL edge of received data byte User must clear CSTR to release SCL
	SMA = 1 and ACKTIE = 1:	Set by hardware on 9th falling SCL edge User must clear CSTR to release SCL

I don't want all these possible clock stretching conditions, I really only want to have one on the ACK bit to give me time to set up the next transmission. The first 2 (RXBF and TXBE) will be set a lot when I really don't need them to be. The TXBE is the worst,

```
    I2C1PIRbits.SCIF = 0;

    //I2C1TXB = 0x00;
    //while(I2C1STAT0bits.R == 0){NOP();};
    I2C1TXB = 0x0F;

    __delay_ms(10);
    NOP();
```

I checked the timing with this. When I have the while commented, I read 15. Otherwise I read 255.

This means that it takes longer to determine whether the device got a read/write command than has to send the data.

I have found that the Client mode gets activated before the R bit gets set

```
    while(I2C1STAT0bits.SMA == 0){};
    if(I2C1STAT0bits.R == 1){
        NOP();
    }
    I2C1TXB = 0x0F;

    __delay_ms(10);
    NOP();
//
```

This means I can stretch the clock until SMA ==1, then determine if it is read/write.

```
    //I2C1TXB = 0x00;
    while(I2C1STAT0bits.SMA == 0){};
    if(I2C1STAT0bits.R == 1){
        I2C1TXB = 0x0F;
    } else {
        int temp = I2C1RXB;
    }
    I2C1CON0bits.CSTR = 0;

    __delay_ms(10);
    NOP();
```

This worked -- I was able to send 0F over the bus.

```
    while(I2C1STAT0bits.SMA == 0){};
    if(I2C1STAT0bits.R == 1){
        for(int i = 1; i < 13; i ++){
            I2C1TXB = i*5;
            __delay_us(50);

            I2C1CON0bits.CSTR = 0;
        }
    }
```

This worked for sending 5-60 in multiples of 5 over the bus. I was able to read it on the PI.

Now I need to get the registers reset to it can be called again after.

Name	Type	Address	Value	Binary
<Enter new watch>				
I2C1ADB0	SFR	0x28E	0x00	00000000
I2C1CNTL	SFR	0x28C	0x08	00001000
I2C1CON0	SFR	0x294	0x80	10000000
I2C1CON1	SFR	0x295	0x00	00000000
I2C1CON2	SFR	0x296	0x40	01000000
I2C1ERR	SFR	0x297	0x00	00000000
I2C1PIE	SFR	0x29B	0xD3	11010011
I2C1PIR	SFR	0x29A	0x04	00000100
I2C1RXB	SFR	0x28A	0x00	00000000
I2C1STAT0	SFR	0x298	0x80	10000000
I2C1STAT1	SFR	0x299	0x20	00100000
I2C1TXB	SFR	0x28B	0x00	00000000

This is before being called

<Enter new watch>				
I2C1ADB0	SFR	0x28E	0xFF	11111111
I2C1CNTL	SFR	0x28C	0x00	00000000
I2C1CON0	SFR	0x294	0x80	10000000
I2C1CON1	SFR	0x295	0x00	00000000
I2C1CON2	SFR	0x296	0x40	01000000
I2C1ERR	SFR	0x297	0x00	00000000
I2C1PIE	SFR	0x29B	0xD3	11010011
I2C1PIR	SFR	0x29A	0xCC	11001100
I2C1RXB	SFR	0x28A	0x00	00000000
I2C1STAT0	SFR	0x298	0x80	10000000
I2C1STAT1	SFR	0x299	0x20	00100000
I2C1TXB	SFR	0x28B	0x00	00000000

This is after (I also turned the module off and back on). I need to clear :

- ADB0
- PIR

I think that is all that matters...

Turns out ADB0 is read only in client mode... hopefully it will be reset on a new start condition.

```
    while(I2C1STAT0bits.SMA == 0){};
    if(I2C1STAT0bits.R == 1){
        for(int i = 1; i < 13; i ++){
            I2C1TXB = i*5;
            __delay_us(50);

            I2C1CON0bits.CSTR = 0;
        }
    }

    __delay_ms(1);
    /*RESET FOR NEXT COMMUNICATION*/
    I2C1CON0bits.EN = 0;
    I2C1ADB0 = 0x00;
    I2C1PIR = 0x00;
    I2C1CON0bits.EN = 1;
```

This worked to be able to read consecutively.

The delay is unideal. Ideally I would just check to make sure that the bus is ready for more data.

Bit 5 – ACKSTAT Acknowledge Status (*Transmission only*)

Value	Description
1	Acknowledge was not received for the most recent transaction
0	Acknowledge was received for the most recent transaction

Maybe I can use this -- wait until an ACK is generated

That did not work.

What if I just wait until the hardware strectches the clock again?

```
while(I2C1STAT0bits.SMA == 0){};
if(I2C1STAT0bits.R == 1){
    for(int i = 1; i < 13; i ++){
        while(I2C1CON0bits.CSTR == 0);
        I2C1TXB = i;
        //__delay_us(10);
        I2C1CON0bits.CSTR = 0;
    }
}
```

This is not working. Every second message comes through -- I get 1, 3, 5, 7, 9, 11.

Which does not make sense to me, cause I am waiting for the clock to get stretched before writing. Maybe the clock is being stretched by something else... What if I clear it and then wait again.

```
while(I2C1STAT0bits.SMA == 0){};
if(I2C1STAT0bits.R == 1){
    for(int i = 1; i < 13; i ++){
        while(I2C1CON0bits.CSTR == 0);
        I2C1CON0bits.CSTR = 0;
        while(I2C1CON0bits.CSTR == 0);
        I2C1TXB = i;
        //__delay_us(10);
        I2C1CON0bits.CSTR = 0;
    }
}
```

This works for the first few some of the time, but is not reliable.

Leave 3:30

20/07/23

Time: 6.5h

July 20, 2023 8:40 AM

Start 8:30

At the end of the day yesterday, I was working on the I2C. I was struggling that the clock stretching was not working to use for timing the data transmission. I was thinking last night and thought I had forgotten to enable the clock stretching on the ACK bit.

```
I2C1PIEbits.CNT1IE = 1;      //Enable interrupt when byte count reaches 0
I2C1PIEbits.ACKTIE = 1;      //Enable interrupt for acknowledge clock stretching
I2C1PIEbits.WRIE = 1;        //Enable interrupt for receiving data to process and determine ACK or NACK
I2C1PIEbits.ADRIE = 0;       //Enable interrupt for processing address
I2C1PIEbits.PCIE = 0;        //Disable interrupt when stop condition detected
I2C1PIEbits.RSCIE = 0;       //Disable interrupt when restart condition detected
I2C1PIEbits.SCIE = 0;        //Disable interrupt when start condition detected
```

Looks like I did turn ACKTIE on which I believe should stretch the clock after each ACK.

I tried using the TXBE flag to wait until the buffer was empty before loading it again. It did not work at all.

Since 1 clock is 10us, a byte should be ~90us

```
101     while(I2C1STAT0bits.SMA == 0){};
102     if(I2C1STAT0bits.R == 1){
103         for(int i = 1; i < 13; i ++){
104             I2C1TXB = i;
105             __delay_us(90);
106             I2C1CON0bits.CSTR = 0;
107         }
108     }
```

This appears to be completely reliable…

I might try and make a script to continuously call this function and check if there are any read errors/how many there are in N readings.

It is not at all completely reliable. With that setup, I ran 2000 iterations, meaning 24k bytes transferred. Of that, 14.3k bytes were incorrect (60%). I have noticed that if I increase the delay on the PIC I get better reliability, but still not great.

Increasing the delay from 90 to 200 us, I got 787 errors (3.3%)

Increasing to 500us I got 113 Errors (0.47%).

1000us reduces to 44 errors

Clearly the major source of this issue is timing.

```
101     while(I2C1STAT0bits.SMA == 0){};
102     if(I2C1STAT0bits.R == 1){
103         for(int i = 1; i < 13; i ++){
104             I2C1TXB = i;
105             __delay_us(200);
106             I2C1CON0bits.CSTR = 0;
107
108         }
109     }
110     __delay_us(200);
111     /*RESET FOR NEXT COMMUNICATION*/
112     I2C1CON0bits.EN = 0;
113     I2C1ADB0 = 0x00;
114     I2C1PIR = 0x00;
115     I2C1CON0bits.EN = 1;
```

With this, I am getting 0.079% error rate.

I upped to 10000 iterations (120k bytes) and I got 146 errors

```
101     while(I2C1STAT0bits.SMA == 0){};
102     if(I2C1STAT0bits.R == 1){
103         for(int i = 1; i < 13; i ++){
104             I2C1TXB = i;
105             __delay_us(200);
106             I2C1CON0bits.CSTR = 0;
107
108         }
109     }
110     __delay_us(200);
111     /*RESET FOR NEXT COMMUNICATION*/
112     //I2C1CON0bits.EN = 0;
113     I2C1PIR = 0x00;
        //I2C1CON0bits.EN = 1;
```

I simplified further to this, and still had a reliability of 136 errors / 120k (0.11%)

Maybe it is time to get something going for the write command. I can optimize for reliability later, it would be nice to have something working for both.

```
112     if(I2C1STAT0bits.R == 0){
113         int index = 0;
114         while(I2C1CNTL > 0){
115             while(I2C1STAT1bits.RXBF == 0);
116             i2cBuffer[index] = I2C1RXB
117             I2C1CON0bits.CSTR = 0;
118             index++;
119         }
120     }
```

I am thinking this should work -- wait for the rxb to fill, then read it to an array, release the clock and incement the index in the array.

I tested this. The RXBF is not working, so I switched it to CSTR for now. The first byte is being written, but that is it. It looks like the issue is that a ACK is not being generated by the PIC.

Maybe I need to set that manually after reading.

```
112     if(I2C1STAT0bits.R == 0){
113         int index = 0;
114         while(I2C1CNTL > 0){
115             while(I2C1CON0bits.CSTR == 0);
116             i2cBuffer[index] = I2C1RXB;
117             I2C1CON1bits.ACKDT = 1;
118             I2C1CON0bits.CSTR = 0;
119             index++;
120         }
121     }
```

This is not generating the ACK on the bus

9. Host hardware transmits the 9th clock pulse. If there are pending errors, such as receive buffer overflow, client hardware automatically generates a NACK condition, sets NACKIF, and the module goes Idle. If I2CxCNT is nonzero (I2CxCNT != 0), client hardware transmits the value of ACKDT as the acknowledgement response to the host. It is up to software to configure ACKDT appropriately. In most cases, the ACKDT bit must be clear (ACKDT = 0) so that the host receives an $\overline{\text{ACK}}$ response (logic low level on SDA during the 9th clock pulse). If I2CxCNT is zero (I2CxCNT = 0), client hardware transmits the value of the Acknowledge End of Count (ACKCNT) bit as the Acknowledgement response, rather than the value of ACKDT. It is up to software to configure ACKCNT appropriately. In most cases, ACKCNT must be set (ACKCNT = 1), which represents a NACK condition. When host hardware detects a NACK on the bus, it will generate a Stop condition. If ACKCNT is clear (ACKCNT = 0), an $\overline{\text{ACK}}$ will be issued, and host hardware will not issue a Stop condition.

Looks like I clear for an ACK bit

```
112     if(I2C1STAT0bits.R == 0){
113         int index = 0;
114         while(I2C1CNTL > 0){
115             while(I2C1CON0bits.CSTR == 0);
116             i2cBuffer[index] = I2C1RXB;
117             I2C1CON1bits.ACKDT = 0;
118             I2C1CON0bits.CSTR = 0;
119             index++;
120         }
121     }
```

This still is not working properly. However it says that the ACKDT will be bypassed if errors have occurred. Let me read the error register.

I2C1CON1	SFR	0x295	0x00	00000000
I2C1CON2	SFR	0x296	0x40	01000000
I2C1ERR	SFR	0x297	0x10	00010000
I2C1PIE	SFR	0x29B	0xD3	11010011

An error did occur -- but this error is that a NACK was detected on the bus. The NACK would be generated by the client...

But RXO isnt even in the error register and that is the quoted example of an error

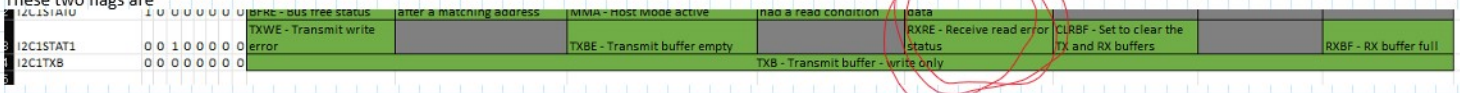
Notes:

1. Software writes to ACKDT must be followed by a minimum SDA setup time before clearing CSTR.
2. A NACK may still be generated by hardware when bus errors are present as indicated by the I2CxSTAT1 or I2CxERR registers.

```
111
112     if(I2C1STAT0bits.R == 0){
113         int index = 0;
114         while(I2C1CNTL > 0){
115             while(I2C1CON0bits.CSTR == 0);
116             i2cBuffer[index] = I2C1RXB;
117             I2C1CON1bits.ACKDT = 0;
118             __delay_us(100);
119             I2C1CON0bits.CSTR = 0;
120             index++;
121         }
122     }
```

This is did not work, I think there is an error set somewhere.

These two flags are



There is a Receive read error status set -- that is probably my issue

Bit 3 – RXRE Receive Read Error Status(1)

Value	Description
1	A byte of data was read from I2CxRXB when it was empty (*must be cleared by software*)
0	No receive overflow occurred

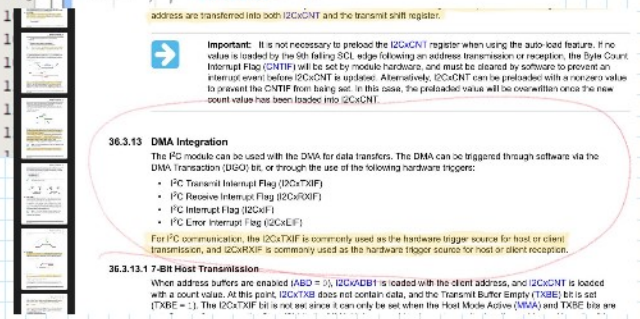
I read the buffer when it was empty.

```
111     CSTR=0
112     if(I2C1STAT0bits.R == 0){
113         int index = 0;
114         while(I2C1CNTL > 0){
115             while(I2C1STAT1bits.RXBF == 0);
116             i2cBuffer[index] = I2C1RXB;
117             I2C1CON1bits.ACKDT = 0;
118             I2C1CON0bits.CSTR = 0;
119             index++;
120         }
121     }
```

I told it to wait until the buffer was full but this failed.

I noticed on the scope that only the address was transmitted the clock after the matching address flag holding a bit to solve my problem.

```
103     while(I2C1STAT0bits.SMA == 0);
104     if(I2C1STAT0bits.R == 1){
105         for(int i = 0; i<=15; i ++){
106             I2C1TXB = i;
```



address are transferred into both I2CxCNT and the transmit shift register.

Important: It is not necessary to preload the I2CxCNT register when using the auto-load feature. If no value is loaded by the 9th falling SCL edge following an address transmission or reception, the Byte Count Interrupt Flag (I2CxCNTIF) will be set by module hardware, and must be cleared by software to prevent an interrupt event before I2CxCNT is updated. Alternatively, I2CxCNT can be preloaded with a nonzero value to prevent the CNTIF from being set. In this case, the preloaded value will be overwritten once the new count value has been loaded into I2CxCNT.

36.3.13 DMA Integration

The I²C module can be used with the DMA for data transfers. The DMA can be triggered through software via the DMA Transaction (DSG) bit, or through the use of the following hardware triggers:

- I²C Transmit Interrupt Flag (I2CxTXIF)
- I²C Receive Interrupt Flag (I2CxRXIF)
- I²C Interrupt Flag (I2CxIF)
- I²C Error Interrupt Flag (I2CxEIF)

For I²C communication, the I2CxTXIF is commonly used as the hardware trigger source for host or client transmission, and I2CxRXIF is commonly used as the hardware trigger source for host or client reception.

36.3.13.1 7-Bit Host Transmission

When address buffers are enabled (ABD = 0), I2CxADB1 is loaded with the client address, and I2CxCNT is loaded with a count value. At this point, I2CxTXB does not contain data, and the Transmit Buffer Empty (TXBE) bit is set (TXBE = 1). The I2CxTXIF bit is not set since it can only be set when the Host Mode Active (MMA) and TXBE bits are

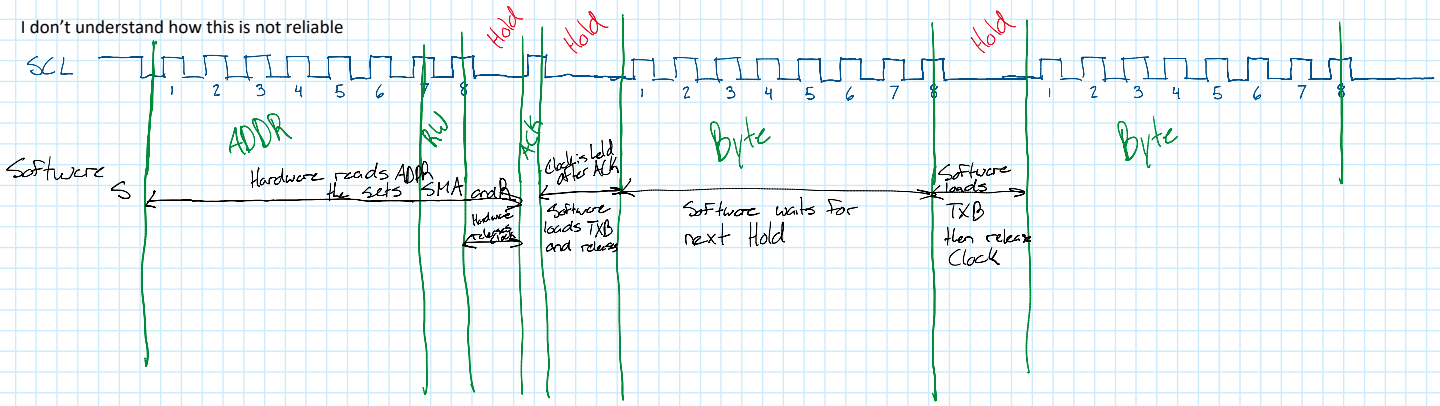
This may be something else to look into — I assume this would function like the IMU where you directly connect to registers with the I2C

36.3.13.4 7-Bit Client Transmission

In 7-bit Client Transmission mode, the state of ADD is ignored. If the client receives the matching 7-bit address and TXBE is set, I2CxTXIF is set by hardware, triggering the DMA to load data into I2CxTXB. Once the data is transmitted from I2CxTXB, I2CxTXIF is set by hardware, triggering the DMA to once again load I2CxTXB with data. The DMA will continue to load data into I2CxTXB until I2CxCNT reaches a zero value. Once I2CxCNT reaches zero and the data is transmitted from I2CxTXB, I2CxTXIF will not be set, and the DMA will stop loading data.

```
103     while(I2C1STAT0bits.SMA == 0){};
104     if(I2C1STAT0bits.R == 1){
105         for(int i = 1; i < 13; i ++){
106             I2C1TXB = i;
107             while(I2C1CON0bits.CSTR == 0);
108             //__delay_us(200);
109             I2C1CON0bits.CSTR = 0;
110
111         }
112         __delay_us(200);
113     }
```

I don't understand how this is not reliable




```
103     while(I2C1STAT0bits.SMA == 0){};
104     if(I2C1STAT0bits.R == 1){
105         for(int i = 1; i < 13; i ++){
106             I2C1TXB = i;
107             __delay_us(5);
108             I2C1CON0bits.CSTR = 0;
109             while(I2C1CON0bits.CSTR == 0);
110         }
111         //__delay_us(200);
112     }
113
```

I changed to this, which seems to have good accuracy every second time. Maybe I have an error when the end condition of my loop is not the same as the start

So when the for loop starts, it is assumed that the clock is being held from after ACK the address.

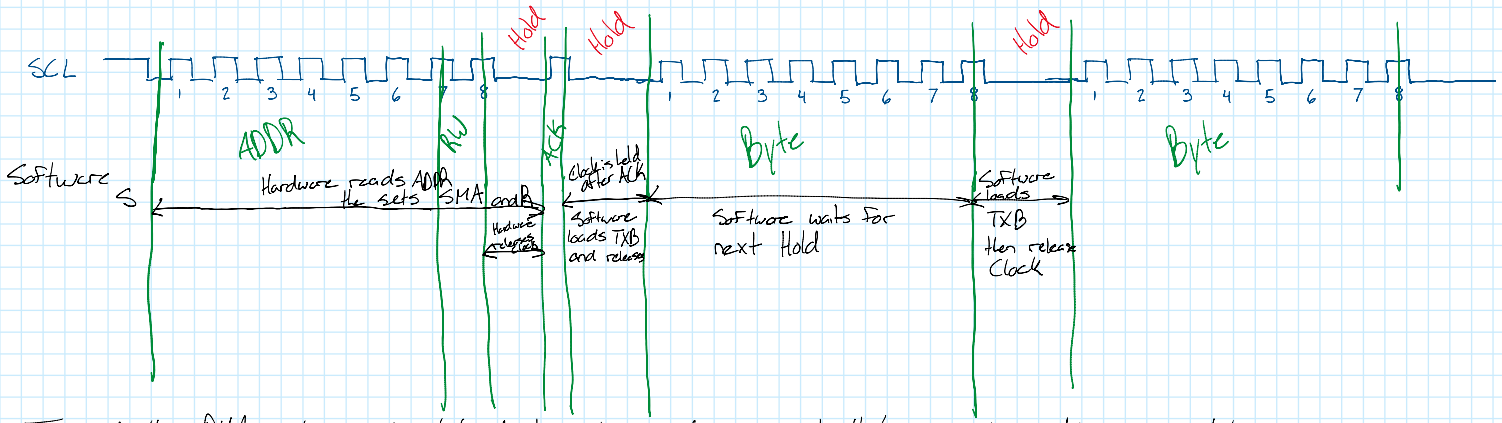
But the for loop ends when the clock is enabled. Therefor when it starts the next time the clock is enabled not disabled. So there is something wrong there. Nevermind, the loop ends when the clock gets held so it is held going into the next iteration.

Leave: 3:00 (Power went out)

21/07/23

Friday, July 21, 2023 9:00 AM

Start 8:00AM



I read the DMA section of the data sheet - not sure how useful it will be when the address is in 2 bytes
My thought was that the I2C would send the address into the destination pointer of the DMA
Then on the next then the DMA would use the RXB as the source and copy the data to the PWM register

Procedure

1. I2C start
2. I2C Address
3. I2C Byte 1 loaded into DMA DSAH
4. I2C Byte 2 loaded into DMA DSAL
5. Set DMA SSA to be the RXB
6. Hardware trigger DMA when RXB is full

On the IMU, you send S - ADDR - $\bar{W}$ - Reg Addr - Restart - ADDR - R/$\bar{W}$ - Data

So on a start condition, DMA uses RXB as a source, and some register A as a destination
Then after a restart, DMA:
- W - DMA uses RXB as a source and the DSA is set to value in A
- R - DMA uses value in A as a source and TXB as a destination

The Datasheet talked about using I2CxTXIF and I2CxRXIF to control the DMA hardware

36.3.13.4 7-Bit Client Transmission

In 7-bit Client Transmission mode, the state of ABD is ignored. If the client receives the matching 7-bit address and TXBE is set, I2CxTXIF is set by hardware, triggering the DMA to load data into I2CxTXB. Once the data is transmitted from I2CxTXB, I2CxTXIF is set by hardware, triggering the DMA to once again load I2CxTXB with data. The DMA will continue to load data into I2CxTXB until I2CxCNT reaches a zero value. Once I2CxCNT reaches zero and the data is transmitted from I2CxTXB, I2CxTXIF will not be set, and the DMA will stop loading data.

36.3.13.5 10-Bit Client Transmission

In 10-bit Client Transmission mode, the state of ABD is ignored. If there is no data in I2CxTXB after the client has received the address high byte with the R/$\bar{W}$ bit set, hardware sets I2CxTXIF, triggering the DMA to load I2CxTXB. The DMA will continue to load data into I2CxTXB until I2CxCNT reaches a zero value. Once I2CxCNT reaches zero and the data is transmitted from I2CxTXB, I2CxTXIF will not be set, and the DMA will stop loading data.

36.3.13.6 7/10-Bit Client Reception

When address buffers are enabled (ABD = 0), client hardware loads I2CxADB0/1 with the matching address, while all data is received by I2CxRXB. Once the client loads I2CxRXB with a received data byte, hardware sets I2CxRXIF, which triggers the DMA to read I2CxRXB. The DMA will continue to read I2CxRXB whenever I2CxRXIF is set.

When address buffers are disabled (ABD = 1), the client loads I2CxRXB with the matching address byte(s) as they are received. Each received address byte sets I2CxRXIF, which triggers the DMA to read I2CxRXB. The DMA will continue to read I2CxRXB whenever I2CxRXIF is set.

The datasheet says that TXIF is set when in transmission mode and RXIF is set in recieve mode - I am wondering if this means whether read or write

I could have a start condition ISR that configures the DMA to use RXB as the source and some memory location as the destination

Then use the RXIF to trigger the DMA

Then on the restart condition, reconfigure the DMA but I need to configure it differently for Reed or Write

How many DMA modules are there? I could just use 2 modules and configure one for the RXIF condition and 1 for the TXIF condition

There are 4 DMA modules

I think I need 3 for this plan

DMA1 → source is RXB
destination is A (some memory)
trigger on RXIF

DMA2 → source Address stored in A
destination is TXB

trigger on MXIF

DMA2 → source Address stored in A
destination is TXB
trigger on TXIF

DMA3 → source RXB
destination Address stored in A
trigger on RXIF

On start condition → Enable DMA1
Disable DMA2
Disable DMA3

On restart condition → Disable DMA1
Enable DMA2
Enable DMA3

One problem is that I am not sure what memory DMA has access to for the 'A' memory location

Key features of the DMA module include:

- Support access to the following memory regions:
  - GPR and SFR space (R/W)
  - Program Flash memory (R only)
  - Data EEPROM memory (R only)

So clearly for this plan, A needs to be in GPR or SFR but I dont know what that means

SFR - Special Function register - things like I2C buffers, PWM duty cycle reg

GPR - General Purpose Register - General memory for user application → I think this is what I need

I forgot for the IMU, there is not a restart condition for write commands You just write the data after the REGADDR

That would be more efficient as the client address only needs to be resent on read commands but the logic for this is less clear

Maybe I still use the 3 registers, and the one for write commands doesnt have a source set until after the Start DMA → but I dont think that the DMA occurs inline with the code I could technically delay but that is lame — avoiding delays is kinda the whole point of figuring out the DMA

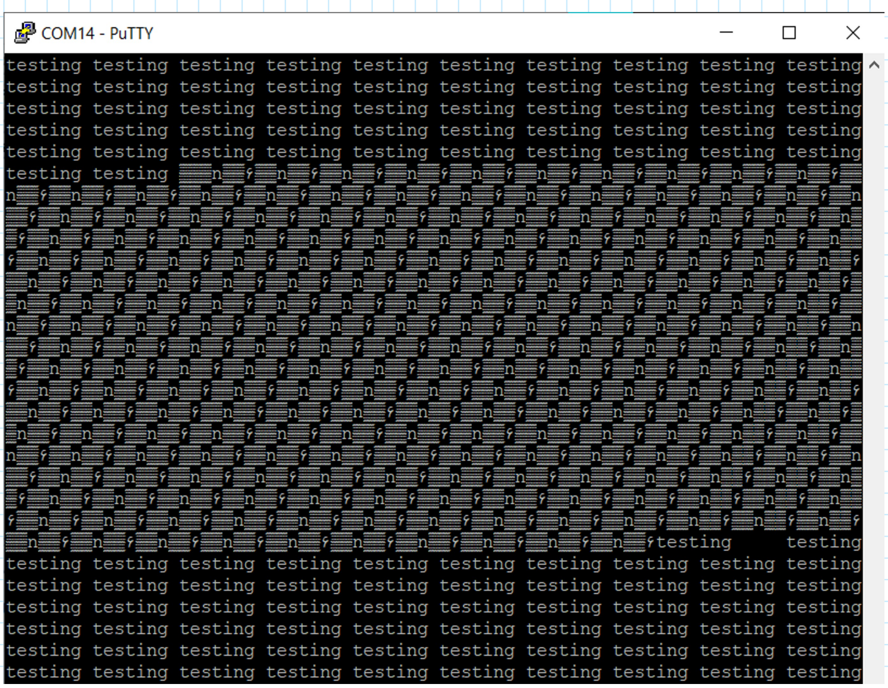
The only other thing I can think of is waiting for the DMA1 to clear the SIRQEN bit - which might be okay

UART

Nick contacted the guy at Intempco about the USB converted and it is wired so that TX connects to TX, which explains why it was not working. I connected it the right way and got strange characters again. Tweaked the BAUD rate value a bit and got it to read properly. It is still not completely reliable.

In that image you can see two errors.

When I add a slight delay into the script between words it seems to become way more reliable.



It will also do that occasionally which is not good…

I am not sure if the delay actually makes it more reliable or if errors just show up less cause there are less transmissions.

I used putty to create a log of the data. I did this for approximately a minute for with 0 delay and 1ms delay.

msDelay

noDelay

To-Do

- Get more reliable I2C
- Get more reliable UART
- Get UART receive working on the PIC
- Fix Accel offset in IMU lib