19/06/23

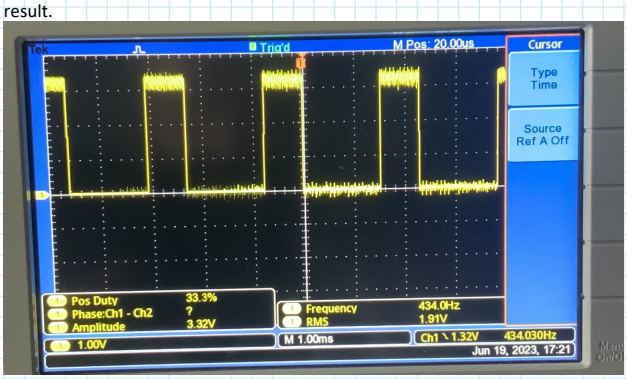
Time 9h

Monday, June 19, 2023 7:59 AM

Start 8:00AM

My first goal of today is to debug the issue where the PWM signal is not stable when high.

My first thought was that maybe there was a hardware issue> When testing, I was using the pin that had the LED connected. I cant think why this would make a difference, however it is one variable that I can remove by switching to another pin. I switched it to RB6 and got the following result.



While there is still some fluctuations, it is substantially better than before. This suggests there was some additional noise coming from the circuit connected to the LED.

I emailed Nick and he said he thinks that we can still do better. Maybe a grounding issue. I moved the ground from the wire on the shield and connected it directly to the pin on the MCU.

I tried a few other things, but eventually found the problem.

If the program does not have a loop to get stuck in, it will re-execute the code from the beginning. This involves the setup of the PWM which involves disabling and re-enabling the PWM signal. If I make a loop for the script to get stuck in, even without anything in it, the signal becomes much better.

I believe that this signal should be good. There is obviously still a small amount of noise but that will never completely go away.

```
#include "Config_18F15Q40.h"
#include <xc.h>
#include <stdio.h>                                  // Standard IO functions
#include <stdlib.h>                                 // Standard library functions
#include <string.h>

uint16_t duty_cycle;

void PWMsetup(){
  /*Setup Pins*/
  RB6PPS = 0x09;  //Assigns RB6 to CCP1 output
  TRISBbits.TRISB6 = 1;  //Disable output driver

  /*Configure PWM*/
  T2PR = 0xFF;          //Set timer pre-scaler for 10bit res
  CCP1CONbits.EN = 1;    //Enable CCP
  CCP1CONbits.FMT = 0;   //Right aligned PWM
  CCP1CONbits.MODE = 0b1100;  //Set mode to PWM

  /*Set Duty cycle*/
  duty_cycle = 0x0000;   //Define 16bit Duty cycle (only 10 relevant bits)
  CCPR1 = duty_cycle;    //Write duty cycle to register

  PIR3bits.TMR2IF = 0;      //Clear interrupt flag bit
  T2CLKbits.CS = 0b00001;    //Set the timer clock source to Fosc/4
  T2CONbits.CKPS = 0b11;     //Set timer pre-scaler value??
  //T2CONbits.RD16 = 1;      //Supposed to configure to 16bit or 8bit
  T2CONbits.T2ON = 1;

  TRISBbits.TRISB6 = 0;

  return;
```

```
}

void ADCsetup(){
   /*Configure Pin*/
//   ADPCH = 0x001101;     //RB5 analog channel
//   TRISBbits.TRISB5 = 1;   //Set RB5 to input
//   ANSELBbits.ANSELB5 = 1; //Set RB5 to analog

   ADPCH = 0x00;           //RA0
   TRISAbits.TRISA0 = 1;   //Set RB5 to input
   ANSELAbits.ANSELA0 = 1; //Set RB5 to analog

   /*Configure ADC*/
   ADCON0bits.FM = 1;   //Right justify
   ADCON0bits.CS = 1;   //ADCRC Clock
   ADACQ = 32;          //Set acquisition time
   ADCON0bits.ON = 1;     //Turn on ADC
}

uint16_t ADCread(){
   /*Read ADC*/
   ADCON0bits.GO = 1;  //Start acquisition
   while(ADCON0bits.GO); //Wait until done
   uint8_t resultHigh = ADRESH;
   uint8_t resultLow = ADRESL;

   /*Convert 2 8bit to 1 16bit*/
   uint16_t result = resultHigh;
   result = result << 8;
   result = result + resultLow;

   return result;
}

void loop(){
   while(1){
      duty_cycle = ADCread();
      CCPR1 = duty_cycle;
   }
}

void main() {
   ADCsetup();
   PWMsetup();
   loop();
   return;
}
```

I created the above script to read an analog value from a potentiometer, and then use that reading to provide a PWM signal. I think that I have this working. However I struggled getting the ADC to work on pin RB5. I reverted back to the one in the example in the datasheet, RA0. When I did this, the program worked.

I am not sure what I am doing wrong with the RB5 channel.

I figured out by problem. I was using 0x001101 for the address instead of 0b001101. The ADC now works on the RB5 pin as well.

I think the only thing left is to get I2C working... But to actually test that, I will need another device for it to communicate with. In this configuration, the MCU would be acting as the client not the host, as the RPI would be calling it for information and it would be sending it. For this reason, it would be good to get a I2C sensor that I can use to develop a working I2C software on the PI. Once I know that I have that working, I can swap to the MCU to see if I have the I2C software on it working properly.

I2C

36.3.2 SDA and SCL Pins

The SDA and SCL pins must be configured as open-drain outputs. Open-drain configuration is accomplished by setting the appropriate bits in the Open-Drain Control (ODCONx) registers, while output direction configuration is handled by clearing the appropriate bits in the Tri-State Control (TRISx) registers. Input threshold, slew rate, and internal pull-up settings are configured using the RxyI2C registers. The RxyI2C registers are used exclusively on the default I^2C pin locations, and provide the following selections:

- Input threshold levels:
  - SMBus 3.0 (1.35V) input threshold
  - SMBus 2.0 (2.1V) input threshold
  - I^2C-specific input thresholds

  - Standard GPIO input thresholds (controlled by the Input Level Control (INLVLx) registers)
- Slew rate limiting:
  - I^2C-specific slew rate limiting
  - Standard GPIO slew rate (controlled by the Slew Rate Control (SLRCONx) registers)
- I^2C pull-ups:
  - Programmable ten or two times the current of the standard internal pull-up
  - Standard GPIO pull-up (controlled by the Weak Pull-Up Control (WPUx) registers)

Important: The pin locations for SDA and SCL are remappable through the Peripheral Pin Select (PPS) registers. If new pin locations for SDA and SCL are desired, user software must configure the INLVLx, SLRCONx, ODCONx, and TRISx registers for each new pin location. The RxyI2C registers cannot be used since they are dedicated to the default pin locations. Additionally, the internal pull-ups for non-I^2C pins are not strong enough to drive the pins; therefore, external pull-up resistors must be used.

36.6 Register Summary - I2C

Address	Name	Bit Pos.	7	6	5	4	3	2	1	0
0x00 ... 0x0289	Reserved									
0x028A	I2C1RXB	7:0	RXB[7:0]							
0x028B	I2C1TXB	7:0	TXB[7:0]							
0x028C	I2C1CNT	7:0	CNT[7:0]							
		15:8	CNT[15:8]							
0x028E	I2C1ADB0	7:0	ADB[7:0]							
0x028F	I2C1ADB1	7:0	ADB[7:0]							
0x0290	I2C1ADR0	7:0	ADR[7:0]							
0x0291	I2C1ADR1	7:0	ADR[6:0]							
0x0292	I2C1ADR2	7:0	ADR[7:0]							
0x0293	I2C1ADR3	7:0	ADR[6:0]							
0x0294	I2C1CON0	7:0	EN	RSEN	S	CSTR	MDR	MODE[2:0]		
0x0295	I2C1CON1	7:0	ACKCNT	ACKDT	ACKSTAT	ACKT	P	RXO	TXU	CSD
0x0296	I2C1CON2	7:0	ACNT	GCEN	FME	ABD	SDAHT[1:0]		BFRET[1:0]	
0x0297	I2C1ERR	7:0		BTOIF	BCLIF	NACKIF		BTOIE	BLCIE	NACKIE
0x0298	I2C1STAT0	7:0	BFRE	SMA	MMA	R	D			
0x0299	I2C1STAT1	7:0	TXWE		TXBE		RXRE	CLRBF		RXBF
0x029A	I2C1PIR	7:0	CNTIF	ACKTIF		WRIF	ADRIF	PCIF	RSCIF	SCIF
0x029B	I2C1PIE	7:0	CNTIE	ACKTIE		WRIE	ADRIE	PCIE	RSCIE	SCIE
0x029C	I2C1BTO	7:0	TOREC	TOBY32	TOTIME[5:0]					
0x029D	I2C1BAUD	7:0	BAUD[7:0]							
0x029E	I2C1CLK	7:0					CLK[3:0]			
0x029F	I2C1BTOC	7:0						BTOC[2:0]		

12:00 lunch
12:30 back

https://www.digikey.ca/en/products/detail/tdk-invensense/EV_ICM-20948/7319741?utm_adgroup=Evaluation%20Boards%20-%20Sensors&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Development%20Boards%2C%20Kits%2C%20Programmers_NEW&utm_term=&productid=7319741&gclid=CjwKCAjw-b-kBhB-EiwA4fvKrI6YMklxsNCmkX84QYu_zHPJIQy_J7xi4OzzIxB3vHpnoz1ipUmcdhoC-A4QAvD_BwE

https://www.digikey.ca/en/products/detail/seeed-technology-co-ltd/104020208/10667534

Meeting with Nick 2h

Read about I2C on the PIC18 chip

Leave 4:00

Start 7:15

I2C Arduino

In my meeting with Nick today, he mentioned using an Arduino to act as the I2C host to be able to test the PIC18 chip when I figure out how to setup the I2C on it.

https://docs.arduino.cc/learn/communication/wire

This website gives a simple example in Arduino on how to use I2C communication protocol.

Controller Reader Sketch

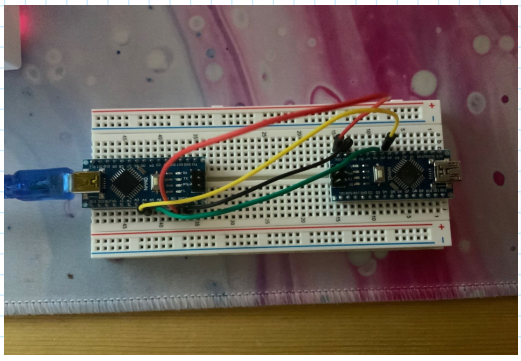
COPY

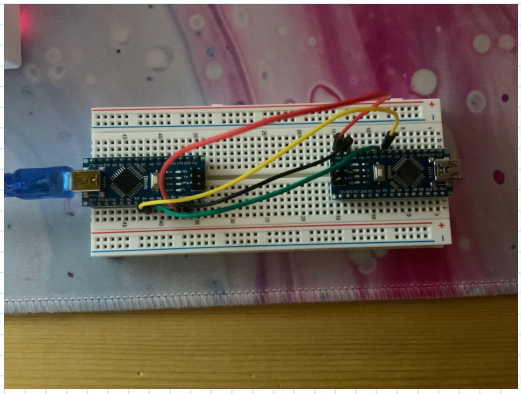
```
// Wire Controller Reader
// by Nicholas Zambetti <http://www.zambetti.com>

// Demonstrates use of the Wire library
// Reads data from an I2C/TWI peripheral device
// Refer to the "Wire Peripheral Sender" example for use wi

// Created 29 March 2006

// This example code is in the public domain.


#include <Wire.h>

void setup() {
  Wire.begin();        // join i2c bus (address optional fc
  Serial.begin(9600);  // start serial for output
}

void loop() {
  Wire.requestFrom(8, 6);    // request 6 bytes from periph

  while (Wire.available()) { // peripheral may send less th
    char c = Wire.read(); // receive a byte as character
    Serial.print(c);         // print the character
  }

  delay(500);
}
```

Controller Writer Sketch

COPY

```
// Wire Master Writer
// by Nicholas Zambetti <http://www.zambetti.com>

// Demonstrates use of the Wire library
// Writes data to an I2C/TWI Peripheral device
// Refer to the "Wire Peripheral Receiver" example for use

// Created 29 March 2006

// This example code is in the public domain.


#include <Wire.h>

void setup()
{
  Wire.begin(); // join i2c bus (address optional for maste
}

byte x = 0;

void loop()
{
  Wire.beginTransmission(4); // transmit to device #4
  Wire.write("x is ");        // sends five bytes
  Wire.write(x);              // sends one byte
  Wire.endTransmission();    // stop transmitting

  x++;
  delay(500);
}
```





I connected two arduinos with I2C and copied the test scripts from the website above for the client and host software.

It is a very basic example, but I was able to get the client to send a message back to the host.

I had originally assumed that I would be able to send information with the request so I would be able to like give a command to base the response on. Like my request could contain an address so that my request could give the result for a specific sun sensor or something. That is probably not really useful. It may be just easier to get them all at once and parse it after. Because I am thinking that I would need to write to the client the address, and then it would save it for the next request to send back the specific data. I could also program a system where you can write an address first to get only one result, or you can just get everything at once. I think there will be enough memory on board to do that and it may give a more robust system. Especially when Nick is saying that he wants to get this code working and then just leave it.

Host

```
#include <Wire.h>
void setup() {
  // put your setup code here, to run once:
  Wire.begin();           //Start I2C
  Serial.begin(9600);     //Start Serial for debug/output
}
void loop() {
  // put your main code here, to run repeatedly:
  uint8_t addr = 0x11;  //Client address
  uint8_t len = 1;      //Length of request in bytes


Wire.beginTransmission(addr);
  Wire.write(0x01);
  Wire.endTransmission();
  Serial.println("Command Set 0x01");

    Wire.requestFrom(addr, len);  //Request data from client
  while(Wire.available()){
    byte c = Wire.read(); //c is one byte
    Serial.print(c, HEX);
  }
  Serial.println("");
  delay(1000);
}
```

Client

```
#include <Wire.h>
//NOTE TO SELF -- Client has no yellow sticker
uint8_t addr = 0x11;
uint8_t command = 0x00;
void setup() {
  // put your setup code here, to run once:
  Wire.begin(addr); //Connect to I2C with address
  Wire.onRequest(respond); //Function to do something when called
  Wire.onReceive(setCommand);
}
void loop() {
  // put your main code here, to run repeatedly:
  delay(100);
}
void respond(){
  //Default command to return all values
  if(command == 0x00){
    Wire.write(0x00);
  }
  //Special command for certain sensor
  if(command == 0x01){
    Wire.write(0x01);
    command = 0x00;
  }
}
void setCommand(){
  byte msg;
  while(1 < Wire.available()){
    msg = Wire.read();
  }
  command = msg;
}
```

I tried implimenting my idea where a command could be sent first but it only works the first time
I am not sure why that is considering the same code is looping.

```
Output   Serial Monitor ×
Message (Enter to send message to 'Arduino Nano' on 'COM7')
Command Set 0x01
1
Command Set 0x01
0
Command Set 0x01
0
Command Set 0x01
0
```

8:45 end

20/06/23 Time 7.5

Tuesday, June 20, 2023 8:03 AM

Start 8:20AM

Read more of datasheet for I2C

11.13.26 PIR7

Name: PIR7
Address: 0x18A

Peripheral Interrupt Request Register 7

Bit	7	6	5	4	3	2	1	0
	PWM3IF	PWM3PIF	CLC3IF		I2C1EIF	I2C1IF	I2C1TXIF	I2C1RXIF
Access	R	R/W/HS	R/W/HS		R	R	R	R
Reset	0	0	0		0	0	0	0

Bit 7 – PWM3IF PWM3 Parameter Interrupt Flag[2]

Value	Description
1	Interrupt has occurred
0	Interrupt event has not occurred

Bit 6 – PWM3PIF PWM3 Period Interrupt Flag

Value	Description
1	Interrupt has occurred (must be cleared by software)
0	Interrupt event has not occurred

Bit 5 – CLC3IF CLC3 Interrupt Flag

Value	Description
1	Interrupt has occurred (must be cleared by software)
0	Interrupt event has not occurred

Bit 3 – I2C1EIF I2C1 Error Interrupt Flag[3]

Value	Description
1	Interrupt has occurred
0	Interrupt event has not occurred

Bit 2 – I2C1IF I2C1 Interrupt Flag[4]

Value	Description
1	Interrupt has occurred
0	Interrupt event has not occurred

Bit 1 – I2C1TXIF I2C1 Transmit Interrupt Flag[5]

Value	Description
1	Interrupt has occurred
0	Interrupt event has not occurred

Bit 0 – I2C1RXIF I2C1 Receive Interrupt Flag[6]

Value	Description
1	Interrupt has occurred
0	Interrupt event has not occurred

Notes:

1. Interrupt flag bits get set when an interrupt condition occurs, regardless of the state of its corresponding enable bit, or the global enable bit. User software must ensure the appropriate interrupt flag bits are clear prior to enabling an interrupt.

The problem I am running into is that I do not understand how to create ISR's and I need to do that for the I2C

```
135 void __interrupt() ISR(void){
136     if(I2C1PIEbits.SCIE && I2C1PIRbits.SCIF){
137         i2cReceive();
138     }
139     return;
140 }
```

I found something similar to this online. I changed the conditions on the if to be the enable interrupt and interrupt flag bits for the I2C start condition.

I also now realize that the Arduino I2C setup I was planning to test with outputs 5V on the pins. The MCU runs on 3.3V so I will need to something to convert the voltage levels

https://www.digikey.ca/en/products/detail/sparkfun-electronics/BOB-12009/5673795

This is a product to convert the logic levels and it specifies I2C in the datasheet so it should be compatible

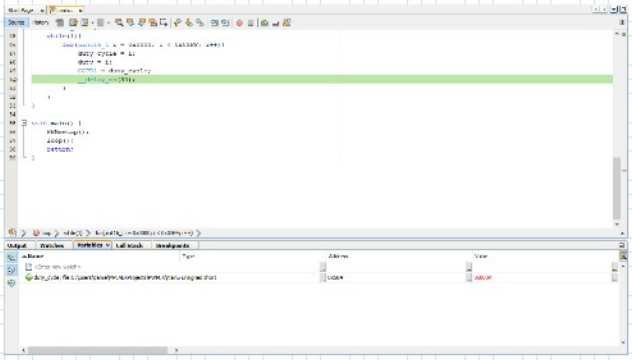
https://www.digikey.ca/en/mylists/list/N5XICY3AZQ

I cant actually test the I2C or do anything with that without the conversion chip. I tried making a script for the setup and then receiving information. The setup should be close, but I have absolutely no idea what is going on with the receiving information part. I also cant create any new projects in the MPLAB software cause it will not detect the SNAP thing.

I figured out for to create new projects that will connect to the SNAP and be programmable

Nick had mentioned that when I got the I2C working we would be able to read out values directly from the board to get a better understanding of what it is doing. I know the SNAP programmer is supposed to be able to do that but I was not able to get it working

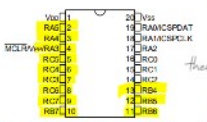
So the programmer will not show the local variable values while it is running. But if I use a breakpoint it will hault and output the variable values,



This will be useful for setting up multi channel ADC.

12 Channel ADC script

Figure 2-2.
20-Pin PDIP
20-Pin SOIC
20-Pin SSOP



these 12 pins will be ADC pins

TrisA = 0bXXX111XXX
TrisB = 0b 1111XXXX
TrisC = 0b 1111XXXX

Equation 40-1. Acquisition Time Example

Assumptions: Temperature = 50°C; External impedance = 10 kΩ; V_{DD} = 5.0V

T_{ACQ} = Amplifier Settling Time + Hold Capacitor Charging Time + Temperature Coefficient

$$T_{ACQ} = T_{AMP} + T_C + T_{COFF}$$

$$T_{ACQ} = 2\,\mu s + T_C + \left[(Temperature - 25°C)\ (0.05\,\mu s/°C)\right]$$

The value for T_C can be approximated with the following equations:

$$V_{APPLIED}\left(1 - \frac{1}{(2^{n+1}) - 1}\right) = V_{CHOLD}$$; [1] V_{CHOLD} charged to within ½ LSb

$$V_{APPLIED}\left(1 - e^{\frac{-T_C}{RC}}\right) = V_{CHOLD}$$; [2] V_{CHOLD} charge response to $V_{APPLIED}$

$$V_{APPLIED}\left(1 - e^{\frac{-T_C}{RC}}\right) = V_{APPLIED}\left(1 - \frac{1}{(2^{n+1}) - 1}\right)$$; Combining [1] and [2]

Note: Where n = ADC resolution in bits

Solving for T_C:

$$T_C = -C_{HOLD}(R_{IC} + R_{SS} + R_S)\ ln\ (1/8191)$$

$$T_C = -28\,pF(1\,k\Omega + 7\,k\Omega + 10\,k\Omega)\ ln\ (0.0001221)$$

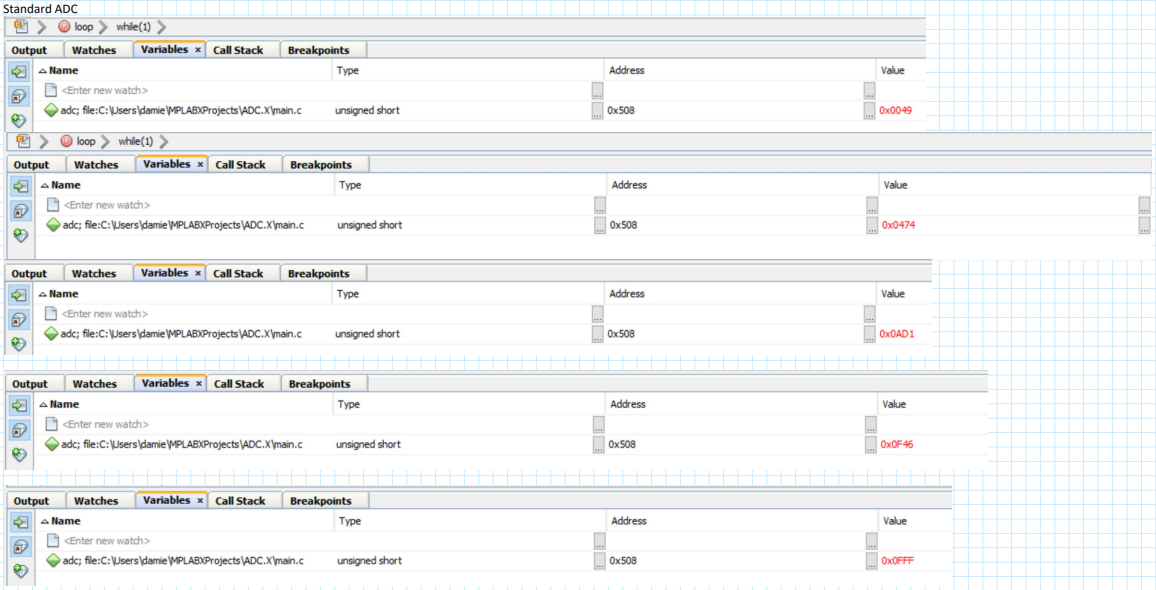
$$T_C = 4.54\,\mu s$$

Therefore:

$$T_{ACQ} = 2\,\mu s + 4.54\,\mu s + \left[(50°C - 25°C)\ (0.05\,\mu s/°C)\right]$$

$$T_{ACQ} = 7.79\,\mu s$$

Standard ADC



As I turn the Pot, I can see the ADC reading going up

12Channel

My 12 channel script did not work for reading the B5 pin... I commented out the new parts and I will uncomment them one by one until it breaks to see what the problem is.

It was not the code where the channel was set in the script.

The problem is here:

```
/*Configure ADC*/
ADREFbits.NREF = 0;     /*Set negative reference
                         * 0: Vref- is connected to AVss,
                         * 1: Vref- is connected to external Vref-*/

ADREFbits.PREF = 11;    /*Set positive reference
                         * 11: Vref+ is connected to internal Fixed Voltage Reference module
                         * 10: Vref+ is connected to external Vref+
                         * 01: Reserved
                         * 00: Vref+ is connected to VDD*/
```

This is causing the output to always be 0x0FFF...

I did not manually set these before, so they must have been set to default values.

Maybe I will try the other options from PREF

Using 00 works properly...

OH, 11 is eleven and not 1 1 in binary. I need 0b11

I changed it to 0b11 and it still was broken... Do I need to enable the Fixed Voltage Reference or something?

I do have to configure it

Name: FVRCON
Address: 0x3D7

FVR Control Register

Important: This register is shared between the Fixed Voltage Reference (FVR) module and the temperature indicator module.

Bit	7	6	5	4	3	2	1	0
	EN	RDY	TSEN	TSRNG	CDAFVR[1:0]		ADFVR[1:0]	
Access	R/W	R	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	q	0	0	0	0	0	0

Bit 7 – EN Fixed Voltage Reference Enable

Value	Description
1	Enables module
0	Disables module

Bit 6 – RDY Fixed Voltage Reference Ready Flag

Value	Description
1	Fixed Voltage Reference output is ready for use
0	Fixed Voltage Reference output is not ready for use or not enabled

Bit 5 – TSEN Temperature Indicator Enable

Value	Description
1	Temperature Indicator is enabled
0	Temperature Indicator is disabled

Bit 4 – TSRNG Temperature Indicator Range Selection

Value	Description
1	V_{OUT} = 3V_T (High Range)
0	V_{OUT} = 2V_T (Low Range)

Bits 3:2 – CDAFVR[1:0] FVR Buffer 2 Gain Selection[1]

Value	Description
11	FVR Buffer 2 Gain is 4x, (4.096V)[2,3]
10	FVR Buffer 2 Gain is 2x, (2.048V)[2,3]
01	FVR Buffer 2 Gain is 1x, (1.024V)
00	FVR Buffer 2 is OFF

Bits 1:0 – ADFVR[1:0] FVR Buffer 1 Gain Selection[2]

Value	Description
11	FVR Buffer 1 Gain is 4x, (4.096V)[2,3]
10	FVR Buffer 1 Gain is 2x, (2.048V)[2,3]
01	FVR Buffer 1 Gain is 1x, (1.024V)
00	FVR Buffer 1 is OFF

Notes:
1. This output goes to the DAC and comparator modules, and to the ADC module as an input channel only.
2. This output goes to the ADC module as a reference and an input channel.
3. Fixed Voltage Reference output cannot exceed V_{DD}.

I configured the ADFVR bits and now it is reading with that reference selected.

I am going to set it back to the VDD reference

I added in two measurement channels with 2 different potentiometers

This is reading independent values for each potentiometer. It appears to be working. There does not appear to be substantial influence from one measurement to the next. Even when the values were very different, they remained equally stable compared to when they were far about. This was NOT rigorously tested but it seems promising.

I added a large amount of comments to specify anything that was unclear or things I found in the datasheet.

C:/Users/damie/MPLABXProjects/ADC12CH.X/main.c

```
/** I N C L U D E S **************************************************************/
#include <xc.h>
#include <stdio.h>// Standard IO functions
#include <stdlib.h>// Standard library functions
#include <string.h>

#define _XTAL_FREQ 4000000      // One instruction cycle is 1 microsaec.

uint16_t B4;
uint16_t B5;

//This array hold the ADPCH bits for each pin
uint8_t channel_map[12] = {
    0b000101,   //RA5
    0b000100,   //RA4
    0b000011,   //RA3
    0b010101,   //RC5
    0b010100,   //RC4
    0b010011,   //RC3
    0b010110,   //RC6
    0b010111,   //RC7
    0b001111,   //RB7
    0b001110,   //RB6
    0b001101,   //RB5
    0b001100    //RB4
};

void ADCsetup(){
    /*Configure Pin*/
    TRISA = TRISA | 0b00111000; //Set RA3, RA4, RA5 to inputs
    TRISB = TRISB | 0b11110000; //Set RB4, RB5, RB6, RB7 to inputs
    TRISC = TRISC | 0b11111000; //Set RC3, RC4, RC5, RC6, RC7 to inputs

    ANSELA = ANSELA | 0b00111000; //Set RA3, RA4, RA5 to analog inputs
    ANSELB = ANSELB | 0b11110000; //Set RB4, RB5, RB6, RB7 to analog inputs
    ANSELC = ANSELC | 0b11111000; //Set RC3, RC4, RC5, RC6, RC7 to analog inputs

    /*Configure ADC*/
    ADREFbits.NREF = 0;     /*Set negative reference
                             * 0: Vref- is connected to AVss,
                             * 1: Vref- is connected to external Vref-*/


    ADREFbits.PREF = 0b00;    /*Set positive reference
                               * 11: Vref+ is connected to internal FVR module
                               * 10: Vref+ is connected to external Vref+
                               * 01: Reserved
                               * 00: Vref+ is connected to VDD*/

    /*IF 0b11 was selected above, configure the FVR module*/
    FVRCONbits.ADFVR = 0b11;    /*Configure the output voltage of the FVR
                                 * 11: 4x = 4.096V
                                 * 10: 2x = 2.048V
```


C:/Users/damie/MPLABXProjects/ADC12CH.X/main.c

```
                                 * 01: 1x = 1.024V
                                 * 00: OFF*/
    FVRCONbits.EN = 1;  //Enable FVR module

    ADCON0bits.FM = 1;      //Right justify
    ADCON0bits.CS = 1;      //Set clock to ADCRC Clock (~600kHz)
    ADACQ = 32;             //Set acquisition time
}

uint16_t ADCread(int channel){
    /*
     * Channel:
     * 0  -- RA5                6  -- RC6
     * 1  -- RA4                7  -- RC7
     * 2  -- RA3                8  -- RB7
     * 3  -- RC5                9  -- RB6
     * 4  -- RC4                10 -- RB5
     * 5  -- RC3                11 -- RB4
     */

    /*Set the ADC channel to ground and read to clear charge*/
    ADPCH = 0b111011;
    ADCON0bits.GO = 1;  //Start acquisition
    while(ADCON0bits.GO); //Wait until done

    /*Set channel to selected pin and measure ADC*/
    ADPCH = channel_map[channel];   //Set ADC to pin
    ADCON0bits.ON = 1;      //Turn on ADC
    ADCON0bits.GO = 1;  //Start acquisition
    while(ADCON0bits.GO); //Wait until done
    uint8_t resultHigh = ADRESH;    //Read high register
    uint8_t resultLow = ADRESL;     //Read low register

    /*Convert 2 8bit to 1 16bit*/
    uint16_t result = resultHigh;
    result = result << 8;
    result = result + resultLow;

    return result;
}

void loop(){
    while(1){
        //Read pin B5 (reference to '10' can be seen in the comment of ARCread() function)
        B5 = ADCread(10);

        __delay_ms(50);

        //Read pin B5 (reference to '11' can be seen in the comment of ARCread() function)
        B4 = ADCread(11);

        __delay_ms(50);
```


C:/Users/damie/MPLABXProjects/ADC12CH.X/main.c

```
    }
}

void main() {
    ADCsetup();
    loop();
    return;
}
```


Stop 3:30

21/06/23

Time: 7.5h

Wednesday, June 21, 2023 8:18 AM

6:00AM

Yesterday I got the ADC script working with 12 channels

One thing I did not look into was using an external V_{REF}^+ the option to select that in software was included — but I did not cover how to set a pin to be the external reference

From in the pinout table, it appears that RA1 is the only option for a V_{REF}^+ pin — I am assuming that this does not need to be configured further as there is no documentation on that

V_{REF}^- is on pin RA2 if we ever need that —

Dual PWM

I know the documentation mentions that 3 16 bit PWM modules are available — each with a dual channel output

If this is true One MCU should be able Run all the PWM as there are 5 PWM signals

I think the first step could be figuring out the dual channel on the one PWM module I have working — then I can try to copy that code to the other ones

I went back to the datasheet and found that there is another Section for PWM, not under the CCP section. I have no idea why there are 2 PWM options, but there are. The second option is way simpler to set up and give a lot more control over the PWM signal.

I am able to set the PWM cycle period which essentially defines the resolution of the signal. It corresponds to how many clock cycles are in the full PWM cycle (not duty cycle, that is set seperately)

Then you can set the Duty cycle for each individual slice (total of 2) which give independent outputs from the same module. There are the 3 modules so I can generate the full 5PWM signals from one device.

C:\Users\damie\MPLABXProjects\DUAL_PWM.X\main.c

```
/*
 * File:   main.c
 * Author: damie
 *
 * Created on June 21, 2023, 8:58 AM
 */


#include <xc.h>

void PWMsetup(){
    //Set pins to output
    TRISCbits.TRISC7 = 0;   //Set C7 to output
    TRISBbits.TRISB6 = 0;   //Set B6 to output

    RB6PPS = 0x0A;  //Configure to PWM1S1P1_OUT
    RC7PPS = 0x0B;  //Configure to PWM1S1P2_OUT

    PWM1ERS = 0b0000;   //Sets the external reset source to disabled
    PWM1CLK = 0b0010;   //Sets the clock source for PWM - Set to Fosc

    /*Set PWM Period*/
    PWM1PR = 0x0003;    //Number of clock periods in PWM period (Effectively the resolution)
    PWM1CPRE = 0x00;    //Clock pre-scaler (n+1)
    PWM1GIE = 0x00;     //Interrupt register -- Disable/Enable interrupts
    PWM1CONbits.LD = 1; //Reload the PR, P1, and P2 registers on the next cycle


    /*Configure Slices (Slice is one of the dual outputs*/
    PWM1S1CFGbits.POL1 = 0; //Set polarity of output 1, 0 = high true
    PWM1S1CFGbits.POL2 = 0; //Set polarity of output 2, 0 = high true

    PWM1S1CFGbits.PPEN = 0; //Disable Push-Pull to alternate outputs
    PWM1S1CFGbits.MODE = 0b001; //Right align

    PWM1S1P1 = 0x0001;  //Set duty cycle
    PWM1S1P2 = 0x0002;  //Set duty cycle

    PWM1CONbits.EN = 1; //Enable PWM module
}

void loop(){
    Nop();
}

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

1.1 of 1 2023.06.21 10:24:25

I have done some testing, and I found that running at the full 16bit resolution gives you a frequency of ~61Hz. This is certainly way too low. On the other hand, using a 2 bit resolution gives you a frequency of 1MHz which is way overkill but really cool. I have found that the math for determining the speed/ resolution is:

$$\frac{F_{OSC}}{2^n} = F_{PWM}$$

Where n is the resolution in bits. So a 10 bit resolution can give ~3.9KHz

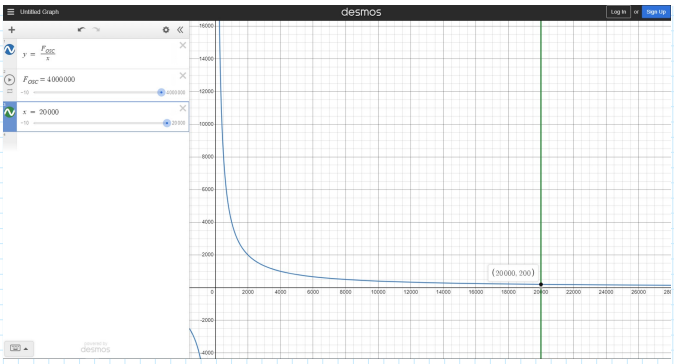
Resolution (btis)	F_PWM	F_OSC	4,000,000.00
16	61.04		
15	122.07		
14	244.14		
13	488.28		
12	976.56		
11	1,953.13		
10	3,906.25		
9	7,812.50		
8	15,625.00		
7	31,250.00		
6	62,500.00		
5	125,000.00		
4	250,000.00		
3	500,000.00		
2	1,000,000.00		

This does not actually have to be controlled in bits.

It can be controlled by a discrete count. To calculate with this, I just need to change the 2^n in the equation to a value.

If I want the resolution as a function of frequency, its just:

$$\frac{F_{OSC}}{F_{PWM}} = 2^n = RES$$



A frequency of 20kHz can be reached with 200 options in the resolution

Setting PWM1PR = 0x00C7; //Number of clock periods in PWM period (Effectively the resolution)

I can get an output that is exactly 20.0kHz on the oscilloscope.

I spoke with Nick, he is thinking that the 10-20kHz range is where we want to be. He also mentioned that we could use an external clock with a higher frequency.

Name: PWMxCLK
Address: 0x461,0x470,0x47F

PWMx Clock Source

Bit	7	6	5	4	3	2	1	0
					CLK[3:0]			
Access					R/W	R/W	R/W	R/W
Reset					0	0	0	0

Bits 3:0 – CLK[3:0] PWM Clock Source Select

CLK	Source	Operates in sleep
1111	Reserved	N/A
1110	CLC4_OUT	Yes(1)
1101	CLC3_OUT	Yes(1)
1100	CLC2_OUT	Yes(1)
1011	CLC1_OUT	Yes(1)
1010	NCO1_OUT	Yes(1)
1001	CLKREF	Yes(1)
1000	EXTOSC	Yes
0111	SOSC	Yes
0110	MFINTOSC (32 kHz)	Yes
0101	MFINTOSC (500 kHz)	Yes
0100	LFINTOSC	Yes
0011	HFINTOSC	Yes
0010	F_{OSC}	No
0001	PWMIN1PPS	Yes(1)
0000	PWMIN0PPS	Yes(1)

Note: Operation during Sleep is possible if the clock supplying the source peripheral operates in Sleep.

Looks like I just need to change the PWM1CLK register to the EXTOSC value.

1:00 lunch
1:30 back

I2C

From my understanding, I2C process is —

1. Host issues start condition
2. Host sends 7-bit address over bus
3. Client recieves address and stores it in the address buffer
4. If Client address matches the address in the buffer Client activates and sends an ACK signal over the bus
5. the Lsb of the address byte corresponds to a R/W command from the host

IF R

1. the next byte from the host will correspond to the number of bytes in the message
2. the client stores this in the CNT register – this will be decremented after each recieved byte
3. Client loads data into TXB register
4. On the next 8 clock cycles data from TXB is transmitted over the bus
5. CNT is decremented
6. Host sends $\overline{ACK}$
7. Repeat 4-6 until CNT = 0

IF W

1. the next byte from the host will be the number of bytes to recad – this will be sent to CNT
2. Client Sends $\overline{ACK}$ over bus
3. Host writes next byte
4. Client recieves byte into RBX and holds clock
5. Client makes data out of RBX and clears it
6. Client releases clock and sends $\overline{ACK}$
7. CNT is decremented
8. repeat 3-7
9. CNT = 0

7. CNT is decremented
8. repeat 3-7
9. CNT = 0

6. Host ends communication

I may be able to read the source code of the arduino library to see if I get more info

https://github.com/arduino/ArduinoCore-avr/blob/master/libraries/Wire/src/Wire.h

Wire.cpp

Wire.h

~\Downloads\Wire.cpp

```
/*
  TwoWire.cpp - TWI/I2C library for Wiring & Arduino
  Copyright (c) 2006 Nicholas Zambetti.  All right reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

  Modified 2012 by Todd Krein (todd@krein.org) to implement repeated starts
  Modified 2017 by Chuck Todd (ctodd@cableone.net) to correct Unconfigured Slave Mode reboot
  Modified 2020 by Greyson Christoforo (grey@christoforo.net) to implement timeouts
*/

extern "C" {
  #include <stdlib.h>
  #include <string.h>
  #include <inttypes.h>
  #include "utility/twi.h"
}

#include "Wire.h"

// Initialize Class Variables //////////////////////////////////////////////////////

uint8_t TwoWire::rxBuffer[BUFFER_LENGTH];
uint8_t TwoWire::rxBufferIndex = 0;
uint8_t TwoWire::rxBufferLength = 0;

uint8_t TwoWire::txAddress = 0;
uint8_t TwoWire::txBuffer[BUFFER_LENGTH];
uint8_t TwoWire::txBufferIndex = 0;
uint8_t TwoWire::txBufferLength = 0;

uint8_t TwoWire::transmitting = 0;
void (*TwoWire::user_onRequest)(void);
void (*TwoWire::user_onReceive)(int);

// Constructors ////////////////////////////////////////////////////////////////

TwoWire::TwoWire()
{
}

// Public Methods //////////////////////////////////////////////////////////////

void TwoWire::begin(void)
{
  rxBufferIndex = 0;
  rxBufferLength = 0;

  txBufferIndex = 0;
  txBufferLength = 0;

  twi_init();
  twi_attachSlaveTxEvent(onRequestService); // default callback must exist
  twi_attachSlaveRxEvent(onReceiveService); // default callback must exist
}

void TwoWire::begin(uint8_t address)
{
  begin();
  twi_setAddress(address);
}

void TwoWire::begin(int address)
{
  begin((uint8_t)address);
}

void TwoWire::end(void)
{
  twi_disable();
}

void TwoWire::setClock(uint32_t clock)
{
  twi_setFrequency(clock);
}
```

What is this

Initialize with device address

end

Set Clock

~\Downloads\Wire.h

```
/*
  TwoWire.h - TWI/I2C library for Arduino & Wiring
  Copyright (c) 2006 Nicholas Zambetti.  All right reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

  Modified 2012 by Todd Krein (todd@krein.org) to implement repeated starts
  Modified 2020 by Greyson Christoforo (grey@christoforo.net) to implement timeouts
*/

#ifndef TwoWire_h
#define TwoWire_h

#include <inttypes.h>
#include "Stream.h"

#define BUFFER_LENGTH 32

// WIRE_HAS_END means Wire has end()
#define WIRE_HAS_END 1

class TwoWire : public Stream
{
  private:
    static uint8_t rxBuffer[];
    static uint8_t rxBufferIndex;
    static uint8_t rxBufferLength;

    static uint8_t txAddress;
    static uint8_t txBuffer[];
    static uint8_t txBufferIndex;
    static uint8_t txBufferLength;

    static uint8_t transmitting;
    static void (*user_onRequest)(void);
    static void (*user_onReceive)(int);
    static void onRequestService(void);
    static void onReceiveService(uint8_t*, int);
  public:
    TwoWire();
    void begin();
    void begin(uint8_t);
    void begin(int);
    void end();
    void setClock(uint32_t);
    void setWireTimeout(uint32_t timeout = 25000, bool reset_with_timeout = false);
    bool getWireTimeoutFlag(void);
    void clearWireTimeoutFlag(void);
    void beginTransmission(uint8_t);
    void beginTransmission(int);
    uint8_t endTransmission(void);
    uint8_t endTransmission(uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t, uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t, uint32_t, uint8_t, uint8_t);
    uint8_t requestFrom(int, int);
    uint8_t requestFrom(int, int, int);
    virtual size_t write(uint8_t);
    virtual size_t write(const uint8_t *, size_t);
    virtual int available(void);
    virtual int read(void);
    virtual int peek(void);
    virtual void flush(void);
    void onReceive( void (*)(int) );
    void onRequest( void (*)(void) );

    inline size_t write(unsigned long n) { return write((uint8_t)n); }
    inline size_t write(long n) { return write((uint8_t)n); }
    inline size_t write(unsigned int n) { return write((uint8_t)n); }
    inline size_t write(int n) { return write((uint8_t)n); }
    using Print::write;
};

extern TwoWire Wire;

#endif
```

```
}

void TwoWire::setClock(uint32_t clock)
{
  twi_setFrequency(clock);
}

/**
 * Sets the TWI timeout.
 *
 * This limits the maximum time to wait for the TWI hardware. If more time passes, the bus is assumed
 * to have locked up (e.g. due to noise-induced glitches or faulty slaves) and the transaction is aborted.
 * Optionally, the TWI hardware is also reset, which can be required to allow subsequent transactions to
 * succeed in some cases (in particular when noise has made the TWI hardware think there is a second
 * master that has claimed the bus).
 *
 * When a timeout is triggered, a flag is set that can be queried with `getWireTimeoutFlag()` and is cleared
 * when `clearWireTimeoutFlag()` or `setWireTimeoutUs()` is called.
 *
 * Note that this timeout can also trigger while waiting for clock stretching or waiting for a second master
 * to complete its transaction. So make sure to adapt the timeout to accommodate for those cases if needed.
 * A typical timeout would be 25ms (which is the maximum clock stretching allowed by the SMBus protocol),
 * but (much) shorter values will usually also work.
 *
 * In the future, a timeout will be enabled by default, so if you require the timeout to be disabled, it is
 * recommended you disable it by default using `setWireTimeoutUs(0)`, even though that is currently
 * the default.
 *
 * @param timeout a timeout value in microseconds, if zero then timeout checking is disabled
 * @param reset_with_timeout if true then TWI interface will be automatically reset on timeout
 *                           if false then TWI interface will not be reset on timeout

 */
void TwoWire::setWireTimeout(uint32_t timeout, bool reset_with_timeout){
  twi_setTimeoutInMicros(timeout, reset_with_timeout);
}

/**
 * Returns the TWT timeout flag.
 *
 * @return true if timeout has occurred since the flag was last cleared.
 */
bool TwoWire::getWireTimeoutFlag(void){
  return(twi_manageTimeoutFlag(false));
}

/**
 * Clears the TWI timeout flag.
 */
void TwoWire::clearWireTimeoutFlag(void){
  twi_manageTimeoutFlag(true);
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity, uint32_t iaddress, uint8_t isize, uint8_t sendStop)
{
  if (isize > 0) {
  // send internal address; this mode allows sending a repeated start to access
  // some devices' internal registers. This function is executed by the hardware
  // TWI module on other processors (for example Due's TWI_IADR and TWI_MMR registers)

  beginTransmission(address);

  // the maximum size of internal address is 3 bytes
  if (isize > 3){
    isize = 3;
  }

  // write internal register address - most significant byte first
  while (isize-- > 0)
    write((uint8_t)(iaddress >> (isize*8)));
  endTransmission(false);
  }


  // clamp to buffer length
  if(quantity > BUFFER_LENGTH){
    quantity = BUFFER_LENGTH;
  }
  // perform blocking read into buffer
  uint8_t read = twi_readFrom(address, rxBuffer, quantity, sendStop);
  // set rx buffer iterator vars
  rxBufferIndex = 0;
  rxBufferLength = read;

  return read;
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop) {
    return requestFrom((uint8_t)address, (uint8_t)quantity, (uint32_t)0, (uint8_t)0, (uint8_t)sendStop);
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)true);
}

uint8_t TwoWire::requestFrom(int address, int quantity)
{
```

```
    using Print::write;
};

extern TwoWire Wire;

#endif


```

```
  // clamp to buffer length
  if(quantity > BUFFER_LENGTH){
    quantity = BUFFER_LENGTH;
  }
  // perform blocking read into buffer
  uint8_t read = twi_readFrom(address, rxBuffer, quantity, sendStop);
  // set rx buffer iterator vars
  rxBufferIndex = 0;
  rxBufferLength = read;

  return read;
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop) {
    return requestFrom((uint8_t)address, (uint8_t)quantity, (uint32_t)0, (uint8_t)0,
(uint8_t)sendStop);
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)true);
}

uint8_t TwoWire::requestFrom(int address, int quantity)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)true);
}

uint8_t TwoWire::requestFrom(int address, int quantity, int sendStop)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)sendStop);
}

void TwoWire::beginTransmission(uint8_t address)
{
  // indicate that we are transmitting
  transmitting = 1;
  // set address of targeted slave
  txAddress = address;
  // reset tx buffer iterator vars
  txBufferIndex = 0;
  txBufferLength = 0;
}

void TwoWire::beginTransmission(int address)
{
  beginTransmission((uint8_t)address);
}

//
//  Originally, 'endTransmission' was an f(void) function.
//  It has been modified to take one parameter indicating
//  whether or not a STOP should be performed on the bus.
//  Calling endTransmission(false) allows a sketch to
//  perform a repeated start.
//
```

```
//  WARNING: Nothing in the library keeps track of whether
//  the bus tenure has been properly ended with a STOP. It
//  is very possible to leave the bus in a hung state if
//  no call to endTransmission(true) is made. Some I2C
//  devices will behave oddly if they do not see a STOP.
//
uint8_t TwoWire::endTransmission(uint8_t sendStop)
{
  // transmit buffer (blocking)
  uint8_t ret = twi_writeTo(txAddress, txBuffer, txBufferLength, 1, sendStop);
  // reset tx buffer iterator vars
  txBufferIndex = 0;
  txBufferLength = 0;
  // indicate that we are done transmitting
  transmitting = 0;
  return ret;
}

//  This provides backwards compatibility with the original
//  definition, and expected behaviour, of endTransmission
//
uint8_t TwoWire::endTransmission(void)
{
  return endTransmission(true);
}

// must be called in:
// slave tx event callback
// or after beginTransmission(address)
size_t TwoWire::write(uint8_t data)
{
  if(transmitting){
  // in master transmitter mode
    // don't bother if buffer is full
    if(txBufferLength >= BUFFER_LENGTH){
      setWriteError();
      return 0;
    }
    // put byte in tx buffer
    txBuffer[txBufferIndex] = data;
    ++txBufferIndex;
    // update amount in buffer
    txBufferLength = txBufferIndex;
  }else{
  // in slave send mode
    // reply to master
    twi_transmit(&data, 1);
  }
  return 1;
}

// must be called in:
// slave tx event callback
// or after beginTransmission(address)
```

```
  return 1;
}

// must be called in:
// slave tx event callback
// or after beginTransmission(address)
size_t TwoWire::write(const uint8_t *data, size_t quantity)
{
```

```
  if(transmitting){
  // in master transmitter mode
    for(size_t i = 0; i < quantity; ++i){
      write(data[i]);
    }
  }else{
  // in slave send mode
    // reply to master
    twi_transmit(data, quantity);
  }
  return quantity;
}

// must be called in:
// slave rx event callback
// or after requestFrom(address, numBytes)
int TwoWire::available(void)
{
  return rxBufferLength - rxBufferIndex;
}

// must be called in:
// slave rx event callback
// or after requestFrom(address, numBytes)
int TwoWire::read(void)
{
  int value = -1;

  // get each successive byte on each call
  if(rxBufferIndex < rxBufferLength){
    value = rxBuffer[rxBufferIndex];
    ++rxBufferIndex;
  }

  return value;
}

// must be called in:
// slave rx event callback
// or after requestFrom(address, numBytes)
int TwoWire::peek(void)
{
  int value = -1;

  if(rxBufferIndex < rxBufferLength){
    value = rxBuffer[rxBufferIndex];
  }

  return value;
}

void TwoWire::flush(void)
{
  // XXX: to be implemented.
}

```

```
// behind the scenes function that is called when data is received
void TwoWire::onReceiveService(uint8_t* inBytes, int numBytes)
{
  // don't bother if user hasn't registered a callback
  if(!user_onReceive){
    return;
  }
  // don't bother if rx buffer is in use by a master requestFrom() op
  // i know this drops data, but it allows for slight stupidity
  // meaning, they may not have read all the master requestFrom() data yet
  if(rxBufferIndex < rxBufferLength){
    return;
  }
  // copy twi rx buffer into local read buffer
  // this enables new reads to happen in parallel
  for(uint8_t i = 0; i < numBytes; ++i){
    rxBuffer[i] = inBytes[i];
  }
  // set rx iterator vars
  rxBufferIndex = 0;
  rxBufferLength = numBytes;
  // alert user program
  user_onReceive(numBytes);
}

// behind the scenes function that is called when data is requested
void TwoWire::onRequestService(void)
{
  // don't bother if user hasn't registered a callback
  if(!user_onRequest){
    return;
  }
  // reset tx buffer iterator vars
  // !!! this will kill any pending pre-master sendTo() activity
  txBufferIndex = 0;
  txBufferLength = 0;
  // alert user program
  user_onRequest();
}

// sets function called on slave write
void TwoWire::onReceive( void (*function)(int) )
```

```
  // alert user program
  user_onRequest();
}

// sets function called on slave write
void TwoWire::onReceive( void (*function)(int) )
{
  user_onReceive = function;
}

// sets function called on slave read
void TwoWire::onRequest( void (*function)(void) )
{
  user_onRequest = function;
}

// Preinstantiate Objects //////////////////////////////////////////////////////

TwoWire Wire = TwoWire();

```

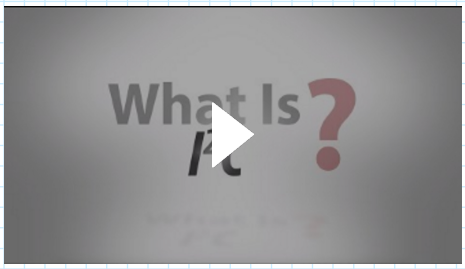
This library is using another library or set of functions - nothing is explicit about the process/sequence is in this code

22/06/23 Time: 7.25

Thursday, June 22, 2023 7:35 AM

Start 7:30

[What Is...I2C?]

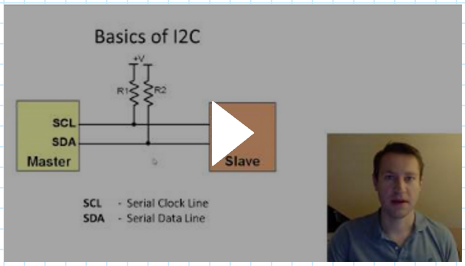


Notes

- If host is writing, client sends ACK
- If host is reading, host sends ACK

- Once started, communication goes until stop condition — there is not a stop condition after each byte

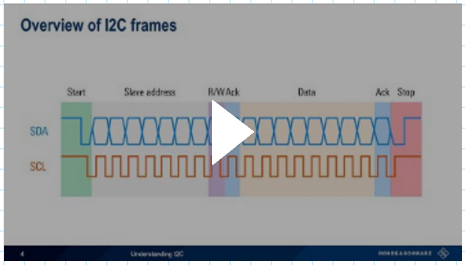
[What is I2C, Basics for Beginners]



Notes:

- Apparently clock stretching is uncommon now

[Understanding I2C]



Notes:

- No videos have mentioned a CNT byte to indicate the length of the transmission

PWM control

From watching videos on I2C, I found a common way of writing memory to a slave is to send the memory address in the first byte/bytes, then the value to be transmitted

I looked up how to write to a memory address in C — I found this

If you work with hardware register in embedded system then standard way is:

```
int volatile * const p_reg = (int *) 0x8000000;
*p_reg = 0x1234;
```

You can have a lot of problems with optimizing compiler, if you omit **volatile**

Where they create a pointer to the address and then use that pointer to set the value

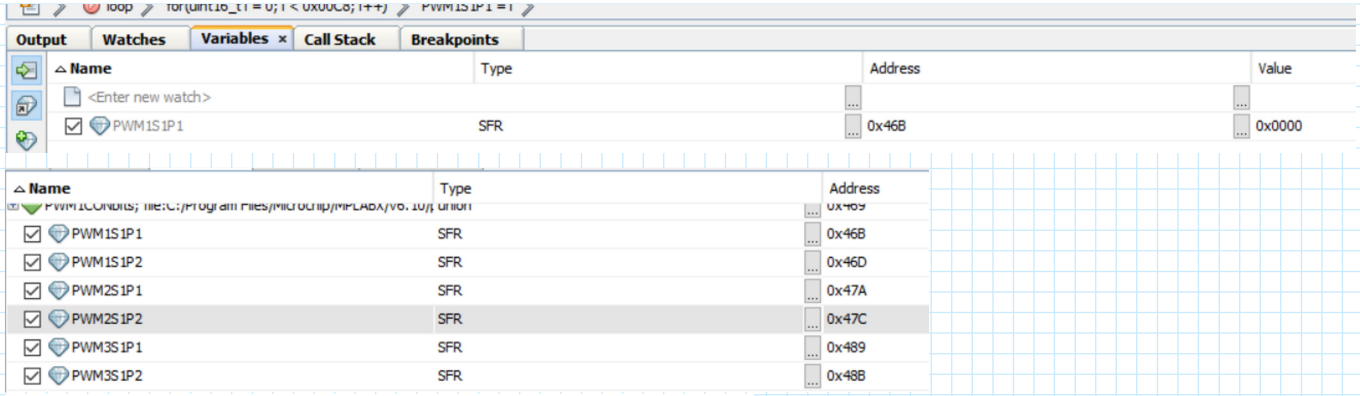
The PWM values I got working yesterday are in

PWM1S1P1 — Address: 0x46B

PWM1S1P2 — Address: 0x46D

OF coorse these registers dont have an address listed...

IF I dont have the address I am not sure I can do it this way...
The debugger will give me the address




```c
void loop(){
    int bufferl = 0x46B;
    int volatile * const pwml1 = (int *) bufferl;
    int volatile * const pwml2 = (int *) 0x46D;
    for(uint16_t i = 0; i < 0x00C8; i++){
        /*NOTE: PWM must be disabled before changing duty cycle*/
        PWM1CONbits.EN = 0; //Enable PWM module
        *pwml1 = i;
        *pwml2 = (0x00C8 - i)/2;

        PWM1CONbits.EN = 1; //Enable PWM module

        __delay_ms(10);
    }
}
```

This script uses a variable that hold an sddress to create a pointer to that address and then write to that address. And it works.

The addresses are all 12bit — then I have the pwm duty cycle which would likely be 8bit

Write PWM

Start | X X X X X X | 0 | 1 | 0 0 0 0 X X X X | 1 | (ACK) | X X X X X X X X | 1 | (ACK) | X X X X X X X X | 0 | (NACK) | Stop

Address | R/W | ACK | PWM Addr Msb | PWM Addr Lsb | duty cycle

I will need to store the Addresses on the PI to be able to write to everything

```c
    /*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
    }
```

This is how I understand the I2C. The software waits for the receive buffer to fill. Once it is filled, generate the

ACK signal, then store the data from the buffer. Then clear the buffer and wait for it to be filled again.

```
/*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
    }
}
```

This is how I think the read would work. The client clears the register and then loads the data into the TXB buffer. Then waits until the host acknowledges the data. I am thinking I shouldn't need the acknowledge and the waiting for the buffer to be clear. I may get rid of the ack because I was concerned if there is no ACK then the MCU will be stuck in that loop forever.

Nick should be ordering the IMU and logic converters so hopefully I will have those next week

With these logic converters, I hope to make more progress with the I2C as I will be able to test and have some feedback. I hope to use the debugger interface to try and see where the data is going and when flags are being set

Once I have the I2C working, I will have everything I need to finish up the Reaction wheel & Magnetorquer MCU, and the Sun sensor PWM MCU.

Then I will likely need to go back to the overall control system, I will likely start by setting up the RPI to communicate with using I2C to the microcontrollers, and ensuring all of the commands work

Once I get some commands on the RPI that work, I might make a library for each MCU to allow the ADCS software to interface in a more human readable manner. I doubt I will be the last one working on on the software, and it may be easier to understand and write software if there is like a

```
PWM.setMagX(duty cycle);
PWM.setMagZ(duty cycle);
PWM.setMotX(duty cycle);
    etc.
```

Instead of having to do the I2C and then pick the right memory addresss and send the data in the right orders etc.

Similarly, I will likely make a function to read the ADCs that will parse the 12 sensor readings into individual values for each sensor.

```
ADC.read()
```

This would also need to return a pointer the the array of readings values as a function can not return an array

I was also considering including an option to read an individual sensor Not sure if this would be THAT useful — but Nick had mentioned wanting to vault this software and only interface with it so the added functionality might be useful at some point

```
ADC.read(int ID=0){ };
```

I could include a parameter that defaults to 0 and CAN be set to read a single channel

Once I have that working, as well as the I2C on the PI

ADCS IO

Output
- Magnetorquers X3 ✓
- Motor X2 ✓
- UHF ✗

Input
- Sun sensor X12 ✓
- Magnometer X3 ✓
- Gyroscope X3 ✓
- accelerometer X3 ✓
- UHF (TLE) ✗

The UHF can connect over the I2C bus, however I was planning to connect it to the UART Pins - so I will need to get that working — I do have the USB to UART cable

- accelerometer X ✓
- UHF (TLE) X

The UHF can connect over the I2C bus, however I was planning to connect it to the UART pins – so I will need to get that working – I do have the USB to UART cable

I could try to connect the PI to my pc this afternoon and see if I can shell into it

Lunch 12:00
Back 12:30

MagX = PWM1S1P1 = 0x46B
MagY = PWM1S1P2 = 0x46D
MagZ = PWM2S1P1 = 0x47A

MotX = PWM3S1P1 = 0x489
MotY = PWM3S1P2 = 0x48B

i2cPWMsim.ino

```ino
#include <Wire.h>

void setup() {
  // put your setup code here, to run once:
  uint8_t = addr = 0x04;
  Wire.begin(addr);
  Wire.onReceive(writePWM);
}

void writePWM(){
  uint16_t addr;
  uint8_t duty_cycle;
  if(3 < Wire.available()){
    addr = Wire.read();

    addr = addr << 8;

    addr |= Wire.read();

    duty_cycle = Wire.read();
    Serial.print("Address: ");
    Serial.print(addr);
    Serial.print("\tDuty cycle: ");
    Serial.println(duty_cycle);
  }
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(100);
}
```

i2cADCsim.ino

```ino
#include <Wire.h>

void setup() {
  // put your setup code here, to run once:
  uint8_t addr = 0x03;  //Sets the client address
  Wire.begin(addr);
  Wire.onRequest(reportADC);
  Serial.begin(9600);
}

void reportADC(){
  uint16_t readings[12] = {
    0x01,
    0x02,
    0x03,
    0x04,
    0x05
    0x06,
    0x07,
    0x08,
    0x09,
    0x0A,
    0x0B,
    0x0C
  };

  for(int i = 0; i < 12; i++){
    Wire.write(readings[i]);
    Serial.print("Writing reading ");
    Serial.println(i);
  }
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(100);
}
```

I created these two Arduino scripts to act as the PIC chips. This way I should be able to get something running on the PI to do the I2C. Both systems use 5V so I should not need the logic converter. I could make a test script to make sure my plan works and get the PI working properly so I can use it to test with the PIC when I get the converter.

I tried to connect the PI to the circuit and run the open command but I got an error. I believe that I need to pull the SDA and SCL lines high with a 4.7k resistor. I don't see any in the room here.

I found some 5.0k Resistors and used those to pull the SDA and SCL high. I also powered the arduinos off the PI and connected a common ground so the voltage levels would be the same. But I still get the same error when I try to communicate to a device on the bus. I am not sure why it is not working. I checked the pins with a meter and they are being pulled high.

I may need to turn on the I2C port in… it is possible that it is off by default.

That was the problem. I enabled the I2C bus and got a handle of 0 instead of -29 which means it should be connected.

When I use the lgI2cReadByte function on my PWMsim arduino, it only reads the first value. The arduino is sending all of them, however the I2C library is configured to only read one byte.
https://raspberry-projects.com/pi/programming-in-c/i2c/using-the-i2c-interface

Leave 3:15

23/06/23

Time: 6.5

Friday, June 23, 2023 8:06 AM

Start 8:00AM

The raspberry pi Zero 2W arrived. My plan with it was to attempt to install the drivers and software. To do this, I am going to need a micro SD card reader & maybe a usb – micro usb adapter to get a keyboard connected.

Task list
- Get RPI I2C to read multiple bytes
- Find correct version of Ubuntu & Ximea Software
- Install software onto RPI
- Test UART Serial connection on a RPI
-

I tried to use a different method of doing I2C on the PI to be able to read multiple bytes. I followed the example that I found yesterday.
https://raspberry-projects.com/pi/programming-in-c/i2c/using-the-i2c-interface

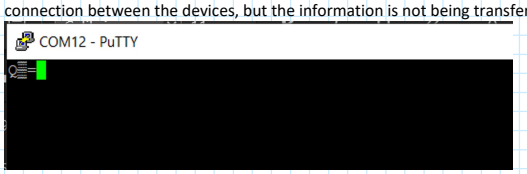
However once again, the kernel that is running on this PI does not have the linux/i2c-dev.h installed. So I cant compile or run the code. I can see that again the file I need is in every other kernel but I don't know how to change. I really want to just install the ubuntu version that I need for the camera and hopefully all these stupid issues will just resolve themselves.

I need the microSD card reader to swap that tho, and I would like to put the test scripts I have created on this PI onto a thumb drive so I can just move them over onto the new OS.

I have all of these things at home, so maybe that is something I will do tonight.

That doesn't leave me with as much to work on during the day today. I might try and get some female-female jumpers from Megan and attach the UART adapter to the PI and see if I can shell into the RPI 4

I connected the UART and configured the config.txt and cmdline.txt files. I am able to get a serial connection between the devices, but the information is not being transferred properly.



That was supposed to be a "t"

So it is detecting when information is being sent properly, it just is not able to interpret that information porperly. Also when I boot the pi, I can see a bunch of information being printed to the serial monitor but none of it is readable.

I have checked and both BAUD rates are at 115200

I tried to turn the BAUD rate on both devices down to 9600 to see if possible the adapter chip couldn't go that high. I know I connected to the motor drivers using this cable at 9600.

I got a similar result where the output was not readable.

Motor Driver

Previously I had found that the motor driver had a torque/current control option.

Range of functions	
Operating modes (RS Versions)	Position, speed and torque control with setpoint specification via interface or analog. Position control with Gearing Mode or stepper motor operation. Operation as Servo Amplifier in voltage controller mode

But another datasheet says that there is only "current limiting" not current torque control. There is on the higher version of the driver. But I just double checked and the first image is for a datasheet on the V2.5 version of the motor driver.

	Generation V2.5		Generation V3.0	
	MCxx 3002	MCxx 3003/06	MC 5004	MC 5005/10
Voltage ranges	▪ Motor: max. 30V ▪ Electronics: max. 30V, optionally separated		▪ Motor: max. 50V ▪ Electronics: max. 50V, separated standard	
Continuous current	2A	3 / 6A	4A	5 / 10A
Peak current	3A	10A	12A	15 / 30A
Motor types	▪ MCDC: DC + Encoder ▪ MCBL: BL + A-Hall ▪ MCLM: LM + A-Hall		▪ DC motors with pos. / speed sensor ▪ BL motors with pos. / speed sensor ▪ LM motors with pos. / speed sensor	
Speed and position sensors	see motor types		▪ DC motors: incremental[1], AES Encoder[1], SSI encoder[1], analog value (potentiometer/tachometer) ▪ BL/LM motors: D-Hall, D-Hall + Encoder[1], A-Hall, AES encoder[1], SSI encoder[1], analog value (potentiometer/tachometer)	
Inputs/outputs	MCDC: DigIn: max. 5 DigOut: max. 1 AnIn ±10V: 1 Optional connection of a second reference encoder (Gearing mode). Not all I/Os available depending on wiring.	MCBL/MCLM: DigIn: max. 3 DigOut: max. 1 AnIn ±10V: 1	DigIn: 8 DigOut: 3 AnIn ±10V: 2 Optional connection of a second reference encoder (Gearing mode).	DigIn: 3 DigOut: 2 AnIn ±10V: 2
Communication	RS232 or CANopen		USB, RS232 and/or CANopen, EtherCAT	
Controller	Position, speed, current limiting		Position, speed, current / torque	
Operating modes	▪ Depending on the interface variant, position, speed and current control with setpoint input via the interface or analog (RS)		▪ Profile Position mode (PP) and Profile Velocity mode (PV), taking into account profile settings ▪ Cyclic Synchronous Position, speed or torque (CSP, CSV or CST) ▪ Analog input for position, speed, torque or voltage (APC, AVC, ATC, volt)	
Profile operation	Linear trapezoidal profiles in all operating modes		Linear or sin² speed in PP and PV modes	
Autonomous processes	Available in the versions with RS232 interface		Up to eight sequential programs in all versions, with optional password protection	

[1] with and without Line driver

Current tasks

- Configure I2C on RPI to read/write n bytes

Current tasks

- Configure I2C on RPI to read/write n bytes
  - issues with kernel/Libraries — should resolve when Linux Distro is changed
- Configure I2C on PIC
  - need logic converters for testing
  - need host device to test
- Set up script to handle the PWM and interface with I2C
  - get I2C working
- Set up script to handle the ADC and I2C interface
  - get I2C working
- Install camera drivers on RPI Zero 2W
  - need microSD card to flash new Distro
- Shell into RPI over UART — No prerequisites
- Better understand ADCS system
  - talk with Nick to go over my understanding
- Read values from IMU
  - Need to get an I2C host working first
  - Need IMU
- Get working software on PI to read ADC, write PWM and read IMU readings
  - Need I2C working
  - Need IMU
- Develop ADCS software
  - Need better understanding of ADCS system
  - Need working software to interface with peripherals
- Capture image in software
  - Need 2W with correct drivers and OS
  - Need Xmega camera
-

Lunch 12:00-12:30

Creating Scripts for Sun Sensor MCU and PWM MCU

I have some scripts made that prove the functionality of the PWM and ADC, but they are not set up to be used in the actual MCUs on board. I cant finish the full script without have the I2C working, but I can work on the components of the script that will need to interface with the I2C.

For the PWM, I want to create a function that takes the PWM address and then writes the duty cycle to that address. The problem I am running into with this is the PWM module needs to be turned off to update the duty cycle - or at least that is what I found in my testing.

To turn off the PWM for one specific module I also need the address of the PWMxCON register. The other option would be to turn all PWM modules off and then back on. This is certainly not ideal.

I am wondering if there is a pattern between the duty cycle address and the con address. For example, maybe the CON register is always 3 addresses before or something.

Address	Name	Bit Pos.	7	6	5	4	3	2	1	0
0x00 ... 0x045F	Reserved									
0x0460	PWM1ERS	7:0						ERS[3:0]		
0x0461	PWM1CLK	7:0						CLK[3:0]		
0x0462	PWM1LDS	7:0						LDS[3:0]		
0x0463	PWM1PR	7:0	PR[7:0]							
		15:8	PR[15:8]							
0x0465	PWM1CPRE	7:0	CPRE[7:0]							
0x0466	PWM1PIPOS	7:0	PIPOS[7:0]							
0x0467	PWM1GIR	7:0							S1P2	S1P1
0x0468	PWM1GIE	7:0							S1P2	S1P1
0x0469	PWM1CON	7:0	EN					LD	ERSPOL	ERSNOW
0x046A	PWM1S1CFG	7:0	POL2	POL1			PPEN		MODE[2:0]	
0x046B	PWM1S1P1	7:0	P1[7:0]							
		15:8	P1[15:8]							
0x046D	PWM1S1P2	7:0	P2[7:0]							
		15:8	P2[15:8]							
0x046F	PWM2ERS	7:0						ERS[3:0]		
0x0470	PWM2CLK	7:0						CLK[3:0]		
0x0471	PWM2LDS	7:0						LDS[3:0]		
0x0472	PWM2PR	7:0	PR[7:0]							
		15:8	PR[15:8]							
0x0474	PWM2CPRE	7:0	CPRE[7:0]							
0x0475	PWM2PIPOS	7:0	PIPOS[7:0]							
0x0476	PWM2GIR	7:0							S1P2	S1P1
0x0477	PWM2GIE	7:0							S1P2	S1P1
0x0478	PWM2CON	7:0	EN					LD	ERSPOL	ERSNOW
0x0479	PWM2S1CFG	7:0	POL2	POL1			PPEN		MODE[2:0]	
0x047A	PWM2S1P1	7:0	P1[7:0]							
		15:8	P1[15:8]							
0x047C	PWM2S1P2	7:0	P2[7:0]							
		15:8	P2[15:8]							
0x047E	PWM3ERS	7:0						ERS[3:0]		
0x047F	PWM3CLK	7:0						CLK[3:0]		
0x0480	PWM3LDS	7:0						LDS[3:0]		
0x0481	PWM3PR	7:0	PR[7:0]							
		15:8	PR[15:8]							
0x0483	PWM3CPRE	7:0	CPRE[7:0]							
0x0484	PWM3PIPOS	7:0	PIPOS[7:0]							
0x0485	PWM3GIR	7:0							S1P2	S1P1
0x0486	PWM3GIE	7:0							S1P2	S1P1
0x0487	PWM3CON	7:0	EN					LD	ERSPOL	ERSNOW
0x0488	PWM3S1CFG	7:0	POL2	POL1			PPEN		MODE[2:0]	
		7:0	P1[7:0]							

0x0483	PWM3CPRE	7:0	CPRE[7:0]								
0x0484	PWM3PIPOS	7:0	PIPOS[7:0]								
0x0485	PWM3GIR	7:0								S1P2	S1P1
0x0486	PWM3GIE	7:0								S1P2	S1P1
0x0487	PWM3CON	7:0	EN						LD	ERSPOL	ERSNOW
0x0488	PWM3S1CFG	7:0	POL2	POL1				PPEN		MODE[2:0]	
0x0489	PWM3S1P1	7:0	P1[7:0]								
		15:8	P1[15:8]								
0x048B	PWM3S1P2	7:0	P2[7:0]								
		15:8	P2[15:8]								
0x048D ... 0x049B	Reserved										
0x049C	PWMLOAD	7:0							MPWM3LD	MPWM2LD	MPWM1LD
0x049D	PWMEN	7:0							MPWM3EN	MPWM2EN	MPWM1EN

It is always 2 registers before the P1 register for that Module. However if the P2 address is given it is 4 registers before…

I could definitely do this with some conditional logic, but that is really messy. Ideally I don't even need to turn the PWM off. I know there was an option to reload on the next cycle which I had enabled, but that didn't seem to be working. Maybe I can focus on that and try to get it to change the cycle without needing to turn off the PWM.

Bit 2 – LD Reload Registers

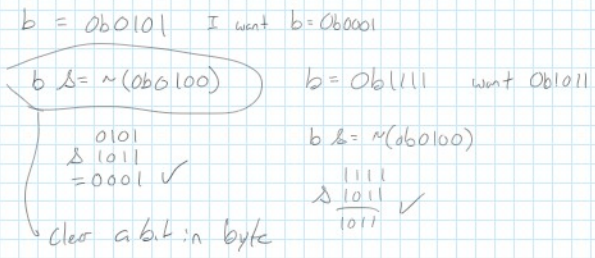
Reload the period and duty cycle registers on the next period event

Value	Description
1	Reload PR/P1/P2 registers
0	Reload not enabled or reload complete

This bit to me sounds like it would make the cycle update once it finishing a cycle

I do have this set to 1 in my software.

I tried setting it to 0 to see what difference it would make. I did not see a difference on the scope.




```
void setPWM(int address, uint8_t duty_cycle){
    int temp = address << 4;
    int conReg;
    if(temp == 0x046){
        conReg = temp >> 4 | 0x0009;
    }
    if(temp == 0x047){
        conReg = temp >> 4 | 0x0008;
    }
    if(temp == 0x048){
        conReg = temp >> 4 | 0x0007;
    }

    conReg &= ~(0b10000000);    //Clear the enable bit of the register
    int volatile * const dutyReg = (int *) address;
    *dutyReg = duty_cycle;
    conReg |= 0b10000000;       //Set the enable bit of the register
}
```

It is not super pretty, but this should work. It clears the last 4 bits (1 character in hex) to determine which PWM it is. Then depending on the PWM module, it sets the register for the PWMxCON register and then clears and sets the enable bit.

It did not work. But it is behaving similar to when the PWM is not being disabled while changing duty cycle. Maybe my system did not work as expected. I will try to use the EN method in that functions to ensure that's whats broken

Using the regular method for setting and clearing the enable made the function work fine. Meaning my method is not working properly.

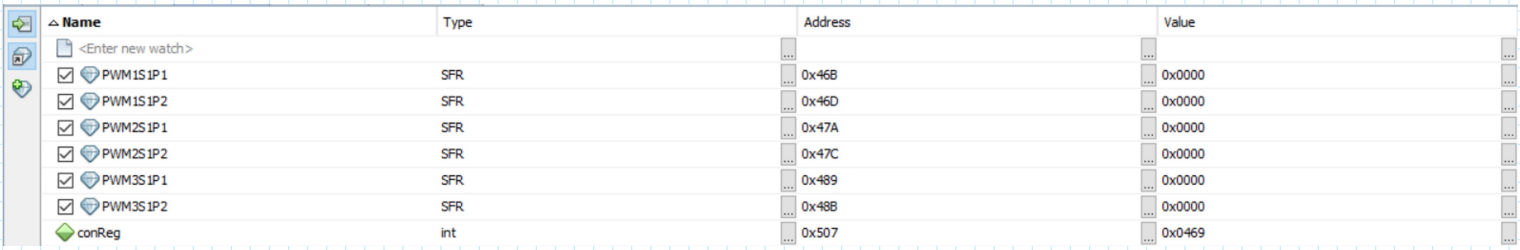
Output | Watches | Variables × | Call Stack | Breakpoints

Name	Type	Address	Value
PWM1S1P1	SFR	0x46B	0x0000
PWM1S1P2	SFR	0x46D	0x0000
PWM2S1P1	SFR	0x47A	0x0000
PWM2S1P2	SFR	0x47C	0x0000
PWM3S1P1	SFR	0x489	0x0000
PWM3S1P2	SFR	0x48B	0x0000
address	int	0x501	0x046B
duty_cycle	unsigned char	0x503	NUL; 0x0
temp	int	0x50B	0x46B0

I guess my shift symbols are backwards. Temp has an extra 0 as opposed to removing the B

That did not solve the issue.

But temp is holding the expected value now, so one bug down.



Name	Type	Address	Value
<Enter new watch>			
PWM1S1P1	SFR	0x46B	0x0000
PWM1S1P2	SFR	0x46D	0x0000
PWM2S1P1	SFR	0x47A	0x0000
PWM2S1P2	SFR	0x47C	0x0000
PWM3S1P1	SFR	0x489	0x0000
PWM3S1P2	SFR	0x48B	0x0000
conReg	int	0x507	0x0469

conReg is also holding the expected value.

OH
I was changing the value of the conReg variable instead of creating a pointer to the memory address.

```
void setPWM(int address, uint8_t duty_cycle){
    int temp = address >> 4;
    int conReg;
    if(temp == 0x046){
        conReg = temp << 4 | 0x0009;
    }
    if(temp == 0x047){
        conReg = temp << 4 | 0x0008;
    }
    if(temp == 0x048){
        conReg = temp << 4 | 0x0007;
    }
    int volatile * const pConReg = (int *) conReg;
    *pConReg &= ~(0b10000000);    //Clear the enable bit of the register
    int volatile * const pDutyReg = (int *) address;
    *pDutyReg = duty_cycle;
    *pConReg |= 0b10000000;       //Set the enable bit of the register
}
```

This function works for PWM1S1P1

I don't want to change the wiring to test others, but I will try the other addresses and check what values for the conReg address are determined using my conditional statement.

PWM1S1P2 works.
PWM2S1P1 works.
PWM2S1P2 works.
PWM3S1P1 works.
PWM3S1P2 works.

All of the possible duty cycle registers are determining the correct CON register for that PWM module.

This is kind of an ugly solution but it is much more efficient than also sending another register over I2C when I can determine the register from the one sent

Leave 3:00