Report: Feedback Controls

Final Project

**Group Members: Ross Mitchell, Krista Stone, Matt Foster, Erin Wiles**

**Class: CSCI 255 Introduction to Embedded Systems**

**Date: 12/15/2014**

Our final project is to design and program a feedback control system for Montana Tech’s Robotics Mining Club. The primary task is to collect and deliver data from multiple robotic sensors and actuators to the Main CPU. This system has several components: a Main CPU, a Master microcontroller and several Slave microcontrollers. Communication between components is serial, either I2C or UART. (See Figure 2.)

For our microcontrollers, we are using Texas Instrument’s MSP430 G2553. Since I2C is designed for communication between one master and multiple slaves, we designated one MSP430 as Master, while the rest were programmed to be Slaves. In this design, each Slave is programmed to collect data from sensors or actuators on the robot or used in conjunction with the robot.

For this feedback system, we gather a distance measurement from the laser range finder (LRF) and pulse width modulation (PWM) signal from a servos motor. The servos motor rotates the platform to which the LRF is secured. Our PWM signal is basically a rotation measurement to determine where the LRF is pointing. The LRF distance measurement data coupled with the PWM signal will be used to track the robot as it moves inside the mining competition arena. (See Figure 1.)

LRF

Servo

Robot

Robot

Figure 1. LRF Tracking Robot in Mining Competition Arena

UART is a serial communication protocol which transfers an ASCII character across either the transmit (TX) or receive (RX) line. In our design, the Main CPU transmits commands from the user to the Master MSP430 G2553. The user designates which Slave sends data by typing in a unique character or character string for each slave (i.e. the Main CPU sends the slave addresses to the Master MSP430 G2553). The Master MSP430 G2553 then uses I2C to send the slave address and receive the measurement data from the designated slave. This I2C data is converted by the Master MSP430 G2553 and sent to the Main CPU across UART.

The user interacts with Master program by running PuTTY on the Main CPU. To properly run PuTTY, the user must designate the serial line (e.g. COMM4) and the baud rate [bps]. The baud rate is a measurement of the symbols per second, in our case it is 9600 characters per second.

I2C is a serial communication protocol designed to send bytes (an 8-bit number) of data, one bit at a time. This process is also known as a “bit-bang”. I2C requires 2 lines: a data line (SDA) which both sends and receives bits and a clock line (SCL) to count off each data bit. The master’s clock controls the timing. In our system slaves are capable of sending multiple bytes.

Since there may be multiple slaves (up to 128), each slave must have its own unique address. For our code, the slave addresses are in hex: 0x48, 0x4A, 0x4C … etc. Note that the slave addresses are even i.e. the least significant bit is 0 in binary. An even address designates the slave to write data according to I2C protocol.

USB

Master MSP430

P1. 6 I2C SDA P1.1 UART RX

P1.7 I2C SCL P1.2 UART TX

VCC

GND

“Programming” computer

Main

CPU

CP2102

RX

TX GND

VCC

USB

GND

10kΩ

10kΩ

Slave MSP430

P1.6 I2C SDA VCC

P1. 7 I2C SCL GND

Slave MSP430

P1. 6 I2C SDA VCC

P1.7 I2C SCL GND

LRF

P1.1 UART RX

P1.2 UART TX

LaunchPad Reminders

1. REMOVE P1.6 jumper to LED2
2. Place TX/RX jumpers parallel with board writing

PWM

Px.x

PSEUDOCODE

Any sensor or actuator

Figure 2. Feedback Control Programming Schematic

We programmed the Slaves and the Master in C using Code Composer Studio 5.4. You can download the program here:

<http://processors.wiki.ti.com/index.php/Download_CCS>

(Note that you will need to register with TI to download.)

For the most part, we modified existing example code provided by TI. While Launchpad and Code Composer Studio are meant to work out of box, we found the UART signal would conflict with the USB communication. Besides interfering with the debugging program, it also interfered with our mouse. If the mouse is jumping all over the screen, it is because UART is interfering with USB communication.

We solved this conflict with the aid of a USB to UART Bridge, CP2102. Essentially, this chip splits off the UART communication from the rest of the USB communication. To use this bridge, it is necessary to install the appropriate drivers. Find the CP2102 driver here:

<http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx>

With CP2102, we had to implement 2 computers: one to program, which we will call the “programming” computer and one to serve as the Main CPU. We were able to program and debug the microcontrollers, no longer did we have to rely solely on voltage measurements with the oscilloscope. When the feedback system is free-running the “programming” computer and its USB connection will be removed.

PuTTy allows the user to handle UART communication. With PuTTY running, the user may enter and/or read out the characters between the "main" CPU and the master MSP430. Find the (compressed) PuTTY file here:

<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

UART sends ASCII characters serially (one-by-one). Each character on a keyboard has an ASCII code. Find the ASCII coed here:

<http://www.asciitable.com/>

If you do not know which port the Cp2102 is connecting to, use Command Prompt and enter the command >mode. The available and recognized ports will appear, you may need to remove and reinsert the CP2102 to determine the COMM port.

Now, that we have given some background on each device and communication method we can now describe the set-up configuration. Figure 1 illustrates the wiring and perhaps most helpful the pin designation and names. We will provide helpful reminders too.

We wasted a lot of time and effort over jumpers: out of the box the LaunchPad connects specific pins with jumpers. Save yourself time and frustration by removing the jumper connecting P1.6 to the LED on every MSP430. It's impossible to implement I2C without this step! The LED must pull the pin low or hi un-expectantly. Also, to use UART on the MSP430 rotate both jumpers connecting TX and RX to be set parallel to the writing on the board.

There are 2 lines for I2C communication: the master clock signal (P1.7) and data line (P1.6). Both of these lines are pulled high (VCC) with 10kΩ resistors. The color code on for this resistance is black brown orange. The last band is tolerance, silver or gold is fine.

When our master free-runs, we can see the clock signal and data signal using an oscilloscope. If Code Composer Studio is acting flakey (as it do), use an oscilloscope to see the state or bits moving across the serial lines. This method helped us realize that our code was working but our debugger was not. Although less precise than reading registers, the oscilloscope was a lifesaver.

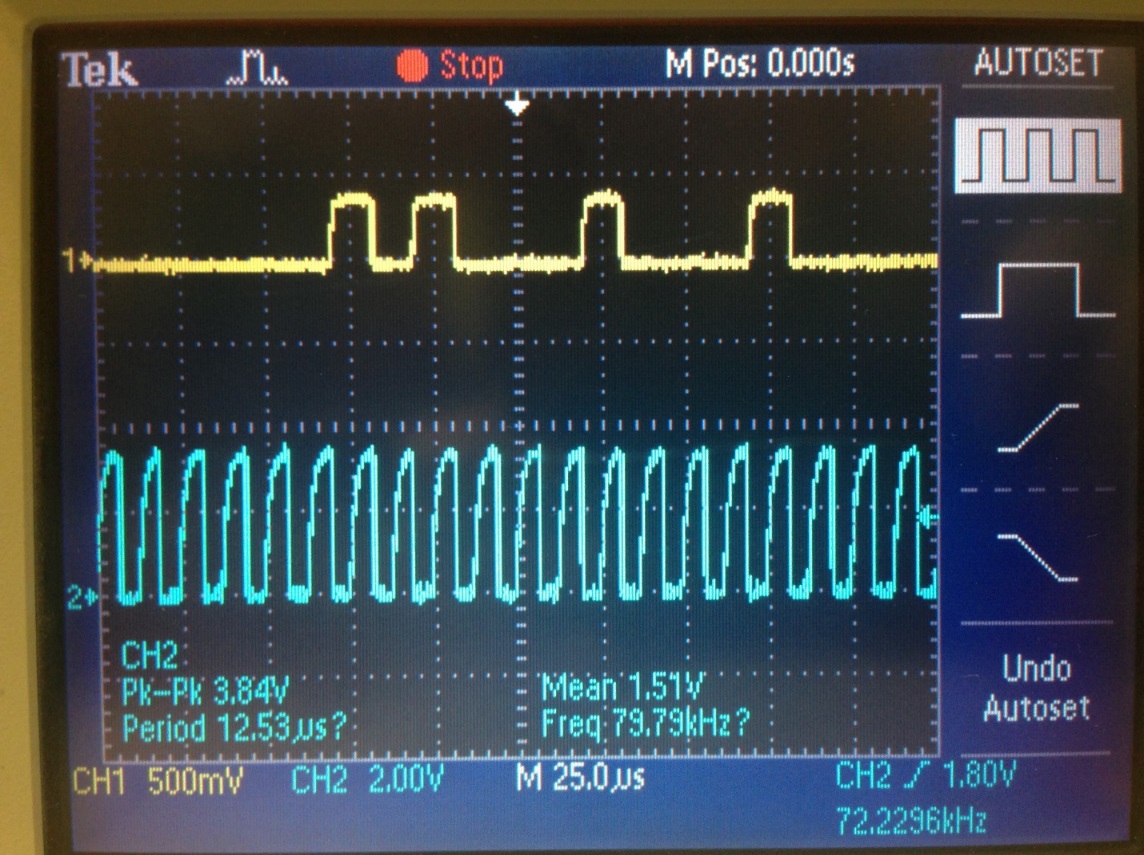


Figure 3. I2C Data and Clock

The figure above is a snapshot of the oscilloscope screen during I2C communication. The blue line is the clock signal: 0 is ~0V and 1 is ~3.84V. The yellow line is the data signal, it is much smaller ~300mV. Alternatively, you may connect a probe to any UART transmit line.

All circuit components, including microcontrollers must share common ground and VCC. Since the oscilloscope is a voltage measuring tool, the probes must share common ground with the circuit. All of these components, except for the CP2102 use 3.3 V for VCC. (The CP2102 Bridge uses 5V but since it is powered by the Main Computer, the different voltages requirement is an non-issue).

Now we can discuss our code and the structure of our Master and Slave programs. Currently, we have 2 working versions of Master code, one to receive the slave address from the Main CPU and another to receive the slave data and transmit that data to the Main CPU. The data transmission follows the following time line:

1A. UART - Main CPU request specific Slaves

Character Sequence or Single Character

2A. I2C - Slave address designated by Master and acknowledged by Slave

1. Transmit Most Recent Data
2. I2C - Start condition sent by Master
3. I2C -Byte loaded into Buffer by Slave

* Each bit transmitted one at a time by Slave
* Each bit received one at a time by Master

1. UART – Convert and transmit byte to character string by Master to Main CPU
2. I2C - Completed byte received by Master
3. I2C - Stop Condition sent by Master
4. Repeat steps a-d for each byte sent.
5. Reset appropriate flags and settings

Meanwhile, every Slave is updating its data and ready to be transmitted:

1B. Receiving data from the LRF via UART

2B. Updating storage Storing LRF data coupled with PWM signal.

Unfortunately, we have not been able to merge steps 1A and 2A into the same program. The slave address is set by both the master and slave MSP430 with the following command:

UCB0I2CSA = 0x4A; // Slave Address initialization

When we set the slave address manually in the initialization settings, we are able to go between different slave addresses, but we must shutdown the microcontrollers (by removing the USB) and rebuild the program. We suspect that initialization settings for UART and I2C conflict and/or multiple slaves are attempting to communicate at the same time. This could also be due to how we structured our code: ISRs may take too long or unable to be called. Alternatively, it may be that the state machine must be reset:

UCA0CTL1 &= ~UCSWRST; // \*\*Initialize USCI state machine\*\*

We are still troubleshooting how to change slave address dynamically, the problem may seems trivial but this chewed up (perhaps wasted) quite a bit of time. It may be fruitful to research and possibly implement the Bus Pirate as a replacement of the Master MSP430:

<https://code.google.com/p/the-bus-pirate/>

<http://dangerousprototypes.com/>

Several factors prevented us from fully testing data retrieval on the Slave side. We are still including example code for Parts 1A and 1B. We regret not dividing tasks equally and accountability between group members. Perhaps the biggest waste of time was not removing the P1.6 jumper to the LED. Anyone one else using the MSP430 LaunchPad needs to: ”REMOVE THAT JUMPER!”

Now we will discuss specific details: since it is both easier to program and discuss code modularly, we will discuss initialization settings, interrupts and functions. For the most part our initialization settings configure clocks and timers, Timers A and B are used by UART and I2C. PWM is currently coded to use Timer A. PWM requires a Timer and will need to be integrated into our code.

Right now, our main clock oscillates at 1MHz. This value in combination with the baud rate determines the modulation settings: UCA0BR0 and UCA0BR1. One calculates the baud rate by dividing the clock frequency by the baud rate. One then converts this value to an 8 bit value. UCA0BR0 is the MSB an 8 bit value and so if your divide value exceeds 255, use UCA0BR1 as the over flow bit. Any remainder is round up or down. The following link is an online tool to calculate the settings:

[http://mspgcc.sourceforge.net/baudrate.html](https://techmail.mtech.edu/owa/redir.aspx?C=Eo7ixjwr6Uixv3VKlulMIfvl7g6u6tEIbRH-_w2EV0E9GlhDzeVRO4J5qEqDXNRHz8LC4U6kFok.&URL=http%3a%2f%2fmspgcc.sourceforge.net%2fbaudrate.html)

We would like to run the clocks as fast as possible, 16MHz, this part is straightforward:

BCSCTL1 = CALBC1\_16MHZ; // Set DCO

DCOCTL = CALDCO\_16MHZ;

However, we also wanted to increase the baud rate of UART signal from 9600bps to 1Mbps by changing the modulation settings. When we ran the program, both the clock signal and UART TX increased in frequency. However the UART character string did not read out on the PuTTY window. (Yes, the COMMx PuTTY settings were changed).

We designed our code to transmit data from the Slave to the Master (via I2C), then to transmit the data from the Master to the Main CPU (via UART). We set-up the main loop to be called by either the UART interrupt or I2C interrupt.

IE2 |= UCA0RXIE; // Enable UART RX interrupt

IE2 |= UCB0RXIE; // Enable I2C interrupt

It is interesting to note that that even though I2C interrupt is designated receiver ‘RX’ and timer B, ‘B’, the ISR is designated transmitter ‘TX’, both timer A and B ‘AB’. The ‘RX’ and ‘TX’ makes a certain amount of sense, since data is received and transmitted on the same line. The I2C & UART appear to use both Timer A and Timer B, how this is done is unclear (to us).

The MSP430 designates pins for UART and I2C communication, see Table 1. Since both types transfer data serially, a buffer is loaded by both protocols. In the case of UART a single character, e.g. ‘A’ is loaded, in the case of I2C a single byte (8-bit number) is loaded. Inside both ISR types the following commands can be found:

UCA0RXBUF='#'; // UART RX Buffer

UCB0TXBUF = \*PTxData++; // I2C RX Buffer

When using the transmitting buffer, always check that the buffer flag is ready:

**while** (!(IFG2&UCA0TXIFG)); // UART TX Buffer ready?

UCA0TXBUF=slave\_addresses[ii]; // SEND BACK slave

Table 1. MSP430 Serial Communication Pin Designations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Variable Name | Pin | Nickname | Serial Type | ISR (Master) | ISR (slave) |
| UCA0RXD | P1.1 | TX | UART | USCIAB0RX | USCIAB0RX |
| UCA0TXD | P1.2 | RX | UART | n/a? | USCIAB0TX |
| UCB0SDA | P1.6 | Data | I2C | USCIAB0TX | USCIAB0TX |
| UCB0CTL0 | P1.7 | CLK | I2C | USCIAB0TX | USCIAB0TX |

The bulk of the UART TX and I2C data transmission (Steps 2A-4) occurs in the I2C interrupt. The counter RXByteCtr counts down to the last byte to be received. The code determines how many bytes will be sent by setting the number of slave bytes to the 1st Bit sent by the slave. The ISR also calls 2 functions, one to convert the bytes to string and the other to load the UART TX buffer.

**#pragma** vector = USCIAB0TX\_VECTOR

**\_\_interrupt** **void** **USCIAB0TX\_ISR**(**void**)

{

RXByteCtr--; // Initialized main while loop

**if** (RXByteCtr){

**if** (ii == 0){

slaveBytes=UCB0RXBUF; // Byte# fits 1st Slave Byte

RXByteCtr=slaveBytes-1;

}

i2c\_storage[ii]=UCB0RXBUF; // Move each I2C byte to storage

convert(i2c\_storage[ii],ii); // Convert byte to string

load\_uart(ii);

// while (!(IFG2&UCA0TXIFG)); // UART TX buffer ready?

// UCA0TXBUF = 'x'; // Test Buffer

ii++;

}

**else**{

ii=slaveBytes-1; // Unclear to team, why necessary

i2c\_storage[ii]=UCB0RXBUF; // Move final RX data to PRxData

convert(i2c\_storage[ii],ii);

load\_uart(ii);

**while** (!(IFG2&UCA0TXIFG)); // USCI\_A0 TX buffer ready?

UCA0TXBUF='\n'; // format easier reading

UCB0CTL1 |= UCTXSTP; // Generate I2C stop condition

ii=0;

\_\_bic\_SR\_register\_on\_exit(CPUOFF); // Exit LPM0

}

}

The convert function takes the I2C value & assumes it is no more the 5 digits in base 10, i.e. bit\_position=10000. The code then converts the byte value into a 5 character string. It’s important to add 0x30, to character values, since that is where 0 begins in ASCII code.

**void** **convert**(**int** i2c\_val, **int** ii){

**volatile** **unsigned** **int** jj=0,temp=0,bit\_position=10000;

**for** (jj=0;jj<slaveBytes;jj++){

temp=i2c\_val/bit\_position;

i2c\_val-=temp\*bit\_position;

bit\_position/=10;

temp+=0x30;

uart\_storage[ii][jj]=temp;

//temp=0;

}// MSB -> LSB

Since multiple bytes are sent, a multiple dimensional matrix is used to store each byte’s character strings. The variable ii indexes each byte, and the variable jj loads each character into the buffer.

**void** **load\_uart**(**int** ii){

**volatile** **unsigned** **int** jj=0;

**for**(jj=0;jj<slaveBytes;jj++){

**while** (!(IFG2&UCA0TXIFG)); // UART TX buffer ready?

//UCA0TXBUF = 'x';

UCA0TXBUF = uart\_storage[ii][jj]; // load buffer one character at a time

}// Load buffer character by character

}

We believe this UART TX and I2C data transmission code is easily adaptable. Now we can discuss the Master code which receives the slave address from Main CPU via UART. The UART RX ISR (Step 1A) stores each character into a character array, slave address. We have allocated 20 characters for a slave address. However, the code is able to handle fewer number of characters.

**volatile** **unsigned** **char** slave\_address[20];

We expect the character sequence to be entered as a hex string, e.g. ‘#0x48’, since we would like to convert that hex string to a hex value and set the slaves address.

We index through the character array using the variable kk. This variable may also be used to determine the length of the address.

**volatile** **unsigned** **int** ii=0,kk=0;

The UAR RX ISR program on the Master checks for a specific sequence of characters, ‘#xxxxx\n.’ The ‘#’ sign is the 1st character, x are variable number of characters and enter is the final character '\n'.

**#pragma** vector=USCIAB0RX\_VECTOR

\_\_interrupt **void** **USCI0RX\_ISR**(**void**)

{

**if** (UCA0RXBUF == '#') // Current character '#'

{

slave\_address[0]=UCA0RXBUF; // Set '#' as 1st Character

kk=1; // index read for 2nd char

}

**if** (slave\_address[0] == '#' && UCA0RXBUF != '#')

{

**if** (UCA0RXBUF != '\n')

{ slave\_address[kk]=UCA0RXBUF;// Store current character

kk++; // Next index

}// Current not '\n'?

**else**{

// store '\n'? no?

// slave\_flag=1;// set flag when complete address entered

}

}// '#' 1st Char & current not '#'

**else**;

}

After we enter the characters via UART, we pause the program to check the values in the character array slave\_address[]. By hovering over the variable, Code Composer Studio will show the current values stored.

We then played the program, entered a different address via UART. Paused the program, hovered over slave\_address[] and the different values appeared. Code composer Studio will highlight the values yellow when they change state.

Therefore, we can only demonstrate that UART RX characters have been received. We would like to write a function which converts a character string to a value, currently we are held up by variable type disagreement.

Since we are including untested code in our code examples, it is prudent to discuss the current design and programming problems. Currently our slave only runs the example code from TI, ‘msp430g2xx3\_uscib0\_i2c\_ll.c’. To complete the task we would need to program the slave to collect data from the LRF and couple this data with a PWM signal.

It appear as though the LRF example code (modified by RJ Hallet) must be calibrated with the LRF device to receive accurate measurements over UART. Regrettably, we have not connected the slave to the LRF device.

Even so, it appears as though this code is too big, when we merge the LRF example code with the existing I2C Slave Write code, the compiler warns there is not enough memory. The error reads,

"error #10099-D: program will not fit into available memory. placement with alignment fails for section "USCIAB0RX" size 0x4 .Available memory ranges:INT07 size: 0x2 unused: 0x2 max hole: 0x2 "

Looking at the LRD example code ISR, It appears as though the ISR was not set-up correctly. We will have to troubleshoot this error.

The original author of the code writes in Spanish, it’s not clear how our environment differs from the original coding environment. Additionally, this code includes a lot of burning time, we want the data from all sensors to update as quickly as possible.

We have not determined how we will control and/or record data from the servo motor with the PWM signal. All of the PWM example code use pins P1.6 and P1.2, which we currently use to transmit and receive data. Although we have not pinned it down, we suspect that we may use P2.4 to control the PWM signal.

To measure the rotation of the servo motor we may need to implement a magnetic encoder, like Avogo’s AEAT-6012-A06 and use the PWM signal to control the platforms. That Avago encoder requires Synchronous Serial Interface (SSI), this would require another chip between the slave and magnetic encoder.

After we determine what exactly we are measuring, we must figure out how to couple the data. Obviously, we will store both data point into the same I2C transmit array, however it seems as though interrupts or functions storing LRF distant measurement and PWM rotation measurement must be independent. Can we assume the times each data point is stored is close enough to synchronous?

A related issue is clock speed, UART, I2C and the PWM signal all use either Timer A, Timer B or both. The UART on the slave side is set communicate with the LRF device at 115200 bps. The LRF Example code is running the clock at 16 MHz instead of 1MHz. Since the PWM is also time dependent, we must configure its modulations setting properly: UCA0BR0 and UCA0BR1. Bryce Hill did the calculation and a full pulse width on PWM should take 32000 clock cycles at 16 MHz.  We must use an unsigned integer to calculate the PWM rate.  Bryce Hill expects the value to be between 16000 and 32000. He also believes this value will be sent back over the UART, how this is possible is not clear. Finally, we would need to test how the I2C serial communication is affected by the change in clock speed.

We covered quite a bit of material doing this project. We coded UART, I2C, and PWM to all be controlled by an individual MSP-430 coded to be the Master. Multiple bytes can now be sent across I2C. The number of bytes received from each slave will be determined by the first byte of data sent from that slave. The PWM code will need to be integrated into the rest of the code we have developed. Our team will continue to work these issues because we are all members of the Montana Tech NASA Robotic Mining Club.