CMPE 121 Project Report

Ariel Anders

Fall 2011

PROJECT OVERVIEW

In this technological day and age, we take electronics for granted; every day we incorporate electronic devices seamlessly in our life without realizing that is there is typically a tiny computer, a "microcontroller", making them function. This course teaches the fundamentals necessary to create microcontroller-based electronics. First, we learn how to interface the microcontroller with hardware devices; such as wiring the microcontroller to a power source or input/output device. Then, we learn how to program the microcontroller to communicate with these hardware devices and perform specific tasks. In our lab, we get hands on practice by making a microcontroller-based device. Although the system designed in the course is unsophisticated, the ***process*** of constructing this system teaches the fundamental of microcontroller-based design: it creates a platform of tools, skills, and protocols that form the basis of our understanding to one day complete more complex engineering design projects.

PROJECT SPECIFICATIONS

The minimum hardware requirements for this project were to design a microcontroller-based system to meet the following specifications:

68HC11E1, Microcontroller
8k x 8 of ROM
8k x 8 of external SRAM
RS-232C serial data interface to the lab PC.
SPI serial interface
Power requirements:

> Capable of operating over a 7 to 12V DC input range from the lab workbench supply, or a single 9V battery. A 3-terminal linear regulator and heat sink is included in your kit. Include in your engineering notes and final report and estimate of the relative temperature rise with and without use of the heat sink.

The minimum software requirements for this project were:

1. Boot code will be written entirely in assembly language and, besides doing any one-time setup and initialization will also have an option, controlled by a hardware jumper, that fully checks all internal and external static ram for bad memory locations. This code must be able to check for stuck address lines and report on any bad memory locations by sending their address over the serial port to the lab PC connected to the HC11's serial port. If no errors are found, it should make an announcement stating this over the serial port. After certifying SRAM, the assembly code will then setup a stack and jump to system and application-level code.

2. System support code will all be written in ANSI C consisting of two device drivers and a system timer:

    a) Device driver 1: design and code a fully interrupt driven device driver for the internal UART capable of supporting full-duplex serial communications at several different baudrates (settable with dipswitches or jumpers or SPI Rom) for communicating with the lab PC.

    b) Optional device driver 2: design and code a driver that fully supports I/O using the internal SPI controller. This should be coded to support whatever specific external SPI device you elect to add.

    c) System timer: design and code an interrupt-driven system clock (timer) capable of being used for general housekeeping and other real-time purposes. It must also be capable of also keeping accurate chronological time.

3. Application's level code:

    a) Add a task to set / modify the current time and write it to the lower-right corner of the screen in

    HyperTerminal on the lab PC (using the serial port device driver).

    b) Add a task that uses the SPI bus to communicate with a peripheral of your choosing to do something useful

Finally yet importantly, there was an optional enhancement option to substitute for 2.b and

3.b.

SYSTEM BOARD OVERVIEW

The design of my microcontroller-based system comes directly from how the MC68HC11E1

works. The following is a brief overview of some of the defining factors that influenced my

design.

The MC68HC11E1 is an 8-bit microcontroller unit (MCU) with several peripheral functions and capabilities provided. In this lab, we are making use of most of its peripheral functions including its "expanded bus" mode. The "expanded bus" mode lets us interface the MC68HC11 with peripheral hardware, such as the external sram and rom.

The MC68HC11 has a 64K-byte "memory-mapped IO" address space: all memory devices (internal and external), and I/O ports are treated the same with their own individual memory address. Each memory location maps to an 8-bit register: one "byte" of data. Hence, the 64 K-byte address spaces has 64K memory registers which is $2^6*2^{10} = 2^{16}$ memory registers and has 16 address bits A0-A15.

Since the MCU has 1-byte data registers, it is intuitively clear that it would have eight pins for data transferred to or from the MCU. To save the number of pins used, the MCU has a multiplexed address/data bus to hold the lower 8 bits of the address, A0-A7, and the data bits, D0-D7. During each machine cycle, the MCU will assert the lower bits of the address on the multiplexed data line, and then during the second half of the cycle it will either read or write to the data pins. The Address Latch Enable is an output signal from the MCU that to "latch" the data into the 74HC573 (an 8-bit latch). Once A0-A7 is 'latched', the second half of the bus cycle can begin where the MCU reads or writes to this data bus. Since we have multiple peripheral devices, there has to be a method to enable or disable a single device at a time; a disable device is in "tri-state", a state of high-impedance that will not cause bus contention. This "method" of selecting the peripheral device is "Address Decoding".

ADDRESS DECODING

The core of address decoding deals with the MCU's memory map. The memory map is simply what it sounds like: a mapping of where each register is located. By default there are a few memory locations predetermined by the MCU. These include the internal eeprom, internal ram, and 64 bytes of registers. Additionally, the "boot vector" for the MCU is $FFFF, this is the address of the first instruction the MCU will load every time upon reset; this follows that the external ROM must have the $FFFF mapped to its memory.

In my project, I use only 16K-bytes of SRAM and ROM even though there is 32K-bytes of SRAM and 128k-bytes of ROM at my disposal. This much memory for a small micro system was overzealous and I decided to use a smaller address space to simplify the address decoding, but still have enough extra memory space if I needed it (more than the 8K-bytes specifications).

The memory map for my project is:

CLASSIC MEMORY MAP

$0000 – 01FF	512 Bytes Internal RAM
$0200 - 0FFF	Unused
$1000 - 103F	64 byte registers
$1040 - 3FFF	Unused
$4000 - 7FFF	16 K-bytes External SRAM
$8000 - B5FF	Unused
$B600 - B7FF	512 bytes EEPROM
$B800 - BFFF	Unused
$C000 - FFFF	16 K-bytes External ROM

DETAILED MEMORY MAP:

Address:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
512 Bytes Int. RAM	0	0	0	0	0	0	0	/ -- 512 bytes Internal Ram --/									$0000 - $01FF
64 byte registers	1	0	0	0	0	0	0	1	0	0	/-- 64B registers --/						$1000 - $103F
16 KB Ext SRAM	**0**	**1**	/ ----------16K –bytes external SRAM ----------/														$4000 - $7FFF
512 bytes EEPROM	1	0	1	1	0	1	1	/ -- 512 bytes internal ROM --/									$B600 - $B7FF
16 KB Ext. ROM	**1**	**1**	/ ----------16K-bytes external ROM ----------/														$C000 - $FFFF

In expanded bus mode, the internal memory has the highest priority. Even with the peripheral SRAM or ROM enabled, the MCU will ignore any data on the data bus and read/write to its internal memory space only. Because of this, I can enable the SRAM whenever A15 is low. Since the chip select is an active low input I can use "linear address decoding" and tie A15 directly to SRAMS's chip select. Similarly, I can invert A15 to ROM chip select; since I am using an inverter this is "partial address decoding".

The two memory devices use Intel style timing. To simplify my read and write enable signals I used a 3-8 decoder 74HC138 that was enabled whenever the MCU's 'E' signal was high. See my lab notebook page 13 for a detailed description and derivation for these signals.

UART

The UART means "Universal Asynchronous Receiver/Transmitter". In this lab we implemented the RS232 protocol using the microcontroller peripheral devices/functions and a MAX232 chip to convert unipolar 0 to 5 volt output from the MCU to bipolar -10 to 10 volt input to the RS232 serial port and vice versa.

The RS232 is a standard protocol for implementing the asynchronous data transfer and receive. It is "asynchronous" because both devices have their own clock cycle. The clock rate or "baud rate" is set to the same value for both systems.

In my project, I am required to write to the serial port using assembly code in the boot program and in the application driver code .c programs.

To utilize the microcontroller's UART system I write directly to the MCU's internal registers.

1) Baud Register
    a. To set the Baud Rate I write to the Baud Register. This register has "pre-scale" bits predetermined by the manufacturer. To use the 9600-Baud rate, I wrote to this register hex value 0x30.
2) The SCI (serial control interface) Status register
    a. In this register, I can enable the Transfer Enable (TE) and Receive Enable (RE) bits, and I can setup the interrupts for more efficient programs. (see interrupt section)
3) SCDR the Serial Control Data Register
    a. The input and output received from this register are mapped to this single address. This does not cause an error because there are actually two registers mapped to this address: a data received register and a data transmit register. Implementing a "read" or "write" command will direct the MCU to access only one of the registers.

MEMORY CHECKER

STUCK ADDRESS LINES

Our Boot code needs to check for stuck address lines. These lines may or may not be able to switch due to improper wiring or a physical chip flaw. The protocol I used to find stuck address lines was described by the Teaching Assistant for this course Jas Condley during a programming section on November 29, 2011.

Since I am using 16k-bytes RAM and ROM I only use 14 address pins: A0 - A13. My RAM starts at address 0x4000 until $7FFF. First I write to 0xFF to 14 registers starting at the base memory address 0x4000. Then I write the register offset to 14 of these registers. The following table shows this protocol:

Register	1)Write 'FF'	2)Write offset
R0 = 0x00	0xFF	--
R1 = 0x01	0xFF	0x01
R2 = 0x02	0xFF	0x02
R3 = 0x03	0xFF	0x03
R4 = 0x04	0xFF	0x04
R5 = 0x05	0xFF	0x05
R6 = 0x06	0xFF	0x06
R7 = 0x07	0xFF	0x07
R8 = 0x08	0xFF	0x08
R9 = 0x09	0xFF	0x09
R10 = 0x0A	0xFF	0x0A
R11 = 0x0B	0xFF	0x0B
R12 = 0x0C	0xFF	0x0C
R13 = 0x0D	0xFF	0x0D

Then read address at offset zero. If the value at R0 is not 0xFF then the numerical value it shows is the address of the pin that is stuck! The downside of this protocol is that it can find one stuck address pin at a time. Once it finds a stuck address pin it prints an error message to

the UART so that you can fix the problem and re-run the code to find the next stuck address pin. Otherwise, it reads 0x00 at the offset 0x00 and prints a success message to the UART.

CHECK MEMORY

The check memory also written in assembly languadge is to check all RAM registers available for use. First, it check sthe internal s-ram located at $0000 – $01FF for bad memory registers. Then it checks my external s-ram $4000-$7FFF for

APPLICATION CODE: WRITING IN C

In this lab, we use the Cosmic 68HC11 cross development tools to program the MCU. This compiler translates ANSI C code into machine code for the MCU. My program had two additional files: "Ivectors.c" and "IO.h". The test program for our class provided both of these. In my program, I modified the Ivectors.c to account for the timer and uart interrupts I integrated with my program. Last but not least, there was a linker ".lf" file the compiler uses to translate the main C application test.c with the assembly code rts.s into one machine code program stored in test.s19. Then, I burned my program test.s19 into my Read-Only-Memory chip using the EEPROM burner in lab.

My main c application: test.c has the following setup:

```
** define constant values
** declare uninitialized global variables
main {
        **set up initial values
        **write to 64-byte registers to initialize interrupts and enable some of the
        microcontrollers peripheral functionality
        infinite for loop{
        ** execute code that should be happening continuously throughout the program
        }
```

}

@interrupts{

** write ISRS to handle interrupt events

}

SETTING THE BAUD RATE

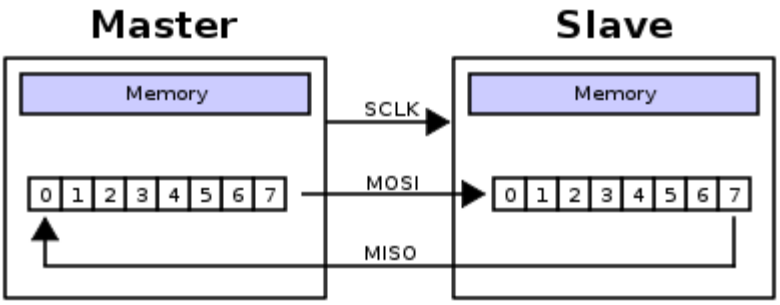
On my board, I have two active low switches. These are connected to pin 5 and 6 on the MCU's Port E. Whenever my update baud rate function is called it will read port E and figure out how the status of these pins. The following Baud Rates are possible:

SWITCH POSITON [1 2]	BAUD RATE
↑↑	9600
↑↓	4800
↓↑	2400
↓↓	1280

UART INTERRUPTS

My test.c program is fairly standard to the general implementation in this course with the exception that I handle full–duplex serial port communication using a single circular buffer.

I based my design off how the Serial Peripheral Interface (SPI) works. The following is an image of how the data is transferred with SPI:



[1]

In my main program, if there is data inside the buffer then it will print the first element of the data out and increment the pointer. If there is data received from the UART It will push the data in to the last element of the queue and increment the pointer.

Note: when the first and last pointers are equal, there is no data to print out.

The queue is circular so once the first or last pointers reach the end of the buffer, they get re-set to the head of the buffer. It differs from SPI in the sense that it is clearly asynchronous; the rate things are pushed out of the buffer is not equal to the way things get pushed into the buffer. Also, in SPI there is always data transfer between both systems on each SCLK transition- in my implementation this is not the case. The queue will dynamically grow and shrink by adjusting its pointers.

It is fully interrupt driven and does not use polling to load or print out data by carefully handling and setting enable flags throughout my interrupt routine.

The problem with this design is that it always causes an "echo" regardless of what the main program is doing. Therefore, I set a command to enable or disable echo(this will be addressed later).

[1] http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

SYSTEM TIMER

I decided to use the main counter overflow interrupt to create my timer. The main counter increments every 500ns. It has 16 bit resolution, hence it overflows every

$$Overflow\ period = 500ns * 2^{16} = 32.77ms$$

Therefore if I have a tick variable keep track of the overflows then

$$\frac{1000\ milliseconds}{1\ second} * \frac{1\ overflow}{32.77\ milliseconds} = \mathbf{30.5\ oveflows\ per\ second}$$

To implement the timer, I have a TOF (timer overflow flag) set each time every 32.77ms. I implement a tick count to increment a global seconds variable:

```
@interrupt timer overflow{
        ** tick ++
         if (tick == 29) then seconds ++
        if (tick == 30) then seconds ++
        if (seconds == 60) then  increment minutes
        if (minutes == 60) then increment hours
        }
```

Then I use VT-100 escape sequences to print the time to the lower right corner.

SUMMARY: RUNNING THE MAIN PROGRAM

1) Run the boot code for testing memory and stuck address lines by setting switches to:

   ↓↓

2) Start the main .c program by setting the switches to: ↑↑

   a. The instructions for the running the program is printed out on teraterm. If you want to view the timer, click '1'. When you do this, you will find a time that does

not start at 0. That is because the time has already started! You can reset the time by inputting 'r'.

b. You can reset the baud rate by 'b'. If you look at the code, the Baud rate is initialized at the beginning based on the switches. However, since I needed a method to run the ram checker or main.c the switches are forced to be set UP causing the program to start in standard 9600 mode***. For best results follow the steps in this order to reset the baud rate:***

i. Set teraterm to 9600 and run program

ii. Se the switches based off this table:

SWITCH POSITON [1 2]	BAUD RATE
↑↑	9600
↑↓	4800
↓↑	2400
↓↓	1280

iii. Type into teraterm 'b'

iv. This will print out "gibberish" because you now have the wrong baud rate set. Go to the teraterm settings and set the baud rate to your desired value.

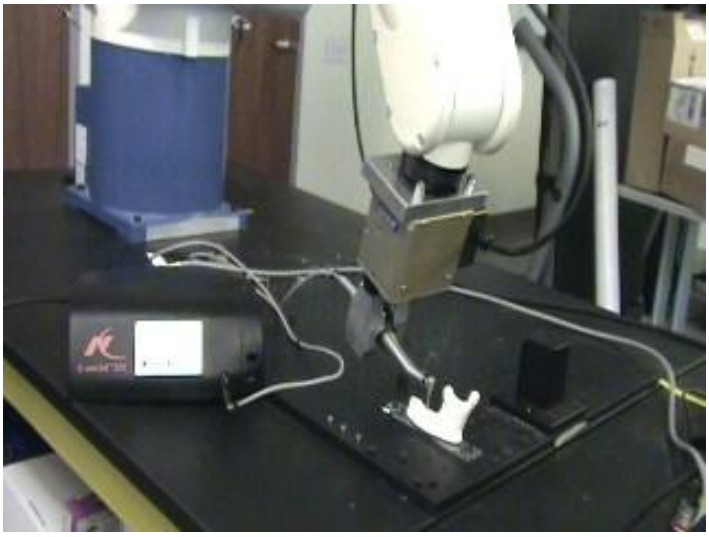
c. Selecting 'e' will make it so your input will not echo to the screen. If you select 'e' again will echo to the screen. To test this, I recommend using CAPS LOCK so that you do not accidently hit 'e' for common words.

d. Hardware mode is to run the expansion. See next section

EXPANSION

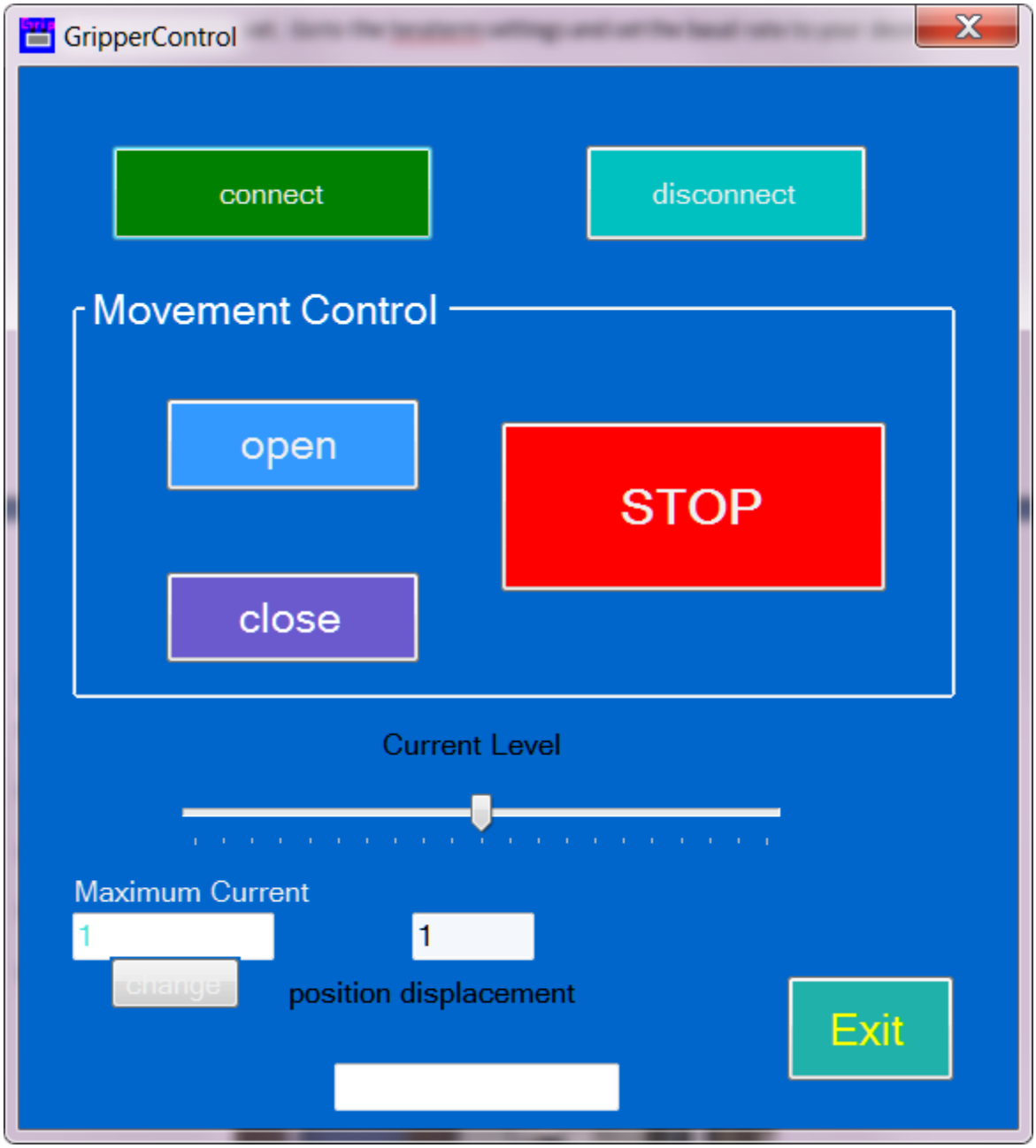
MOTIVATION AND BACKGROUND INFORMATION

My expansion for this project was using my board to control the mechanical grippers that I use in my research project at the Robotic lab. These parallel plate grippers, Schunk PT-AE70, are designed to hold a dental tool and are attached to the end of the robot. My research project is to execute autonomous dental procedures using the dental tool. The following picture shows the grippers in action:



The grippers have large gray "jaws" attached that were designed by a member of the RADR senior designed group last spring.

Previously I created a GUI to control these mechanical grippers:



The grippers program must run during the procedure; however, I do not want to use the computer to control these because I want the computer to solely implement dynamic

equations and update the robot in real-time. Any delay of running external programs has made it impossible to receive real-time processing, but this is a good step in the right direction.

The grippers are programmed with standard RS232 protocol. The current configuration for the mechanical grippers I used is set for 9600 Baud (this can be changed) and have an ID of 2. Previously I used a dynamic link library .dll to write an abstracted c++ program. However, I recalled that the documentation for finding the hex commands to write to the gripper was well documented in the "Schunk Motion" application notes.

The data transferred uses this protocol:

1.3.2 Data frame

The data frame of the motion protocol always contains the following elements.

- D-Len (1 byte)
- Command Code (1 byte)

Figure 1.1: Data frame

D-Len (data length) indicates the number of subsequent useful data items including the command byte. The data frame consists of one byte, so that a motion protocol message can consist of maximum 255 data bytes.

Since I use RS232 I had to adhere to this special requirement:

1.3.3 Special requirements with RS232

As the RS232 was not intended as a bus system when devised, a number of elements must be added to the data frame in order to enable several modules to communicate through a single serial interface.

Figure 1.2: RS232 data frame

The data frame is followed by two bytes (group/ID) indicating the module to be targeted or the module that sent the response. Only the first three bits of the first byte are used. The second byte constitutes a unique module ID. => up to 255 different modules can be addressed. The first three bits of the first byte are encoded as follows:

- 0x03 Error signal from module
- 0x05 Message from master to a module
- 0x07 Response from module

The CRC value is from a check sum algorithm and data is interpreted in "Little Endian" Format

with the IEEE 754 standard for binary floating-point arithmetic.

MY SOLUTION:

I control the grippers based off the amount of current is applied (current is almost directly

proportional to force). I implement "move cur" commands and set the current anywhere from

zero to 14 amps. In my project, I use fragile equipment and have experimentally found the range of current I desire is from 0.25 to 2 amps.

I decided to forgo calculating the check sum algorithm or data conversion to IEEE 754 by computing (by hand) a set of 8 commands for opening the current at even intervals from 0.25 to 2 amps. I created a similar set for closing the gripper and then I found the command for stopping the gripper.

HEX COMMANDS

STOP: 02 04 10 82 10 82 00 08 03 02 04 41 00 45 03

Current (A)	OPEN HEX COMMANDS
0.25	02 04 46 0B 08 00 00 80 3E 1C 03
0.5	02 04 46 0B 08 00 00 00 3F 9C 03
0.75	02 04 46 0B 08 00 00 40 3F DC 03
1	02 04 46 0B 08 00 00 80 3F 1D 03
1.25	02 04 46 0B 08 00 00 A0 3F 3D 03
1.5	02 04 46 0B 08 00 00 C0 3F 5D 03
1.75	02 04 46 0B 08 00 00 E0 3F 7D 03
2	02 04 46 0B 08 00 00 00 40 9D 03

Current: (A)	CLOSE HEX COMMANDS
0.25	02 04 46 0B 08 00 00 00 80 DD 03
0.5	02 04 46 0B 08 00 00 00 BF 1D 03
0.75	02 04 46 0B 08 00 00 40 BF 5D 03
1	02 04 46 0B 08 00 00 80 BF 9D 03
1.25	02 04 46 0B 08 00 00 A0 BF BD 03
1.5	02 04 46 0B 08 00 00 80 BF 9D 03
1.75	02 04 46 0B 08 00 00 E0 BF FD 03
2	02 04 46 0B 08 00 00 00 C0 1E 03

I created a double array of char, rather an array of character strings. Each array had eight strings for eight different move current commands.

To make the gripper implement a desired current based move I had to send one of these hex strings over the UART.

System Board Hardware

I used two push buttons and a potentiometer to execute tSSShese commands. One button will execute an 'open' command the other will execute a 'close' command. These buttons are active low on pins E0 and E1.

Then, I have a potentiometer on E7: the scrollable knob on the board is simply a varying voltage divider. The voltage at E7 will be proportional to the amount the knob is rotated.

Software

In my main event loop I constantly update the value of the global variables buttons and trimpotscale. When these change, I call an update hardware function. First I decipher whether an "open" or "close" function is desired. Then, I will obtain the Analog to Digital converted value on pin E7. This value is from 0 to 255 since it has 8 bit resolution and starts counting at zero. Based off this number I run a current lookup command to obtain the desired hex string. I load the hex string to the output buffer and let my previous UART interrupt routines do the rest of the work!

Modifications to Gripper Hardware

I connected the two systems with a double male RS232 connector. To my great dismay this did not work! However, I was confident that my board was working and printing out the correct commands so I decided to investigate the connection instead.

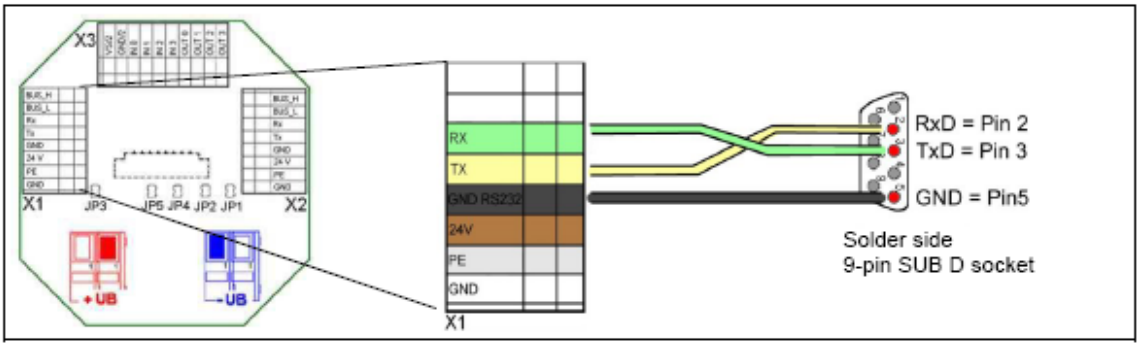


Figure 10 Connection board: Terminal strip X1 and connection to 9-pin SUB D socket

Connection	Terminal	SCHUNK cable color
RS232 interface	Tx	yellow
	Rx	green
	GND (Rx/Tx)	shield (from Rx/Tx)
Logic connection	24V	brown
	PE	shield
	GND	white
Output power supply	+UB	red
	-UB	blue

Table 11 RS232 connection: Assignment of terminal strip X1

I used the oscilloscope to verify the connecting pins and then I used jumpers to directly connect the TX, RX, and GND signals. It worked!

Then I tried using my male-male RS232 connector and it still did not work! What I did not realize was that I had to mirror the connecting pins for one of the connectors. The gripper power converter box has capability for two serial ports; however, only one was being use. The solution was simple: I rewired the extra serial port to interface with my board.

FUTURE WORK

My "stop" command needs some extra control logic. When both buttons are pushed the stop command is printed transferred to the grippers, however, when I release both buttons it's impossible for me to release them at exactly the same time and my main event loop reads one of the buttons down, calls my hardware function and executes a move command. Essentially, I need to figure out a way to delay reading the next buttons values following a 'stop'.

My ram checker found two-memory address registers wrong. I am not sure if this is a problem with my RAM, my wiring is shoddy, or something else entirely. It is a confusing problem because my address lines work and only two registers out of 16K do not work. If I had more time, I would spend even more time troubleshooting this error, since I have not been able to fix it thus far.

I dislike how my timer prints to the screen. I think I should find a more efficient way to count time and possibly see if I can remove more code from ISRS to the main event loop.

I would rewire and solder the entire board. Unfortunately, I have terrible wire wrapping and soldering skills. The only reason my board worked as soon as it did was I could deftly use the digital multimeter(DMM) and the oscilloscope to check for continuity errors or bus contention. I was able to thoroughly troubleshoot to make a working board and expansion but I still had difficulty fixing errors from the beginning of the quarter. For example, I did not fix the insulating the capacitors problem because I was concerned I would ruin the wire wrapping I had already completed on top. I did my best to decrease the solder or bulky wiring lines and fixed

the power problem connecting problems, but I think the only way I could truly fix these were if I redid the board. In fact, I bought all the components to redo the board, just never had time to. Nevertheless, I will probably use the skills acquired in this class and the extra components to do so something cool!

Overall, I am pleased with my project. I was able to do an expansion that at times seemed impossible especially since, it required using skills that I did not have. I think my programming could be better and more efficient, but I understand the process of coding a microcontroller now. By the time, I interfaced my switches I was comfortable figuring out how they worked. I used a DMM tested to see if up or down was connected and then set up a simple schematic with the using the extra pull up resistor pins. Overall, it was fun to put the knowledge I had acquired through computer architecture and digital logic to the test.

Power Dissipation with Heat Sink: