

**CMPE 121 L Project  
Laura Miller  
June 2, 2004**

## Table of Contents

1. Project Specifications .....	3
1.1. Hardware Specifications .....	3
1.2. Software Specifications .....	3
2. Hardware Design .....	3
2.1. Power .....	4
2.2. 68HC11E2 Processor .....	4
2.2.1. Clock .....	4
2.2.2. Serial Communication .....	4
2.2.3. Reset .....	4
2.2.4. Mode .....	5
2.2.5. Address/Data .....	5
2.3. Multiplexed Data/Address Lines .....	5
2.4. Memory .....	5
2.5. I/O Ports .....	6
2.6. Serial Communication .....	6
2.6.1. DIP Switches .....	6
2.7. LCD Display .....	7
3. Software Design .....	7
3.1. Boot Code .....	7
3.2. RAM Tester .....	8
3.3. Fully Interrupt Driven, Full-Duplex Serial Communication .....	8
3.4. Interrupt Driven System Clock .....	8
3.5. ROM Loader .....	9
3.6. LCD Driver .....	9
3.7. DIP Switch Driver .....	10
4. Conclusion .....	10
<b>Appendix A-1: Main Engineering Schematic .....</b>	<b>11</b>
<b>Appendix A-2: DIP Switch Schematics .....</b>	<b>12</b>
<b>Appendix A-3: LCD Display Schematic .....</b>	<b>13</b>
<b>Appendix A-4: Picture of Board .....</b>	<b>14</b>
<b>Appendix B-1: Classic Memory Map .....</b>	<b>15</b>
<b>Appendix B-2: Detailed Memory Map .....</b>	<b>16</b>
<b>Appendix C-1: Boot Code .....</b>	<b>17</b>
<b>Appendix C-2: RAM Tester Code .....</b>	<b>26</b>
<b>Appendix C-3: Interrupt Driven, Full Duplex Serial Communication .....</b>	<b>33</b>
<b>Appendix C-4: Interrupt Driven Timer .....</b>	<b>36</b>
<b>Appendix C-5 ROM Loader .....</b>	<b>38</b>
<b>Appendix C-6: User Program That Runs the LCD .....</b>	<b>41</b>

## 1. Project Specifications

### 1.1. Hardware Specifications

- CPU - 68HC11E2
  - 8 MHz Clock
- 28k x 8 SRAM – 62256LP-12
- 32k x 8 EEPROM – AM29F010
- Serial Data Interface – RS232C
- 5V Voltage Regulator
  - Operable over 7 to 12 V DC
- LCD
- DIP Switch

### 1.2. Software Specifications

- Boot code to setup CPU
- SRAM Tester
  - Check for adjacent cell dependencies
  - Check for bad memory locations
- Interrupt Device Driver
  - Supports full duplex serial communication
  - Operable at 9600 bps and 4800 bps
- Interrupt Driven System Clock
- DIP Switch Driver
- LCD Driver

## 2. Hardware Design

To begin the hardware design I first implemented the power circuit and placed the power and ground rails as close to the 7805T to reduce power line length. Then I placed the HC11 near the power rail, from there I placed the 74HC573 used for latching the addresses  $A_0$  through  $A_7$  in line with the address on the HC11. From there I placed the SRAM and EEPROM next to each other for maximum pin symmetry with address and data lines. I placed all the logic I needed for all the enable signals in line with the latch. The MAX232 was then connected to the HC11 and serial header. I then needed another latch to latch the data from the DIP switches so I aligned it next to the ROM for pin symmetry. I hooked the DIP switches up to the data bus and up to power (through a 4.7 k $\Omega$  network resistor) and ground. Then I connected the header for the LCD onto the data bus. In my design I worked very hard to ensure the power lines were short and not daisy chained off each other, I also used pin symmetry to make wiring uncomplicated. The layout and engineering schematic can be seen in Appendix A.

## 2.1. Power

The voltage, into the board, needs to be regulated to 5 V from 7 V – 12 V. This is done by using the 7805T Voltage Regulator, U12 in Appendix A-1. This is a three pin chip with a 7 V -12 V input, 5 V output, and ground I/O. Before the voltage goes through the regulator, it needs to go through the IN4004 diode to prevent current from going into the power source. After the power goes through the regulator it is then connected in parallel to two capacitors. The capacitors are .1  $\mu$ F and 47  $\mu$ F to filter out both low and high frequencies.  $V_{CC}$  and  $G_{ND}$  then go to power and ground rails that supply all the chips on the board.

## 2.2. 68HC11E2 Processor

I configured the HC11 to handle memory usage, I/O, serial communication, and reset.  $V_{CC}$  and  $G_{ND}$  have two bypass capacitors, 1  $\mu$ F and .1  $\mu$ F, between them for noise reduction. Currently Port A is not being used and all of its inputs are tied up to pull-up resistors, I did nothing with the outputs. The serial clock signal (SCK) and slave select (SS) inputs were also tied high with pull-up resistors. XIRQ and IRQ are also pulled up because they're inputs. For analog to digital conversion I did nothing with Port E, but I tied VRH and VRL to  $G_{ND}$  because it does not matter what their values are because they are never used, however they are inputs. The engineering schematic for the HC11 can be found in Appendix A-1.

### 2.2.1. Clock

There is an 8 MHz clock connected to XTAL and EXTAL on the HC11. A 10 M $\Omega$  resistor and two 22 pF capacitors connected to the clock in parallel. The 8 MHz clock generates a 2 MHz E clock signal that controls the board.

### 2.2.2. Serial Communication

TXD, RXD, MISO, and MOSI are all used to communicate with the serial communication. RXD and TXD are used for receiving and transmitting data. I connected MISO and MOSI even though I'm not using them currently because in the future I may wish to use them.

### 2.2.3. Reset

The HC11 has an input pin used for resetting the board. To lower the noise from the reset switch, I used a .1  $\mu$ F capacitor and a 4.7 k $\Omega$  pull-up resistor.

#### 2.2.4. Mode

I want to run the chip in expanded mode, since it is the normal mode that allows access to external memory. To be in expansion mode MODA and MODB must be high, I tied it to a pull up resistor to keep current low.

#### 2.2.5. Address/Data

The HC11 has a 16 bit address and an 8 bit data bus. The first half of the address bus is multiplexed with the data bus. To control which is kind of data is being sent, the HC11 has an address strobe to tell the address latch when the address is being sent. The HC11 also has a read/write signal that is used to tell the EEPROM, SRAM, and DIPs when to read and write. This signal is very helpful in reducing bus contention between the memory and I/O. The E Clock is used for timing purposes for all of the external chips.

#### 2.3. Multiplexed Data/Address Lines

The HC11's lower eight address bits ( $A_0-A_7$ ) also carry data. The multiplexed line carries the address during the first part of the bus cycle and the data afterwards. The address strobe on the HC11 carries this information which is connected to the clock to the 74HC573 (U2) Octal D-Type Transparent Latch. The 74HC573 latches the address and is connected to both the SRAM and EEPROM lower address bits and the Register Selection on the LCD. The data on the bus is then able to be accessed during the remaining of the bus cycle.

#### 2.4. Memory

32 KB of EEPROM and 28 KB of SRAM are utilized in my board design. The memory decoding scheme I used is exhaustive decoding since every address line from the HC11 is used and it is not linear because I used logic to access each address. This memory decoding scheme prevents bus contention because the addresses are a 1-to-1 mapping and output enable and chip enable ensure that no outputs will be outputting information at the same time. The bus timing has been modeled so that it emulates Intel timing. The classic and detailed memory maps are located in Appendix B and the engineering schematic describing the wiring of the memory can be located in Appendix A-1.

The 29F010B EEPROM (U3) is a 128k x 8, 32 pin chip.  $A_0-A_{14}$  are utilized for addressing 32KB of ROM.  $A_{14}-A_{15}$  are tied to ground because they are not used in my memory decoding scheme.  $D_0-D_7$  are used to send and receive data from the data bus. Write Enable is tied to a 4.7 k $\Omega$  network resistor because I do not want to write to the ROM. Output

enable is connected to the read signal, a negation of the R!/W signal from the HC11, which only allows for the ROM to be on during a read cycle. The Chip Enable is connected to  $A_{15}$  so that ROM is only selected when  $A_{15}$  is high. ROM is addressable using addresses 0x8000-0xFFFF. There is a .1  $\mu\text{F}$  bypass capacitor between  $V_{CC}$  and  $G_{ND}$  to level out noisy signals.

The HM62256 SRAM (U4) is a 32k x 8, 28 pin chip.  $A_0$ - $A_{14}$  are utilized for addressing 28KB of SRAM. The missing 4KB of data is due to the condition that if  $A_{15}$  is low and  $A_{12}$ - $A_{14}$  are high then I/O ports will be used.  $D_0$ - $D_7$  are used to send and receive data to and from the data bus. Write enable is connected to the write signal to allow writing only during the write cycle. Output enable is connected to the read signal, a negation of the R!/W signal from the HC11, which only allows for the SRAM to be on during a read cycle. Chip select is only one when  $A_{15}$  is low and  $A_{12}$ - $A_{14}$  are high. SRAM is addressable using addresses 0x0000-0x6FFF. There is a .1  $\mu\text{F}$  bypass capacitor between  $V_{CC}$  and  $G_{ND}$  to level out noisy signals.

## 2.5. I/O Ports

I/O is accessed using addresses 0x7000-0x7FFF in 0x200 increments. 0x7000 is used to access the DIP switches. 0x7200 is used to access the LCD Display. 0x7400-0x7FFF is not implemented, but more expansions could be easily implemented. When  $A_{15}$  is low and  $A_{12}$ - $A_{14}$  are high then a 74HC138 3-to-8 Decoder is enabled.  $A_9$ - $A_{11}$  are used to select which device is used for I/O. There is a .1  $\mu\text{F}$  bypass capacitor between  $V_{CC}$  and  $G_{ND}$  to level out noisy signals.

## 2.6. Serial Communication

The serial port signals TXD, RXD, RTS, and CTS first must go through the MAX232ACPE Driver/Receiver. Five .1  $\mu\text{F}$  capacitors needed to be used to configure the driver/receiver correctly. One between  $C1+$  &  $C1-$ ,  $C2+$  &  $C2-$ ,  $V+$  &  $V_{CC}$ ,  $V-$  &  $G_{ND}$ , and a bypass capacitor between  $V_{CC}$  and  $G_{ND}$ . The voltage will be double on the signals after it passes through the driver/receiver. The driver/receiver and the header are connected to the serial signals for serial communication with the HC11.

### 2.6.1. DIP Switches

The DIP Switches can be read from address 0x7000-0x71FF, it doesn't matter which address in that range is used. An 8 bit string will be read from the switches. At boot-up  $A_8$  is used to tell the processor to run the RAM tester and  $A_7$  signifies the BAUD Rate. Each switch is connected on one end to a pulled up  $V_{CC}$  or  $G_{ND}$ , which is read as a logic high or low. Another 74HC573 Octal D-Type Transparent Latch (U9) is used to hold

the switch data until it can be read from the data bus. Output enable is a negation of the R/W signal XOR-ed with the I/O signal from the 3-to-8 decoder used for I/O Ports. This only enables output when it is the read cycle and any address between 0x7000-0x71FF is selected. Latch enable is connected to power causing it to never be enabled. There is a .1  $\mu$ F bypass capacitor between  $V_{CC}$  and  $G_{ND}$  to level out noisy signals on the latch. The engineering schematic for the DIP switches, transparent latch, and logic used can be found in Appendix A-2.

## 2.7. LCD Display

The LCD Display used in my board is a 20 character, 1 line, 14 pin LCD. The LCD can be written to use addresses 0x7200-0x73FF, if the number is odd the control commands will be sent to the LCD and if the number is even character commands will be sent to the LCD. This is because the register selection bit on the LCD is toggled by  $A_0$ . When  $A_0$  is a 0 then control commands are being used and when  $A_0$  is a 1 then a character is being written. A 10 k $\Omega$  trimpot is used to adjust the contrast on the LCD. Enable is a negation of the E Clock XOR-ed with the I/O signal from the 3-to-8 decoder used for I/O ports.  $D_0$ - $D_7$  are the only other connections that needed to be made for the LCD.

## 3. Software Design

### 3.1. Boot Code

As seen in Appendix C-1, I modified `ivectors.c` so that it is an assembly file (`ivectors.s`), allowing me place the vector table in the `.interrupts` section so that there are not two `.const`'s sections as the test program originally had. In `ivectors.s` I added interrupts for SCI for serial communication and for `PULSE_ACC_OVERF` so that I could use it for my timer.

Then I modified linker file (`eprom.lkf`) by adding a `.boot` and an `.interrupt` section to provide more structure to the program. The `.boot` section is now where boot code is, differentiating it from the `.text` area which has more general routines. Additional object files were included to implement the different programs I had written. I also defined 0x1040 to be where user programs begin and 0x6FFF to be where the stack begins.

Since I made a `.boot` section I needed to create a boot stub to declare what it was to do. In `bootstub.s` the program jumps to the RAM Tester and the ROM Loader calls it to set up the stack for the user programs.

The functions for getting time, putting a byte from the serial connection, and getting a byte from the serial connection are all found in `hc11.h` so that user programs can use them. In `hc11.s` I defined constants to be placed where I had them in my memory map. It also checks the DIP switch 7 before initializing the BAUD rate to either 9600 (when ON) or

4800 (when OFF), the DIP switches are active low. I renamed the original hc11.h file to hc11\_reg.h and did not significantly alter it. It is used to define registers and ports.

### 3.2. RAM Tester

The RAM Tester, only runs when the 8<sup>th</sup> DIP switch is turned on, tests for stuck address bits, adjacency dependent bits, and bad address lines. I use two different methods for checking the RAM. The first is to write alternating bits 10101010 and 01010101 to the memory, then read it back from the RAM to make sure it was written correctly. This checks for both stuck memory bits and adjacency dependent bits and if there is a problem, it reports back the problem and prints out the error.

Since the bit pattern is symmetrical it may not guarantee that incorrect address locations are being reported incorrectly. To ensure my RAM testing further, I generate a 257 byte long string. I found this implementation strategy in an article called “Testing RAM in Embedded Systems” at <http://www.ganssle.com/testingram.pdf>.

Creating this program was a bit difficult because it all had to be done in assembly and could not use the stack. The lack of memory usage caused many issues that spawned ingenuity for I/O and tests. I created my own I/O that I don't use in any other programs.

For testing purposes I tried a few different scenarios. When the RAM is plugged in normally, all addresses are reported okay. When the RAM is entirely unplugged, every address is reported badly. When the RAM has a broken data pin, every address is reported badly. When the RAM has a broken address pin, unfortunately all the addresses are reported okay. When the RAM has connected address pins, the HC11 won't boot.

### 3.3. Fully Interrupt Driven, Full-Duplex Serial Communication

For full-duplex, interrupt driven serial communication I first had to create a circular queue for both input and output. If the queue is full with 256 bytes, then nothing can be added to the queue. To keep track of SCI interrupts that are caused by received or transmitted bits, I placed a line in ivectors.s to implement the interrupt. The code for the serial connection can be found in intrserial.c in Appendix C-3.

### 3.4. Interrupt Driven System Clock

I used the output compare register 2 to generate a clock signal to make a timer. I originally set it to 2000 so that it would overflow every millisecond and from there I add it again so keep track of the milliseconds. I keep

track of how many hours, minutes, and seconds there are and print them out.

First I had to initialize the vt100 screen so that I could print the time at the bottom of the screen. I then just had it print out my name at the top of the screen and so it's just the timer. Then I had the time print out every second to the screen. My clock was fast 1.5 seconds every hour, which would mean in a month it is off  $30 \text{ days} * 24 \text{ hours} * 60 \text{ minutes} * 60 \text{ seconds} * .00042 \text{ error} = 1036 \text{ seconds}$  or 17.28 minutes.

### 3.5. ROM Loader

For convenience I created a program to be burned onto the ROM so that I can load programs onto the RAM through the serial connection. First the S19 file needed to be decoded and this was done as follows:

- All lines starts with S
- If second digit is
  - 0 is a comment
  - 1 is data
  - 9 is end of file
- The next byte (in HEX) is the length
- The next 2 bytes (in HEX) are the address
- The rest of the bytes (in HEX), except for the last byte, are data
- The last byte (in HEX) is checksum

### 3.6. LCD Driver

Writing the driver for communication with the LCD involves sending control commands and characters. First an initialization sequence must be sent to the LCD. When register selection is low ( $A_0$  low), control commands are sent to the LCD. First 0x30 is sent to set the interface data length and select the one line display. Then 0x01 turns the LCD on, 0x1F places the cursor on the LCD, 0x0E turns the cursor on, and 0x06 sets mode to increment the address by one and shifts the cursor to the right.

After the initialization sequence is sent to the LCD then characters can be written to the LCD. When register selection is high ( $A_0$  high), characters are sent to the LCD. The characters that the LCD displays are in direct correlation with the ASCII table. To print to the LCD all that is needed is a simple, printf-like statement that write to an odd address in the LCD's memory range.

The LCD does unfortunately have a timing constraint so characters cannot just be sent to the LCD without a waiting period. Luckily the manufacturers included a "wait" bit so it is easy to check if it is safe to write a character to the LCD. To accomplish this task I have the code wait until the wait bit is not set, and then send another instruction to it.

When writing this code, Ryan Cormier and I worked together because we both have the same LCD. We used the data sheet to learn how to initialize the LCD and worked writing the code.

Currently, when the driver is loaded through the RAM Loader it turns the LCD on, prints out a message, and allows the user to type through HyperTerminal to the LCD. The LCD accepts ASCII characters, backspaces, new lines, and ESC will terminate the program and reboot the system.

Adam Freidin helped us with the macros and some of the design of the code.

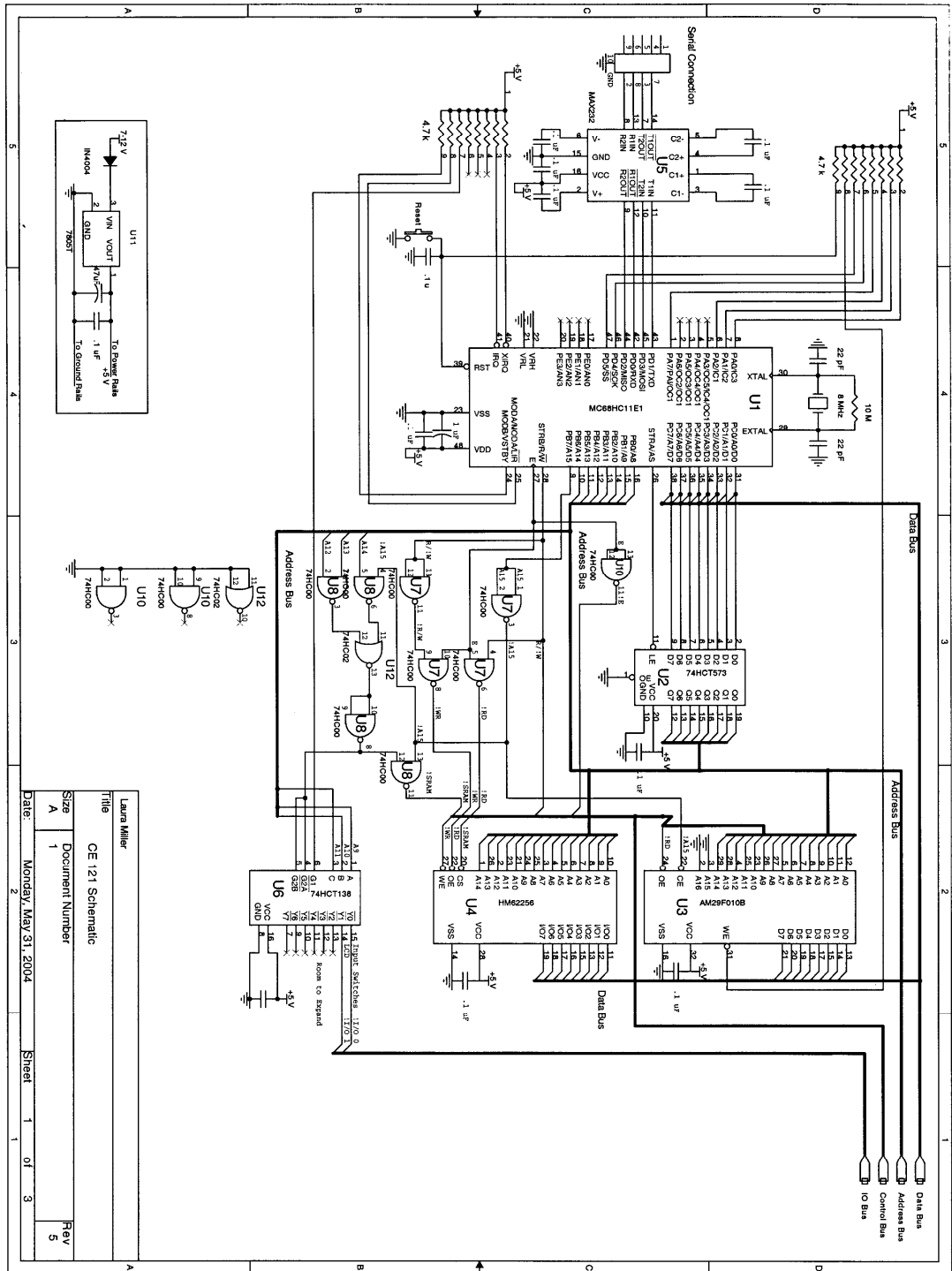
### 3.7. DIP Switch Driver

During boot up, the DIP switches signify if the RAM tester is run and which BAUD rate is to be chosen. Reading the DIP switches requires accessing any address between 0x7000 and 0x71FF. An 8 bit long number is read in and then masked to determine which bits are set.

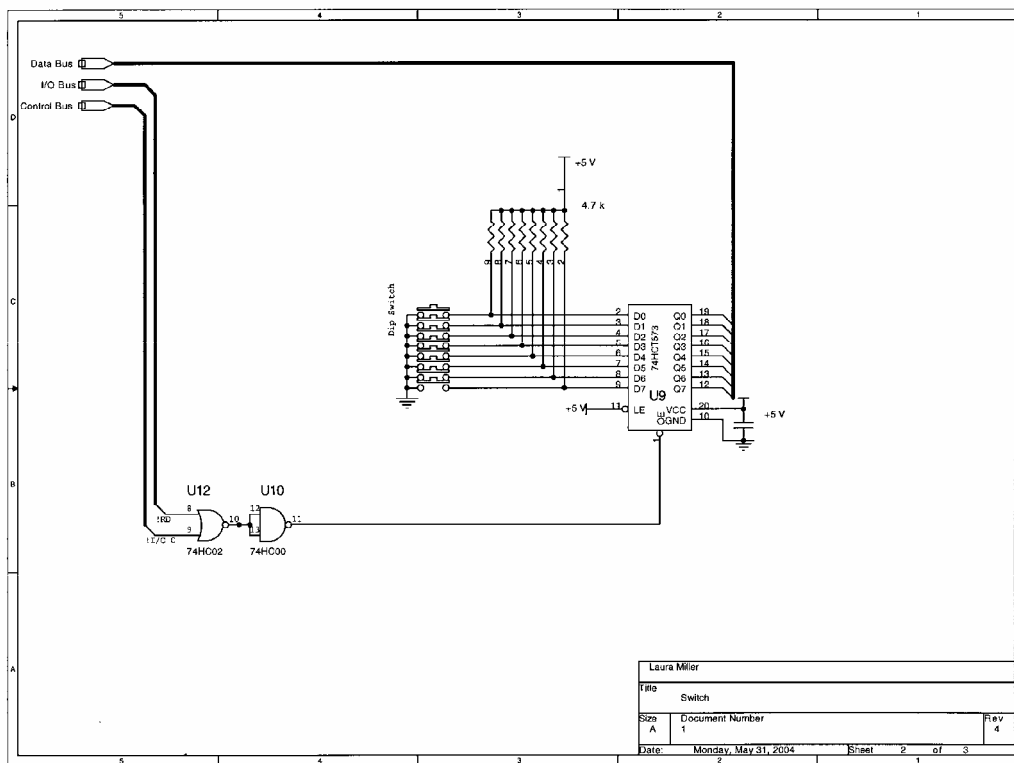
## 4. Conclusion

This lab stretched me top my limits as far as doing work, but it was rewarding in the end having a finished project and not at the last minute. Once my hardware was done and working nicely, the software was fairly easy to implement. The RAM Tester was a little annoying but I quickly got used to not being able to use the stack. I wish I could have started something before the 4<sup>th</sup> or 5<sup>th</sup> week of school because I feel I could have gotten a lot more out of it. I also should worked at normal hours and my hardware would have been more correct the first time.

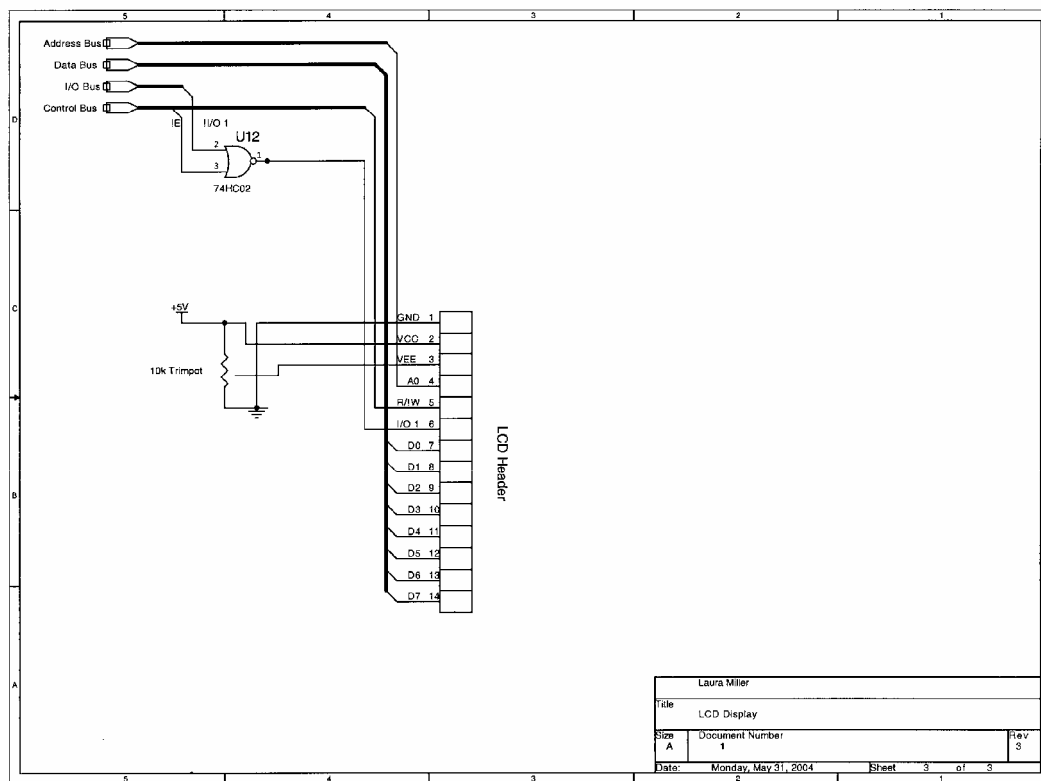
Appendix A-1: Main Engineering Schematic

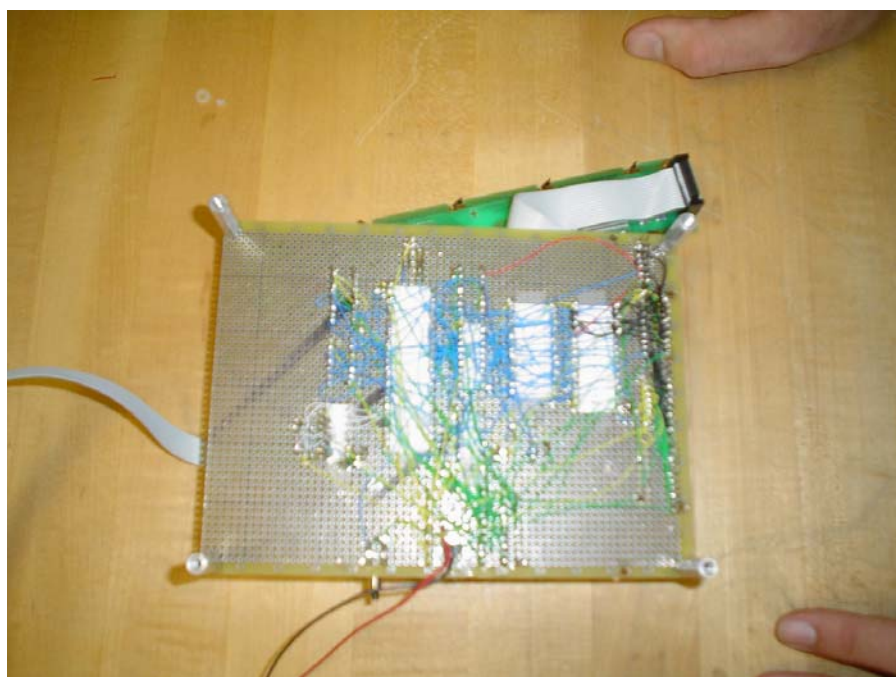


### Appendix A-2: DIP Switch Schematics

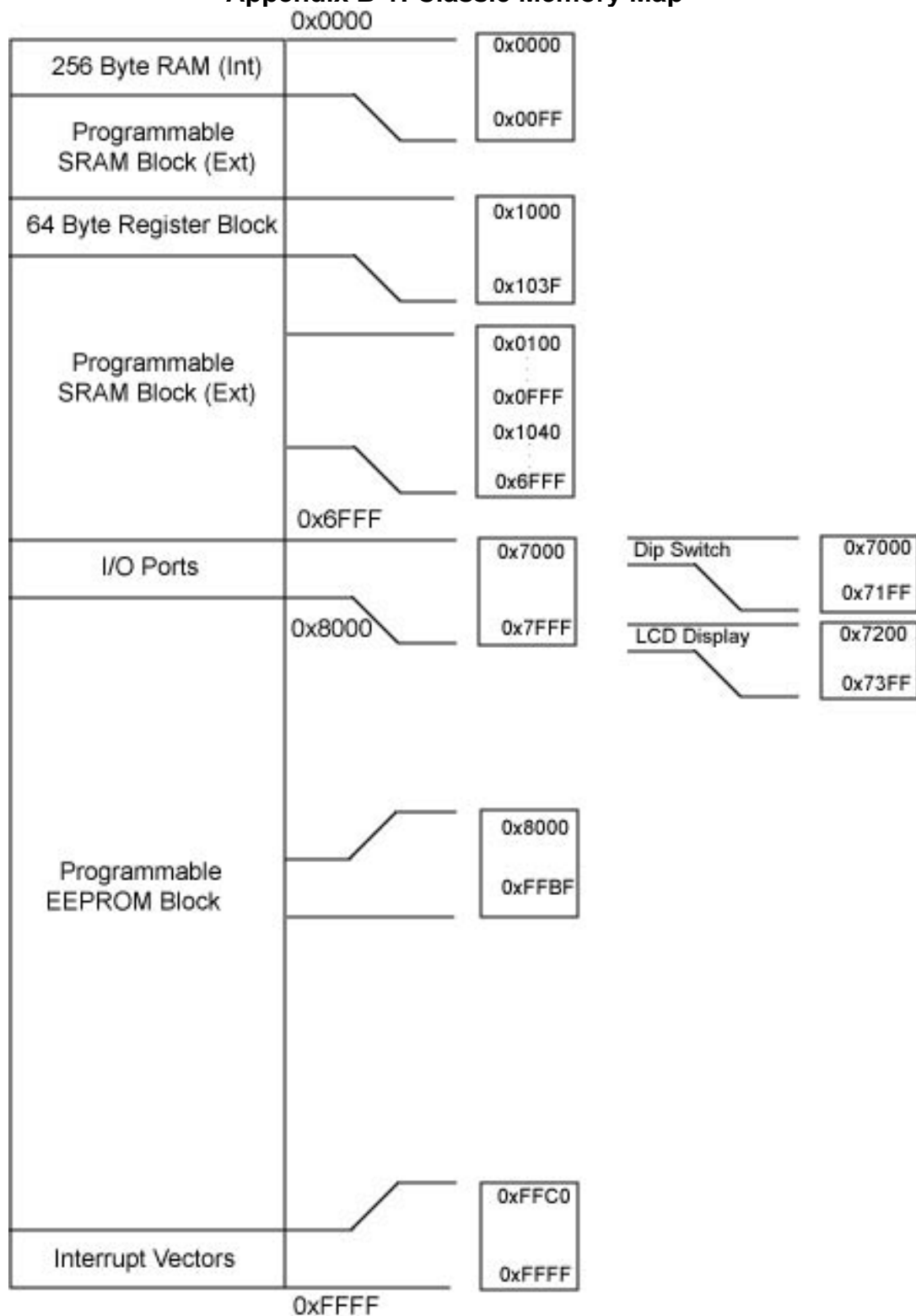


### Appendix A-3: LCD Display Schematic



**Appendix A-4: Picture of Board**

### Appendix B-1: Classic Memory Map



## Appendix B-2: Detailed Memory Map

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	0	0	0	/					28 KB RAM				/	/		0x0000-0x06FF SRAM (Ext)		
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...			
0	1	1	0	/									/	/				
0	0	0	0	0	0	0	/									/		0x0000-0x01FF RAM (Int)
0	0	0	1	0	0	0	0	0	0	/					/		0x1000-0x103F Registers	
0	1	1	1	0	0	0	/									/		0x7000-0x71FF DIP Switch
0	1	1	1	0	0	1	/									/		0x7200-0x73FF LCD
0	1	1	1	0	1	0	/									/		0x7400-0x75FF Not used
0	1	1	1	0	1	1	/									/		0x7600-0x7FFF Not used
0	1	1	1	1	0	0	/									/		0x7800-0x79FF Not used
0	1	1	1	1	0	1	/									/		0x7A00-0x7BFF Not used
0	1	1	1	1	1	0	/									/		0x7C00-0x7DFF Not used
0	1	1	1	1	1	1	/									/		0x7E00-0x7FFF Not used
1	/									32 KB ROM				/	/		0x8000-0xFFFF EEPROM	

## Appendix C-1: Boot Code

```

# FILE: eeprom.lkf

# This file is used by the assembler and linker to map where
# objects of all kinds will be placed in memory. It should
# precisely agree with your particular memory map.

# This block defines where objects (code and data) will be placed.
# These are placed in one of six different "segments"
# Syntax: +seg .xxxx options
# Once a "segment" base address is defined, it remains in effect
# until redefined later on.  RAM segments can overlap or even
# have the same base address as is done in this default example.

+seg .boot  -b 0x8000 -n .boot  # program start address  (ROM)
+seg .text  -a .boot -n .text  # other program code    (ROM)
+seg .const -a .text          # constants follow program (ROM)
+seg .data  -b 0x0000 -n .data # initialized data      (RAM)
+seg .bss   -a .data          # uninitialized data    (RAM)
+seg .interrupt -b 0xffd6     # vectors start address

# These object files are linked into appropriate segments
# defined above in the order shown.
bootstub.o           # assembly boot code
ramchecker.o         # RAM tester
romloader.o          # files that are loaded onto ram
intrserial.o         # full duplex serial communication
ivectors.o           # interrupt vectors
vt100.o              #printing out to hyperterminal

# The linker will look in these libraries to resolve externally
# defined calls.
# libm.h11 -> machine lib. (always used)
# libi.h11 -> integer lib.
# libd.h11 -> double precision lib.
# libf.h11 -> floating point lib.

"C:\COSMIC\CX11\lib\libi.h11"
"C:\COSMIC\CX11\lib\libm.h11"

# Set the _memory base pointer to be the same as bss section.
# The use of this pointer in
# programming usually defines where heap space begins and
# normally follows immediately after
# the bss section.
+def __memory=@.bss      # symbol used by library
+def __userstart=0x1040 # boot vector for serially loaded programs
+def __userstack=0x6FFF # stack top for serially loaded programs
+def _DIPSWITCH=0x7000 # dipswitch access area

```

```

;
;   FILE: IVECTORS.S
; PURPOSE: interrupt vector table for 68HC11 Project
;   CODE: Cosmic C v4.1m
;   AUTHOR: Laura Miller, converted from Peterson's C-file.
; HISTORY: Spring-01: created from Cosmic's original template.
;           Spring-04: placed into a format which is more hard core.
;

```

```

; This vector table contains ordinal 16-bit pointers to ISR's,
; and is located at the top of addressable memory in ROM
; beginning at FFD6h.

```

```

    xref __boot, _oc_intp, _sci_intp
    xdef __interrupt

```

```

SCI:                equ _sci_intp
SPI:                equ 0000h
PULSE_ACC_INPUT:   equ 0000h
PULSE_ACC_OVERF:   equ 0000h
TIMER_OVERF:       equ 0000h
OUTPUT_COMPARE_5:  equ 0000h
OUTPUT_COMPARE_4:  equ 0000h
OUTPUT_COMPARE_3:  equ 0000h
OUTPUT_COMPARE_2:  equ _oc_intp
OUTPUT_COMPARE_1:  equ 0000h
INPUT_CAPTURE_3:   equ 0000h
INPUT_CAPTURE_2:   equ 0000h
INPUT_CAPTURE_1:   equ 0000h
REAL_TIME:         equ 0000h
IRQ:               equ 0000h
XIRQ:              equ 0000h
SWI:               equ 0000h
ILLEGAL:           equ 0000h
COP_FAIL:          equ 0000h
COP_CLOCK_FAIL:   equ 0000h
RESET:             equ __boot

```

```

; dc.w = declare constant .word

```

```

.interrupt:section
    switch .interrupt
__interrupt:
    dc.w SCI
    dc.w SPI
    dc.w PULSE_ACC_INPUT
    dc.w PULSE_ACC_OVERF
    dc.w TIMER_OVERF
    dc.w OUTPUT_COMPARE_5
    dc.w OUTPUT_COMPARE_4
    dc.w OUTPUT_COMPARE_3
    dc.w OUTPUT_COMPARE_2
    dc.w OUTPUT_COMPARE_1
    dc.w INPUT_CAPTURE_3
    dc.w INPUT_CAPTURE_2
    dc.w INPUT_CAPTURE_1
    dc.w REAL_TIME

```

```
dc.w IRQ
dc.w XIRQ
dc.w SWI
dc.w ILLEGAL
dc.w COP_FAIL
dc.w COP_CLOCK_FAI
dc.w RESET
```

```
; Name: Laura Miller
; Date: 5/21/04
; bootstub.s
; Description: Code used at bootup to start the RAM checker and
;             loads the stack to the user stack pointer and goes
;             to user start address.
    xdef __boot, __runuser
    xdef __after_ramcheck
    xref __start_ramcheck
    xref __userstart, __userstack

include "hc11.s"

.boot:section
    switch .boot
__boot:
    jmp __start_ramcheck

    switch .text

; Sets up the stack after the RAM checker has completed
__runuser:
    clrb
    ldy #0
    ldx #__userstack
    txs
    jmp __userstart
```

```
/* eeprom.h
 * Name: Laura Miller
 * Date: 5/30/04
 * Description: The functions for getting time, getting and
 * putting bytes using the serial connection are prototyped here
 */
#ifndef EEPROM_H_
#define EEPROM_H_

int get_time(void);
int sci_put_byte(char c);
int sci_get_byte(void);

#endif
```

```

/* hc11.h
 * Name: Laura Miller
 * Date: 5/21/04
 * Description: Defines the bits used in serial communication and
 * timer interrupt flag.
 */
#ifndef HC11_H_
#define HC11_H_

#include"hc11_compat.h"

/* Serial Communications Status Register (SCSR) */

#define REG_SCDR SCDR
#define REG_BAUD BAUD

#define REG_SCSR SCSR
#define SCSR_TDRE 0x80 // Transmit Data Register Empty; 1 when empty
#define SCSR_TC 0x40 // Transmit Complete; 1 when idle
#define SCSR_RDRF 0x20 // Recieve Data Register Full; 1 when full
#define SCSR_IDLE 0x10 // Idle register
#define SCSR_OR 0x08
#define SCSR_NF 0x04
#define SCSR_FE 0x02

/* Serial Communications Control Register 2 (SCSR2) */
#define REG_SCCR2 SCCR2
#define SCSR2_TIE 0x80 // Transmit interrupt enable bit
#define SCSR2_TCIE 0x40 // Transmit complete interrupt enable bit
#define SCSR2_RIE 0x20 // Recieve interrupt enable bit
#define SCSR2_ILIE 0x10 // Idle interrupt enable bit
#define SCSR2_TE 0x08 // Transmit enable bit
#define SCSR2_RE 0x04 // Receive enable bit
#define SCSR2_RWU 0x02

/* Timer Interrupt Flag 2 (TFLG2) real time timer */
#define REG_TFLG2 TFLG2
#define TFLG2_RTIF 0x40

#endif

```

```

/*
 * NAME: Include file for 121 Lab projects
 * FILE: IO.H
 * CODE: Cosmic's C/ASM tools ver4
 * EMBEDDED I/O DEFINITIONS FOR THE MC68HC11
 */

#define _BASE    0x1000                // default base of 64 byte
register block
#define _IO(x)   @_BASE+x             // offset macro

#if _BASE == 0
    #define _PORT @dir
#else
    #define _PORT @port
#endif

_PORT volatile char PORTA  _IO(0);    // port A
_PORT          char PIOC   _IO(2);    // parallel control
_PORT volatile char PORTC  _IO(3);    // port C
_PORT volatile char PORTB  _IO(4);    // port B
_PORT volatile char PORTCL _IO(5);    // port C latched
_PORT          char DDRC   _IO(7);    // data direction port C
_PORT volatile char PORTD  _IO(8);    // port D
_PORT          char DDRD   _IO(9);    // data direction port D
_PORT volatile char PORTE  _IO(0xa);  // port E
_PORT          char CFORC  _IO(0xb);  // compare force
_PORT          char OC1M   _IO(0xc);  // ocl mask
_PORT          char OC1D   _IO(0xd);  // ocl data
_PORT volatile int  TCNT   _IO(0xe);  // timer counter
_PORT volatile int  TIC1   _IO(0x10); // timer capture 1
_PORT volatile int  TIC2   _IO(0x12); // timer capture 2
_PORT volatile int  TIC3   _IO(0x14); // timer capture 3
_PORT          int  TOC1   _IO(0x16); // output compare 1
_PORT          int  TOC2   _IO(0x18); // output compare 2
_PORT          int  TOC3   _IO(0x1a); // output compare 3
_PORT          int  TOC4   _IO(0x1c); // output compare 4
_PORT          int  TOC5   _IO(0x1e); // output compare 5
_PORT          char  TCTL1  _IO(0x20); // timer control 1
_PORT          char  TCTL2  _IO(0x21); // timer control 2
_PORT          char  TMSK1  _IO(0x22); // timer interrupt mask 1
_PORT volatile char  TFLG1  _IO(0x23); // timer interrupt flag 1
_PORT          char  TMSK2  _IO(0x24); // timer interrupt mask 2
_PORT volatile char  TFLG2  _IO(0x25); // timer interrupt flag 2
_PORT          char  PACTL  _IO(0x26); // pulse accumulator control
_PORT          char  PACNT  _IO(0x27); // pulse accumulator count
_PORT          char  SPCR   _IO(0x28); // SPI control register
_PORT volatile char  SPSR   _IO(0x29); // SPI status register
_PORT volatile char  SPDR   _IO(0x2a); // SPI data register
_PORT          char  BAUD   _IO(0x2b); // SCI baud rate
_PORT          char  SCCR1  _IO(0x2c); // SCI control register 1
_PORT          char  SCCR2  _IO(0x2d); // SCI control register 2
_PORT volatile char  SCSR   _IO(0x2e); // SCI status register
_PORT volatile char  SCDR   _IO(0x2f); // SCI data register
_PORT volatile char  ADCTL  _IO(0x30); // A/D control register
_PORT volatile char  ADR1   _IO(0x31); // A/D result 1
_PORT volatile char  ADR2   _IO(0x32); // A/D result 2

```

```
_PORT volatile char ADR3    _IO(0x33); // A/D result 3
_PORT volatile char ADR4    _IO(0x34); // A/D result 4
_PORT          char BPROT   _IO(0x35); // block protect
_PORT          char PORTG   _IO(0x36); // port G
_PORT          char DDRG    _IO(0x37); // data direction port G
_PORT          char OPTION   _IO(0x39); // system config options
_PORT          char COPRST   _IO(0x3a); // COP arm/reset
_PORT          char PPROG    _IO(0x3b); // EEPROM control register
_PORT          char HPRIO    _IO(0x3c); // highest priority register
_PORT          char INIT     _IO(0x3d); // RAM-IO mapping register
_PORT          char TEST1    _IO(0x3e); // factory test control
_PORT          char CONFIG   _IO(0x3f); // (EEPROM) config register
```

```

; hc11.s
; Name: Laura Miller
; Description: Defines where things are in memory, and initializes
; the BAUD rate.
; Modeled from v2_17g2.asm

REGBASE: equ 1000h ; Start of 64 B Reg Block
ROMBASE: equ 8000h ; Start of ROM
RAMBASE: equ 1040h ; Start of user RAM area 0x1000-0x103F not used
RAMTOP: equ 6FFFh ; Top of user RAM area
DIPS: equ 7000h ; DIP Switch Port
LCDCMD: equ 7200h ; LCD Command Port
LCDCHR: equ 7201h ; LCD Char Port
IO2: equ 7400h ; Expansion
IO3: equ 7600h ; Expansion
IO4: equ 7800h ; Expansion
IO5: equ 7A00h ; Expansion
IO6: equ 7C00h ; Expansion
IO7: equ 7E00h ; Expansion

NULL: equ 00h ; End of text/table
CR: equ 0dh ; Carriage return
LF: equ 0ah ; Line Feed
ESC: equ 1Bh ; Escape Character

CHECKRAM_MASK: equ 80h
ALTBAUD_MASK: equ 40h

TDRE: equ 80h

BAUD : equ 102Bh
SCCR1: equ 102Ch
SCCR2: equ 102Dh
SCSR : equ 102Eh
SCDR : equ 102Fh

; Set up the HC11 UART for 9600 or 4800 bps
SETUPSCI:macro
    ldx #DIPS
    brclr 0,x,#ALTBAUD_MASK,otherbaud
    ldaa #30h ; for 9600 baud
    bra sci_continue
otherbaud:
    ldaa #31h ; for 4800 baud
sci_continue:
    staa BAUD
    ldaa #00h
    staa SCCR1
    ldaa #2Ch
    staa SCCR2
endm

```

## Appendix C-2: RAM Tester Code

```

; ramchecker.s
; Name: Laura Miller
; Description: A ram checker for an HC11

; the entrypoint for the next program to run after the
; rom checker.
    xref __after_ramcheck
    xdef __start_ramcheck

include "hc11.s"
include "io.s"

; Two patterns to test for RAM errors
; Covers Conditions:
; 1. If a bit is dependent on an adjacent bit
; 2. If a bit is stuck
PATTERN_A: equ 01010101b
PATTERN_B: equ 10101010b

; Read only data for rom loading
; The regions are marked because SRAM is not contiguous.
; NOTE: they must be sequential!
.const: section
switch .const
blockl:
    dc.w 0000h; low boundary of memory region 0x0000-0x1000
    dc.w 1040h; low boundary of memory region 0x1040-0x7000
blockh:
    dc.w 1000h; high boundary of memory region 0x0000-0x1000
    dc.w 7000h; high boundary of memory region 0x1040-0x7000

; MACRO: chooseblock
;
; checks the block indexed by \1 iff x is lower then the high
; address and lower than the low address.
;   usage: chooseblock n
;   input: x -- address to check
;   clobbers: everything except x
; operation:
;   if x is lower than blockl[n]: set x to blockl[n].
;   jumps to checkblock if x is less than blockh[n].
chooseblock:macro
    cpx blockh+\1*2 ; check x against the high address in block
    bhs \@nextblock ; if x is too high, skip to next block
    cpx blockl+\1*2 ; x is in range, but the low address may be too
low
    bhs \@checkblock ; if it isn't go check x,
    ldx blockl+\1*2 ; otherwise make x the next low block address
\@checkblock:
    putc_imm #'a' ; printf("at %04X\r\n", addr);
    putc_imm #'t'
    putc_imm #' '
    puthex4
    crlf

```

```

        ldy #blockh+\1*2 ; set y to point to the top address
        ; go to the code labeled with checkblock_X_pass
        jmp checkblock_\2_pass
\@nextblock:
        endm

; MACRO: selects the next block to be checked for bad memory.
chooseblocks:macro
        ldx #00h          ; start back at addr = 0
checkram_\1_pass:      ; label to come back too from checkblock_X_pass
; try choosing each block
        chooseblock 0, \1
        chooseblock 1, \1
        endm

hexascii:
        dc.b '0123456789ABCDEF' ; hex characters used for printing
addresses

switch .text

; ENTRY POINT !!!
__start_ramcheck:
        SETUPSCI ; initializes UART

        putc_imm #27
        putc_imm #'['
        putc_imm #'2'
        putc_imm #'J'

        putc_imm #27
        putc_imm #'['
        putc_imm #'1'
        putc_imm #59
        putc_imm #'2'
        putc_imm #'3'
        putc_imm #'r'

        putc_imm #27
        putc_imm #'['
        putc_imm #'1'
        putc_imm #59
        putc_imm #'1'
        putc_imm #'H'

        putc_imm #'R'
        putc_imm #'A'
        putc_imm #'M'
        putc_imm #' '
        putc_imm #'C'
        putc_imm #'h'
        putc_imm #'e'
        putc_imm #'c'
        putc_imm #'k'
        putc_imm #'e'
        putc_imm #'r'
; don't skip check if CHECKRAM bit is not set in switch

```

```

    ldx #DIPS
    brclr x,#CHECKRAM_MASK,noskip
    jmp skip
noskip:

; choose blocks for the first pass, and again for the second.
    chooseblocks first
    chooseblocks second

    ; when ram checking is done... printf("done\r\n");
    putc_imm #'d'
    putc_imm #'o'
    putc_imm #'n'
    putc_imm #'e'
    crlf

    ; ram checking is done, or not, but now it is
    ; time to continue with the next boot step
skip:
    ldx #6ffff
    txs
    cli ; enable interrupts
    jmp __after_ramcheck

; MACRO: printerr
; arguments: X -- first, second... pass
; does printf("Error in %04X.", x++); goto checkram_X_pass
printerror:macro
    putc_imm #'E'
    putc_imm #'r'
    putc_imm #'r'
    putc_imm #'o'
    putc_imm #'r'
    putc_imm #' '
    putc_imm #'i'
    putc_imm #'n'
    putc_imm #' '
; weird code here clobbers the stack pointer but
; preserves the x register (addr).
    txs
    xgdx
    tsx
    puthex4
    putc_imm #'.'
    crlf
    inx
    jmp checkram_\1_pass
endm

; MACRO: reporterror
; operation: jumps to printerror if the CCR says "not-equal".
reporterror:macro
    beq \@ok
    printerror \1
\@ok:
endm

```

```

; MACRO: checkpattern
; loads a pattern, stores it, reads it back,
; reports an error if the pattern changes
checkpattern:macro
    ldaa #PATTERN_\1
    staa 0,x
    ldab 0,x
    cba
    reporterror first
endm

; computes a = (LOW(x) - HI(x))&0xFF
; generates a sequence in terms of x with period 257
generateprimecheckbyte:macro
    txs
    xgdx
    tsx
    cba
    bge \@
    incb
\@:
    nega
    aba
endm

; write the prime checkbyte to the current address
writeprimecheckbyte:macro
    generateprimecheckbyte
    staa 0,x
endm

; ensure that the current address contains the correct checkbyte
checkprimecheckbyte:macro
    generateprimecheckbyte
    ldab 0,x
    cba
    reporterror second
endm

; checkblock function:
; x is current address,
; y points to the top address
; does the stuck-bit pattern check, followed by
; writing the checkbyte in preparation for
; the second pass.
checkblock_first_pass:
    checkpattern A ; check *x with each pattern
    checkpattern B
    writeprimecheckbyte
    inx             ; increment x
    cpx 0,y        ; check if we have reached the end of the section
    bne checkblock_first_pass; if not, repeat
    putc_imm #'.'
    putc_imm #'.'
    putc_imm #'.'
    putc_imm #' '
    puthex4

```

```
    crlf
    jmp checkram_first_pass ; otherwise return

; The second pass checks that every memory cell contains
; its checkbyte left from the first pass.
; The checkbyte is designed to locate stuck or tied
; address lines.
checkblock_second_pass:
    checkprimecheckbyte
    inc x ; increment x
    cpx 0,y ; check if we have reached the end of the section
    bne checkblock_second_pass; if not, repeat
    putc_imm #'.'
    putc_imm #'.'
    putc_imm #'.'
    putc_imm #' '
    puthex4
    crlf
    jmp checkram_second_pass; otherwise return
```

```
; io.s
; Name: Laura Miller
; Description: Macros used for putc
putc:macro
    tys
    ldy #SCSR
    brclr y,#TDRE,* ; loop until serial is ready for output
    staa SCDR
    tsy
    endm

putc_imm:macro
    ldaa \1
    putc
    endm

crlf:macro
    ldaa #CR
    putc
    ldaa #LF
    putc
    endm

txd:macro
    txs
    xgdx
    tsx
    endm

lsrd4:macro
    lsrd
    lsrd
    lsrd
    lsrd
    endm

puthex4:macro
    txd
    lsrd4
    lsrd4
    lsrd4
    puthex
    txd
    lsrd4
    lsrd4
    puthex
    txd
    lsrd4
    puthex
    txd
    puthex
    endm

puthex:macro
    andb #0Fh
```

```
ldy #hexascii
aby
ldaa 0,y
putc
endm
```

### Appendix C-3: Interrupt Driven, Full Duplex Serial Communication

```

/* Name: Laura Miller
 * Date: 6/8/04
 * File: intrserial.c
 * Description: Implements full-duplex, interrupt driven serial
communication.
 *           Creates two circular queues to hold I/O. Also
implements the
 *           the interrupt thrown when a character is ready to be
sent or
 *           received.
 */
#include "hcl1.h"
#include "hcl1_reg.h"

#define TDRE 0x80          // Transmit Ready Enable
#define RDRF 0x20          // Recieve Ready Enable
#define TIE 0x80          // Transmit Interrupt enable
#define RTIF 0x40

#define MAX_SIZE 256      // Max Size of the buffer
#define null -1

// Initializes the buffer
#define buff_init(buffer) \
    { buffer.count = 0; buffer.head = 0; buffer.tail = 0; }

// Returns true if the buffer is full
#define buff_full(buffer) (buffer.count >= MAX_SIZE)

// Returns true if the buffer is empty
#define buff_empty(buffer) (buffer.count == 0)

// Queue buffer used to hold the input and output
typedef struct{
    unsigned char head;
    unsigned char tail;
    unsigned int count;
    unsigned char buffer[MAX_SIZE];
} queue_buffer;

// Create two buffers, one for input and one for output
queue_buffer input_buff;
queue_buffer output_buff;

// io_buff_init()
// Initialize both the input and output buffers
void io_buff_init(void){
    buff_init(input_buff);
    buff_init(output_buff);
}

// put_buff()
// Puts a character into the input buffer and increases the count of
the number

```

```

// of items in the buffer.
void put_buff(void){
    if(!buff_full(input_buff)){ // If the buffer still has space in it
        input_buff.count++; // Increase the count of items in the
buffer
        input_buff.buffer[input_buff.tail] = SCDR; // Set the tail of the
buffer
// equal to the
character coming // in

        if(input_buff.tail < MAX_SIZE) input_buff.tail++; // If we're at
the end
        else input_buff.tail = 0; // Wrap the tail
around
    }
}

// get_buff()
// Gets a character from the output buffer and decreases the count of
the number
// of items in the buffer
unsigned char get_buff(void){
    if(!buff_empty(output_buff)){ // If the buffer has something in it
        output_buff.count--; // decrease the amount of items

        if(output_buff.head < MAX_SIZE) output_buff.head++; // If the head
is at the
        else output_buff.head = 0; // end, wrap
it around

        return output_buff.buffer[output_buff.head]; // return the
character
    }
    return null;
}

// sci_intp()
// If there is an input interrupt, put in buffer
// If there is an output interrupt, take out of buffer
@interrupt void sci_intp(void) {
    if(!(REG_SCSR & SCSR_RDRF)) put_buff();
    if(!(REG_SCSR & SCSR_TDRE)) SCDR = get_buff();
}

// getc()
// Get a character from the input buffer, decrease the count and
increase
// the head
unsigned char getc(void){
    if(!buff_empty(input_buff)){ // If there's something in the buffer
        input_buff.count--; // decrease the count

        if(input_buff.head < MAX_SIZE) input_buff.head++; // If we're at
the end
        else input_buff.head = 0; // wrap the head
around
    }
}

```

```

    return input_buff.buffer[input_buff.head];        // return the
character
}
return null;
}

// putc()
// Puts a character into the output buffer, increases the count and the
tail
void putc(unsigned char c){
    if(!buff_full(output_buff)){ // If there's space in the buffer
        output_buff.count++;    // increase the count
        output_buff.buffer[output_buff.tail] = c; // put the character in
// the buffer

        // If we're at the end, wrap the head around
        if(output_buff.tail < MAX_SIZE) output_buff.tail++;
        else output_buff.tail = 0;

        SCCR2 |= SCSR2_TIE0;
    }
}

// puts()
// Write a string to the serial port
static void puts(const char *s) {
    while(*s) {
        putc(*s++);
    }
}

// timer_main()
// Set up the screen, initialize the buffers, set the flags and write
out the time
void timer_main(void){
    setup_vt100();
    io_buff_init();

    SCCR2 |= 0x2c; // Set the RDRF, IDLE, Overrun Error Flags

    TMSK1 = 0x40; // Set the Pulse Accumulator Overflow Interrupt Enable
Bit
    TOC2 = 2000; // Set up the timer output compare 2 to overflow in
millisecond
// increments

    while(1){
        SCCR2 &= 0x2c;
        set_time();
    }
}

```

## Appendix C-4: Interrupt Driven Timer

```

/* Name: Laura Miller
 * Date: 6/8/04
 * File: vt100.c
 * Description: Sets up the vt100 HyperTerminal, implements the
 *              timer interrupt and handles it by setting the
 *              time and printing it out to the
 *              screen every second
 */

#include "hc11.h"
#define ESC "\033"

unsigned int hours = 0,
            minutes = 0,
            seconds = 0,
            milliseconds = 0;

// setup_vt100()
// Sets the vt100 HyperTerminal up at boot up time. Clears the screen,
sets
// the screen to be from lines 1-23 and column width to 80. Moves the
// cursor to 1,1 on the screen
void setup_vt100(){
    puts(ESC "[2J"); //clear the screen
    puts(ESC "[1;23r"); // set window
    puts(ESC "[?31"); // set column width to 80
    puts(ESC "[1;1H"); // Move Cusor to location 1, 1
}

// print_time()
// Calls time to update hours, minutes, seconds on the terminal
void print_time(){
    time();

    puts(ESC "7"); //Save cursor position and attributes
    puts(ESC "[24;1H"); // Move cursor to bottom of screen
    printf("%2d : %2d : %2d", hours, minutes, seconds); //Print time
    puts(ESC "8"); //Restore cursor position and attributes
}

// set_time()
// Sets seconds every 1000 milliseconds and prints the time every
second
void set_time(){
    if(milliseconds >= 1000){ // For each second
        seconds += milliseconds/1000; // Update 'seconds'
        milliseconds %= 1000; // Correct milliseconds
        print_time(); //Print the time
    }
    time();
}

// time()

```

```
// Sets hours, minutes, and seconds accordingly
void time(){
    minutes += seconds/60;
    hours += minutes/60;
    minutes %= 60;
    seconds %= 60;
}

// oc_intp()
// The interrupt that happens every time output compare 2 overflows
@interrupt void oc_intp(void){
    TFLG1 |= 0x40;
    milliseconds++;    // Each time the clock overflows add a
                       // millisecond
    TOC2 += 2000;     // Makes it overflow every millisecond
}
```

## Appendix C-5 ROM Loader

```

/* Name: Laura Miller
 * Date: 5/20/04
 * File: romloader.c
 * Description: The program that's burned onto the ROM and after the
 *              RAM Checker and bootup code is executed it is run, it
 *              then allows for an S19 to be sent through the serial
 *              connection and loads them onto the RAM
 */
#include "hc11.h"

static unsigned char getc() {
    while(!(REG_SCSR & SCSR_RDRF));
    return SCDR;
}

static unsigned char putchar(char c) {
    while(!(REG_SCSR & SCSR_TDRE));
    SCDR = c;
}

static void puts(char *s) {
    while(*s)
        putchar(*s++);
}

// Gets the high half and low half of a byte from hex input and returns
// the number and also incorporates the byte into the running checksum.
#define getbyte() (\
    data = __unhex[getc()] * 0x10, \
    data += __unhex[getc()] * 0x01, \
    checksum += data, data \
)

// Converts hex ascii characters (upper & lowercase chars) to numbers
static const unsigned char __unhex[128] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0,
    0,10,11,12,13,14,15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0,10,11,12,13,14,15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};

// error()
// Output an error message
static void error(const char *msg) {
    puts("Error loading program: ");
    puts(msg);
    puts("\r\nReset and try again.\r\n");

    for(;;);
}

```

```

// zeroAllMemory()
// Sets all the memory from 0x0-0x1000 to zero
static void zeroLowMemory(void) {
    char *addr;
    for(addr = (char*)0x0000; addr < (char*)0x1000; addr++)
        *addr = 0;
}

// _after_ramcheck()
// After the ram checker has been run, this function runs and will
// decode
// the S19 file it receives from the serial port and runs it on the RAM
extern char DIPSWITCH;

void _after_ramcheck() {
    char type;
    unsigned char checksum;
    unsigned char data;
    unsigned char len;
    unsigned short address;

    // Clears all the memory to use
    zeroLowMemory();

    // If Dipswitch 6 is selected, run the timer instead
    if(~DIPSWITCH & 0x20) {
        timer_main();
        return;
    }
    _asm ( "sei" );

    puts("Ready to download programs on to the RAM through \r\n");
    puts("the serial connection!\r\n");

    do {
        // Skip spaces and new lines at the beginning of every record
        do type = getc();
        while(type == '\n' || type == '\r' || type == ' ');
        // If line doesn't begin with S it's not a
        // valid s19 file and throw an error
        if(type != 'S') error("invalid s-record");
        type = getc();

        checksum = 0;

        len = getbyte();
        // Get the high and low bytes of the address which is 2 bytes
        long/16 bits
        address = getbyte() * 0x0100;
        address += getbyte() * 0x0001;

        if(address < 0x0100 && type != '0') {
            error("Space below 0x0100 is in my stack! Change your lkf.");
        }

        // Address = 2 CheckSum = 1 => 3
        len -= 3;
    } while(1);
}

```

```

//While length is not 0, load data into memory from S-record.
while(len) {
    getbyte();
    if(type != '0') {
        *(volatile unsigned char *)address = data;
        if(*(volatile unsigned char *)address != data) {
            error("Failed to write s-record to ram");
        }
        address++;
    }
    len--;
}

// Store the 1s Complement of checksum in length
len = ~checksum;

// If the next byte is not the check sum, throw an error
if(getbyte() != len) error("checksum doesn't match computed
value");
//Stop at EOF S Record
} while(type != '9');

// Once the s19 file is loaded, we goto the user program.
puts("... S19 file loaded, jumping to 0x1040, stack at
0x6FFF!\r\n\r\n");
_asm (" xref __runuser\n"
      " cli\n"
      " jmp __runuser");
}

```

## Appendix C-6: User Program That Runs the LCD

```

// LCD Driver
// Date: 5/31/04 in the wee hours of the morning
// Authors: Laura Miller, Ryan Cormier, Adam Freidin
// Description: Turns the LCD on, prints out a message, then allows
the user
//           to type whatever message they want to through
hyperterminal.
//
#include <stdio.h>
#include "../eeprom/eeprom.h"
#include "hcl1.h"

// Checks the ready bit to make sure that the LCD is ready to receive
another
// command.
#define LCDwait_and_do(X) do { while(addrctrl & 0x80); X; } while(0)
#define LCD_CONTROL(x) LCDwait_and_do(addrctrl = x)
#define LCD_CHARACTER(x) LCDwait_and_do(addrchar = x)

// Need to different access different addresses
// A0 is used to decide if a control command or a character command
// If address is even and in the 7200-73FF it will be a control command
// If address is odd and in the 7200-73FF it will be a character
extern volatile char addrctrl,
    addrchar;

// writeLCD()
// Writes out a string to the LCD
// Waits until the ready bit is set to send another character
static void writeLCD(const char *s) {
    while(*s) {
        LCD_CHARACTER(*s++);
    }
}

static char getc_raw() {
    while(REG_SCSR & 8) (void)SCDR;
    while(!(REG_SCSR & SCSR_RDRF));
    return SCDR;
}

static void putc_raw(char c) {
    while(REG_SCSR & 8) (void)SCDR;
    while(!(REG_SCSR & SCSR_TDRE));
    SCDR = c;
}

static void puts_raw(const char *s) {
    while(*s) putc_raw(*s++);
}

// LCD()
// Initializes the LCD, writes a string to signify that it's working

```

```

void LCD(void) {

    // Initializes the LCD
    // 0x30 - Sets interface data length and selects the 5x7 dot, 1 line
display
    LCD_CONTROL(0x30);
    // 0x01 - Turns the LCD ON
    LCD_CONTROL(0x01);
    // 0x1F - Makes the cursor move to the right
    LCD_CONTROL(0x1F);
    // 0x0E -Turns on cursor
    LCD_CONTROL(0x0E);
    // 0x06 - Sets mode to increment the address by one and to shift
cursor to
    //           the right at the time of write to internal RAM
    LCD_CONTROL(0x06);

    writeLCD("    I'm working!  ");
    puts_raw("I'm working!\r\n");
    puts_raw("want characters: ");
    for(;;) {
        char c;
        c = getc_raw();
        putc_raw(c);
        if(c == '\033') break; // If ESC is pressed stop taking in
characters
        if(c == '\r') {           // If Return is entered, create a new line
            LCD_CONTROL(0x01);   // Clear the LCD
            LCD_CONTROL(0x02);   // Reset LCD scroll and cursor
            LCD_CONTROL(0x1F);   // Move cursor to the RIGHT home
        } else if(c == '\b'){    // If a backspace is entered
            LCD_CONTROL(0x10);   // Move the cursor back
            LCD_CHARACTER(' ');  // Write a space over the character
            LCD_CONTROL(0x10);   // Move the cursor back
        } else {
            LCD_CHARACTER(c);    // Write a space over the character
        }
    }

    // ESC has been pressed, reboot!
    writeLCD("done!");
    // 0x08 - Turn off the LCD
    LCD_CONTROL(0x08);

    // reboot
    (*((void (**)))(0xFFFFE))();
}

```

```
# user.lkf
+seg .boot -b 0x1040 -n .boot
+seg .text -a .boot -n .text # program start address (ROM)
+seg .const -a .text -n .const # constants follow program (ROM)
+seg .bss -a .const -n .bss # uninitialized data (RAM)
+seg .data -a .bss -n .data # initialized data (RAM)
+seg .unused -a .data # unused ram

+def _addrctrl=0x7200
+def _addrchar=0x7201

# These object files are linked into appropriate segments defined above
in the order shown.
crt0.o
user.o
# "..\eeprom\eeprom_sym.h11"

# The linker will look in these libraries to resolve externally defined
calls.
# libm.h11 -> machine lib. (always used)
# libi.h11 -> integer lib.
# libd.h11 -> double precision lib.
# libf.h11 -> floating point lib.
# Do not declare them all, since some implement variations of the same
function. For example,
# printf() is found in both the integer and floating point libraries.
Obviously, the integer
# version is much smaller and is preferred unless you need floating
point support.
"C:\COSMIC\CX11\lib\libi.h11"
"C:\COSMIC\CX11\lib\libm.h11"
```