EEL4744 – Summary of Lab Tips

Mauricio J. Vacas
EEL4744 Spring 2008 TA
May 23, 2008

Contents

Introduction 3

Lab 2 – UF 68HC12 Board Construction & Software Tools 4

Lab 3 – Assembly Programming & Elementary Wiring 5

Lab 4 – Elementary Programming & Adding a Keypad 7

Lab 5 – Software Switch Debounce & CPLD Programming 9

Lab 6 – Microprocessor Port & Memory Expansion 11

Lab 7 – LCD Display & Real Time Interrupt (Stop Watch) 13

Lab 8 – Asynchronous Serial I/O: Interfacing to an External UART and using the Internal UART 16

Lab 9 – A/D & D/A Conversion 21

Extra Stuff – How to debug your non-working board 23

Introduction

This document summarizes a set of tips I created for each lab while I was TA for Microprocessor Applications. The purpose is to supplement the lab documents and clarify any confusion as to the tasks and goals for each lab. Each lab is based on the labs taught in the Spring of 2008 by Dr. Tao Li, although the work is very similar to my own experience when I took the class.

The end goal is to guide students to not only understand how to do the labs, but see how the labs connect with one another and see the big picture of using microprocessors and their interactions with components.

I try to not just regurgitate the information on the lab documents, but expand on it and provide references for more information on the subject. At the end of the document, I provide a set of tips on how to debug boards that don't work.

I hope these set of tips are useful in providing clarity and understanding to students as they progress through this fascinating field.

-Mauricio J. Vacas

Lab 2 – UF 68HC12 Board Construction & Software Tools

The main purpose of this lab is for you guys to finish your boards and boot them so you can run programs on it. In order to do this, you guys should have the following programs installed on your laptops:
a) MiniIDE - http://www.mgtek.com/miniide/
b) Quartus - http://www.altera.com/products/software/products/quartus2/qts-index.html
c) HC12 Simulator - http://hcs12text.com/freesim.html

Because of this you should bring your laptop to this and every lab in the semester.

I'm sure by now everyone's already run through the Board Construction Manual. In order to get more information and start understanding your boards better, I recommend reading through the Board Manual. It will answer a great deal of questions regarding the purpose of jumpers on the board as well as a handy reference when making connections to the headers. I recommend printing it out as it will be heavily used throughout the semester.

The other reference material which will not be used as much this week, but will be used heavily throughout the semester is the Programming Reference Manual. This manual provides all of the assembly instructions the microprocessor supports and how to use them. Therefore, it is very useful for assembly language programming.
You can order a hardcopy of this manual by going to
http://www2.hibbertgroup.com/freescale/main?action=hibbertgroup.client.freescale.ui.inventory.InventorySearchController

Writing under description: 68HC12
Under Type of Item, select Reference Manual
Click search

The item number is CPU12RM and its description is: HCS12 AND M68HC12 CPU12 REFERENCE MANUAL. Just add it to the cart and checkout.

Lab 3 – Assembly Programming & Elementary Wiring

This section serves to clarify any ambiguities in the lab 3 document as well as to provide guidance on how to accomplish the tasks for this lab.
As the lab purpose states, lab 3 is used to introduce students into the basics of assembly language programming and wire wrapping of components.
Part I.

- For the dummy vector, different TA's might approach this differently, but you can assume that the TA might give you this vector as an assembly program to compile and download into the board.
- Remember that using EQU to define a constant is different than a label used to mark an address. In other words:
  - CONST EQU $01, is not the same as

  -
    ```
              ORG  $9000
    LABEL     ABA
    ```
  - The value of CONST is $01, whereas the value of LABEL is $9000
  - Regardless, you can also store an address label as a constant:
    - LABEL EQU $9000
- The breakdown of the address labels/constants are as follows; the following values will be provided in the lab, but for testing students can assume their own values:
  - score_addr: starting address of dummy vector containing values 0-100
  - score_vector_len: length of vector, number of elements in vector
  - grade_addr: starting address of vector containing letter grades determined from the numeric values in the dummy vector
  - You should store all of these values as constants in your program, so that when the actual values are given in lab you can modify them easily.

Part II.

- All wire wrapping is done in the large unused area of the board. Instead of putting the components (LEDs, R-packs) directly on the holes, you should put them on the sockets with large pins for easier wire wrapping.

- You do not need to solder all of the pins of the sockets on the board, just the corners is sufficient to keep the socket in place.
- For the resistor, you can use the 330 Ohm DIP RPACK(should look different than the other chips and have two rows of pins, 8 on each side) included in your kit. To determine which ones they are, use a multimeter to test the resistance between the two sides of the RPACK.
- To write "1s" to DDRT, you can use the memory modify (mm) command on the console you get when you boot up the board. What you want to do is use mm to write $0F on address $00AF. This will configure the lower 4 bits of Port T (which are used for general

I/O) as outputs, instead of inputs. Refer to page 35 and page 50 of the M68HC12B Family Technical Data Guide for more information.

- After setting these bits to output, when you write to address $00AE, you should see the corresponding values light up on your LED. (e.g. if you write $03, you should see the bottom two lights on your LED light up)

Part III.

- The SIP resistor pack is the one that only has one row of pins. Remember how these work, there is one pin (the one on the side where the marking is) on that resistor which is connected to all of the other pins. This pin should be connected to +5V. The other pins represent the other side of the resistor which you connect to the switches. By using this resistor you don't have to do as much wire wrapping (if you were using a DIP package you would have to connect the other end of each resistor to +5V)
- In order to read the values of the switches, you use memory display (md) command on the console on address $00AE. Remember you configured the upper 4 bits of Port T to work as inputs (by setting them to 0 on DDRT (i.e. address $00AF)), therefore when you read in this address you should see the values of the switches.
- With this information, you should be able to read in the values from the switches and use that information to write it out to the LEDs continuously to perform the task specified in the lab document.

Lab 4 – Elementary Programming & Adding a Keypad

Part I

- As with Lab 3, use constants for easy editing of addresses.
- In this part you will be given a vector of 2's complement numbers (i.e. they may be negative) and order them from greatest to smallest. The term magnitude refers to value, not to absolute value. Therefore, if you have the following vector: (1, -9, 5, -3), you should order it and store it like this: (5, 1, -3, -9).
- There are many sort algorithms in existence (i.e. bubble sort,merge sort, quick sort, etc...), so if you don't know where to start you can take a look at these to get some idea. Some are more complicated than others, so don't just pick one and try to implement as it may be very difficult to write in assembly. Try to look at them and visualize which one you can write in assembly.
- The lab suggests you run the sort program as a subroutine of your main program. There are specific instructions which can help you with this:
    - JSR: Jump to Subroutine (pg 171 of CPU12 Reference Manual)
        - Jumps to an address (e.g. your subroutine)
        - Stores the address of the instruction that follows JSR as the return address. This address (which gets stored in the stack) is then used by the instruction RTS to return back to the main program.
    - RTS: Return from Subroutine (pg. 246 of CPU12RM)
        - You use this instruction to return to the main program, executing the instruction right after JSR. The return address is retrieved from the stack.
    - You can see an example of how these instructions work by looking at and running in the simulator the following example program found in the MIL 4744 site:
        - http://mil.ufl.edu/4744/examples/sub_cntr.asm
- The other task which you need to do is pass parameters to the subroutine using the stack.
  A stack is a very common computer science data structure, you can imagine it as a stack of pancakes, you can only put or get pancakes from the top of the stack.
  In this case, the pancakes are data stored in the stack that you either PUSH (put) or PULL (get) from the stack. The stack exists in memory and the address where you PUSH into or PULL from is determined by the stack pointer (SP). This type of structure is a LIFO (last-in, first-out) type structure.
  There are specific instructions you can use to perform this:
    - LDS: Load Stack Pointer (pg. 193 CPU12RM)
      At the beginning of the program you need to define where in memory you're going to store your stack. This is the instruction you use to do that. Remember to choose an address which is high in memory since every time you store something into the stack, the stack pointer will decrement. (so the stack goes

from the address you specify downwards through the address space with every PUSH).
    - PSHA, PSHB, PSHX, PSHY: Push the value stored in A, B, X, or Y in the stack.
    - PULA, PULB, PULX, PULY: Retrieve value from stack and store into A, B, X, or Y.
    - All of the PSH and PUL instructions can be found in pgs 222-233 of CPU12RM.
    - **REMEMBER:** When you use JSR, the return address is pushed into the stack, therefore before you retrieve your parameters, you need to store the return address somewhere (i.e. one of your registers)
    - Run through the following example to understand how stack operations work
        - http://mil.ufl.edu/4744/examples/stack1.asm
- Hopefully, with this information you'll know how to do your subroutine and pass the parameters (orig_addr, orig_len, sorted_addr) to it through the stack.

Part II

- **NOTE:** The address of the DDRP (Data Direction Register for Port P) and PORTP can be found on page 44 of the M68HC12B Family Technical Data Guide .
- Before reading the instructions for the keypad, read the Keypad Handout keypad.pdf.
- Therefore, you use the output pins of PORTP (P3:0) to activate a specific column. Since the column pins are active low, you activate a specific column by sending a low ("L") value to it. This is why when you send an $E signal on P3:0, you activate COL1 on the keypad.
- Once you have activated a column, you can detect whether a key on that column has been pressed. When a key is pressed, it will reflect on P7:4 of PORTP.
- By writing a loop that cycles through sending the signals $7, $B, $D, and $E to the columns (P3:0) and constantly reads the input from the rows (P7:4), you can read which keys have been pressed on the keypad. You can send this value to your LEDs using PORTT.

Lab 5 – Software Switch Debounce & CPLD Programming

Prelab materials
Part 1
1. ASM and LST files hardcopies and softcopies for Part 1 program.
Part 2
1. Logic design for inverter

Part I.

1. The goal of the previous lab was for you guys to create a program that can read and decode a key pressed in the keypad and send this value to the LEDs. The big picture is that with that program you can now use the keypad for any application. In this particular one you are going to use it to increment a counter and clear it.
2. The problem with this application is that since the voltage "bounces" when you press and release a key**.** This would increment the counter several times, instead of just the one time the key was pressed. Therefore, a delay needs to be inserted on both ends to debounce it**:**

- **DELAY:** To insert a delay you need to use a loop. Basically the way to think about is you want a certain number of clock cycles to pass so that eventually you're sitting in the loop for between 50-100ms. The microprocessor is running on a 4 Mhz clock, so let's say that you run a loop 5000 times and each time you are 4 clock cycles in the loop, therefore you are spending 20,000 clock cycles in the loop. That, in ms, is 20,000/(4 * 10^6) = 50ms. You have to check the number of clock cycles it takes to execute each instruction in the loop to perform this calculation.

3. Therefore the instructions are:
   a. Test for key pressed: same way as you did in Lab 3.
   b. If pressed, insert delay (as explained earlier)
   c. Check when a key is released: **This means that if the key has not been released, then you shouldn't go back and check whether another key has been pressed. Keep the program waiting until you detect the release of the key.**
   d. Once released insert another delay and return to step 1.
4. The TAs will show you how to use the LSAs to measure the bouncing and delays on your program.

Part II.
1. Similar to what you did when building your board, the task here is to take an input from one of the switches to one of the free I/O pins of your CPLD, pass it through an inverter, and output it to one of the LEDs.
2. The key in this lab is knowing how to map pins created in your Quartus design to pins on the CPLD, which eventually go to pins on the J16 CPLD I/O header.
3. Therefore, you would connect your I/O (one switch and one LED) to a pin on J16. This pin

corresponds to a pin on the CPLD, the specifics can be seen in the board manual on page 7.
4. Then after identifying the pin and compiling your design, you can go to the pin assignment editor on Quartus to specify the physical pin on the CPLD that would correspond with the input and output pins on your logic design.
5. Most of the hardware for this part should already have been installed (switches, resistors, LEDs) from your work in Lab 3.

Lab 6 – Microprocessor Port & Memory Expansion

Pre-lab requirements:
Part 1
1. Hand drawn schematic of latch with pin numbers connected to the board headers with appropriate pin #s reflected.
2. Logic design (BDF or VHDL) of decode circuitry

Part 2
1. Data pin-out for the SRAM device
2. Logic design (BDF or VHDL)
3. .LST hardcopy of memory tests T1-T3

Part I. Port Expansion
1. The process of searching for the datasheet of any IC is a common and important task. Usually Googling the part number of the chip is enough to give you the datasheet. Note that the chip that you have on your kit might vary from the exact part number mentioned on the lab document. Remember that there are various manufacturers of the same type of chips so even though the part is different the function should be the same, regardless try to find the data sheet for your specific part.
2. As per the datasheet, the 74574 chip is an 8-bit D flip-flop with tri-stated outputs (remember from Digital Logic). What does this mean? It has:
   a. 8 inputs (D)
   b. 8 outputs (Q)
   c. A clock input signal
   d. An active low output enable, which if true, then Q = D, else Q = Z (high impedance)
   e. Ground and power pins
3. As has been mentioned in the past, the pinouts for the databus (header J30) can be found in the board manual. You should connect the inputs of your latch (D) to the pins on the databus (the specific pins can be seen on page 12 of the board manual)

KEY TIP #1: How to decode an address
4. You need to connect your clock signal to your CPLD through an I/O pin. Remember that the flip flop will only store values when it is triggered by a rising edge of the clock signal. Therefore, if you only want the flip flop to store values when you are in a specific address range, you need combinatorial logic to do this.
For example, let's say that I wanted a flip flop to only store values when the address is somewhere in the range of $1000-$1FFF, then you would have the following logic equation:
Clk = /A15 * /A14 * /A13 * A12 * ECLK

In the case of the lab, you need the address range to be from $9000-$97FF. Therefore, you might need to connect a couple of address pins to the CPLD I/O in order to create your decode equation.

What this all means, is that when you write to any address location in the range $9000-$97FF, the data will be stored in the D flip flop latch you added.

6. To verify that it works correctly you can use your multimeter to measure the voltages on the output of the latch (e.g. if you write $71 to address $9100, you should see the following sequence of voltage output on your flip flop LHHH LLLH).

Part II. SRAM interfacing

1. This process is similar to the D Flip Flop. You have a 32K x 8 SRAM that you want to treat as an 8K x 8 SRAM, therefore you need to limit the upper limit of the address range in your decode equation. Your starting address is $A000, so based on the fact you want the SRAM to act as an 8K memory, what should the upper limit of your address range be?
You will use this for your decode equation.

2. This time, instead of hooking up the Clk signal to your CPLD, you will connect the active low chip enable (CE) to a CPLD I/O pin. You will use your decode equation to determine the value of this signal. Do you need ECLK in your decode equation?

3. A couple of clarifying points:
KEY TIP #2: There's more wiring to be done beyond the lab document

I/O corresponds to SRAM data input/output, where do you connect these?
CS is the Chip Enable (CE)
/WE acts the same as your R/~W, this pin needs to be connected so the SRAM knows whether you are writing to it or reading from it. Refer to page 2 of the SRAM datasheet for more info.

Remember to connect the 0.1 uF capacitor between power and ground to reduce noise.

Part III. SRAM memory testing

These procedures are fairly simple. If your SRAM works correctly, these programs should run similarly as if you were writing to and reading from the 1K SRAM on the board, but for a different address range.

Lab 7 – LCD Display & Real Time Interrupt (Stop Watch)

Pre-lab material
1. Schematic of the LCD connected to the latch
2. Timing diagram of writing nibbles to the LCD
3. .LST file for LCD code to sent out your full-name
4. .LST file RTI code to create the stopwatch

Part I

1. Verify the LCD documents posted on the class website. Here are the addresses for more information:
http://mil.ufl.edu/4744/docs/lcdmanual/lcdmanual.html
and
http://mil.ufl.edu/4744/software.html

2. Connect the LCD to the latch as shown in step 1 of the lab document.

3. You need to connect the potentiometer (variable resistor) to power and ground and connect the middle pin of the potentiometer to the contract adjustment pin. As you move the knob on the potentiometer, you adjust the voltage of the signal going to the middle pin. The ideal contrast for the LCD is between 3.3 - 3.7 V.

4. The idea of this lab is that you send commands/data to the LCD by writing to the latch you attached in the previous labs. The upper four bits (Q7:Q4) correspond with the nibble you're sending. Then you have R/W (1 for Read, 0 for Write), RS (which decides whether you are sending commands (0) or data (1)), and the enable (E) signal. Every time that you send a nibble you have to send it two, one with the enable signal asserted and another with the enable signal cleared.
For example, to send the byte $38 to the LCD, assuming you're sending a command, you would have to do the following:
mm $9000 $34 ; send $3, E set
mm $9000 $30 ; send $3, E clear
mm $9000 $84 ; send $8, E set
mm $9000 $80 ; send $8, E clear
By doing this you are toggling the enable signal which is used to capture the nibble by the LCD.

5. You can send nibbles through the console and not have to worry about delays since the time it takes to write two nibbles is much greater than a couple of ms. For the program, you do have to worry about delays between sending commands, these can be seen in the LCD documentation. To create delays you will do what you did for Lab 4. My advice is to create a

subroutine which a delay specified by a variable you pass through the stack, but you can choose your own method.

6. The first thing you need to do when you're programming the LCD is initialize it. Again, this can be found in the documentation. When you send all of the commands you should see a cursor on the LCD screen.

7. Remember that for the digital design part, the output of the AND gate for the latch decode equation must be inverted and the clock input should not be inverted.

8. The commands needed to write your name on the LCD and the values of the ASCII characters can be found in the LCD documentation.

Part II.

1. A couple of examples on how to use RTI can be found:
http://mil.ufl.edu/4744/examples/rtia.asm
and
http://mil.ufl.edu/4744/examples/rtib.asm

2. The idea of the interrupt is it is a way of interrupting a program whenever an event has occurred. In this lab, that event is going to be when a certain amount of time has elapsed using the real time interrupt (RTI). Once the interrupt has been initialized and enabled, whenever the interrupt is activated it will interrupt the normal program and look in the interrupt's vector location (IVR).

3. The data located in the IVR located in the EEPROM is the address of the interrupt service routine (ISR). The ISR is the (you can call) subroutine that is executed when the interrupt is triggered.

4. Therefore, in this lab, you will be setting the RTI to fire every tenth of a second. When the program enters the ISR, it should check whether the program is in start mode or pause mode (use a global variable) . If it is in start mode, it should increment the time on the LCD. Otherwise, it should not do anything.

5. If at any time in the program the 0 key is pressed, the time in the LCD should be reset to 00.0 (should you have this in your ISR?). If the 1 key is pressed, you should update the global variable accordingly.

6. Remember, that if you do not use the procedure in lab 5, if you pressed the 1 key, the bouncing effect would make it seem like you pressed it several times. The program might start and pause the LCD time several times.

7. Before you can use the RTI, you need to initialize it by configuring certain RTI registers. These

can be found in the examples posted above and in the 6812 manual. The manual will tell you what configuration you need in order to set the RTI to fire at the time you want it to.

Lab 8 – Asynchronous Serial I/O: Interfacing to an External UART and using the Internal UART

IMPORTANT: Lab 8 is a long lab that requires a lot of wire wrapping and going through documentation in order to understand how to use the UART and the SCI system. Therefore I recommend starting this lab early to allocate enough time for all of the parts.

Note: While the pre-lab documents mentioned here tries to summarize the info on the lab document, you should always verify with the lab document for the specific pre-lab requirements.

Prelab
1. Create a hand-drawn wiring schematic of the entire UART system. This includes the 68HC12 pins, DB9 Header, Clock, C550, and DS229 chips. Use labels instead of wires to show connections
2. Provide the UART chip-enable equation and BDF printout of the added CPLD logic
3. Provide .LST for the input read/polled write program
4. Provide .LST for the polled read/write display program
5. Provide .LST for the interrupt-based UART read/write program
6. Provide .LST for the Baud Rate Selection program

Part I
1. As the document states, look over the lab documents to get an idea of the parts you will be using. In particular, focus on the UART / RS-232 notes.
Keep in mind that it's a lot of information at first glance, so don't worry if you're overwhelmed, as you progress through the lab you'll understand.

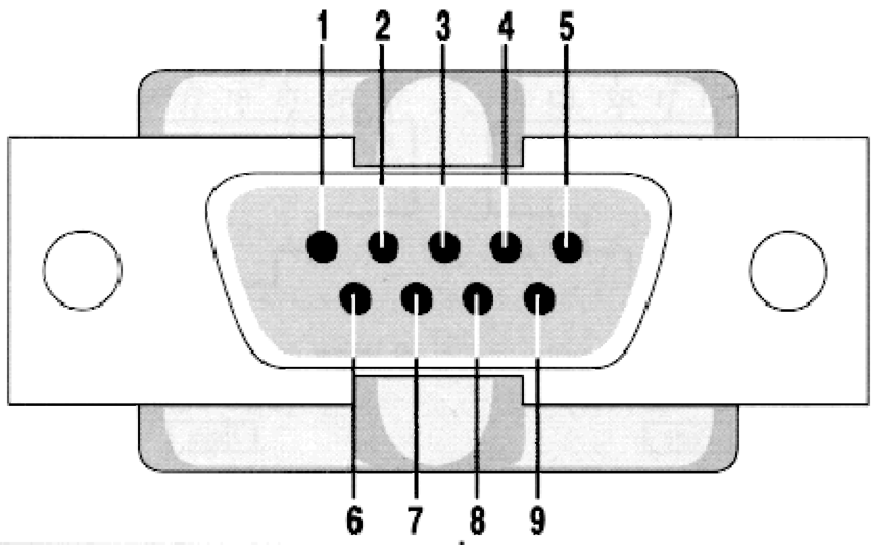
2. Before wire wrapping, you should identify the parts you require. The UART is a big chip (40-pins) therefore it should be pretty easy to identify. Always remember the trick is to match the number of the part required (i.e. 16C550) to the part # on the chip. It doesn't have to match up exactly as different manufacturers use additional symbols to identify the part.
Furthermore, I know I had a student who had a 16-pin RS-232 part instead of a 20-pin. In this case, you should look for the datasheet of your particular part before progressing to your wire wrapping. I will try to find which part this is and send it to Dr. Li to post it on the website.

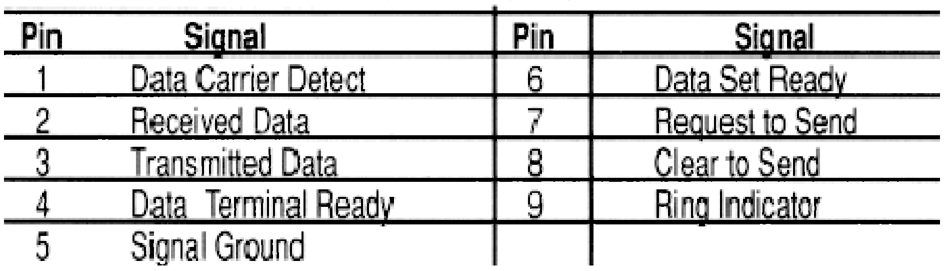
3. Page 2 of the UART/RS-232 Notes shows how to wirewrap the UART to the uP.

4. You will need to create a new decode equation for the UART and map it to the address range $8000-$8FFF. You should be able to write and read from the UART, just like you do in the SRAM

(i.e. therefore your decode equation should not differ too much from the SRAMs).
The GDF is a typo in the lab document, assume that wherever this is stated it means either a BDF or VHDL program.

5. Wirewrap the RS-232 chip as stated in page 3 of the UART / RS-232 notes. Also you have to connect the RS-232 TX/RX signals and ground to the DB9 connector as stated in the document. Below is an image of the DB-9 connector for clarification of the part, the pin #s are actually engraved on the DB-9 connector, use those, as the image shows the pins in reverse position:





Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

6. The next step is to check to see if you can write to the line control register (LCR) on the UART at address $8003. Use mm and md commands to write to and read from this address to verify if you can write/read the UART. Information regarding the UART's accessible registers can be found on pages 19-20 of the UART spec sheet. Even though you have the UART mapped to the address range $8000-$8FFF, there are only three address lines connected to the UART (A2:A0). Between A2:A0 and the divisor latch access bit (DLAB), you can access all registers in the UART. The DLAB is a bit that is part of the LCR and can be modified by writing to the MSB of the LCR in address $8003. The LCR is a very important register in general which you will need to use to

operate the UART, for more information on this register, look at **page 24** of the UART spec sheet.

7. Connect a serial cable to your board using the DB-9 and your PC. Use Hyperterminal (or similar) on your PC to communicate serially with the board. In the settings you can adjust properties such as the BAUD rate (explained below)

8. The program you need to write is not very different from other programs you've done in the past.

a. You know how to detect keys pressed on the keypad using your programs in the past. Use this info to create a program that scans the keypad to determine which key has been pressed and then sends out its ASCII equivalent to the PC. (Do you need to debounce the key?)

b. In order for the UART on your board to communicate with your PC, you need to set the BAUD rates for these two equal. The BAUD rate is a measure of the transmission speed of data. Therefore, if two systems do not use the same BAUD rate then one system will be sending information too quickly or too slowly for the other system to capture.

c. The BAUD rate of your UART is controlled by modifying the divisor latches on the UART. Essentially, the UART has a 16-bit divisor latch (split into two   bytes located in $8000-$8001) which divide the input clock (thereby slowing it down) in order to send data at a specific BAUD rate. To access these latches you need to set the DLAB bit to 1 and then modify the address locations $8000-$8001 as can be seen on the table in page 20. For more information on how to set specific BAUD rates look at **page 29** of the UART spec sheet. Remember that the clock mentioned there might not necessarily be the same clock you are using.

d. Remember that in order to access the Transmitter Holding Register (THR), you need to clear the DLAB bit, otherwise you would be reading the divisor latches.

e. As the lab document states, you want to poll the Transmitter Holding Register Empty (THRE) flag which is bit 5 on the Line Status Register (LSR) until you get a high signal. Once this happens, then you check whether a key has been pressed, if so you send the ASCII equivalent of the key to the THR register ($8001 when DLAB=0). Repeat this process continuously to keep trasnmitting keys to the PC.

f. Through this procedure, you should be able to type keys on your keypad and see them displayed on Hyperterminal on your computer.

As I had mentioned previously, for the DB9 header, make sure you use the pin #s that are marked on the actual device, not those in the image. The ones in the image are from a bottom-up look (upside down) than what you see on the part.

Part II.

NOTE: This part is overall similar to Part I, except instead of transmitting data to the Transmitter Holding Register (THR) (address $8000), you will be receiving data to the Receiver Buffer Register (RBR) (address $8000). Both of these are in the same address location, what differentiates them is whether you are writing to or reading from that address location.
The data being received will be from the keyboard on the computer you are running HyperTerminal from.

1. Initialize the UART to the settings that will match the HyperTerminal settings (BAUD rate, etc...). Also, initialize your LCD.

2. Before reading, you need to make sure that there is actually data waiting on the RBR. That is what the DR flag is for (found in the Line Status Register (LSR)) address can be found in table 3 on page 20. So you should poll this flag until it is high and then read from the RBR.

3. Take the value received (ASCII character) and display it on your LCD.

Part III. Write/Read using interrupts

NOTE: The basic idea of this part is to use interrupts for reading and writing data on the UART. In the last lab, you used the RTI (Real Time Interrupt), which interrupted after a given time had passed. In general, interrupts trigger when a specific event has happened.

1. In this part, that event will be a true value from the INTRPT output of the UART. When the IRQ is configured properly, this signal will interrupt the execution of a program running on the microprocessor (i.e. will go into the IRQ's Interrupt Vector Table, look for the Pseudo-vector, and then go to the IRQ's ISR to perform some function). The pseudo-vector will be different than that of the RTI and can be found in the 68HC12 Technical Manual (go to the interrupts section (Ch. 4))
Also, you will need to find the IRQ enable bit which is in one of the control registers (can also be found in this chapter)

2. Furthermore, for the UART, you will also have to enable interrupts. These can be found in the Interrupt Enable Register (address $8001). The trick is to alternate between enabling the Enable Received Data Available Interrupt (ERB) and the Enable Transmitter Holding Register Empty Interrupt (ETBEI).

The former sends an interrupt when it detects that there is data available in the RBR. The latter sends an interrupt when it detects that the THR is empty.
In order to determine which interrupt has been fired, you use the Interrupt Identification Register (IIR) (address $8002), look at page 24 on the UART datasheet for the id codes.

3. Remember that regardless of whether an RX or a TX interrupt fired, they will both go to the same ISR (the IRQ's ISR). Therefore, you will have to identify at the beginning of you ISR which interrupt has fired.
The sequence is: initialize IRQ and UART, enable the RX interrupt. For an RX interrupt, read the RBR and store value in memory, disable the RX interrupt, enable the TX interrupt. When the TX interrupt fires, read memory location where you stored that character received and write the character to the transmit buffer (do you need to poll the THRE flag?), disable the TX interrupt, and enable the RX interrupt.

The effect of this program is that data sent from the keyboard connected to the PC running HyperTerminal will be echoed back to the PC. You will see a duplicate value of each character pressed on the keyboard.

Part IV. 68HC12 Serial Communication Interface (SCI) System

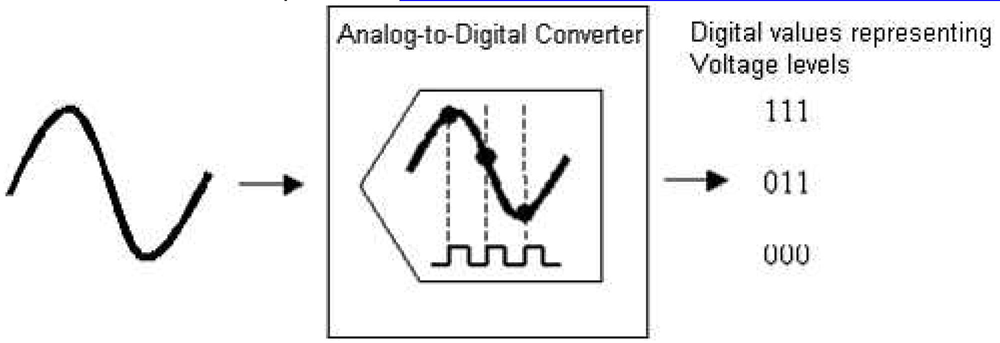
1. The SCI system works very similarly to the UART system (i.e. you have to alternate RX and TX interrupt enables like in the UART, you have flags which tell you whether the Transmit Register is empty or there is data available)

2. All of this info is compiled in Ch. 14 of the 68HC12 Technical Manual. It provides info on the registers you need to use to enable/disable interrupts, flags, and configuring the appropriate BAUD rates. With the information there and the instructions in the lab document, you should be able to program the SCI to perform the functions needed.

Lab 9 – A/D & D/A Conversion

Part I.

NOTE: An A/D converter takes as an input a continuous (analog) signal and digitizes it by taking discrete samples of the analog signal at periodic times (e.g. the sampling rate). A D/A converter conversely takes discrete samples (such as bytes in memory) and converts them to continuous analog signals by filling the gaps between the samples.
A/D conversion example from http://zone.ni.com/devzone/cda/tut/p/id/3016:


Analog-to-Digital Converter
Digital values representing Voltage levels
111
011
000

1. The documentation for the on-board HC12 A/D converter can be found in Chapter 17 of the 68HC12 Technical Manual.

2. The ANx pins are located in header J4. The pinout can be seen in page 14 of the board manual. It says ANA pins, but they are these are connected to the ANx pins of the microprocessor, therefore treat them the same (i.e. ANA1 => AN1)

3. Pay close attention to section 17.5 of the 68HC12 Technical Manual, details regarding initialization and reading from a potentiometer can be found there. The addresses used in the example do not reflect the addresses you should use in your program (i.e. there's no memory in address $7000).
In particular pay close attention to the values sent to the control registers ATDCTL2, ATDCTL3, ATDCTL4, ATDCTL5 and why these values are being set. Read the descriptions of the bits which are being set/cleared so you understand why these values are used (i.e. LDAA $80, STAA ATDCTL2 will enable the A/D converter, while disabling other options).

4. For ATDCTL4, the description states:
The reset state divides the P clock by a total of four and is appropriate for nominal operation at a bus rate of between 2 MHz and 8 MHz.
This is why the LSB of ATDCTL4 is set.

5. The ATDCTL5 is used for initiating single conversion sequences or setting a continuous conversion sequence. In the example, they initiate a conversion on channel AN6 based based on the values written to the control register.

6. The ATDSTAT contains the flag which indicates whether the conversion has completed (in its MSB).

7. The result of the conversion is stored in ADR2H (again, this was determined in what was written to ATDCTL5). ADR2H is part of a set of ATD Result Registers (17.3.10) which store the result of the A/D conversion. Do not confuse these with the Port AD Data Input Register (17.3.9) which takes in digital input data.

Part II.

1. Add the latch HC574 to any open address range that has 2^10 bytes. Connect the HC574 D inputs to the data bus on the board and the Q outputs to the DAC.

2. Wire wrap the DAC as the diagram illustrates. Hardware wiring instructions are pretty detailed in the lab document.

3. For details on how to use the Output Compare feature on the board look at Chapter 12 of the 68HC12 Technical Manual. In particular, pay close attention to section 12.5 which describes how to use the output compare to generate a square wave (not the same as the triangle wave you need to create.

4. Create a vector of 20 values which represent a triangle wave (values should start at $0, ascend to $FF, and descend to $0)

Part III takes what you learned in Parts I and II together so you can configure the particular frequency you want your triangle wave to output at.

Extra Stuff – How to debug your non-working board

You've finally finished soldering all of your boards' headers, you've programmed the CPLD with the correct equations, and you connect your board to your PC and press the reset button and… NOTHING HAPPENS!

This is by far one of the most frustrating occurrences for students after dedicating so much work to building their boards. I have outlined a series of steps that are helpful in debugging non-working boards. There are two possibilities regarding non-working boards:

1. The board never worked
2. The board worked until I attached XYZ, now it doesn't boot anymore

# 1 is by far more complicated as the list of possible problems associated with your board range from a badly soldered pin on your microprocessor to a malfunctioning CPLD chip. I will outline some debugging steps to take for #1.

For # 2, the easiest remedy is to bring the board back to its "last working state". Take off any new components you've added including their wirewraps (don't just pull the chips out). These problems can range from wire wrapping the wrong pins to two wires touching somewhere. Now, why would two wires touching cause your board to not boot? Imagine if the two wires are D1 and D0 on your databus. All of a sudden all the values that D0 have will be the same as those that D1 has. Since the board uses the databus to transmit the data of the monitor program located on the EEPROM, if this data is getting corrupted it will not boot. The same thing would happen with address lines.

Now, here are things to watch out for if you're stuck in #1:

1. Didn't push chips completely in the sockets
2. Did not set up the equations correctly on their CPLD
   a. Reset, DBE, and ROM_OE are active low, students need to take this into account in their designs
   b. All of the I/O need to be assigned to the appropriate pins on the CPLD, these can be found on the Board Manual
3. Forgot to wirewrap the two pins (Step 25 of the Board Construction Manual)
4. USB Cable (Have to go to Device Manager and watch out for an exclamation mark next to the driver)
5. Incorrect COM port set up in MiniIDE options
6. Pins might not be correctly soldered

   1. To detect whether you have a broken trace, you need to perform connectivity tests. To do this you use your multimeter (MM)(particularly useful if you can use ones that beep when there is connectivity, there are a couple in the lab). I would suggest testing connectivity in this order:
      - Microprocessor pins

   - o CPLD pins
   - o EEPROM pins

2. This is where the board manual comes in handy. By looking at the schematics (MP - page 19, CPLD - page 11, EEPROM - page 13), you can see which pin goes where on the board. For example, for the MP, pin 45 (PA7/DATA15/ADDR15) goes to the address latch U7 (page 8) to pin 18 (AD15). By putting one of the multimeter leads on pin 45 of the microprocessor and the other on pin 18 of U7, you can test whether there is connectivity between these two pins. Remember that, pin 1 on the chip corresponds to where the dot on the chip lies.

   By going through this process you can test whether there are any broken traces (no connectivity between two pins where there should be).

3. The other way of testing what is wrong with your board is by using an oscilloscope. By putting the black alligator clip on ground (you can use pin 10 of the JTAG header or solder some headers on where it is marked ground near the unused parts of the board) and the other sensor on the pin you're trying to observe, you can see on the oscilloscope the actual analog signal coming from it. Some useful pins to observe are: the pins on the oscillator, the control signals on J6, the CPLD jumpers on J5, the microprocessor, CPLD, and EEPROM pins. The oscillator should have a 4 Mhz sine wave coming out of one of its pins, the ECLK should have a periodic quasi-square wave.
   Do not focus too much on the oscilloscope, as they might suggest that there are problems on the board, but not explain what or where. Connectivity tests give more definite answers as to where the problems are on your board.

4. Another step is to use the Logic State Analyzer (LSA) to observe what is happening on the databus and the address lines and detect whether there is any erroneous behavior (e.g. such as two pins that always have the same value no matter what).