Experiment 8

Chad Huard :: ee6681@wayne.edu :: Lab 6: 3/23/10 :: Report: 3/28/10

Introduction:

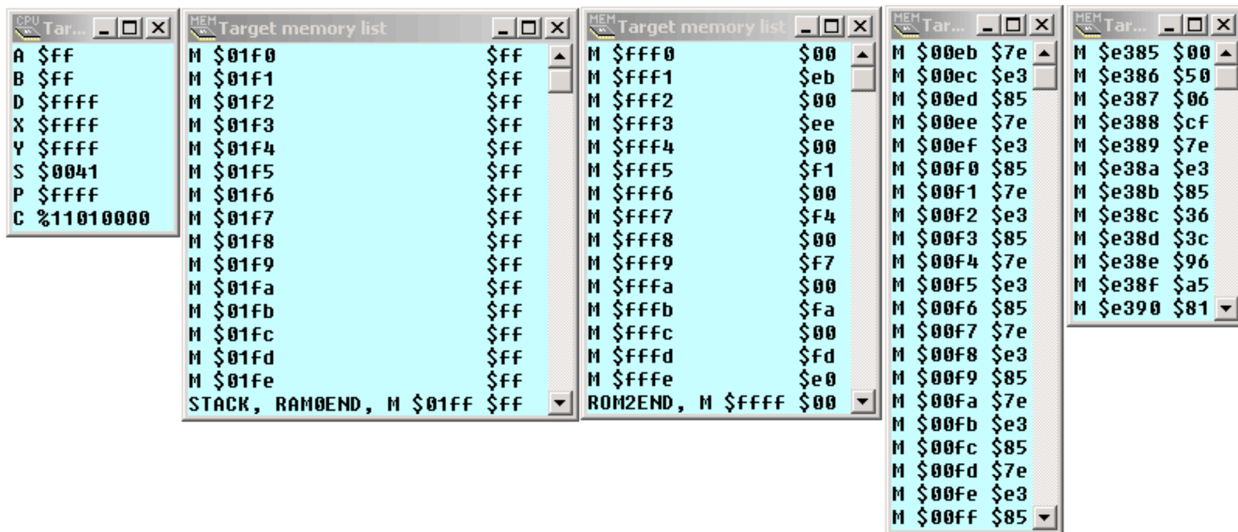
This lab explores the interrupt, its uses and its use of the stack. Without interrupts, a processor is required to poll each condition that it is required to react to. Whether it is checking to see if a button is pressed, or if a bit has been received on a serial pin, the processor is spending valuable time checking on environmental conditions when it it may have other things that are waiting to be done. To solve the problem of needing to constantly poll the status of certain conditions the HC11 implements an interrupt system. This interrupt system allows specific events to grab the processor away from what it is doing and handle the event. The ability to do this is crucial for events where timing is important (such as serial or parallel data transfer).

This lab will deal with the IRQ interrupt, which will be used to react to a button press, and the SWI interrupt, which will be used to illustrate the interrupt's use of the stack to store the environment.

Experiment:

1. Use the THRSim11 'Target' menu to follow the jump vectors for the SWI instruction:

Here are the various memory locations and processor registers on the board after reset:



The interrupt vector for the SWI command (at $FFF6) is $00F4 which contains the command JMP $E385. Therefor the actual location of the SWI service routine is $E385. Using the target disassembler we can see the commands at this location:

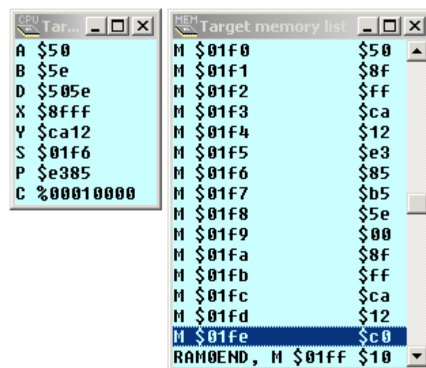
The first two lines of code are used to set the X and I masks and clear the S bit in the condition codes. This allows the processor to be stopped. Since the I and X bits are set, the only thing that will wake the computer is an XIRQ interrupt, but the XIRQ service routine *will not be performed.* If the XIRQ interrupt occurs the program will execute the next line of

code which will jump back to the same location as SWI. This makes it essentially impossible to wake the processor from a SWI with this configuration. If you want the processor to be able to use the SWI and awake on an XIRQ you could re-write the service routine replacing the JMP at the end with RTI. This new service routine would need to be placed in user RAM and the jump vector changed to point to the new service routine instead of the default routine in the EEPROM.

To illustrate how the interrupt uses the stack to store the environment at the moment of interrupt we use the program:

```
ORG    $C000
LDS    #$01FF
LDX    #$8FFF
LDY    #$CA12
LDAA   #$A2
LDAB   #$5E
ABA
SWI
```

Stepping through the program one line at a time, I noticed that the processor seems to use the memory just under the stack pointer for many things that are not specifically mentioned, but once data is explicitly written to the stack it is not overwritten. After reaching the STOP command in the SWI service routine described above the registers and stack memory look like this:



Notice the stack pointer is pointing to $01F6 and all data below that is garbage as described above. If we were to pull from the stack at this point we would retrieve the condition codes ($B5), ACCB ($5E), ACCB ($00), XR ($8FFF), YR ($CA12) and finally the PC ($C010). All of these are the values that the registers had when the SWI was called. The program counter points to next instruction after the SWI in the main program. If you continue to hit the step ahead button you will come to the JMP $E385 and just stay in the SWI loop forever.

2. Consider the following program and its associated IRQ service routine:

a) Estimate number times IRQ will be called with a quick flick, then test estimate with on-board experiment:

This is a fairly straightforward calculation. When the button is held down the IRQ will be called repetitively with a period to the length of the IRQ service routine.

Stack Env.†		14
LDAA	EXT	4
STAA	EXT	4
INCA		2
CMPA	DIR	3
BNE		3
STAA	EXT	4
RTI		12
	Total Cycles:	46
	Time:	23µs

† Assuming the IRQ takes the same time as SWI

With with a period of 23µs we can estimate that the number of times the IRQ will be called with a very short flick of the button (between 1ms and 5ms) will be between 44 and 218 times. In my experiment using the board I found my average flick to cause the IRQ to cycle 136 times. This gives an average time of 3.218ms per flick. I must note that the technique of flicking the switch takes some practice, a “normal” fast press of the button will cause the counter to overflow 3 or 4 times (I modified the code to increment an overflow counter to verify this).

b) Add the code below at the proper location in the IRQ service routine and test the program. Document what you observe.

```
PULA
ORAA  #$10
PSHA
```

This code, when placed inside of the interrupt service routine, masks the IRQ interrupt after the program returns from the routine. Essentially it pulls the condition codes from the where it was stacked when the IRQ was called and replaces it with the IRQ bit set. When the RTI pulls the environment form the stack the IRQ bit will be set masking the interrupt. Therefor, the IRQ will only be called once, and any further pressing of the button will not effect the program because the interrupt is masked.

c) Modify the main program so that Port B increments by one every time the switch is pressed and then released:

There are many different ways to accomplish this. The hint in the problem statement recommends masking the IRQ and delaying for 1/3s outside of the IRQ service routine. After experimenting with this technique, I found it to be difficult to implement. If the delay is inside the main loop there is a chance that the IRQ will be called right before the end of the delay, in which case the delay could be very short. This could be solved by checking the I bit every time through the loop, then running the delay and only clearing the I bit at the end of the delay. The problem with this is that the processor

spends the entire 1/3s delaying, which wastes a lot of processor time. This technique is also limited to very quick presses of the button.

Another route is to place the delay inside of the interrupt service routine. This alleviates the need to check the I bit in the condition codes every time through the main loop, but it still wastes and entire 333ms of processor time doing nothing. This technique also will only work for very short presses of the button.

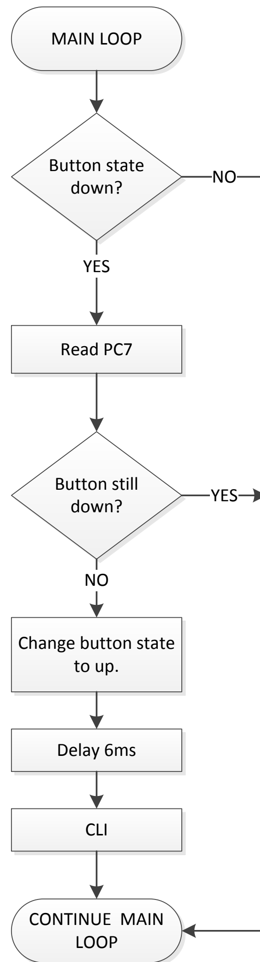
The technique I chose to use requires the button to be connected to the IRQ and a pin on port C, in this case PC7. The code uses a variable ‘STATE’ to keep track of the button state - whether the button is currently up or down - and the program will not clear the I bit until the button has been released. This solves the problem of needing to press the button very quickly: it can now be held for as long as you like without the IRQ cycling. Also, it allows the processor to run other code for the vast majority of the time even when the button is held down.

This technique adds some overhead to the main loop. When the button is up there is one branch that checks the button state, but when the button is down each time through the main loop port C is polled to see if the button is released. This is still preferable to the other options since the processor is free to do other things and only polls the port once through the loop. This assumes that the contents of the main loop itself take an appropriately short period of time for the accurate polling of port C.

Even with the program toggling the ‘STATE’ condition, there is still the need for some delay in the system. This is because of the mechanical bounce in the button itself. If the processor is allowed to run full speed, when the button is pressed there is a bouncing condition where the next sample could give a false up reading. The same thing can happen - though it is much less likely - on release. To compensate for this a 6ms delay is used when the state is changed in each direction before the next polling of the button state. This delay still consumes processor time, but much less than the alternative methods.

```
*====================================
* Use IRQ to read button
* By: Chad Huard
* 3/28/10
*====================================
portb         equ    $1004
PORTC         EQU    $1003
stack         equ    $01ff
              org    $0000
value         fcb    $01
overflow      fcb    $00
STATE         FCB    $00
TOC1_OFF      EQU    $16
TFLG1_OFF     EQU    $23
TCNT_OFF      EQU    $0E

*====================================
* MAIN LOOP
*====================================
              org    $c000
              lds    #stack
              LDX    #$1000
              CLI
LOOP          BRCLR  STATE,$80,SKIP
              LDAA   PORTC
              BPL    SKIP
              CLR    STATE
              JSR    DELAY
              CLI                         Enable the IRQ interrupt
SKIP          bra    LOOP                 keep looping (waiting for interrupt)

*====================================
* 6 mS DELAY FOR DEBOUNCE
*====================================
DELAY         LDD    TCNT_OFF,X
              ADDD   #12000
              STD    TOC1_OFF,X
              LDAA   #$80
              STAA   TFLG1_OFF,X
LOOP2         BRCLR  TFLG1_OFF,X,$80,LOOP2
              RTS

*====================================
* IRQ SERVICE ROUTINE
*====================================
              org    $c200        Interrupt service routine starts at #C200
              LDAA   #$80
              STAA   STATE
              PULA
              ORAA   #%00010000
              ldaa   value
              staa   portb        Display on Port B
              inca
              cmpa   #$00
              bne    ends
              inc    overflow
ends          staa   value
              JSR    DELAY
              rti

              org    $00ee        Place a JMP $C200 at address $00EE
              jmp    $c200
              end
```

Main Loop Logic:



Conclusion:

Interrupts are an invaluable tool in embedded design. Without the ability to interrupt on an event many time sensitive processes would be impossible. For instance, if you wish to create a PWM signal using the HC11 the obvious choice is to use the timing port, which uses very precise interrupts to toggle on and off the pins of port A. If you were to try to replicate this functionality without interrupts, the processor would be spending all of its time calculating and stepping through delays to drive the PWM. Using interrupts the PWM is more accurate and leaves the processor free for other calculations for the majority of the time.

Some of the particulars of the interrupt system are difficult to follow at first. The vector tables leading to a jump table finally leading to a service routine is a process that must be used a few times before it is fully understood. The use of the stack to store the environment more straightforward than the vector tables, but using tricks like pulling variables from the stack and replacing them (as was done in problem 2) requires practice and a thorough understanding of the entire process. Once interrupts are fully understood they quickly become an indispensable tool for many time sensitive applications.