**ECE-3226-50:** Lab #1

Interrupts

Robert Campbell

**Objective:**

The purpose of this lab is to gain familiarity with each aspect of utilizing external interrupts. Specifically, appropriately using Interrupt Vector Table, enabling interrupts, triggering an interrupt with an outside signal, and observing the priority of the different interrupts when multiple signals are received.

**Equipment:**

AVR Studio 7

STK500 Starter Kit

JTAG mkII Debugger

**Procedure:**

Description: Write a program that continually increments a binary counter that counts from 0 to 255, and rolls back over to 0, in 1 second increments. Have this counter output to PORTA and connect PORTA to LED7 through LED0, with LED0 being the least significant bit.

Also configure INT0, INT1, and INT2 to interrupt the counter and flash their respective number 5 times in 1 second intervals. INT0 will flash 0x03, INT1 will flash 0xC0, and INT2 will flash 0xAA.

Connect INT0 (PD2) to SW0, INT1 (PD3) to SW1, and INT2 (PB2) to SW2.

;1A

Jmp start

Jmp InterRout0

Jmp InterRout1

Jmp InterRout2

.org 0x0030

Start:

Ldi r16, low(ramend)

Out spl, r16

Ldi r16, high(ramend)

Out sph, r16

;Set INT0,1,2 as inputs

CBI DDRB, 2; PB2 as input (INT2)

CBI DDRD, 2; PD2 as input (INT0)

CBI DDRD, 3; PD3 as input (INT1)

;Set PORTA as output

LDI r16, 0xff

Out DDRA, r16

Out PORTA, r16 ; Initialize active low output to all 0s (0xff)

SBI PortB, 2 ; INT2 pullup resistor

SBI PortD, 2 ; INT0 pullup resistor

SBI PortD, 3 ; INT1 pullup resistor

Ldi R16, 0x0A ; Load 0000 1010

Out MCUCR, R16 ; Sets INT0,1 to negative edge

Clr R16 ; 0x00 to write to MCUCSR

Out MCUCSR, r16 ; Sets bit 6 to 0 for negative edge for INT2

LDI R16, 0xE0; 1110 0000 for GICR

Out GICR, R16; Activates INT0,1,2

SEI ; Enables global interrupt

LDI R17, 0xff; counter register

Counter:

Out PORTA, r17 ; Initialize active low output to all 0s (0xff)

; Then write the new output each loop

Dec r17;

Call delay

Rjmp counter

end: rjmp end

Delay:

push r0

Push r18;

Push r19;

Push r20;

clr r0

Ldi r18, 0x40; 0x030D40

Ldi r19, 0x0d; is 200,000

Ldi r20, 0x03;

DelayLoop:

Subi r18, 1;

Sbc r19, r0;

Sbc r20, r0;

Brne DelayLoop; delays 5microseconds \* value of r20:r18

; 5 \* 200000 = 1000000microsec or 1 second

Pop r20;

Pop r19;

Pop r18;

pop r0;

Ret

;1B

InterRout0:

Push r21

Push r22

Push r23

In r21, sreg

Push r21

ldi r22, 0xff

Ldi r23, 5;

Ldi r21, 0xFC ;0x03 with active low

Flash0:

Out PortA, r21

call delay; flash on 1sec

Out PortA, r22;

call delay; flash off 1 sec

Dec r23

Brne Flash0

;Restore output to previous value

Out PortA, r17;

;Restore registers

Pop r21

Out sreg, r21

Pop r23

Pop r22

Pop r21

Reti;

;1C

InterRout1:

Push r21

Push r22

Push r23

In r21, sreg

Push r21

ldi r22, 0xff

Ldi r23, 5;

Ldi r21, 0x3f ; 0xc0 active low

Flash1:

Out PortA, r21

call delay; flash on 1sec

Out PortA, r22;

call delay; flash off 1 sec

Dec r23

Brne Flash1

;Restore output to previous value

Out PortA, r17;

;Restore registers

Pop r21

Out sreg, r21

Pop r23

Pop r22

Pop r21

Reti;

;1D

InterRout2:

Push r21

Push r22

Push r23

In r21, sreg

Push r21

ldi r22, 0xff

Ldi r23, 5;

Ldi r21, 0x55 ;0xAA active low

Flash2:

Out PortA, r21

call delay; flash on 1sec

Out PortA, r22;

call delay; flash off 1 sec

Dec r23

Brne Flash2

;Restore output to previous value

Out PortA, r17;

;Restore registers

Pop r21

Out sreg, r21

Pop r23

Pop r22

Pop r21

Reti;

Observations: (2.1) The counter incremented as expected from 0 to 255, and successfully rolled-over to 0 to start counting up again.

(2.2 – 2.4) When an interrupt button is pressed, the counter stops counting, flashes the appropriate number (0x03 for INT0, 0xC0 for INT1, and 0xAA for INT2) 5 times, and then returns to counting from where the interrupt came in.

(2.5) Pressing INT2 once flashes 0xAA. After pressing INT2 again while 0xAA was flashing, it flashed an additional 5 times for a total of 10 before returning to counting.

(2.6) Pressing INT0 while INT1 was being serviced resulted in 0xC0 flashing 5 times followed by 0x03 flashing 5 times, despite INT0 being a higher priority. This happened because new interrupts are disabled while the interrupt process is still running, since the i-bit is disabled until the process finishes. However, the flag for INT0 was set in the GIFR so that it could be processed once global interrupt, the i-bit, was reenabled.

(2.7) Pressing INT2 while INT0 was being serviced resulted in 0x03 flashing 5 times followed by 0xAA flashing 5 times before returning to counting. INT2 cannot interrupt the INT0 process for the same reason written in (2.6), new interrupts cannot be processed while an interrupt is in the middle of being serviced.

(2.8) While INT2 is being serviced, press the buttons for INT1 and INT0. This results in 0xAA flashing 5 times, then 0x03, and finally 0xC0. This is because the INT0 and INT1 flags were both set in the GIFR while INT2 was being serviced. Once the INT2 process finished and the global interrupt was reenabled, the next highest priority interrupt that has its flag set in the GIFR is processed next. In this case that was INT0. INT1 was then processed once INT0’s process was complete.

(2.9) Toggle both INT1 and INT0 simultaneously. To do this, wire both interrupts (PD2 and PD3) to the same button (SW0). After the button was pressed, 0x03 flashed 5 times and then 0xC0 flashed 5 times. In other words, INT0 was processed first and then INT1 was processed.

**Discussion/Conclusion:**

The purpose of this experiment was to gain familiarity with external interrupts and their priority.

We encountered a few smaller issues throughout the experiment. One came from trying to take some extra steps to return the counter to the previous value that the register held before returning from the interrupt process. In the end, just pushing and popping the necessary registers was the most effective method.

Another issue arose from a simple syntax mistake of using a jump instruction to call the delay loop instead of a call instruction. When a button was pressed, this resulted in the display showing each of the interrupt numbers one time and then staying on 0x03 without ever returning to the counter.

Early on, we had missed a twist in the ribbon cable for the LEDs. Weirdly, this didn’t just affect which LEDs turned on, but for some reason it also made the counter run faster. Correcting the cable fixed both problems with our LEDs.

Beyond just making sure that the code was written correctly, the experiment helped us to internalize how exactly the interrupt priority works. Unfortunately, due to the nature of interrupts we couldn’t add in debouncing logic. This lead to a few instances where the same interrupt ended up being processed multiple times.