Jose Franco Baquera

April 22, 2018

CS - 273

Homework #11

1. Find the memory and I/O (if available) addresses of the following timer/counter related registers from ATmega328P Register Summary (in the back of the booklet):

`--- means that the I/O address does not exists/is unavailable.`

(NOTE: Memory addresses are two bytes in size. This is why we add the two preceding zeros, but they are not needed. We will place them in the table below to stress that memory addresses are two bytes in size while I/O addresses are one byte in size.)

Register	I/O Address (if available)	Memory Address
TCNT0	0x26	0x0046
TCCR0A	0x24	0x0044
TCCR0B	0x25	0x0045
OCR0A	0x27	0x0047
OCR0B	0x28	0x0048
TIMSK0	---	0x006E
TCNT1H	---	0x0085
TCNT1L	---	0x0084
TCCR1A	---	0x0080
TCCR1B	---	0x0081
TCCR1C	---	0x0082
OCR1AH	---	0x0089

OCR1AL	---	0x0088
OCR1BH	---	0x008B
OCR1BL	---	0x008A
TIMSK1	---	0x006F
TCNT2	---	0x00B2
TCCR2A	---	0x00B0
TCCR2B	---	0x00B1
OCR2A	---	0x00B3
OCR2B	---	0x00B4
TIMSK2	---	0x0070

2. Read Section 14 Interrupts and Timers of the booklet and answer the following questions.

2.1 What are the benefits of the interrupts mechanism in a computer?

There are multiple benefits for using interrupt mechanisms in a computer. One benefit is that interrupts in a computer allow the CPU to take care of a sudden condition or event *as soon as it occurs.* This benefit was apparent in our Lab 9 assignment since it allowed for us to detect *exactly* when the ultrasonic sensor's echo line when up and when it went down. If we used other methods like "polling" instead of interrupts, we might miss this state change and get wrong measurements/results. Another benefit of using interrupts is that it does not "tie down" the CPU, meaning that it can operate normally and do whatever instruction is fed without constantly spending all of its time checking if an event or condition has occurred (i.e. we make the CPU "multitask"). One last benefit of using interrupts is that it allows a system to be much more efficient and faster. This scenario is especially apparent with I/O devices since such devices are usually much slower than a computer's CPU. If we made the CPU wait until the I/O device was finished, the computer would run much slower.

2.2 Explain what an interrupt vector is.

According to section 14 of the booklet, an interrupt vector is a table of addresses where "each entry is the address of the code that should be executed in response to the interrupt that is associated with that array index." It is important to note that the interrupt vector table is always at a fixed location in memory (in our AVR case the interrupt vector table is found at the very low end of program memory). In essence, an interrupt vector tells "us what our next program counter is going to be."

2.3 Why does an interrupt service routine (ISR) need to save all register it uses including SREG?

An interrupt service routine (ISR) needs to save all registers it uses (including the SREG register) and must set them back to their original state before returning because, in most cases, an ISR might execute an instruction that changes the value of a register or sets a certain flag in the SREG that was not part of the original program. This may lead to unwanted or erroneous interference with the original program once the ISR returns. In essence, we must make sure that all the data of the original program (e.g. data contained in registers, etc.) is saved before the interrupt happens in order to put it back the way it was after the interrupt is done. This is done by pushing the necessary registers onto the stack and popping them back once interrupt is done but before ISR returns.

2.4 Write an ISR for TIMER2 COMPB event so that a two-byte integer at memory location 0x0200 (low byte) and 0x0201 (high byte) will be increased by one every time the ISR is invoked. Please give both C and assembly code.

C Code Version:

```
#include <avr/interrupt.h>

ISR ( TIMER2_COMPB_vect ) {

   int *loPtr = 0x0200; // Create a pointer to the low byte address

   int *hiPtr = 0x0201; // Create a pointer to the high byte address

   int low = *lowPtr; //store contents of low pointer into variable low

   int high = *highPtr; //store contents of high pointer into variable high

   // Check for overflow

   if( low == 255 ) {

      low = 0; // Reset the low byte back to 0

      high++; // Increment high

   } // end if.

   else {
```

```
      low++; // Increment low by 1
   } // end else
   *loPtr = low; // Update the low byte in memory address
   *hiPtr = high; // Update the high byte in memory address
} // end function
```

Assembly Code Version:

```
.SET SREG, 0x3F
TIMER2_COMPB_ISR:
   # Protect all registers that we will use
   PUSH R18
   PUSH R19
   PUSH R20
   PUSH R21
   IN R20, SREG ; save original SREG register into R20
   LDS R18, 0x0200 ; load low byte from memory into r18
   LDS R19, 0x0201 ; load high byte from memory into r19
   INC R18  ; increment the low byte
   IN R21, SREG ; check when low byte reaches past 255 by checking SREG
   SBRC R21, 1 ; if low byte went pass 255 (i.e. 256 = 0), increment high byte
   INC R19 ; This instruction will be skipped if no overflow occurred.
   STS 0x0200, R18 ;store new value into original memory location for low byte
   STS 0x0201, R19 ;store new value into original memory location for high byte
   OUT SREG, R20 ; restore original SREG so we don’t mess with the orig program
   # Restore all registers to their original state
   POP R21
   POP R20
```

```
   POP R19
   POP R18
   RETI

# NOTE: There various ways to do the above assembly code. For example,
# we could have used "ADC" AVR command to add with a carry. The ADC
# command does the following: ADC Rd, Rr ; Rd = Rd + Rr + C.
# However, using a "SBRC" command made more sense to me. We would have
# to do something like the following if we wanted to use the ADC
# command:
#     CLR R21
#     INC R18  ; increment the low byte
#     ADC R19, R21 ; add with carry (if any)
```