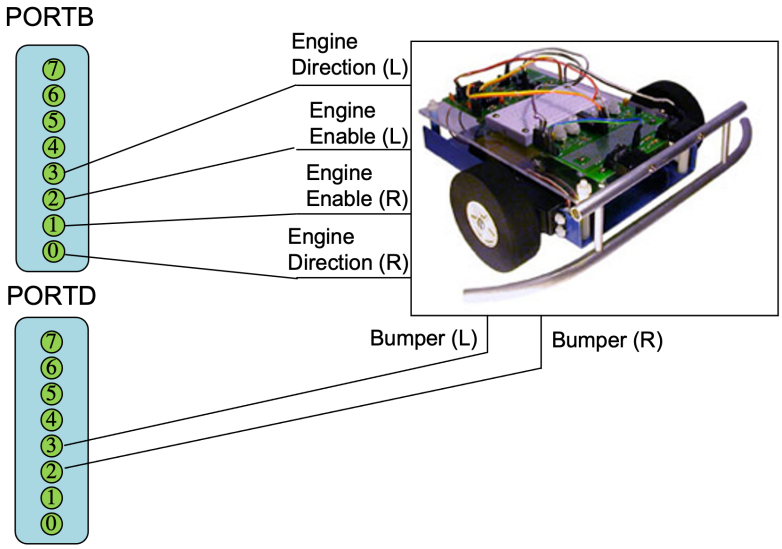
ECE 375
Computer Organization and Assembly Language Programming
Fall 2022
Solutions Set #3

[25 pts]

1- Consider the AVR code segment shown below that initializes I/O and interrupts for Tekbots shown below (with some information missing).

```
.include "m128def.inc"
.def  mpr = r16
.org  $0000
      rjmp  INIT
.org  _________________(i)
      rjmp  HitRight
.org  _________________(ii)
      rjmp  HitLeft
      …
.org  $0056
INIT: ____________________________(1)      ; Control engine
      ____________________________(2)      ;
      ____________________________(3)      ; Detect whiskers
      ____________________________(4)      ;
      ____________________________(5)      ; Enable pull-up resisters for L/R bumpers
      ____________________________(6)      ;
      ____________________________(7)      ; Detect on the proper edge
      ____________________________(8)      ;
      ____________________________(9)      ; Turn on interrupts for L/R bumpers
      ____________________________(10)     ;
      sei                                  ; Turn on global interrupt
```



(a) Fill in the lines 1-2 with the necessary code to set Data Directional Register x to control engine enable and engine direction for both left and right wheels.

Solution

```
ldi       mpr, 0b00001111       (1)
out       DDRB, mpr             (2)
```

These two instructions set the DDRB so that PORTB pins 0-3 are set to output.

(b) Fill in the lines 3-4 with the necessary code to set Data Directional Register x to detect left and right bumper movements.

Solution

```
ldi     mpr, 0b00000000     (3)
out     DDRD, mpr           (4)
```

These two instructions set the DDRD so that PORTD pins 2 and 3, which are mapped to INT 2 and 3, respectively, are set to input. Note that these two instructions are not needed since PORTD pins will be zeros on reset. However, we keep these instructions more as a guidance on what needs to be done.

(c) What are the addresses needed in the lines (i) and (ii) to properly control the execution of interrupt service routines for left and right bumpers. Fill in the lines 5-6 to enable the pull-up resisters for these whiskers.

Solution

```
.org    $0006               (i)
.org    $0008               (ii)

ldi     mpr, 0b00001100     (5)   ; Enable pull-up left and right bumpers
out     PORTD, mpr          (6)
```

The interrupt vectors for INT 2 and INT 3 are at addresses $0006 and $0008, respectively. The pull-up registers on these pins also need to be enabled.

(d) Fill in the lines 7-8 with the necessary code to set External Input Sense Control to detect bumper hits (i.e., interrupts) on a falling edge.

Solution

```
ldi     mpr, 0b101000000    (7)
sts     EICRA, mpr          (8)
```

These two instructions set INT 2-3 on PORTD pins 2-3 to be detected on a falling edge. Since EICRA is in the Extended I/O Registers space, we need to use `sts`.

(e) Fill in the lines 9-10 to enable interrupts for whisker movements.

Solution

```
ldi     mpr, 0b00110000     (9)
out     EIMSK, mpr          (10)
```

These two instructions set up the external interrupt masks for INT 2-3

[25 pts]

2- Consider the `WAIT` subroutine for Tekbot discussed in class that waits for 1 sec. and returns. Rewrite the `WAIT` subroutine so that it waits for 1 sec. using the 16-bit Timer/Counter1 with the highest possible resolution. Assume that the system clock frequency is 8 MHz and the Timer/Counter1 is operating under the CTC mode. This is done by doing the following:

(a) Timer/Counter1 is initialized to operate in the CTC mode.

(b) The `WAIT` subroutine loads the proper value into OCR1A and waits until OCF1A is set. Once OCF1A is set, it is cleared and the `WAIT` subroutine returns.

Use the skeleton code shown below. Also, show the necessary calculations for determining *value* and *prescale*. Note that your code may not use any other GPRs besides `mpr`.

```
.include "m128def.inc"
.def mpr = r16
...
.ORG    $0000
        RJMP  Initialize
.ORG    $0046                ; End of interrupt vectors
Initialize:
        ...
        ...Your code goes here...
        ...
WAIT:
        ...
        ...Your code goes here...
        ...
        RET
```

Solution

The first thing that needs to be done is to calculate the value to be loaded onto the 16-bit OCR1A. This is done by evaluating the following equation:

$$value = \text{TOP} = (1 \text{ sec}/(\text{prescale} \times 125 \text{ ns})) - 1 = (8{,}000{,}000/\text{prescale}) - 1$$

We want to use a prescale value that would lead to the highest resolution (i.e., lowest prescale value) and yet satisfy the above equation, i.e., TOP ≤ 65535, thus prescale = 256. This leads to *value* = 31,250.
Obviously, there are many ways to write this code, but here is one possibility:

```
.include "m128def.inc"
.def mpr = r16
...
.ORG    $0000
  RJMP  Initialize

.ORG    $0056                ; End of interrupt vectors
Initialize:
  LDI   mpr, 0b00001100      ; Set prescalar to 256,WGM13:0=0100
  OUT   TCCR1B, mpr          ; By default WGM11:0=00, CTC mode
WAIT:
  LDI   mpr, high(31250)     ; Load the value for delay
  OUT   OCR1AH, mpr          ; Load high byte first
  LDI   mpr, low(31250)      ;
  OUT   OCR1AL, mpr          ;
LOOP:
  SBIS  TIFR1, OCF1A         ; Skip if OCF1A flag in TIFR1 is set, OCF1A = 1
  RJMP  LOOP                 ; Loop if OCF1A not set
  LDI   mpr, 0b00000010      ; Reset OCF1A
  OUT   TIFR, mpr            ; Note - write 1 to reset
  RET
  ...
```

The initialization part of the code sets the prescalar to 256, i.e., CS12 = 1, CS11 = 0, and CS10 = 0. Note that WGM11:10 in TCCR1A does not have to be explicitly configured for the CTC mode of operation since these bits are initialized to 0's at reset.

The `WAIT` subroutine first sets the Timer/Counter1 to *value* = 62,499. Note that to do a 16-bit write, the high byte (TCNT1H) must be written before the low byte (TCNT1L) is written.

In `LOOP`, the OCF1A bit in TIFR1 is tested using SBIS to see if it is set. If it is set, the loop exits and OCF1A is reset by writing a 1 and the `WAIT` subroutine returns. Otherwise, the loop continues until OCF1A is set.

[25 pts]

3- Consider the AVR code segment shown below (with some missing information) that configures Timer/Counter0 for Fast PWM operation, and modifies the Fast PWM duty cycle whenever a specific button on Port D is pressed.
   (a) Fill in lines (1-2) with the instructions necessary to configure Timer/Counter0 for Fast PWM mode to toggle OC0A.
   (b) Fill in lines (3-4) with the instructions necessary to set the prescale value to 8.
   (c) Based on the prescale value used in part (b), what is the frequency of the PWM signal (f_{PWM}) being generated by Timer/Counter0? Assume the system clock frequency is 8 MHz.
   (d) Fill in lines (5-6) to provide the compare value for Timer/Counter0 so that the initial duty cycle is 0%.
   (e) What would be the value necessary for the variable `step` to increase the duty cycle by 10% each time the `DUTY_STEP` subroutine is executed? Ignore the case when/if the compare value overflows.

```
.include    "m32U4def.inc"
.def        mpr = r16
.def        temp = r17
.equ        step = ___

INIT:
     …
     ; stack pointer is initialized
     …

     ; I/O ports
     ldi    mpr, 0b10000000     ; set Port B, pin 7 (OC0A) as output
     out    DDRB, mpr

     ldi    mpr, 0b00000000     ; set pin 0 as input
     out    DDRD, mpr
     ldi    mpr, 0b00000001     ; enable pull-up resistor for pin 0
     out    PORTD, mpr

     ; Timer/Counter0
     ; Fast PWM mode, non-inverting, prescale = 8
     _____________________________(1)
     _____________________________(2)
     _____________________________(3)
     _____________________________(4)

     ; Initial compare value for PWM output
     _____________________________(5)
     _____________________________(6)

MAIN:
     sbis   PIND, 0
     rcall  DUTY_STEP
     rjmp   MAIN

DUTY_STEP:
     push   mpr
     push   temp
```

```
in    mpr, ________        ; read the current PWM compare value
ldi   temp, step
add   mpr, temp            ; add step value to compare value
out   ________, mpr        ; write new PWM compare value

pop   temp
pop   mpr
ret                        ; return
```

Solution

(a) Fill in lines (1-2) with the instructions necessary to configure Timer/Counter0 for Fast PWM mode to toggle OC0A.

```
ldi    mpr, (1<<WGM01|1<<WGM00|0<<COM0A1)|1<<COM0A0)  (1)
out    TCCR0A, mpr                                    (2)
```

Note that WGM02 in TCCR0B is already initialized to 0.

(b) Fill in lines (3-4) with the instructions necessary to set the prescale value to 8.

```
ldi    mpr, (0<<CS02|1<<CS01|0<<CS00)                 (3)
out    TCCR0B, mpr                                    (4)
```

(c) Based on the prescale value used in part (b), what is the frequency of the PWM signal (f_{PWM}) being generated by Timer/Counter0? Assume the system clock frequency is 8 MHz.

The frequency (f_{PWM}) for an 8-bit timer is found using the following equation: f_{PWM} = 8 MHz / (prescale x 256). Since the prescale value from (b) is 8, this leads to a PWM frequency of f_{PWM} = 3.906 kHz.

(d) Fill in lines (5-6) to provide the initial compare value for Timer/Counter0.

```
ldi    mpr, 0                  (5)
out    OCR0, mpr               (6)
```

Since Timer/Counter0 is configured to Toggle OC0A on compare match, the duty cycle % is determined as OCF0A / 255. Thus, setting `OCF0A` to zero leads to a duty cycle of 0%. This is achieved by loading zero into `mpr` and then writing it out to `OCF0A`.

(e) What would be the value necessary for the variable `step` to increase the duty cycle by 10% each time the `DUTY_STEP` subroutine is executed? Ignore the case when/if the compare value overflows.

The `DUTY_STEP` subroutine adds `step` to OCF0A each time it is executed, which can be determined using the equation (`step` / 255) = 10%. Solving for `step` leads to $\lceil 25.5 \rceil = 26$.

[25 pts]

4- Write a subroutine `initUSART1` to configure ATmega32U4 USART1 to operate as a transmitter and sends a data every time USART1 Data Register Empty interrupt occurs. The transmitter operates with the following settings:

- 8 data bits, 1 stop bit, and odd parity
- 9,600 Baud rate
- Transmitter enabled
- Normal asynchronous mode operation
- Interrupt enabled

Assume the system clock is 8 MHz. The skeleton code is shown below:

```
.include "m32U4def.inc"
.def mpr = r16
.ORG $0000
     RJMP initUSART1
…
.ORG $0034
     JMP  SendData
…
.ORG $0056
initUSART1:
     …
     …Your code goes here…
     …
Main:
     ld   mpr, X+         ; Send first data
     sts  UDR1, mpr
Loop:
     RJMP Loop

SendData:
     ld   mpr, X+         ; Send next data
     sts  UDR1, mpr
     reti
```

Solution

There are many ways to write this code but here is one possible code for `initUSART1`.

```
initUSART1:
     ; Port D set up – pin3 output
     ldi  mpr, 0b00001000           ; Configure USART1 TXD1 (Port D, pin 3)
     out  DDRD, mpr                 ; Set pin direction to output
     ; Set Baud rate
     ldi  mpr, 51                   ; Set baud rate to 9,600 with f = 8 MHz
     sts  UBRR1L, mpr               ; UBRR1H already initialized to $00
     ; Enable transmitter and interrupt
     ldi  mpr, (1<<TXEN1|1<<UDRIE1) ; Enable Transmitter and interrupt
     sts  UCSR1B, mpr               ; UCSR1B in extended I/O space, use sts
     ; Set asynchronous mode and frame format
     ldi  mpr, (1<<UPM11|1<<UPM10|1<<UCSZ11|1<<UCSZ10)
     sts  UCSR1C, mpr               ; UCSR1C in extended I/O space, use sts
     sei                            ; Enable global interrupt
```

The first two instructions configure Pin 3, Port D (TXD1) for output since the USART1 is acting as a transmitter.

The next two instructions set the Baud rate. This is done by calculating the UBRR value, which is (8MHz/(16x9600))-1 = 51, and then writing it into the UBRR1L register. The UBRR1H register was not written to since upper byte is initialized as $00.

The next two instructions enable the transmitter and the interrupt, which is done by setting the 3rd-bit (i.e., TXEN1) and the 5th-bit (i.e., UDRIE1) of UCSR1B.

The next pair of instructions sets it to the asynchronous mode, which is done by setting the 6th bit (i.e., UMSEL1) of UCSR1C to 0. Note that `0<<UMSEL1` is not necessary since it is already initialized to 0 on reset. This is also the case for `0<<UCSZ12` and `0<<UPM10`.

The frame format with 8 data bits, 1 stop bit, and odd parity is set by selecting UCSZ12:0 to be 011, UPM11:0 to be 11, and USBS1 to 0. These bits are configured using

```
(1<<UPM11|1<<UPM10|1<<UCSZ11|1<<UCSZ10)
```

Also note that `sts` is used because `UCSR1C` is in the extended I/O space.