Consider the AVR code segment shown below that initializes and handles interrupts.

(a) Explain in words what the code accomplishes when it is executed. That is, explain what it does and how it does it.

(b) Write and explain the interrupt initialization code (lines 1-7) necessary to make the interrupt service routine (starting at ISR:) work properly. More specifically,
   (i) Fill in lines (1-2) with the necessary code to set the interrupt in question to detect an interrupt on a rising edge.
   (ii) Fill in lines (3-4) with the necessary code to mask out all other interrupts except the interrupt in question.
   (iii) Fill in lines (5-6) with the necessary code to set the port in question for input.
   (iv) Fill in line (7) to enable interrupt.

```
.include "m128def.inc"
.def mpr = r16 ; Multi-purpose register
.def count = r17 ; Assume R17 is initially 0

.ORG $0000
START: RJMP INIT
.ORG $0002
JMP ISR
.ORG $0046

INIT: ______________________ (1)
                   ______________________ (2)
                   ______________________ (3)
                   ______________________ (4)
                   ______________________ (5)
                   ______________________ (6)
                   ______________________ (7)
       LDI XH, high(CTR)
       LDI XL, low(CTR)
       LDI YH, high(DATA)
       LDI YL, low(DATA)

WAIT: RJMP WAIT
.ORG $100F
ISR: IN mpr, PINA
      ST Y+, mpr
      INC count
      ST X, count
      RETI

.DSEG
CTR: .BYTE 1
DATA: .BYTE 256
```

Solution

(a) This code reads an 8-bit value latched on to Port A when an interrupt occurs from INT0. It then stores it to memory starting at address DATA then increments a count, which will be stored in memory location at CTR.

(b) Here is the modified code with initialization code to make ISR work properly. Note that I have set the input sense control for pin INT0 to detect an interrupt on a rising edge.

```
.include "m128def.inc"
.def mpr = r16 ; Multi-purpose register
```
2- The AVR code below (with some missing information) is designed to initialize and service interrupts from three I/O devices (DevA, DevB, and DevC).
(a) There are 8 external interrupt pins (INT7-INT0) in AVR. Which three interrupt pins are these I/O devices connected to?
(b) Which I/O device’s interrupt is detected on a falling edge?
(c) The interrupt pins referred to in part (a) are connected to two of the 7 ports (PORTA-PORLG) in AVR. Which ports are they?
(d) There are important instructions missing in lines (1-2) of the code. Fill in the missing instructions in lines (1-2) so that the code will work correctly.
(e) Suppose DevA requires that no interrupts are detected while it is being serviced. Fill in lines (3-4) with the necessary code to clear any latched interrupts at the end of ISR_DevA.
(f) Suppose interrupts are detected from all three interrupt pins at the same time. Which subroutine (ISR_DevA, ISR_DevB, or ISR_DevC) will be executed first?

```
.include "m128def.inc"
def mpr = r16
START:
.org $0000
RJMP INIT
.org $0002
JMP ISR_DevA
.org $0008
JMP ISR_DevB
.org $000C
JMP ISR_DevC

INIT:
ldi mpr, 0b10000011
sts EICRA, mpr
ldi mpr, 0b00001100
out EICRB, mpr
ldi mpr, _________
out EIMSK, mpr
ldi mpr, $00
```
Solution:

(a) There are 8 external interrupt pins (INT7-INT0) in AVR. Which three interrupt pins are these I/O devices connected to?

This can be determined by looking at the addresses where JMPs are located, i.e., addresses $0002$, $0008$, and $000C$, which correspond to INT0, INT3, and INT5. These pins can also be recognized by values written into EICRA and EICRB. However, this may not always work since a pin can be set to “00” to indicate low-level trigger.

(b) Which I/O device’s interrupt is detected on a falling edge?

Among the bits in EICRA and EICRB, bits 7 and 6 in EICRA are set to falling edge (i.e., 10). Therefore, DevB (INT3) is set to detect an interrupt on a falling edge.

(c) The interrupt pins referred to in part (a) are connected to two of the 7 ports (PORTA-PORTE) in AVR. Which ports are they?

INT7-INT0 are spread between PORTD (INT3-0) and PORTE (INT7-4).

(d) There are important instructions missing in lines (1-2) of the code. Fill in the missing instructions in lines (1-2) so that the code will work correctly.

Since PORTE also needs to be setup for input, we need to have the following instructions:

\[
\text{out} \quad \text{DDRE, mpr} \quad (1)
\]

Since INT3 is set to detect on a falling edge, this pin needs to be set to Input with pull-up. Thus, we need to have the following instructions:

\[
\text{out} \quad \text{DDRD, mpr} \quad (1)
\]

\[
\text{ldi} \quad \text{mpr,0b00001000} \quad (2)
\]

\[
\text{sei}
\]

\[
\text{MAIN:}
\]

\[
\{\quad \ldots \text{do something}\ldots
\}
\]

\[
\text{ISR_DevA:}
\]

\[
\ldots
\]

\[
\quad (3)
\]

\[
\text{RETI}
\]

\[
\text{ISR_DevB:}
\]

\[
\{\quad \ldots
\]

\[
\ldots
\]

\[
\quad (4)
\]

\[
\text{RETI}
\]

\[
\text{ISR_DevC:}
\]

\[
\{\quad \ldots
\]

\[
\ldots
\]

\[
\text{RETI}
\]
(e) Suppose DevA requires that no interrupts are detected while it is being serviced. Fill in lines (3-4) with the necessary code to clear any latched interrupts at the end of ISR_DevA.

We need to clear EIFR so that any interrupts latched during ISR are ignored. This is achieved by

```assembly
ldi  mpr, 0b00101001  (3)
out EIFR, mpr       (4)
```

(f) Suppose interrupts are detected from all three interrupt pins at the same time. Which subroutine (ISR_DevA, ISR_DevB, or ISR_DevC) will be executed first?

The priority is based on the address of JMP with the lowest address having the highest priority. Thus, ISR_DevA will be called first.

[25 pts]

3- Write an AVR assembly code that waits for 1 sec using the 8-bit Timer/Counter0 with the system clock frequency of 16 MHz operating under Normal mode. This is done by doing the following:
(1) Timer/Counter0 is initialized to count for 10 ms and then interrupts on an overflow;
(2) The main part of the program simply loops, and for each iteration, tests whether the interrupt has occurred 100 times; and
(3) On each interrupt, Timer/Counter0 is reloaded to interrupt again in 10 ms.

Use the skeleton code shown below:

```assembly
.include "m128def.inc"
def  mpr = r16
def  counter = r17
...
.ORG $0000
RJMP Initialize
.ORG $0020 ; Timer/Counter0 overflow interrupt vector
    JMP Reload_Counter
.ORG $0046 ; End of interrupt vectors
Initialize:
    ...
    ...Your code goes here...
    ...
LOOP:
    ...
    ...Your code goes here...
    ...
    ...
Reload_counter:
    ...
    ...Your code goes here...
    ...
RETI
```
Solution

The first thing that needs to be done is to calculate the value to be loaded onto Timer/Counter0. This is done by evaluating the following equation:

\[ \text{value} = 256 - \left( \frac{10 \text{ ms}}{\text{prescale} \times 62.5 \text{ ns}} \right) = 256 - \left( \frac{160,000}{\text{prescale}} \right) \]

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, thus \( \text{prescale} = 1024 \). This leads to \( \text{value} = 100 \).

Obviously, there are many ways to write this code, but here is one possibility:

```assembly
.include "m128def.inc"
def mpr = r16
def counter = r17
.
.ORG $0000
RJMP Initialize
.
.ORG $0020 ; Timer/Counter0 overflow interrupt vector
JMP Reload_counter
.
.ORG $0046 ; End of interrupt vectors

Initialize:
  LDI mpr, 0b00000001 ; Enable interrupt on Timer/Counter0 overflow
  OUT TIMSK, mpr
  SEI ; Enable global interrupt
  LDI mpr, 0b00000011 ; Set prescalar to 1024
  OUT TCCR0, mpr ;
  LDI mpr, 100 ; Load the value for delay
  OUT TCNT0, mpr ;
  LDI counter, 100 ; Set timer
LOOP:
  CPI counter, 0 ; Repeat until 100 interrupts
  BRNE LOOP
.
Reload_counter:
  PUSH mpr
  LDI mpr, 100 ;
  OUT TCNT0, mpr ; Load the value for delay
  DEC counter
  POP mpr
  RETI
```

The initialization part of the code first enables Timer/Counter0 overflow interrupt and the global interrupt. Then, the prescaler is set to 1024, i.e., \( \text{CS02} = 1, \text{CS01} = 1, \text{and CS00} = 1 \). Note that the Normal mode of operation does not have to be explicitly configured since the Waveform Generation Mode bits are all 0s at reset, i.e., \( \text{WGM01:0} = 00 \) (Normal mode). The next part sets the Timer/Counter0 to \( \text{value} = 100 \).

Once the Timer/Counter0 is set, the program enters a loop waiting for a Timer/Counter0 overflow interrupt to occur. In addition, counter is checked to see if it has reached zero.

Finally, the Reload_counter routine reloads the value, decrements the counter, and returns.

[25 pts]
4- Write a subroutine \texttt{initUSART1} to configure ATmega128 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, 2 stop bits, and even parity
- 9,600 Baud rate
- Transmitter enabled
- Normal asynchronous mode operation
- Interrupt enabled

Assume the system clock is 16 MHz. The skeleton code is shown below:

```assembly
.include "m128def.inc"
.def mpr = r16
.ORG $0000
RJMP initUSART1
...
.ORG $003E
JMP SendData
...
.ORG $0046
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`.

```assembly
initUSART1:
    ; Port D set up - pin 3 output
    ldi mpr, 0b00001000          ; Configure USART1 (Port D, pin 3)
    out DDRD, mpr                ; Set pin direction to output
    ; Set Baud rate
    ldi mpr, 103                 ; Set baud rate to 9,600 with f = 16 MHz
    sts UBRR1L, mpr              ; UBRR1H already initialized to $00
    ; Enable transmitter and interrupt
    ldi mpr, (1<<TXEN1|1<<UDRIE1) ; Enable Transmitter and interrupt
    sts UCSRB, mpr               ; UCSRB in extended I/O space, use sts
    ; Set asynchronous mode and frame format
    ldi mpr, (1<<USBS1|1<<UPM11|1<<UCSZ11|1<<UCSZ10)
    sts UCSRC, mpr               ; UCSRC in extended I/O space, use sts
    sei ; Enable global interrupt
```

The first two instructions configure Pin 3, Port D 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 

\[ \frac{16\text{MHz}}{(16x9600)} - 1 = 103 \]

and then writing it into the UBRR1L register. The UBRR1H register was not written to since upper byte is $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 UCSRB.

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, 2 stop bits, and even parity is set by selecting UCSZ12:0 to be 011, UPM11:0 to be 10, and USBS1 to 1. These bits are configured using

$$(1<<USBS1|1<<UPM11|0<<UPM10|1<<UCSZ11|1<<UCSZ10)$$

Also note that stdio.h is used because UCSR1C is in the extended I/O space.