

19 Interrupt Basics

At the end of this unit the student will be able to:

explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.

Computer systems do not exist in isolation but are usually interfaced to the external world via I/O devices. The problem with I/O devices is that they are usually much slower than the computer system, therefore it is necessary to synchronize the interactions between the I/O devices and the computer system. With an input device, such as a keyboard, there will be times when the CPU has to wait for new data to be ready. Only when the data is ready, can it be read by the CPU and processed. If the time to create new data is longer than the processing time then the system is said to be *I/O bound*. If the data is ready before the CPU is ready, then the system is *CPU bound*. Systems can be *unbuffered* where the I/O writes/reads directly to the CPU or *buffered* where a buffer or memory storage exists between the I/O device and the CPU. There are several ways through which synchronization can be accomplished.

- Blind cycle
- Busy polling
- Interrupts
- Direct Memory Access (DMA)

Blind Cycle

This type of synchronization is appropriate when the delay due to I/O is fixed and known. It is called blind because it provides no feedback from the I/O device back to the computer. For example, if a printer can print at 40 characters/sec but cannot inform the computer when the last character has been printed. Figure 8 shows a diagram of how this synchronization is achieved. Blind cycle synchronization is simple and predictable, however if the output/input rate is variable

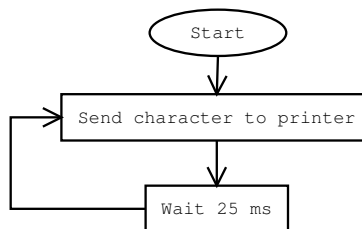


Figure 8: Blind cycle flowchart for a printer

or unknown then this method cannot be used. Computing time is wasted while the CPU is waiting and, because no feedback is available, error checking cannot be performed nor can special conditions be handled.

Busy polling

With this method of synchronization, the I/O device has a status bit that can be checked by the CPU to determine if data transfer can occur. Figure 9 shows how this type of synchronization is achieved. Busy polling can accommodate variable or unknown output/input rates, however the

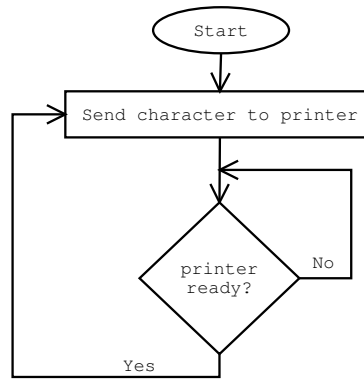


Figure 9: Busy wait flowchart for a printer

CPU still wastes time waiting for the I/O device to become ready. Therefore, the CPU is unable to do any other task while polling is taking place. A better method is to have the external device signal the CPU when servicing is required i.e. interrupts.

Direct Memory Access (DMA)

All the interfaces, seen so far, between the I/O device and the CPU have been controlled by software. In this type of control, if it is required to transfer data from an input device to RAM, the data must first be transferred to the CPU registers, then from the registers to RAM. Performance can be greatly improved if the data can be transferred directly between I/O devices and RAM. Direct Memory Access (DMA) allows this type of direct transfer without CPU intervention. DMA can be read or write and in both cases, the CPU is halted and the data transferred between memory and I/O device. The PIC16F877 does not support DMA.

Interrupts

Interrupts are a way of signaling the CPU in an asynchronous fashion that servicing is required. When an interrupt occurs the CPU finishes execution of its current instruction, saves any needed registers and the program counter (PC) on the stack and transfers execution to the Interrupt Service Routine (ISR). The CPU then executes the ISR and when finished, restores the PC and continues execution of the main program. This sequence is shown in Figure 10. The CPU is only interrupted when a service request is received, so no waiting is takes place and useful work can be done between interrupts. Interrupts can be *vectored* which means that each interrupt has a unique interrupt vector, and therefore a separate ISR. With vectored interrupts, the CPU will automatically execute the appropriate ISR when an interrupt occurs and there is no need to determine which device generated the service request. PC's have a standard interrupt vector structure. An interrupt subroutine may be "installed" by changing the instruction at the appropriate vector. The PIC16F877 does not

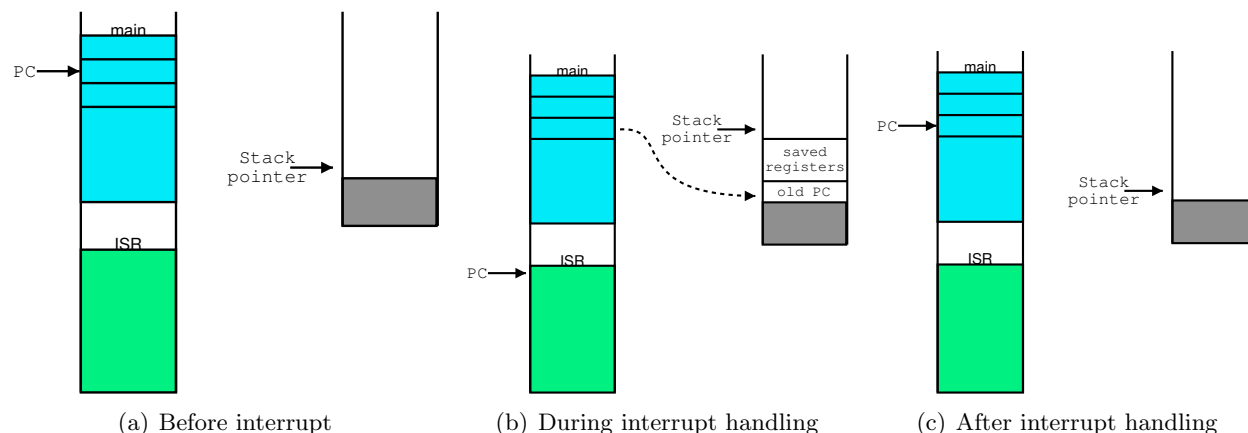


Figure 10: Main memory and stack when an interrupt occurs

support multiple vectored interrupts, but the same effect can be achieved by *polling* the interrupt sources to determine who raised the interrupt and then executing an appropriate sub-routine.

When multiple interrupts occur simultaneously, the CPU must have some way of determining which interrupt to service first, i.e. which interrupt has *priority*. Further most CPU's have a single interrupt line, which is shared by all the interrupt sources. To implement hardware priority ordering, a peripheral interrupt controller (PIC) may be used. This is in turn a peripheral, which accepts all the hardware interrupt signals, and asserts the CPU interrupt line iff an *enabled* signal is received. The PIC control registers are used enable signals, and set their relative priorities. The PIC data registers report which of the many signals have been exerted, and the location of the ISR vector. If a PIC is not used (or does not support priority ordering) then the priority of the interrupt is determined by the order in which the interrupt sources are polled in the ISR, and the sequence in which the interrupts are enabled/disabled.

The interrupts which we have discussed, are all triggered by a hardware signal. It is also possible to trigger an interrupt handler from software (typically using a special interrupt instruction). In essence, this is the same as a subroutine call. Typically hardware interrupts are generated by peripheral reporting task completion or requesting CPU service. Software interrupts may be used to access BIOS (and other low level) functions, as well as to allow programs to access shared memory and peripherals in a structured or "known" fashion.

Software interrupts are not supported by the PIC16F877.

The PIC16F877 family has 14 sources of interrupts, including Timer Overflow, and signals/changes on Input Ports. The interrupt control register (INTCON) records individual interrupt requests in flag bits (Note that the flag bits will be set whether or not the interrupt is enabled). It also has individual and global interrupt enable bits. A global interrupt enable bit, GIE (INTCON<7>) enables (if set) all unmasked interrupts, or disables (if cleared) all interrupts. When bit GIE is enabled, and an interrupts flag bit and mask bit are set, the interrupt will vector immediately.

Individual interrupts can be disabled through their corresponding enable bits in various registers. Individual interrupt bits are set, regardless of the status of the GIE bit. The GIE bit is cleared on RESET. The "return from interrupt" instruction, RETFIE, exits the interrupt routine, as well as sets the GIE bit, which re-enables interrupts.

The RB0/INT pin interrupt, the RB port change interrupt, and the TMR0 overflow interrupt flags are contained in the INTCON register. The peripheral interrupt flags are contained in the special function registers, PIR1 and PIR2. The corresponding interrupt enable bits are contained in special function registers, PIE1 and PIE2, and the peripheral interrupt enable bit is contained in special function register INTCON.

When an interrupt is responded to, the GIE bit is cleared to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with 0004h. Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits.

It is the responsibility of the ISR to clear the interrupt flag bit corresponding to the interrupt that caused the ISR to be executed, otherwise the CPU would assume that another interrupt of the same type had occurred during execution of the ISR. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid recursive interrupts.

For external interrupt events, such as the INT pin or PORTB change interrupt, the interrupt latency will be three or four instruction cycles. The exact latency depends on when the interrupt event occurs. The latency is the same for one or two-cycle instructions. Individual interrupt flag bits are set, regardless of the status of their corresponding mask bit, PEIE bit, or GIE bit.

If interrupt occurs during the execution of the ISR it is not serviced until after the ISR completes and the GIE bit re-enabled. At this point, if the postponed interrupt's enable bit is set, it is serviced. – (PIC 2001)

Timing & Timers

One use to which the PIC can be put to is to control the times when events happen. A typical example is reading a port or A/D at specified intervals. If polling or blind cycle synchronization is used then what is required is some means of generating a delay to control the loop that reads the port or A/D.

There are several ways in which fixed delay times can be obtained. One of the most straightforward is by using the times taken by instructions to create the delay. Another is to use the built-in timer facility available.

It can be assumed for the rest of this discussion that a 10 ms delay time is required. If the clock frequency is 4 Mhz, then the number of cycles required is 10,000³³. Let us see how to generate smaller delays, for example 1 ms. This can be done by counting down a value in the W register until it becomes zero. A first pass at the delay routine can be:

```
delay_1ms    movlw    <unknown>
:Loop4      addlw    0xFF      ; add -1, cannot use decrement
            btfss    STATUS,Z
            goto     :Loop4
            return
```

³³The cycle frequency is the 1/4 the clock frequency

Peripherals on PIC16F877

- Parallel I/O port
 - PORTx -- data
 - TRISx -- configuration
- Timer0
 - TMR0 -- data
 - T0CS -- control/configuration (bit)
 - T0IE -- control (bit)
 - T0IF -- state

Using Peripheral registers/flags

- Four categories
 - control
 - configuration
 - state
 - data
- Use once

action:

- configure (maybe once)
- set control
- wait for state (optional)
- read/write data
- wait for state (optional)

- Polling
 - blind (keep looping)

```
repeat forever
  action
  delay (optional)
```

- busy (loop using state)

```
repeat forever
  do something
  if state ok
    action
```

Timing/Delay

- Count the instructions (know the clock frequency)

```
delay_1ms      movlw 0xF9      ; 0xF9 = 249
                nop
usec4           addlw 0xFF      ; add -1
                btfss STATUS,Z
                goto usec4
                return
```

- Use the timers
 - Watchdog
 - nominal time-out period of 18 ms. Period varies with temperature, VDD
 - Scale up to 1:128 (max. 2.3 seconds)
 - reset/wake-up on timeout
 - Timer0/1/2
 - period determined by external oscillators or the instruction cycle
 - pre/post scaling
 - poll for timeout/interrupt on timeout

Software Interfacing

- CPU vs. I/O Bound
- Buffered vs. Unbuffered
- Synchronization
 - Blind cycle ★
 - Busy polling ★
 - Interrupts ★
 - vectored
 - Direct Memory Access (DMA)
- I/O Modules/Ports
 - isolated
 - memory mapped ★

★ Supported by PIC16F877

Now to count how long it takes to do the `:Loop4` loop. The `addlw` instruction takes 1 cycle, the `btfss` takes 1 and the `goto` takes 2. So a single traversal of the loop takes 4 cycles. The last traversal of the loop will take one more cycle when the bit test jumps to the `return`. Since it is only one cycle in a millisecond, the error is small. The final part is the setting of the `<unknown>` value.

The call to the delay routine will take 2 cycles and the loading of `W` will take another cycle, and since the loop takes 4 cycles, we need to round out the overhead to 4 cycles and compensate for it in the initial value of `W`.

The final routine that generates 1 ms delay is show below

```
delay_1ms    movlw    0xF9        ; 0xF9 = 249
              nop
usec4        addlw    0xFF        ; add -1, cannot use decrement, why?
              btfss    STATUS,Z
              goto     usec4
              return
```

This routine can be expanded through the use of a variable to allow times in multiples of 1 ms. For example

; routine is passes the amount of ms to delay in `W` ; it uses an additional register variable `msec_count`

```
delay_ms     movwf    msec_count
msecloop     movlw    0xF8        ; 0xF8 = 248
              call     usec4      ; 248 * 4 + 2 (call) = 994
              nop
              nop
              decfsz   msec_count,F
              goto     msecloop
              return
```

The overhead for the call to `delay_ms` is neglected as well saving `W` in `msec_count`.

Go through the routine carefully making sure that you understand its operation.

Using built-in timers

The Timer 0 of the PIC will generate an interrupt, if enabled, when an overflow occurs. This can be an advantage over the previous type of delay because the CPU can be doing other useful work during the waiting. In non-time critical applications, the use of ISR's are not necessary and simple polling will suffice. It is fairly straightforward to generate small delays by initializing Timer 0 to specific values and letting it timeout while continuously checking for the timeout. Normally Timer 0 runs continuously whenever the PIC is turned on, but there may be times when the user may want to set the time when it starts. This can be done by ensuring that the `RA4/T0CK1` pin is grounded and setting Timer 0 in counter mode. Since the input to the counter is zero, no counting will take place and initialization of Timer 0 can take place without timing being started. When timing is required Timer 0 can then be switched to timer mode.

Simple timing (no interrupts)

The following section of code illustrates how simple timing can be done.

```

Init    bsf      STATUS,RPO      ; bank 1
        movlw    B'00100110'    ; set in counter mode, prescaler = 128
        movwf    OPTION_REG
        bcf      STATUS,RPO      ; back to bank 0
        bcf      INTCON,TOIF     ; clear to be sure
        .
        call     Delay
        .
Delay    movlw    106             ; time for 256 - 106 = 250 cycles
        movwf    TMRO
        bsf      STATUS,RPO      ; switch to bank 1 to access OPTION_REG
        bcf      OPTION_REG,TOCS ; start timing
        bcf      STATUS,RPO
Loop     btfss    INTCON,TOIF     ; keep checking for overflow
        goto     Loop
        bcf      INTCON,TOIF     ; clear flag
        return

```

Writing an Interrupt Service Routine (ISR)

The PIC16F877 has only one interrupt vector, located at 0x004, and as a result, polling must be done at the start of the ISR to determine which interrupt source generated the interrupt. Once the source is determined the appropriate handler can be executed. Within the handler, the interrupt flag bit must be cleared before re-enabling interrupts. Using the template given previously, an ISR designed to handle Timer 0, external and Port B change interrupts looks like the following.

```

ISR      movwf    w_temp          ; save W and STATUS
        movf     STATUS,W         ;
        movwf    status_temp     ;
Poll     btfsc    INTCON,TOIF     ; test if TMRO overflow occurred
        call     Timer_hndlr      ; call handler for TMRO
        btfsc    INTCON,INTF     ; test if external interrupt occurred
        call     Extern_hndlr     ; call handler for external interrupt
        btfsc    INTCON,RBIF     ; test if PORTB change interrupt occurred
        call     Change_hndlr     ; ...etc
ISRDone  movf     status_temp,W   ; restore pre-isr STATUS register
        movwf    STATUS          ;
        swapf    w_temp,F        ; restore pre-isr W register contents
        swapf    w_temp,W        ; ... without affecting the zero flag
        retfie                   ; return from interrupt

Timer_hndlr
        bcf      INTCON,TOIF     ; clear the overflow flag
        ; do processing
        return

Extern_hndlr
        ; clear appropriate flag etc
        return

Change_hndlr
        ; clear appropriate flag etc
        return

```

Simple timing (with interrupts)

If a 10 ms delay is required then there is no combination of TMR0 and prescaler that will give that time repeatedly so other methods must be devised. It can be seen that using a prescale value of 8, Timer 0 will overflow every 2.048 ms, and counting this five times will give 10.24 ms or approximately 10 ms. The following routine illustrates how this can be done.

```

COUNT EQU 0x05
clr     cntr
.
Delay   btfss  INTCON,T0IF
        goto   Delay
        bcf     INTCON,T0IF
        incf    cntr,F
        movlw   COUNT
        xorwf   cntr,W
        btfss   STATUS,Z
        goto    Delay
        clr     cntr
        return

```

Instead of polling the T0IF flag, an ISR can be used that is triggered by the overflow interrupt, in addition the counter, `cntr` is changed by the ISR. So the routine can now be broken into two parts and also can be made simpler by doing the check for the five times overflow of TMR0 by decrementing instead of incrementing `cntr`.

```

COUNT equ 0x05 ; ISR
; initialization of ISR etc
Poll    btfsc  INTCON,T0IF
        goto   Timer
        btfsc  INTCON,.
.
Timer    decf   cntr
        bcf     INTCON,T0IF
        goto    Poll

; end of ISR
; Main routine
Init     movlw  COUNT
        movwf   cntr
.
Delay    btfss  cntr,7
        goto    Delay
        movlw   COUNT
        movwf   cntr
        return

```

More precise timing

The time delay achieved in the preceding section is only an approximation to the desired value of 10 ms which means that TMR0 must overflow every 2 ms. Now the total count in TMR0 with a prescaler of 8 is $256 \times 8 = 2048$ and $2000 = 256 \times 8 - 6 \times 8$. Therefore if TMR0 is incremented by 6 every time it overflows then the cycle time will be 2 ms. This can be done in the `Timer` handler of the ISR, for example,

```
Offset    equ        6

Timer      movlw      Offset
           addwf      TMR0,F
           decf       cntr
           bcf        INTCON,T0IF
           goto       Poll
```

Note that whenever a write is done to TMR0, it waits for two cycles to re-synchronize before counting starts again. In this case the wait is not significant and the error in 10 ms is only 0.1%.

References

2001. PIC16F87X Data Sheet: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. Technical Reference Document 30292c, MicroChip Technology Inc.
- Katzen, Sid. 2003. *The quintessential pic microcontroller*. Springer. 2nd printing.
- Stallings, William. 2000. *Computer architecture and organization*. 5th ed. Prentice-Hall, Inc.

Review Exercises

1. Label the following statements about interrupts True/False:
 - (a) Interrupts may be triggered either by an electrical signal, or a special software instruction.
 - (b) Before handling an interrupt, all processor registers which may be affected by the handler must be saved, so they can be restored later.
 - (c) When an interrupt occurs the instruction which is being processed is stopped, and re-executed after the interrupt has been handled.
 - (d) It is possible for an interrupt handler to be interrupted by another interrupt.
 - (e) Interrupt prioritisation may be achieved either by using a dedicated peripheral, or by checking the interrupt sources in a pre-determined order.

Interrupts vs. Polling

Initialise -- configuration Initialise -- flags Initialise -- mask Command -- control	Initialise -- configuration
Repeat <i>Job</i>	Repeat **Command -- control Do <i>Job</i> Until -- state **Read -- data
ISR: save context **Command -- control **Read -- data **Clear -- flags restore context	

Software Interrupts

- Trigger does not come from an external device – it is an instruction (which may indicate the appropriate vector).
- These are used to:
 - facilitate BIOS and other low level functions
 - allow programs to access memory and peripherals in a structured “known” fashion

Prioritizing hardware interrupts

- Priority in software handler
 - Ordering of the handler calls
 - Disabling/Enabling other triggers
- Priority using an external peripheral: Peripheral interrupt controller (PIC)
 - Only allows one interrupt to be serviced at a time.
 - Priority is either determined by the connection position or can be programmed using a peripheral register.

Typical applications

- Hardware Interrupts
 - Peripheral reporting completion
 - A/D conversion done
 - Interval expired
 - DMA request complete
 - Peripheral requesting service
 - Key-press
 - Clock tick
- Software Interrupts
 - Time triggered actions
 - System clock update
 - Protected mode functions
 - BIOS function call (joystick read)

2. All PIC16F877 ISR routines must save the working register before anything else is done. This is because:
- (a) using the `retfie` instruction will corrupt the working register.
 - (b) after 10 instruction cycles without use, the Working register will lose its value.
 - (c) the ISR routine is likely to change the Working register, and the main program needs the original value.
 - (d) we can't poll the interrupt sources unless the Working register is empty.
 - (e) none of the above

3. Choose the word combination which best completes the following sentence(s):

The interrupt 3I is the address of the instruction executed when the interrupt occurs. Interrupts are typically used when the 3II needs to communicate information to the 3III .

- (a) 3I: service routine; 3II:CPU ; 3III:peripheral
 - (b) 3I: vector; 3II:CPU ; 3III:peripheral
 - (c) 3I: service routine; 3II:peripheral; 3III:CPU
 - (d) 3I: vector; 3II:peripheral; 3III:CPU
 - (e) 3I: handler; 3II:peripheral; 3III:CPU
4. (a) In your own words, explain what happens when an interrupt occurs, and when the main program resumes.
- (b) In your own words, explain what is meant by saving and restoring the "context", and why these processes are needed in interrupt procedures.
5. (a) Differentiate between hardware and software interrupts.
- (b) Which is preferable: software or hardware prioritisation of interrupts? Justify your answer.
- (c) Student A tells Student B that hardware interrupts are interrupts that are prioritised using hardware. Is student A correct? Explain your answer.
6. (a) What is the reason for having the interrupt sources in a particular order?
- (b) When the ISR is entered in response to one specific interrupt, is it possible that a different source might be serviced first? Discuss your answer.
- (c) If while one interrupt source is being serviced, a second interrupt source requests service, will the second source be serviced before the `retfie` instruction at the end of the ISR? Discuss your answer.
7. Identify one source of interrupts on the PIC 16F877 (apart from the timers), and locate the flag and enable bits for your chosen interrupt source.

8. “The speed of a rotating shaft can be measured by using a coded disk to generate a pulse on each angular advance of 10° , which can be used to interrupt a PIC. If the top speed is 20,000 revolutions per minute, what is the absolute maximum duration of the ISR in this worst case situation to avoid missing pulses? You may assume a crystal frequency of 4MHz. ”(Katzen 2003; Q7.4 p.193)

Where code is requested, it is sufficient to simply include the relevant snippets/fragments. Use comments to indicate any assumptions you make about the rest of the code.

9. In a typical ISR, part of the initialization saves the W and STATUS registers, however this assumes that the main code that will be interrupted will never be in Bank 1 at the time of the interrupt. If the ISR ever switches banks then the values of W and STATUS that are restored would be incorrect³⁴. In processors where the Bank 0 addresses are different from the Bank 1 addresses several things can be done to relax this assumption.

- Interrupts can be disabled before switching to Bank 1 and re-enabled after switching back to Bank 0.
- Use two addresses, one in each bank, to hold the temporary value of W, i.e. W_TEMP. For example,

```
W_TEMP    equ      0x20
W_TEMP_    equ      0xA0
```

Now when W is stored in W_TEMP using direct addressing at the beginning of the ISR it will go in either of two locations: 0x20 or 0xA0, depending on which bank is active at the moment the interrupt occurs.

- (a) Rewrite the initialization of the ISR that saves W and STATUS in this case, switching to Bank 0 for the duration of the ISR. There would be need only for one location for STATUS_TEMP defined in Bank 0.
 - (b) Rewrite the ending of the ISR to restore the STATUS and W.
10. You are responsible for writing an application for a PIC16F877 with an 8MHz external clock signal, which reads 8-bit information from a device. The device is configured so that the most recent byte appears on PortB every 40ms, and stays there until the next byte is available. When the byte changes, the device raises the READY signal line for 5ms and then drops it.
- (a) **In your own words**, explain the difference(s) between using blind cycle polling, busy-wait polling, and interrupts to accomplish your task.
 - (b) **In your own words**, explain how indirect addressing may be used to implement a circular buffer.
 - (c) For each of the three approaches (blind cycle polling, busy-wait polling, and interrupts)
 - explain where the READY signal line needs to be connected,
 - write a program for the PIC16F877 which will store the 8 most recently read bytes in a circular buffer,
 - determine the number of program memory locations needed to store the code and the number of data memory or register locations required to store constants/variables.

³⁴In the PIC16F8x, Bank 1's General purpose registers are mapped to Bank 0, so that this argument does not hold

Tutorial Exercise 7B Offline Version³⁵

ID# _____

The Bank of Nowhere has recently installed a Passive Infra-Red sensor at the doorway to their bank vault. The sensor generates a 5V signal when a person is standing in front of the doorway, and 0V when no-one is at the doorway. The sensor is connected to `PORTB<0>`. The bank wishes to maintain a count of the number of people who ARRIVE at the vault doorway.

1. Write a routine called `Init` which will enable the appropriate interrupts. Your code should be appropriately commented. *4 marks*

2. Write an interrupt handler which will increment a register named `Cnt` when a person arrives. *6 marks*

PLAIGIARISM DECLARATION:

For the purposes of this exercise, unauthorised collaboration is any form of collaboration which does NOT fall into one of the following categories:

- verbal or written discussion/clarification of question and/or related concepts

Department of Electrical and Computer Engineering

PLAGIARISM Plagiarism is the presentation by a student of an assignment which has in fact been copied in whole or in part from another student's work, or from any other source (e.g. published books or periodicals), without due acknowledgement in the text.

COLLUSION Collusion is the presentation by a student of an assignment as his or her own which is in fact the result in whole or part of unauthorised collaboration with another person or persons.

DECLARATION I declare that this assignment is my own work and does not involve plagiarism or collusion. I have read and understood University Examination Regulations 73,75,76 and 79 regarding cheating.

Signed:

Date:

(Department of Electrical and Computer Engineering)

³⁵Students are advised that Offline versions are to be used either with the explicit permission of the course lecturer OR at times when the electronic course support site is unavailable for more than 48 hours.

Unit 19 Write the letter you have been assigned here_____.

- A: Count the number of times a single pole double throw switch is closed.
- B: Turn an output pin on/off when a non-latching push-button is pressed.
- C: LED flashes at a constant frequency
- D: Speaker beeps at a constant frequency

For the task which you have been assigned:

1. Describe one way in which interrupts could be used to accomplish your task.
2. Identify 2 issues you need to consider in order to decide if using interrupts is appropriate for your task.

Reflection & Feedback

- Indicate the objectives that you feel you have achieved in this unit.
 - explain the operation of interrupts, the different types of interrupts (h/w, s/w), how interrupts may be prioritized (peripheral interrupt controller), and identify common applications of interrupts.
- Which aspect of this unit did you have the most difficulty understanding?
- Which aspect of this unit did you like best? Why did you like it?
- Identify one thing (if any) that you learned while doing this unit/tutorial.
- Identify one way in which this unit/tutorial could be improved.

