

Programming Interrupts for DOS-Based Data Acquisition on 80x86-Based Computers

Table of Contents

1. Introduction
2. What Are Interrupts?
3. Interrupts for Data Acquisition
4. Interrupts on the IBM PC
5. Interrupt Programming for DOS
6. Examples of Interrupt Programming-IRQ, ESP, ASM
7. Conclusion

Introduction

This application note explains how to use interrupt programming on computers based on the 80x86 family of microprocessors running the DOS operating system. This family includes the 8086, 8088, 80286, 80386, and 80486 microprocessors that are used in PC/XT/AT, EISA, and IBM PS/2 computers. This application note describes interrupts, explains how interrupts work and how they are implemented on 80x86-based computers, and discusses programming guidelines when designing an interrupt-driven DOS application. Finally, this application note contains example programming code to demonstrate how to program interrupts for a DOS-based data acquisition application.

What Are Interrupts?

Interrupts are very important to the operation of any computer. Interrupts give the processor the ability to respond quickly to its peripherals (such as the keyboard and the hard disk) and to the outside world in general. Without interrupts, a processor would be unable to service more than one task efficiently and reliably. The importance of the interrupt is illustrated when comparing an interrupt to a doorbell. If your door did not have a doorbell, you would have to periodically go to the door to see if anyone happened to be there at that time. Of course, that would be very inefficient. With a doorbell, you only need to go to the door when the doorbell rings, and you are then confident that someone is there waiting. Likewise, it is not efficient for the processor to continually check whether any of its peripherals require attention at a given time. An interrupt is a doorbell to the processor to signal that some device needs service.

During normal execution of a program, instructions are read from memory and executed sequentially by the processor. The processor uses a special register called the *instruction pointer* to keep track of the next instruction to be executed. A set of general-purpose registers are used for manipulation and temporary storage of any data used by the program.

An interrupt is a special input to the processor. When the processor is signaled that an interrupt has occurred, the processor finishes the instruction currently being executed and saves the instruction pointer and a status word to the stack. The *stack* is a special block of memory used to keep track of information during function calls and interrupts. The processor uses a special register called the *stack pointer* to keep track of the location of the *top* of the stack, which is where the last item was added to the stack.

After the processor saves this information to the stack, the processor branches to a special routine called an *interrupt service routine* or *interrupt handler*. If needed, this routine can determine the source of the interrupt and then service the device that requested the interrupt. This process may involve reading a character from the keyboard and updating the monitor, or moving data from an adapter board into computer memory. After handling the interrupt, the last instruction executed in the interrupt service routine is an *interrupt return instruction*. This instruction causes the processor to restore the instruction pointer and status register with the values that were saved to the stack at the time of the interrupt and to resume execution.

Interrupts for Data Acquisition

Many data acquisition applications can benefit greatly from the interrupt capability of the processor. Interrupts can be used for immediate communication between the data acquisition hardware and the computer. They can be used to time or synchronize data acquisition with external events or to respond promptly to external events and alarm conditions. However, interrupts are not always the best approach in data acquisition applications. You must consider the advantages and disadvantages of various data acquisition methods before deciding to use interrupt-driven acquisition.

Data acquisition involves the monitoring of one or several sources of physical data at a regular rate, which is often determined by a clock signal. Commonly, the data originates from analog-to-digital converters (ADCs), counter/timer devices, or switches. In general, there are three approaches to acquiring data from an external device or synchronizing communication between devices. These three approaches are described as follows:

- **Polling** – This method involves periodically reading the status of the device to determine whether the device needs attention.
- **Direct Memory Access (DMA)** – A dedicated processor, the DMA controller transparently transfers data from the device to computer memory, or vice versa.
- **Interrupts** – the device is configured to interrupt the processor whenever the device requires attention.

For some applications, polling a device can be a very effective way to service a data acquisition operation. The polling method is simple, easy to understand, and easy to debug. If the processor can be dedicated to the single task of monitoring and servicing the device, high acquisition rates can be realized. However, dedication to a single task is often not possible. Other devices may also require timely monitoring, or the processor may be required to do other work, such as handling previously acquired data. DMA or interrupts would then be a better approach.

DMA uses a DMA controller to move data from one location to another, making fast data transfers of large blocks of data possible. DMA is fast because dedicated hardware transfers the data and requires only one or two read/write bus cycles per transfer. In addition, the processor is free for other activities (such as processing acquired data) because the DMA controller proceeds with the data acquisition in the background. However, the processor may need to access the data on a point-by-point basis as it is being acquired. DMA is not well-suited for this type of application, and interrupts or polling may be preferable. Furthermore, the number of DMA channels in a PC is limited, and a dedicated DMA channel is required for each unique data acquisition source.

With interrupts, the processor can respond quickly to an event (such as a clock signal) to transfer data or to synchronize different events. Unlike DMA, an interrupt-driven system has the advantage of being able to process the data on a point-by-point basis, which can be particularly important in control applications. Consider a typical control application synchronized with a clock signal, for example. The application may require that, with every pulse of a clock, the processor read the result of an A/D conversion, read the state of several digital lines, use these results to compute a new control value, and write this value to a digital-to-analog converter (DAC). DMA would not be useful for such an application, and a polling routine would not directly provide synchronization with the clock signal. Interrupts, however, make it easier to synchronize the input and output with the clock signal and for the processor to perform other general processing tasks in the foreground, if needed.

One disadvantage of interrupts, however, is that they require processor time, and there can be a significant amount of software overhead time associated with the interrupt service routine. When designing a data acquisition system, you must estimate the amount of this overhead time, which consists of the time required by the processor to vector to the proper interrupt service routine, known as *interrupt latency*, and the time necessary to perform the task required within the service routine. If the overhead is too large for the required acquisition rate, then DMA or a very efficient polling routine must be used. In applications that are time critical, the interrupt service routine must be written to be as fast as possible. As a result, interrupt service routines for data acquisition applications are often written in assembly language.

Many driver software packages use both interrupt and DMA methods to handle data acquisition servicing. NI-DAQ®, for example, includes sophisticated, high-level routines for data acquisition that takes advantage of the interrupt and DMA capabilities of the hardware. Application developers should seriously consider using such a driver package, which spares the developer the complexity and details of programming the low-level hardware of the computer and the adapter board. Using a driver package can significantly decrease the amount of time required to develop an application.

Interrupts on the IBM PC

The 80x86 Family Register Set

The 80x86 family of microprocessors use 14 separate 16-bit registers that can be grouped into four categories: general-purpose registers, segment registers, offset registers, and the flags register. The registers, their categories, and their use are listed in Table 1.

Table 1. Registers of the 8086 Family of Microprocessors

Register	Category	Use
AX	General-purpose	—
BX	General-purpose	—
CX	General-purpose	—
DX	General-purpose	—
CS	Segment	Code segment
DS	Segment	Data segment
ES	Segment	Extra segment
SS	Segment	Stack segment
SP	Offset	Stack pointer
BP	Offset	Base pointer
SI	Offset	Source index
DI	Offset	Destination index
IP	Offset	Instruction pointer
Flags	Flags	Status flags

The general-purpose registers are used by the processor for temporary storage of data. To facilitate use of 8-bit and 16-bit values, each of these 16-bit registers can also be addressed as a pair of 8-bit registers. AL, AH, BL, BH, and so on are used to address the lower (L) or higher (H) bytes.

The segment registers are used to store the base address of 64 kilobyte memory segments. The absolute or complete address of a memory location is obtained by combining the segment registers with an offset value stored in one of the offset registers. A segment register identifies a distinct 64 kilobyte segment. A particular memory location within the segment is indicated by a 16-bit offset register. For example, the memory location 01F0:0009 refers to byte 9 within segment 01F0.

The instruction pointer (IP) holds the offset address of the next instruction to be executed by the processor. The absolute address of the instruction is determined by combining the IP offset register with the Code Segment (CS) Register. The IP Register is automatically updated each time the processor loads an instruction.

Another offset register is the Stack Pointer (SP) Register, which, when combined with the Stack Segment Register (SS), points to the current top of the stack. The stack is a last-in first-out (LIFO) memory structure. As items or data are added, the stack gets larger. When an item is removed, the first item to come off the stack is the last item added to the stack. When data is added to or removed from the stack, the value of the stack pointer is appropriately updated.

The Flags Register, as the name implies, stores the state of nine CPU status and control flags. Six of the bits are status flags, which are automatically set or cleared after the execution of every instruction. The three other bits are used for control purposes. These bits include the direction flag, the trap flag, and the interrupt flag. The interrupt flag (IF) bit is used to enable or disable hardware interrupts.

The 80x86 Family Interrupts

The first 1024 bytes of memory are reserved for the interrupt vector table. This table holds up to 256 vectors, each of which is a 4-byte pointer to a specific interrupt service routine that is executed when the corresponding interrupt is processed. The design of the 80x86 family requires certain interrupt vectors to be used for specific functions. While many of these vectors are reserved for software interrupts, a limited number of vectors are reserved for hardware interrupts.

When a hardware interrupt occurs, the processor first responds by pushing the contents of the Flags, CS, and IP Registers onto the stack and disabling any further hardware interrupts by clearing the IF bit in the Flags Register. The processor then looks to the system bus for an 8-bit interrupt number, multiplies the number by 4 (the size of each vector in bytes), and uses the result as an offset into the interrupt vector table. The vector address pointed to by the result is then loaded into the CS and IP Registers, and the processor resumes operation at this new location pointed to by the instruction pointer. After processing the interrupt, the processor restores the Flags, CS, and IP Registers with the values saved to the stack when the interrupt occurred. Because the IF bit is cleared after the Flags Register is saved to the stack, restoring the Flags Register also resets the IF bit, which enables hardware interrupts.

Hardware Interrupts and the Intel 8259A Programmable Interrupt Controller

Hardware interrupts are sent to the processor through an Intel 8259A programmable interrupt controller, which provides the system with eight prioritized hardware interrupts. When the 8259A receives an interrupt request, typically from a peripheral device, the controller drives high its output (INT), which is connected to the processor's interrupt input pin (INTR). The INTR pin is used by the processor to signal the occurrence of a maskable interrupt. If the IF bit in the Flags Register is set, the processor sends an acknowledge signal to the 8259A controller.

After receiving the interrupt acknowledge signal (INTA) from the processor, the interrupt controller places the appropriate interrupt vector on the system bus. The eight hardware interrupts (IRQ0 through IRQ7) on the 8259A are mapped by the controller to interrupt vectors 8 through 15 (0Fh). The interrupts are prioritized, IRQ0 having the highest priority and IRQ7 having the lowest priority. The PC design assigns interrupts IRQ0 through IRQ7 for particular peripherals. Table 2 lists these assignments. IRQ0 and IRQ1 are reserved for the system timer and the keyboard, respectively, while IRQ2 is reserved for cascading additional interrupt controllers. IRQ3 through IRQ7 are assigned to various peripherals but are connected to the I/O bus (see Figure 1). If these peripherals are not present, interrupts IRQ3 through IRQ7 can be used for other purposes.

Table 2. Interrupt Assignments on the IBM PC/XT Computer

Interrupt	Vector Index	Action
IRQ0	08h	Timer tick (18.2 times/sec)
IRQ1	09h	Keyboard input
IRQ2	0Ah	Reserved for cascading interrupts
IRQ3	0Bh	COM2
IRQ4	0Ch	COM1

IRQ5	0Dh	Fixed disk controller
IRQ6	0Eh	Floppy disk controller
IRQ7	0Fh	Printer controller

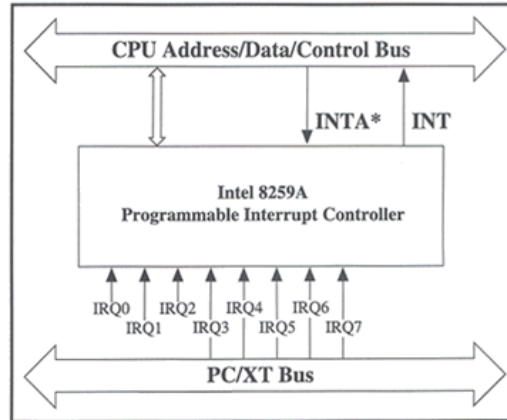


Figure 1. Block Diagram of 8259A Interface to CPU and PX/XT Bus

The PC AT computer is designed with a second interrupt controller for eight additional IRQ levels. The INT line of the second controller, the *slave*, is cascaded to the IRQ2 line of the first controller, the *master*, as illustrated in Figure 2. The eight interrupts (IRQ8 through IRQ15) from the second controller are mapped to interrupt vectors 112 (70h) through 119 (77h). Because these additional interrupts are effectively connected to the IRQ2 line of the master 8259A controller, they take priority over IRQ3 through IRQ7 of the master 8259A.

Table 3. Additional Interrupt Assignments on IBM PC AT Computers

Interrupt	Vector Index	Action
IRQ8	70h	Real-time clock
IRQ9	71h	Redirect cascade
IRQ10	72h	Reserved
IRQ11	73h	Reserved
IRQ12	74h	Auxiliary device
IRQ13	75h	Math coprocessor exception
IRQ14	76h	Fixed disk controller
IRQ15	77h	Reserved

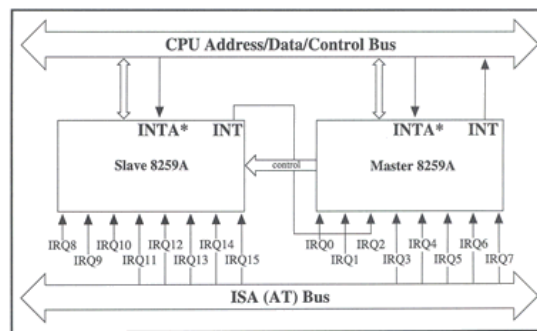


Figure 2. Cascaded Configuration of the Intel 8259A as Used in PC AT Computers

A device, such as a plug-in adapter board, issues an interrupt by causing a high-to-low transition on one of the IRQ lines. With most adapter boards you can choose which IRQ line to use, usually by positioning a jumper. The board is then programmed to assert an interrupt signal under certain conditions, such as the completion of an A/D conversion or the pulse of a counter/timer output. Many adapter boards are able to cause interrupts under more than one condition. Therefore, the interrupt service routine must determine the cause of the interrupt, usually by reading a status register on the adapter board.

The Micro Channel bus, used in many of the IBM PS/2 computers, and the EISA bus also use the cascaded Intel 8259A configuration shown in Figure 2 for prioritized interrupts. The previous discussion applies to these computers as well. However, there is one significant difference concerning how the interrupt line works. While interrupts on PC/XT/AT computers are triggered by a high-to-low transition, or edge, of the interrupt line, the Micro Channel bus defines an interrupt signal as asserted by the state of the interrupt line. With this configuration, referred to as *level mode*, an interrupt is considered to be asserted, or active, as long as the interrupt line is held in the high state. With the EISA bus, on the other hand, both level mode and edge-triggered interrupts can be used.

Interrupt Programming for DOS

There are three basic operations to programming an interrupt-driven DOS application – installing the interrupt vector, which points to the interrupt service routine, into the interrupt vector table; handling the interrupt with an interrupt service routine; and removing the interrupt vector. These operations are described as follows.

Installing the Interrupt Vector

Installing an interrupt vector consists of putting the correct address of the interrupt service routine, or interrupt vector, in the interrupt vector table, and enabling interrupts. First, you determine the proper location in the interrupt vector table to install the handler address. As mentioned previously, interrupts IRQ0 through IRQ7 map to interrupt level 8 through 15, and interrupts IRQ8 through IRQ15 map to interrupt levels 112 (70h) through 119 (77h). For example, the address of a handler that is to service an IRQ0 interrupt must be installed at location or index 8 in the vector table. DOS has two functions, accessible through the DOS function call INT 21h, for setting and retrieving interrupt vectors. Function 25h is used to set an interrupt vector. When function 25h is called, the AL Register holds the interrupt table index, and the DS and DX Registers point to the address of the interrupt service routine. Function 35h is used to *get*, or return the value of, an interrupt vector from the table. This function should be called before the interrupt is set so the original vector can be saved and then restored when removing the interrupt.

For the interrupt controller to recognize an interrupt, the appropriate interrupt must be enabled. Enabling interrupts is done via the Interrupt Mask Register of the interrupt controller. This register contains bits that mask each of the interrupt request lines. If the bit is cleared or set, then the corresponding interrupt is enabled or disabled, respectively.

Whenever the interrupt controller is to be programmed, interrupts to the processor *must* be disabled. Interrupts are disabled with the assembly language call `c1i` and re-enabled with the function `sti`. These two instructions manipulate the interrupt flag (IF) bit in the Flags Register. Interrupts should be also be disabled during any manipulation of the interrupt vector table.

Some high-level languages include function calls that operate in the same manner as functions `25h` and `35h`. These functions provide a convenient way to install interrupt vectors without having to resort to assembly language programming. Table 4 lists these functions for some popular languages. Consult the user manual for the particular language you are using for more information concerning these or similar functions.

Table 4. Some Available Functions to Set and Get Interrupt Vectors

Language	To Set Interrupt Vector	To Get Interrupt Vector
DOS Function	INT 25h	INT 35h
Microsoft C 5.0	<code>_dos_setvect</code>	<code>_dos_getvect</code>
Turbo C	<code>setvect</code>	<code>getvect</code>
Turbo Pascal 4.0	<code>SetIntVec</code>	<code>GetIntVec</code>

Interrupt Handling

In an interrupt handler or service routine you must first save any registers that the interrupt service routine will modify so the registers can be restored before returning from the service routine. The interrupt handler may then need to address the device to verify that the interrupt originated from the device. Many devices are capable of causing interrupts for a number of different reasons, so reading a Status Register may be necessary to determine the exact cause of the interrupt.

Having verified the source and cause of the interrupt, the interrupt service routine then does the particular task for that application. For example, in a typical data acquisition application, the routine reads a value from the adapter board, stores the value into memory, and increments a counter. After handling the interrupt, the routine must restore the processor registers that were saved at the beginning of the routine.

Finally, a non-specific end of interrupt (EOI) is sent to the interrupt controller. If the EOI is not sent, no further interrupts of equal or lower priority can occur.

Note: *In systems with cascaded interrupt controllers (non-XT computers), EOI instructions must be sent to both the master and the slave 8259A interrupt controllers. Again, interrupts to the processor must be disabled while writing the EOI instructions to the interrupt controllers.*

When writing an interrupt service routine, do *not* use DOS functions unless special precautions are taken. DOS is not re-entrant. If the processor is interrupted during a DOS call, and the service routine attempts to call another DOS function, the computer may lock up. To prevent this problem, you may use an undocumented `INT 21h` function, function `34h`, to determine if a DOS function is in progress. The `34h` function returns a pointer in the ES Register and the BX Register that points to a DOS busy flag (InDOS flag). When an `INT 21h` function starts, the flag is incremented by one; when the function ends, the flag is decremented by one. If the flag is zero, then no DOS functions are currently being executed.

Note: *The INT 21h function is undocumented and not officially supported by IBM and Microsoft. Therefore, the function may not be available in all versions of DOS.*

Removing Interrupts

Finally, the application must remove the interrupt and restore the vector table to its original state. First, disable the interrupt level using the Interrupt Mask Register. Then, using the data stored during the interrupt installation, the original interrupt vector is restored using function `25h` or a similar high-level function call.

Examples of Interrupt Programming-IRQ, ESP, ASM

```
install_ISR:      This function performs the steps necessary to install an interrupt
                  service routine and to enable the interrupt on the interrupt control-
                  ler. The IRQ number and the address of the interrupt service routine
                  are passed as parameters.

remove_ISR:      This function uses the data saved by install_ISR to restore the orig-
                  inal interrupt vector and disable the interrupt in the interrupt
                  controller.

IRQ_Handler:     This function contains the shell for an example interrupt service rou-
                  tine. The example reads data from an adapter board, stores the data
                  into an array, and increments a counter. In addition, the function
                  sends the EOI to the interrupt controller.
```

Also included in the listing is an example of a C program that would use these routines.

```
***** IRQESP.ASM *****
;
;   Engineering Software Package
;   Rev A.0
;   Copyright 1991 National Instruments Corporation.
;   All rights reserved.
;
;   This file contains code that installs and removes interrupt vectors
;   and programs a PC/XT/AT interrupt controller. Included is an
;   example of a user-written interrupt service routine. This code was
;   written for the Microsoft Macro Assembler.
;
*****

Public _install_ISR          ; _install_ISR and _remove_ISR are
Public _remove_ISR          ; PUBLIC so that they can be called
                             ; from a C program.
Public _IRQ_Handler         ; _IRQ_Handler is PUBLIC so that its
                             ; address can be passed to
                             ; _install_ISR by a C program.

; These external declarations enable the assembly language interrupt
; service routine to access data declared in the calling C program.
; Because the C compiler renames io_buffer as _io_buffer, these names
; are prepended with an underscore.
```

```

EXTRN  _io_buffer:WORD
EXTRN  _io_count:WORD

MASTER_IRQ_BASE  EQU  20h          ; I/O addresses of the master interrupt
MASTER_IRQ_MASK  EQU  21h          ; controller.

SLAVE_IRQ_BASEEQU  0A0h          ; I/O addresses of the slave interrupt
SLAVE_IRQ_MASKEQU  0A1h          ; controller.

NON_SPEC_EOIEQU  20h              ; The bit pattern for a non-specific
                                   ; end of interrupt command.

IRQ_TEXT  SEGMENT WORD PUBLIC 'CODE'
          ASSUME CS:IRQ_TEXT

IRQ_numDB  ?                      ; IRQ_num contains the IRQ number (0
                                   ; to 15).
INT_numDB  ?                      ; INT_num contains the INT number or
                                   ; index into the interrupt vector
                                   ; table (8 to 15 or 112 to 119).
old_int_offDW  ?                  ; old_int_off and old_int_seg contain
old_int_segDW  ?                  ; the OFFSET and SEGMENT of the
                                   ; original interrupt vector.

;*****
;  _install_ISR performs the steps necessary to install a user-
;  defined interrupt service routine and to enable the interrupt. The
;  name is prepended with an underscore so that _install_ISR can be
;  called from a C program (the C compiler automatically prepends an
;  underscore to every global name). The syntax for the call from a C
;  program is as follows:
;
;  install_ISR (IRQ_num, ISR_proc);
;
;  Because _install_ISR is declared as a far procedure, _install_ISR
;  must also be so declared in the calling C program or, equivalently,
;  the C program must be compiled with the /AL flag.
;*****

_install_ISR  proc                far

    push  bp                      ; Save the registers to be used and
    mov  bp, sp                  ; set up the base pointer to the stack
    push  ax                      ; frame containing the arguments to
    push  bx                      ; this procedure.
    push  cx
    push  dx
    push  ds
    push  es

    mov  ax, cs                  ; Set up ds to point to the segment
    mov  ds, ax                  ; containing the data items. In this
                                   ; case the data segment is also the
                                   ; CODE segment.

    mov  al, [bp+6]              ; Grab the IRQ number from the stack.
    mov  IRQ_num, al              ; Store the IRQ number for later use.
    cmp  al, 7                   ; If the IRQ number is 0 to 7, you must
    ja   slave                   ; add 8 to it to index the right place
    add  al, 8                   ; in the interrupt vector table. If the
    jmp  getvec                  ; IRQ number is 8 to 15, you must add
                                   ; 104 to index the right place. This
slave:add    al, 104              ; index is also called the INT number.

getvec:mov  INT_num, al           ; Store the INT number for later use.
    mov  ah, 35h                 ; Get the previous ISR vector in es and
    int  21h                     ; bx. al already contains the correct
    mov  old_int_off, bx         ; index into the interrupt vector
    mov  old_int_seg, es         ; table. The old vector is needed when
                                   ; removing the user-defined ISR vector.

    cli                          ; Disable interrupts while installing
                                   ; a vector and changing the masks.
    push  ds                     ; Save the current value of ds.
    mov  al, INT_num             ;
    mov  ah, 25h                 ; To install your own vector, ds and dx
    mov  dx, [bp + 10]           ; must point to it when calling DOS
    mov  ds, dx                  ; function 25h and al must contain
    mov  dx, [bp + 8]           ; the INT number, or index, into the
    int  21h                     ; interrupt vector table.

    pop  ds                      ; Restore ds.

; For the interrupt controller to recognize your interrupt, the interrupt

```

```
; must be "unmasked". The MASTER_IRQ_MASK Register controls IRQ levels
; 0 to 7. Clearing bit x (for example, bit 5) enables IRQ level x (for
; example, IRQ 5). Setting a bit disables the corresponding IRQ level.
; The SLAVE_IRQ_MASK Register controls IRQ levels 8 to 15. Clearing bit
; x (for example, bit 2) enables IRQ level x + 8 (for example, IRQ 10).
```

```
mov     bx, 1                ; bx holds the new mask.
mov     cl, IRQ_num          ; The new mask contains all ones,
shl     bx, cl               ; except in the bit position
not     bx                   ; corresponding to your IRQ level.

in      al, MASTER_IRQ_MASK  ; Read the current master mask.
and     al, bl               ; AND the mask with the low byte of the
out     MASTER_IRQ_MASK, al  ; new mask, and write out the result.
jmp     $+2                  ; This process provides a delay between
                             ; reads and writes between these chips.

in      al, SLAVE_IRQ_MASK   ; Read the current slave mask.
and     al, bh               ; AND the mask with the high byte of the
out     SLAVE_IRQ_MASK, al   ; new mask, and write out the result.

pop     es                   ; Restore all the registers.
pop     ds
pop     dx
pop     cx
pop     bx
pop     ax
pop     bp
sti                                           ; Enable interrupts and
ret                                           ; return.
```

_install_ISRendp

```
;*****
;      This is the shell of an interrupt handler.
;*****
```

_IRQ_Handlerprocfar

```
push    ax                   ; Save ax and ds.
push    ds

; Code to handle the interrupt goes here - usually a register on the
; adapter board is read to make sure that is where the interrupt
; came from and, if the adapter can generate an interrupt for more
; than one reason, to determine the reason for the interrupt. Once
; verified that the interrupt indeed came from the correct
; adapter board, this routine might read a value from the board,
; store the value in an array, and increment a counter.
; Here is an example of such code:

push    dx                   ; Save any other registers used
push    di                   ; in this section.

mov     ax, SEG _io_count    ; Set up ds to point to the segment
mov     ds, ax               ; where _io_count is declared.

mov     di, _io_count        ; Load the sample count into di.
shl     di, 1                ; Multiply the integer count by 2 to
                             ; convert it to a byte count.

mov     dx, 0236h            ; Load an adapter address.

in      ax, dx               ; Read something from the adapter.

mov     _io_buffer[di], ax   ; Store the value; this assumes that
                             ; _io_buffer and _io_count are in the
                             ; same segment and that adding di to
                             ; the beginning of the array does not
                             ; cross over into the next segment.

inc     word ptr _io_count    ; Increment the sample count.

pop     di                   ; Restore registers used in the
pop     dx                   ; adapter handling code.

; Code to tell the adapter that the interrupt has been serviced goes
; here - usually the adapter contains a location that, when written
; to, causes the adapter to cease asserting the interrupt. This must
; be done before the next step (sending EOI to the interrupt
; controller) or the same interrupt is asserted again, and the
; computer may lock up (caught in an endless loop of asserting and
; servicing the same interrupt).
; Now send a non-specific end of interrupt to the interrupt con-
; troller. If the IRQ level is 8 to 15, send EOI to the slave
; first and then to the master. Otherwise, only send EOI to
```

```

; the master. It is not necessary to explicitly disable
; interrupts here because the CPU has already done so during the
; interrupt acknowledge cycle.

mov     ax, cs                ; Set ds to point to the segment
mov     ds, ax                ; where IRQ_num is declared.
mov     al, NON_SPEC_EOI      ; Move the EOI code into al.
cmp     byte ptr IRQ_num, 8    ; If the IRQ level is less than 8,
jb      Ack_Master            ; send EOI to the master only.
out     SLAVE_IRQ_BASE, al     ; Otherwise, send EOI to the slave.
jmp     $+2                   ; Delay.

Ack_Master:
out     MASTER_IRQ_BASE, al    ; Send EOI to the master.

pop     ds                    ; Restore remaining registers.
pop     ax

iret                             ; Return with the IRET instruction.

_IRQ_Handler endp

;*****
; _remove_ISR uses the data saved by _install_ISR to restore the
; original interrupt vector and to set the proper bit in the mask
; registers to disable the interrupt. _remove_ISR can be called from a
; C program by the following statement:
;
; remove_ISR();
;
; Remember, this is a far procedure like _install_ISR.
;*****

_remove_ISR    proc            far

push     ax                    ; Save the registers.
push     bx
push     cx
push     dx
push     ds

mov     ax, cs                ; Set up ds to point to the segment mov     ds, ax    ; containing the data; in this case, ; ds is also the CODE segment. mov
bx, 1        ; bx holds the new mask. mov     cl, IRQ_num    ; It contains all zeroes except shl     bx, cl    ; in the bit position
corresponding to ; your IRQ level.

cli     ; Disable interrupts while changing ; masks and installing ISR vectors. in     al, MASTER_IRQ_MASK    ; Read the current master mask. or
al, bl      ; OR the current master mask with the ; low byte of the mask and out     MASTER_IRQ_MASK, al    ; write the new mask out. jmp
$+2        ; Delay.

in      al, SLAVE_IRQ_MASK    ; Read the current slave mask. or     al, bh      ; OR the current slave mask with the ; high byte of the mask and
out     SLAVE_IRQ_MASK, al    ; write the new mask out.

mov     al, INT_num           ; Restore the original vector by mov     dx, old_int_off    ; installing the vector the same way mov     ds, old_int_seg
; you installed your ISR vector. mov     ah, 25h int     21h

pop     ds                    ; Restore the registers. pop     dx pop     cx pop     bx pop     ax sti     ; Enable interrupts. ret _remove_ISR    endp
_IRQ_TEXT    ENDS
END

/*****
* Here is an example of a C program that would use the routines in
* IRQESP.ASM.
*****/

extern void far install_ISR(); extern void far remove_ISR(); extern void far IRQ_Handler();

int io_buffer[1000]; int io_count;

main() {    int i;

/* Initialize the sample counter. */    io_count = 0;

/* Install the interrupt service routine, IRQ_Handler(), for IRQ 5. */    install_ISR (5, IRQ_Handler);

/* Program your adapter board as necessary to generate the interrupts. */
/* When done (perhaps when io_count reaches 1000), disable your board's
   interrupts. */

/* Re-install the previous interrupt service routine. */
remove_ISR ();

/* Look at the data. */
for (i=0; i<io_count; i++) printf("\n%d",io_buffer[i]);
}

```

Conclusion

Interrupts can be a powerful mechanism for servicing data acquisition and control operations. With interrupts, the processor can respond quickly and efficiently to data acquisition hardware. Besides simply transferring acquired data, interrupts can be used to synchronize different events, or process data as it is acquired. However, interrupts are not always the best method for data acquisition. DMA is useful for moving blocks of data at high rates with minimal processor intervention. Polling is the most straightforward technique, but not very effective for multiple devices or high throughput rates.

The PC design uses the Intel 8259A programmable interrupt controller to provide several prioritized interrupts for the I/O bus. A data acquisition board plugged into the I/O bus can interrupt the processor by asserting the appropriate interrupt line. The processor then jumps to an interrupt service routine located at a preprogrammed address. The example program, `IRQESP.ASM`, included in this application note shows you how to program interrupts under DOS.

Alternatively, National Instruments has developed sophisticated, flexible driver software for PC/XT/AT, PCI, and PCMCIA data acquisition hardware. This software, NI-DAQ, includes high-level, easy-to-use functions that take advantage of the interrupt and DMA capabilities of the data acquisition hardware. Many applications developers who use driver-level software instead of low-level hardware programming realize substantial savings in time and effort.

Legal

This tutorial (this "tutorial") was developed by National Instruments ("NI"). Although technical support of this tutorial may be made available by National Instruments, the content in this tutorial may not be completely tested and verified, and NI does not guarantee its quality in any way or that NI will continue to support this content with each new revision of related products and drivers. THIS TUTORIAL IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND AND SUBJECT TO CERTAIN RESTRICTIONS AS MORE SPECIFICALLY SET FORTH IN NI.COM'S TERMS OF USE (<http://ni.com/legal/termsofuse/unitedstates/us/>).