1 Il software d'interruzione

La routine di risposta all'interruzione (ISR) è asincrona rispetto al programma in esecuzione, scatta quando ce n'è bisogno, non quando vuole il programmatore.
La CPU può essere interrotta mentre sta eseguendo un qualsiasi programma, in un suo qualsiasi punto, per cui la routine di risposta non può fare affidamento sul programma che sta interrompendo; deve essere completa ed autosufficiente e ripristinare registri e stack come erano al momento del suo intervento.
Per questo deve effettuare operazioni di salvataggio e ripristino di tutti i registri che la ISR utilizza. Queste operazioni vengono dette "**salvataggio del contesto**" (context saving).
Dunque occorre rimarcare che nel campo delle routine d'interruzione è obbligatorio lasciare le cose come le si erano trovate, pena il blocco dell'intero computer.

1.1 Problemi delle routine d'interruzione

Dati della ISR
Essendo necessario non modificare nulla dell'informazione che costituisce lo stato del programma che si è interrotto (cioè di ciò che viene detto il contesto del programma interrotto), la ISR dovrà salvare ogni cosa che essa utilizza (registri di dati e di segmento, stack, particolari locazioni di memoria che non vanno lasciate modificate, ..) provvedendo a ripristinare il contenuto iniziale prima di eseguire la IRET. Per la memorizzazione temporanea del contesto del programma interrotto e per eventuali dati locali, la ISR potrà utilizzare la memoria o, per i dati temporanei, anche lo stack.

Aree di memoria locali alla ISR
Un'area di memoria locale può far comodo ad una ISR, in qualche caso anche al posto dello stack, per la memorizzazione del contesto o per contenere dati che possano essere utilizzati da altri programmi.
La sua definizione è analoga a tutte le definizioni di aree di memoria in Assembly 8086. Si potranno definire diverse etichette, una per ogni "variabile" definita. Naturalmente il segmento dati definito deve essere considerato FAR. Ciò significa che si dovrà salvare DS del programma interrotto, scrivere in DS il valore del segmento dati locale e ripristinare il valore vecchio prima della IRET.
Un esempio, che utilizza l'extra segment:

```
LocalDATA SEGMENT
    Buffer DW 50 DUP (?) ;  buffer di appoggio: un vettore di word per la memorizzazione
dei dati letti dalla ISR
;  i dati verranno successivamente usati dai programmi che utilizzano questa ISR
    StringaDiAppoggio DB 30 DUP (?)   ; una stringa temporanea per l'uso interno da parte
della ISR
LocalDATA ENDS
..
ASSUME ES: LocalDATA ;  in questo esempio utilizziamo ES per la lettura e scrittura nel
segmento dati della ISR

ISR:
    ; salva l'Extra Segment del programma interrotto:
    PUSH ES
    PUSH AX
    ; fa puntare ES al nostro segmento dati locale:
    MOV AX, SEG LocalDATA
    MOV ES, AX

; .. da qui c'è il codice della ISR, che usa normalmente il segmento dati definito;
p.es.:
    MOV SI, OFFSET StringaDiAppoggio
    MOV AL, ES:[SI] ; prende il primo carattere della stringa temporanea (override con
ES)

; .. prima della fine della ISR:

     ; ripristina i registri usati
     POP AX
     ; ripristina l'Extra Segment del programma interrotto:
     POP ES
IRET
```

Un'area di memoria locale, per variabili che vanno cancellate al ritorno dalla ISR, si può ricavare anche dallo stack, spostando lo stack pointer verso gli indirizzi più bassi di tante locazioni quante servono per la memorizzazione tempora-

nea (tecnica dello "stack frame" già illustrata nel capitolo su C e Assembly") ed usando quelle locazioni dello stack così "allocate".

Salvataggio nello stack

Se i dati sono temporanei, possono essere cancellati prima della fine della ISR. In questo caso il salvataggio può avvenire anche nello stack. Bisogna fare attenzione a non usarne troppo, perché non si sa quale sia la situazione dell'area di stack che il programma interrotto sta usando e si potrebbe correre il rischio di riempirlo. Se una ISR ha bisogno di molto stack, deve definire una propria area di stack (e di conseguenza salvare SS e SP del programma che ha interrotto). Esempio: per definire una propria area di stack il programma che installa la ISR potrà fare così:

```
LocalSTACK SEGMENT
    EtichettaCheNonUsoMai DB 1024 DUP (?)     ; così alloca lo spazio in memoria per uno
stack di 1 kbyte
    InizioDelloStack LABEL WORD
   ; la direttiva LABEL WORD fa definire un'etichetta che verrà usata con un trasferimen-
to a 16 bit ma non fa allocare
   ; memoria
LocalSTACK ENDS

LocalData SEGMENT
    SSvecchioStack   DW  ?   ; deve contenere il puntatore al segmento del vecchio stack
    SPvecchioStack   DW  ?   ; deve contenere il puntatore all'offset del vecchio stack
LocalData ENDS

ASSUME SS: LocalSTACK ; ora il compilatore sa che l'area dati chiamata LocalStack è uno
stack

; .. poi, nel codice della ISR:
ISR:
    PUSH AX  ; salvo, nello stack del programma interrotto, AX, perché lo utilizzo subito
    MOV AX, SEG LocalData    ; parcheggio in AX perché DS non ammette la modifica in im-
mediato
    MOV DS, AX           ; carico DS con l'indirizzo di segmento dell'area dati locale
    MOV [SPvecchioStack], SP  ; memorizzo SP nell'area dati locale
    MOV [SSvecchioStack], SS  ; memorizzo SS nell'area dati locale
    ; ora sistemo SS e SP del mio nuovo stack:
    MOV AX, SEG LocalSTACK   ; parcheggio in AX perché SS non ammette la modifica in im-
mediato
    CLI              ; disabilito gli interrupt perché se un interrupt si infila fra le
due MOV seguenti
; trova lo stack inconsistente (mezzo vecchio e mezzo nuovo)
; (da notare che le nuove CPU rifiutano automaticamente l'interrupt dopo l'istruzione MOV
SS, AX, in questo caso la
;     CLI e la STI non sono indispensabili)
    MOV SS, AX
    MOV SP, OFFSET InizioDelloStack
    ; faccio puntare lo stack (SS:SP) all'indirizzo segmentato di valore più alto del mio
stack locale
    ; (ricordiamoci che lo stack aumenta calando di indirizzo)
    STI              ; ora posso anche essere interrotto

; .. qui c'è il codice della ISR, che usa liberamente lo stack, che è grande 1kbyte

; .. poi, prima della fine della ISR:
    ; ripristina, dalla memoria locale, il puntatore allo stack del programma interrotto:

    CLI
    MOV AX, [SSvecchioStack]
    MOV SS, AX
    MOV SP, [SPvecchioStack]
    STI
    ; ripristina AX del programma interrotto, che era nel suo stack (ora si punta di nuo-
vo a quello!):
    POP AX
IRET
```

Come si vede, il procedimento è abbastanza macchinoso e pericoloso. Se è possibile, è meglio limitare l'uso dello stack e usare quello del programma interrotto.

Passaggio di parametri ed accesso ai dati

Dato che una routine di servizio di un'interruzione hardware non viene chiamata esplicitamente da nessun programma, ma parte in qualsiasi momento sotto controllo dell'hardware, essa non può ricevere parametri né attraverso i registri, né attraverso lo stack (non si sa cosa c'è nei registri, né nello stack al momento del lancio di una ISR).

Locazioni fisse
E' facile passare parametri se si possono usare locazioni fisse in memoria, nelle quali sia la routine di risposta, sia chi utilizza il suo servizio, potranno leggere o scrivere.
Esempio:
Supponiamo che si debba passare un parametro attraverso la locazione di memoria FF123h (indirizzo scritto a 20 bit). Per raggiungerla, con indirizzo segmentato, bisogna stabilire un valore per la parte di segmento dell'indirizzo. Supponiamo di voler usare il valore FF12h come segmento. Allora, per raggiungere l'indirizzo dato, il valore dell'offset dovrà essere 0003h (si ricordi il meccanismo per la segmentazione, con l'aggiunta di quattro bit a zero al registro di segmento e la successiva addizione con l'offset).
Per la scrittura del valore la ISR, o il programma principale, potranno fare così:

```
; scrittura di ES:
MOV AX, FF12h
MOV ES, AX
MOV ES:[0003h], BX ; scrittura in memoria, se BX contiene il valore da salvare
```

Esempio

L'esempio più semplice che si possa immaginare è quello in cui il programma principale non fa altro che attendere che una ISR modifichi il valore di una variabile. Per quanto semplice questo programma può essere utile per verificare se l'hardware dell'interrupt funziona ed esso viene effettivamente lanciato.

```
INTnumber EQU 8  ; numero del vettore d'interruzione che interessa

UNICO SEGMENT
; definiamo un unico segmento, nel quale memorizziamo l'unico dato della ISR

ISReseguita DB 0 ; parametro della ISR, usato come un flag:
                 ; viene messo a 0 (TRUE) all'inizio del programma
                 ; viene messo a -1 (FALSE) dalla ISR
; segue il codice della ISR, che è conveniente mettere proprio qui:
EtichettaISR:
    ; la ISR trasforma la variabile ISReseguita da 0 a -1 (tutti 1),
    ; segnalando al programma principale che la procedura è stata eseguita

    ; cambio del parametro ("flag" ISReseguita):
    NOT CS:[ISReseguita]
    ; l'override con CS ci permette di interrompere qualsiasi programma
    ; che punti a qualsiasi registro

    IRET

Inizio:
; codice che viene fatto eseguire all'inizio del programma:
ASSUME CS:UNICO
    ; faccio puntare ES all'inizio della memoria:
    XOR AX, AX
    MOV ES, AX

    ; installazione del nuovo vettore all'etichetta della nostra ISR:
    CLI
    MOV word PTR ES:[INTnumber * 4], offset EtichettaISR
    MOV word PTR ES:[INTnumber * 4 + 2], seg EtichettaISR
    STI

    ; PROGRAMMA PRICIPALE
    MOV CS:[ISReseguita], 0 ; inizializzazione del flag
    ; attesa che la ISR sia lanciata: rimane in
    ; un loop fino a che la ISR non modifica ISReseguita
Attesa:
    CMP CS:[ISReseguita], 0
    JE Attesa
    ; !! se arriva qui l'ISR è scattata, altrimenti rimane in loop !!

    ; conclusione:
    MOV AH, 4Ch
    INT 21h
UNICO ends
END Inizio
```

Il programma mostrato omette per semplicità di comprensione alcuni dettagli importanti, come l'esecuzione dell'"end of interrupt" (EOI) all'interno della ISR ed il ripristino del vecchio vettore d'interruzione alla conclusione.

Questo significa che, eseguito così com'è scritto, provoca facilmente il blocco del computer (o del programma, se eseguito in Windows). Il codice del programma si può trovare nel file ISRminim.ASM, nel CD ROM allegato al testo.
Il file INTtest.ASM contiene una versione più "educata", che esce in modo corretto, ripristinando il vecchio vettore d'interruzione, ed emette il comando EOI.

Corse

Una condizione molto pericolosa che può svilupparsi quando si hanno variabili condivise che vengono modificate da più di un interrupt è la "corsa" o "corsa critica" ("**race** condition").

Si definisce "**corsa"** una condizione anomala che si manifesta a causa di una inattesa dipendenza di un programma dall'ordine e dagli istanti in cui accadono certi eventi.

Proprio per limitare il problema delle corse la CPU provvede ad azzerare il flag di interrupt quando entra nelle ISR.
Quando si verifica una corsa critica il valore delle variabili non è più "coerente". In questo caso si dice che la variabile non è "**consistente**", traducendo direttamente l'aggettivo inglese "consistent", che significa "regolare", "coerente" o "affidabile".
Vediamo con un esempio come si può sviluppare una corsa critica.
Supponiamo di avere un'area di memoria di 4 kByte destinata a fare da buffer per i dati letti da due convertitori A/D da 7 bit. Quest'area di memoria costituisce un "buffer circolare", nel quale vengono accumulati dati fino a che l'area di 4kByte non è piena, poi si ricomincia dall'inizio.
Quando la conversione in uno dei due A/D è terminata esso lancia un interrupt; i due convertitori sono collegati alla stessa linea di interrupt, perciò innescano la stessa ISR; per riconoscere quale dei due convertitori ha lanciato l'interrupt si usa il bit più significativo dello stesso byte in cui è presente la lettura del convertitore. Se quel bit vale 1 vuol dire che quel convertitore A /D in quel momento sta lanciando un'interruzione.
Supponiamo anche che la semplice lettura del port faccia "cancellare" il bit che segnala l'interruzione (riconoscimento dell'interrupt).

Analizziamo dunque la seguente ISR che, a causa dell'esecuzione di una procedura di lunga durata, richiede che il sistema delle interruzioni sia abilitato (istruzione STI, punto [0] del sorgente):

```
..
IsrA_D:          ; scatta al completamento di ogni conversione A/D
    ; salvataggio del contesto del programma interrotto:
    PUSH AX
    PUSH BX
    STI                  ; riabilito gli interrupt perché      [0]
                         ; devo eseguire una lunga procedura
    CALL UnaLungaProcedura   ; procedura che prende "molto tempo"

    IN AL, IndirizzoA_D1 ; lettura di A/D n.1
    ; controlla se è 1 che lancia l'interrupt:
    TEST AL, 10000000b
    JZ NonEuno
eUno:    ; interrupt lanciato da A/D n.1
    IN AL, IndirizzoA_D2 ; lettura di A/D n.2

NonEuno:     ; interrupt lanciato da A/D n.2

    AND AL, 011111111b   ; cancella il bit più significativo,
                         ; che segnala la fine conversione
    MOV BX, [ProssimoIN] ; legge l'indirizzo del prossimo      [1]
                         ; valore da scrivere nel buffer
    MOV [BX], AL         ; scrive nel buffer                   [2]
    INC [ProssimoIN]     ; fa avanzare il buffer per il prossimo valore
    AND [ProssimoIN], 0FFFh ; dopo 4kByte rimanda all'inizio del buffer
                         ; il puntatore ProssimoIN (offset 0)
                         ;                                     [3]
    ; (^ trucco documentato nel capitolo "Tecniche di programmazione" del
    ;    primo volume di questo libro, Capitolo "tecniche di programmazione",
    ;    paragrafo "Contatori a modulo")

    ; ripristino del contesto:
    POP BX
    POP AX
IRET
```

Il programma è piuttosto semplice. Ad ogni interrupt la ISR:
- legge il valore dell'A/D
- determina quale dei due A/D ha lanciato l'interruzione

- legge il puntatore nel buffer
- scrive il valore letto all'indirizzo del puntatore
- aggiorna il puntatore

Naturalmente un'altra parte del programma, che qui non interessa, "svuota" il buffer per analizzare i dati, modificando il puntatore UltimoOut.
Dato che la ISR include un'istruzione STI, un ADC può interrompere mentre è in corso la ISR che sta servendo l'altro ADC. Dunque una ISR può interrompere un'altra ISR.
Questa interruzione non causa nessun problema nella gestione dello stack e le due procedure vengono eseguite e ritornano regolarmente.
In verità se proviamo ad eseguire il programma così com'è esso sembra funzionare regolarmente, solo che, molto raramente, diciamo una volta su diecimila, un dato da un ADC viene misteriosamente perduto.
La ragione è dovuta all'innesco di una corsa, in quell'unico caso su diecimila. Ciò accade quando scatta un secondo interrupt mentre è in esecuzione l'istruzione indicata con [1] nel sorgente procedente.
Proviamo ad eseguire "a mano" la procedura, facendo le seguenti ipotesi esemplificative:
- ProssimoIn vale 700
- la ISR viene lanciata perché ADC 1 ha prodotto il valore 100
- la ISR di ADC 1 viene interrotta perché viene letto il valore 200 su ADC 2

Se l'ordine di esecuzione delle istruzioni più importanti è quello illustrato successivamente le ISR mettono valori diversi nella stessa posizione del buffer. Il primo dei due valori che viene scritto viene perciò sovrascritto, e viene perduto.
Supponiamo dunque che si susseguano queste istruzioni:

```
..
ISR dovuta ad ADC 1:

IsrA_D:
    PUSH AX
    PUSH BX
..
    IN AL, IndirizzoA_D1  in AL finisce il numero 100
..
    MOV BX, [ProssimoIN]  in BX finisce 700, indirizzo corrente del buffer

!! ORA SCATTA L'INTERRUPT DI ADC 2 !!
ISR dovuta ad ADC 2:
esegue la seconda ISR, "dentro" alla prima:

IsrA_D:
    PUSH AX
    PUSH BX                  ; IN BX SI SALVA 700!

..
Dopo le altre istruzioni della ISR interna si giunge a:

    IN AL, IndirizzoA_D2 ; in AL finisce il numero 200
..
    MOV BX, [ProssimoIN]  ; in BX finisce ANCORA 700, infatti la prima ISR
                          ; non ha aggiornato ProssimoIN perché è stata interrotta

    MOV [BX], AL         ; nel buffer viene scritto il numero 200
                         ; ma NELLA POSIZIONE CHE COMPETE AL VALORE DI A/D n.1

..
    si prosegue nella ISR "interna" fino a concluderla:

    POP BX               ; ripristina il numero 700, messo nello stack all'inizio
                         ; della ISR interna
    POP AX
IRET

    MOV [BX], AL         ; punto [2] della ISR esterna, in BX è appena
                         ; stato ripristinato il valore 700, per cui
                         ; questa MOV SOVRASCRIVE il valore precedente
..
IRET
```

La conseguenza di questo esempio è che in alcuni momenti "sfortunati" si può perdere un valore letto da uno dei due A/D; come avremo luogo di vedere meglio nella parte di questo volume relativa ai Sistemi Operativi, le conseguenze possono essere più gravi; in molti casi il programma si potrebbe bloccare in uno stallo (deadlock).

Una lezione che impariamo da questo problema è che la sussistenza di corse genera **errori intermittenti** ("intermittents") che non si presentano per la gran parte del tempo e colpiscono solo quando accadono alcune sequenze "sfortunate" di eventi.
Chiaramente, dato che non si può contare sulla fortuna, il programma funzionante non dovrà avere corse e perciò funzionerà sempre.
Per curare il problema, nel caso del nostro esempio, bastava impedire all'altro ADC di interrompere nel momento in cui si legge e/o si modifica la variabile condivisa.
Per far ciò bastava inserire un'istruzione CLI prima del punto [1]:

```
..
    CLI                  ; ora nessuno interrompe mentre leggo
                         ; e modifico ProssimoIN
    MOV BX, [ProssimoIN] ; legge l'indirizzo del prossimo      [1]
                         ; valore da scrivere nel buffer
    MOV [BX], AL         ; scrive nel buffer                   [2]
    INC [ProssimoIN]     ; fa avanzare il buffer per il prossimo valore
    AND [ProssimoIN], 0FFFh ; dopo 4kByte rimanda all'inizio del buffer
                         ; il puntatore ProssimoIN (offset 0)
                         ;                                     [3]
    STI[1]               ; ora possono anche interrompere, tanto
                         ; ProssimoIN è già a posto
..
```

1.1.1 Accesso ad altre routine

All'interno di una ISR l'uso delle procedure deve essere sottoposto a scrutinio molto accurato, perché esse possono causare gravi problemi, dovuti al fatto che l'avvio della ISR avviene sotto il controllo dell'hardware.

Riusabilità

Se da una ISR si vuole chiamare un'altra routine bisogna chiedersi: "cosa succede se la ISR ha interrotto proprio quella routine che vorrebbe chiamare?"
Consideriamo una qualsiasi routine che usi delle variabili globali, od esterne ad essa. Se essa le usa e le modifica senza riinizializzarle, ogni volta che viene eseguita vedrà un valore diverso nelle variabili globali, e lo cambierà ulteriormente.
Questo tipo di procedure sono soggette a problemi di corse quando vengono interrotte da altre procedure che accedono a quelle variabili globali. Per questa ragione vengono dette non riusabili.
Se una routine usa variabili globali, ma le inizializza sempre, ogni volta si riparte dallo stesso punto per cui se viene chiamata con gli stessi parametri avrà sempre lo stesso risultato. Una routine del genere viene detta riusabile in serie.

Rientranza

Se una routine usa solo variabili che definisce ed inizializza localmente (e, di solito, scarta alla fine, per risparmiare memoria), essa è tale che ogni chiamata alla routine genera ed usa i propri dati, non modificando dati di altri. Routine come questa vengono dette rientranti ("reentrant").
Routine rientranti possono essere interrotte in ogni punto, anche da sé stesse, e dare comunque un risultato corretto.
Siccome una ISR potrebbe avere interrotto l'esecuzione di una routine qualsiasi, e non ha modo di sapere quale, essa non può chiamare un'altra routine se essa non è rientrante.
Se una routine ne chiama un'altra non rientrante non è rientrante. Nella realizzazione di procedure rientranti si deve considerare che i linguaggi ad alto livello fanno spesso chiamate a librerie "nascoste", che quasi sempre non sono rientranti.
Le routine del BIOS e dell'MS-DOS non sono rientranti. Molte delle funzioni di libreria dei linguaggi ad alto livello non sono rientranti (per usarle in ISR è necessario verificare nella documentazione del linguaggio).
Le procedure del nucleo (kernel) dei Sistemi Operativi multiprogrammati (vedi i capitoli sui Sistemi Operativi) devono essere rientranti.

Possibilità di interruzione da parte di altre routine

Per limitare il problema delle corse sarebbe ideale eseguire tutte le ISR con interrupt disabilitati, cosa che d'altronde viene incentivata dalla CPU stessa, che disabilita gli interrupt mentre salta alla ISR.
Peraltro potrebbe essere necessario scrivere ISR che durino per troppo tempo. In questo caso bisogna eseguire una STI e riabilitare le interruzioni. Questo significa che la nostra ISR potrebbe essere interrotta da altre routine d'interruzione.
Considerare questa evenienza significa proteggere dagli altri interrupt tutte le locazioni di memoria che ci servono nella ISR e che gli altri interrupt potrebbero usare. Ciò si ottiene disabilitando le interruzioni tutte le volte che si accede a locazioni di memoria che possono servire ad altre routine d'interruzione, e ripristinandole subito dopo. (CLI, poi STI). Inoltre potrebbe darsi di dover scrivere la nostra routine in modo che sia rientrante, perché, se scatta ancora l'interrupt che stiamo servendo, la nostra routine verrà interrotta da sé stessa!

Frequenza degli interrupt

Supponiamo che una ISR non faccia in tempo a concludersi prima dell'interrupt successivo. In questo caso si corre incontro a due pericoli.
- se durante la ISR gli interrupt sono disattivati la risposta al secondo interrupt potrebbe giungere troppo tardi (alla fine della prima ISR)
- se invece gli interrupt sono abilitati e la ISR interrotta è rientrante non dovrebbero esserci problemi.

[1] Quest'ultima STI non era indispensabile in questo caso specifico, dato che ci pensa la IRET che segue subito dopo.

Questo è vero solo se la situazione descritta non si presenta regolarmente, perché se giunge un flusso costante di interrupt che non lascia mai concludere le ISR lo stack si può riempire molto rapidamente e se si esaurisce .. (BUM!).
Ecco dunque un'altra ragione per scrivere ISR brevi.

1.2 Interrupt in MS-DOS

Passaggio dei parametri

Se i dati devono essere passati attraverso locazioni non fisse in memoria, si può fare una distinzione fra programmi ".COM " e ".EXE".

Programmi .COM
Se il programma che usa, installa e contiene la ISR è di tipo .COM, tutti i segmenti coincidono, così si può accedere a variabili dell'applicativo semplicemente utilizzando CS come registro di segmento ed usando l'override (CS contiene il giusto valore del segmento, perché esso è nella vector table e viene settato al momento del salto FAR alla ISR).
Esempio:

```
MOV CS:[DatoDellaISR], AX     ; scrive nel segmento giusto perché i valori di CS e DS,
                          ; in un .COM, coincidono.
```

In questo modo non è necessario modificare DS e perciò neppure salvarlo e ripristinarlo.

Programmi .EXE
Qualora il file sia .EXE, è necessario che la ISR conosca il segmento dell'indirizzo al quale lavorare. La ISR dovrà perciò richiedere al DOS un "data segment fix-up", ottenendo un valore da mettere in DS, dopo averne salvato il valore vecchio.
Esempio:

```
ISR:
    MOV AX, SEG SegmentoDatiDellaISR  ; data segment fix-up
    MOV DS, AX
```

Naturalmente il vecchio valore di DS va conservato e ripristinato, come per ES nell'esempio precedente.
In alternativa si può usare la tecnica già vista nel primo esempio completo (INTtest.ASM) e "nascondere" i dati nel segmento di codice della ISR, usando poi l'override con CS per accedere al segmento.

Uso del controllore d'interruzione

Come già detto nel capitolo precedente, non è il caso di cambiare le modalità di funzionamento dei PIC (8259) del PC, né il numero di vettore che essi mettono sul data bus, quando sollecitati dai dispositivi (08h).
Le cose che si possono (o si debbono) fare in modo abbastanza sicuro sono: spedizione del comando di fine dell'interruzione e mascheramento di singole sorgenti di interruzione a livello del PIC, esempi su come farlo sono stati già presentati nel precedente capitolo.

Installazione e rimozione di ISR dalla tabella dei vettori

Nell'esempio precedente abbiamo visto l'installazione di una nuova ISR scrivendo direttamente nella tabella del vettori d'interruzione.

```
    ; installazione del nuovo vettore all'etichetta della nostra ISR:
    CLI
    MOV word PTR ES:[INTnumber * 4], offset EtichettaISR
    MOV word PTR ES:[INTnumber * 4 + 2], seg EtichettaISR
    STI
```

Si noti che gli interrupt devono essere disabilitati perché se l'interrupt "colpisce" fra un'istruzione e l'altra il salto usa l'offset della nuova ISR ed il segmento della vecchia!

Ogni programma che installi una sua ISR deve ripristinare, prima di terminare, quella che era presente precedentemente nella tabella dei vettori.
Infatti quando il programma finisce la sua memoria viene liberata e successivamente riutilizzata. Lasciando che il vettore d'interruzione continui a puntare al programma terminato ci si espone ad un salto "nel vuoto" quando il codice del nostro programma sarà sovrascritto da uno nuovo.
Per questo quando un programma che usa una nuova ISR fa modifiche alla tabella dei vettori d'interruzione, deve leggerne prima il contenuto, memorizzare da qualche parte il valore del vecchio vettore e, quando finisce, prima di tornare al DOS, rimettere nella Interrupt Table il valore vecchio, che era stato conservato.
Vediamo un esempio. Per "ricordarsi" del puntatore alla vecchia ISR, utilizziamo quattro byte di memoria:

```
    OffsetVecchioCodice DW ?
    SegmentoVecchioCodice DW ?
```

poi, nel programma principale, dobbiamo memorizzare il vecchio vettore:

```
    ; memorizzazione del vecchio vettore d'interruzione:
```

```
    MOV AX, ES:[INTnumber * 4]    ; lettura del vecchio offset dalla tabella
    MOV CS:[OffsetVecchioCodice], AX  ; memorizzazione temporanea
    MOV AX, ES:[INTnumber * 4 + 2] ; lettura del vecchio segmento dalla tabella
    MOV CS:[SegmentoVecchioCodice], AX  ; memorizzazione temporanea
    ; !! in ES ci deve essere 0 !!
```

Successivamente installiamo la nuova ISR la usiamo, eseguendo il programma principale.
Infine, prima della conclusione del programma, rimettiamo a posto la vecchia ISR:

```
; ripristino il vecchio vettore d'interruzione:
    MOV AX, CS:[OffsetVecchioCodice]
    CLI
    MOV ES:[INTnumber * 4], AX
    MOV AX, CS:[SegmentoVecchioCodice]
    MOV ES:[INTnumber * 4 + 2], AX
    STI
```

Per l'installazione di servizi nella tabella dei vettori MSDOS mette a disposizione un servizio specifico, che fa le cose appena indicate ed è più, specialmente se il computer è in rete o si usa Windows. Infatti in questo caso il sistema operativo avrà la notifica che la tabella dei vettori è stata modificata e potrà riportarlo nelle sue tabelle.
Il servizio predisposto allo scopo e il numero 25h dell'interrupt software 21h e serve per scrivere un vettore d'interruzione al posto giusto della tabella dei vettori, il servizio 35h invece legge la tabella dei vettori.
Questi servizi funzionano così:
Lettura del vettore:

```
AH = 35h     Servizio "get interrupt vector"
AL = numero del vettore da leggere (0-255)
ES = segmento dell'indirizzo della routine di risposta
BX = offset dell'indirizzo della routine di risposta
Chiamata del servizio: INT 21h
```

Scrittura del vettore d'interruzione:

```
AH = 25h     Servizio "set interrupt vector"
AL = numero del vettore da modificare (0-255)
DS = segmento dell'indirizzo della routine di risposta
DX = offset dell'indirizzo della routine di risposta
Chiamata del servizio: INT 21h
```

Questi servizi provvedono a disabilitare le interruzioni prima di fare le modifiche e a riabilitarle quando ciò è sicuro.

Un problema potrebbe insorgere se il programma finisce senza passare dalla parte di codice che fa l'operazione di ripristino della vecchia ISR. Questo potrebbe accedere, per esempio, se c'è un'eccezione dovuto ad una divisione per zero, che fa finire il programma per una via non prevista.
Se ciò accade la vecchia ISR non viene ripristinata ed il sistema si blocca non appena carichiamo un nuovo programma.
I linguaggi ad alto livello mettono a disposizione librerie per la gestione delle eccezioni che si prendono carico di questo problema.

1.2.1 TSR d'interruzione

In alcuni casi è utile installare routine d'interruzione che continuino a funzionare anche quando il programma che le ha installate finisce.
I programmi che terminano non rilasciando al DOS, tutta o in parte, la memoria che usano, si chiamano "Terminate but Stay Resident" (**TSR**).
Essi, per continuare a funzionare anche dopo che il programma che li ha installati è finito, devono necessariamente essere agganciati a qualche vettore di interruzione, che li fa funzionare come ISR di servizi hardware o software (p.es. il programma TSR potrebbe intercettare la chiamata al DOS per la scrittura su video (interrupt software) o la pressione di particolari combinazioni di tasti (interrupt hardware)).
Un programma Assembly scritto per essere TSR si deve dividere due parti: una parte, quella ad indirizzi più bassi, è destinata a rimanere in memoria anche dopo che il programma è finito
La parte rimanente invece verrà liberata dalla memoria al momento della fine del programma; al momento dell'esecuzione da parte dell'utente di un programma successivo, la parte non residente verrà sovrascritta, mentre la parte residente rimarrà intoccata.
Per installare un TSR, alla fine dell'esecuzione del main, invece del servizio AH = 4Ch dell'interrupt 21h, si usa il servizio AH=31h.
Servizio "terminate and stay resident":

```
AH = 31h     Servizio "terminate and stay resident"
```

```
AL = codice di ritorno da parte del servizio
DX = numero dei paragrafi del programma che devono rimanere residenti (un paragrafo = 16
Byte)
Chiamata del servizio: INT 21h
```

Questo servizio può installare come TSR programmi .COM e .EXE.
Prima di chiamare INT 27h, si dovrà copiare in DX 1/16 della dimensione della parte del programma che si vuole lasciare in memoria.
Di conseguenza tutte le locazioni dall'inizio del programma a quella precedente alla dimensione passata in DX rimarranno residenti in memoria.
Si farà in modo che in esse stiano la routine che vogliamo installare come TSR ed i suoi dati locali.
Il programma principale, che verrà eseguito una sola volta e che installerà la ISR nel giusto vettore d'interruzione, potrà di solito essere cancellato dopo che è stato eseguito.
Esempio:

```
; Uscita come TSR con servizio DOS AH = 31h
    ; ciò che mi interessa lasciare in memoria inizia da offset 0, per
    ; cui l'offset dell'etichetta dalla quale voglio "tagliare" è il numero
    ; di Byte che vanno lasciati in memoria.
    ; ne calcolo 1/16 per avere il numero dei paragrafi, poi aggiungo 1 per
    ; averne almeno 1:
    MOV DX, offset TaglioDaQui
    SHR DX, 4
    INC DX
    MOV AH, 31h [2]
    INT 21h      ; chiamata al DOS per TSR
```

L'installazione di una ISR residente toglie dal vettore d'interruzione la vecchia ISR. Siccome alla fine del programma la nuova ISR deve rimanere residente, non si deve più ripristinare il contenuto del vecchio vettore d'interruzione.
E' quindi buona norma fare in modo che la vecchia routine d'interruzione venga ancora eseguita, chiamandola da qualche punto della nostra nuova ISR e facendo uso, per il salto, dell'indirizzo del vecchio vettore d'interruzione. Questo indirizzo si può acquisire con le stesse modalità già viste e scrivere nella memoria locale della ISR.
Le versioni più moderne di Windows fanno automaticamente qualcosa di analogo quando si lancia un qualsiasi programma DOS che modifica la tabella dei vettori d'interruzione. In questo modo il programma DOS può funzionare regolarmente pur senza costituire un grosso pericolo per gli altri programmi e per il S.O. stesso.
Per verificare questo fatto si può eseguire questo esperimento:
1. far partire il computer in modalità MSDOS o con un dischetto DOS
2. eseguire il programma TSRinstl, descritto nel paragrafo successivo e presente sul CDROM allegato. TSRinstl installa una ISR sull'interrupt del system clock e la lascia TSR, per cui l'interrupt del clock DOS rimane "staccato"
3. eseguire il comando TIME alcune volte: come ci si poteva aspettare il tempo è "congelato", dato che l'interrupt DOS del clock non arriva più
4. far partire Windows e ripetere le stesse operazioni, installando TSRinstl. Ora l'ora di TIME va avanti, confermandoci il fatto che la richiesta di modifica della tabella dei vettori viene intercettata da Windows che fa in modo che possano eseguire sia la sua ISR, sia quella del programma DOS.

Una TSR di esempio
I programmi TSRinstl.ASM, TSRusa.ASM e TSRtogli.ASM, presenti nel CDROM allegato, permettono di provare una TSR molto semplice.
Il programma TSRinstl installa come TSR una ISR che risponde all'INT8, quello del realtime clock che viene lanciato 18,2 volte/s. Questa ISR non fa altro che contare in una locazione di memoria al suo interno il numero di volte che la ISR viene eseguita.
Il programma TSRusa va a leggere il conteggio nella memoria della ISR e lo visualizza.
Il programma TSRtogli toglie la nostra TSR dalla tabella dei vettori, rimettendo il puntatore alla procedura che c'era prima.
La Figura 1 mostra il funzionamento delle tre applicazioni. Se si lancia TSRusa prima di aver installato la TSR il risultato è un numero, prelevato dalla memoria da una locazione "sbagliata". Quel valore rimane costante.
Con l'esecuzione di TSRinstl il contatore viene aggiornato ad ogni ISR sull'IRQ0, per cui aumenta di 18,2 numeri al secondo. Se si esegue ora TSRusa esso visualizza un valore crescente con il tempo trascorso.
Se si toglie la TSR dal vettori d'interruzione di INT8 con il programma TSRtogli la lettura produce lo stesso numero fisso di prima, perché la nostra ISR non viene più eseguita (ed anche perché il programma TSRusa legge alla stessa locazione "sbagliata" di prima).
<FILE>
usoTSR.BMP
</FILE>

Figura 1: uso della TSR di esempio

[2] Attenzione: per i programmi TSR piccoli può essere necessario passare in DX numeri più grandi di quelli ottenuti con questo procedimento!

Analizziamo ora parte dei tre programmi.
Il programma di installazione scrive nella tabella dei vettori l'indirizzo della nuova ISR, solo dopo aver salvato l'indirizzo della vecchia in un'area di memoria di 8 Byte, che rimarrà anch'essa TSR.
L'area di memoria in questione si definisce così:

```
NumeroISR DW 0               ; questa locazione ha offset 0 nel segmento
OffsetVecchioCodice DW ?     ; questa locazione ha offset 2
SegmentoVecchioCodice DW ?   ; questa locazione ha offset 4

EtichettaISR:
```

Come si può notare dall'etichetta di inizio della ISR, quest'area di memoria risiede immediatamente prima dell'inizio del codice della ISR. Quest'area di memoria contiene il contatore che verrà aggiornato dalla ISR e l'indirizzo segmentato del vecchio vettore di interruzione.
Il salvataggio del puntatore alla vecchia ISR viene fatto così:

```
    ; lettura con servizio DOS dell'indirizzo segmentato dalla tabella vettori
    ; AH = 35h: servizio 35h (get interrupt vector)
    ; AL = 08h: richiesta dell'indirizzo del vettore 08h
    MOV AX, 3508h
    INT 21h      ; chiamata al DOS
    MOV CS:[OffsetVecchioCodice], BX  ; il servizio dà l'offest in BX
    MOV CS:[SegmentoVecchioCodice], ES    ; e il segmento in ES
```

Poi è necessario scrivere nella tabella dei vettori l'indirizzo della nuova ISR, così:

```
    ; installazione del nuovo vettore all'etichetta della nostra ISR:
    ; scrittura con servizio DOS dell'indirizzo segmentato nella tabella vettori
    MOV AX, SEG EtichettaISR
    MOV DS, AX   ; in DS il servizio 25h vuole il segmento da installare
    MOV DX, offset EtichettaISR ; in DX l'offset
    ; AH = 25h: servizio 25h (set interrupt vector)
    ; AL = 08h: richiesta di installare nel vettore 08h
    MOV AX, 2508h
    INT 21h      ; chiamata al DOS
```

Infine il programma principale si conclude usando il servizio 31h per rimanere TSR.

```
; Uscita come TSR con servizio DOS AH = 31h
    ; in DX ci andrebbe il numero di paragrafi che si vuole lasciare
    ; ma ha funzionato solo con numeri più grandi:
    ; il numero giusto sarebbe (offset TaglioDaQui / 16) + 1
    ; ma non ha funzionato!!
    MOV DX, offset TaglioDaQui
    ; SHR DX, 4
    ; INC DX
    MOV AH, 31h
    INT 21h      ; chiamata al DOS per TSR
```

La ISR è molto semplice e dovrebbe bastare il suo stesso codice per descriverla:

```
EtichettaISR:
    ; la ISR incrementa NumeroISR, contando il numero di volte che la ISR
    ; è partita
    ; salvataggio del contesto:
    PUSH AX

    ; corpo della ISR:
    INC CS:[NumeroISR]

    ; end of interrupt:
    MOV AL, 20h
    OUT 20h, AL

    ; ripristino del contesto:
    POP AX
    IRET
```

Quando viene installato TSRinstl il contatore NumeroISR viene creato ed inizializzato e comincia subito ad essere incrementato per 8,2 volte/s.

Il programma TSRusa visualizza il conteggio che la ISR sta incrementando, andando a leggere nel segmento della ISR. La parte più importante di questo programma è la seguente:

```
; prepara l'indirizzo di segmento in cui andare a leggere il valore
; del contatore, il segmento lo legge dalla tabella dei vettori:
MOV AX, 0
MOV ES, AX
MOV AX, ES:[8 * 4 + 2]    ; lettura del segmento dalla tabella vettori
MOV ES, AX   ; ora ES punta al segmento (unico) della TSR

; lettura della variabile della TSR:
MOV AX, ES:[NumeroISR]
```

Nel codice soprastante NumeroISR è l'offset del contatore, che abbiamo visto essere 0 nel programma TSRinstl.

Il programma TSRtogli sostituisce nella tabella dei vettori la nostra ISR con quella che c'era prima. In modo analogo al programma precedente si procura, in ES, il puntatore al segmento della nostra ISR e lì trova l'indirizzo della vecchia, ripristinandola nella tabella dei vettori.

```
MOV AX, ES:[SegmentoVecchioCodice]
MOV DS, AX   ; il servizio 25h vuole in DS il segmento da installare
MOV DX, ES:[OffsetVecchioCodice] ; in DX l'offset

; installazione del vecchio vettore con servizio DOS
; AH = 25h: servizio 25h (set interrupt vector)
; AL = 08h: richiesta di installare nel vettore 08h
MOV AX, 2508h
INT 21h      ; chiamata al DOS
```

A questo punto la nostra ISR è "sganciata" dalla tabella dei vettori e non progredisce più nel conteggio.