# Esercitazioni PIC Midrange.

## Inserto_d: Introduzione all'interrupt

Alcune considerazioni sugli interrupt nei PIC, principalmente quelli a 8 bit.

---

Con i Midrange inizia, nei PIC a 8 bit, ad essere presente la funzione di interrupt.

Un interrupt è l'azione conseguente ad una chiamata proveniente da una periferica, la quale interrompe il flusso principale delle istruzioni e devia l'esecuzione al contenuto di una altra area di memoria programma.

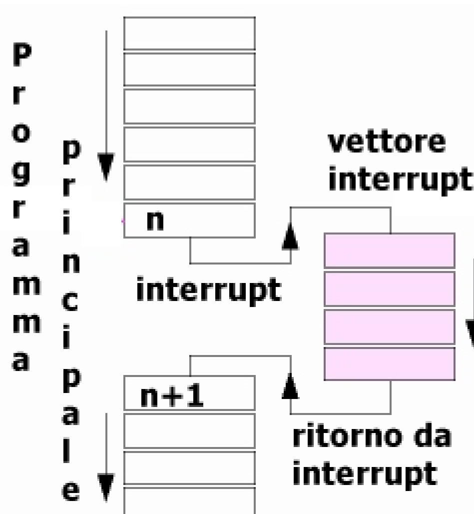
---

## Interrompere.

E' possibile che per una qualunque attività siano necessari eventi che richiedono una attenzione immediata.
Facendo un esempio: stiamo leggendo un libro e siamo impegnati nello scorrere le frasi e a comprenderne il significato (questo è il flusso principale del "programma"). Ma ecco che improvvisamente squilla il telefono: occorre interrompere (*interrupt*) la lettura e rispondere alla chiamata. Dopo di che si potrà nuovamente riprendere dal punto in cui si è stati interrotti.

Identicamente all' esempio, il programma sta eseguendo una procedura quando arriva un dato da un ingresso seriale. Occorre analizzarlo ed agire di conseguenza. Il flusso principale viene interrotto, si attiva la gestione dell'informazione acquisita, quindi si riprende l'attività principale.

Una richiesta di interruzione viene generata quando un dispositivo esterno all'unità logica centrale e per questo denominato "periferica", invia un segnale opportuno. Il programma risponderà terminando l'istruzione corrente in esecuzione e saltando (*vectoring*) alla routine di servizio di interrupt (**ISR** - *Interrupt Service Routine*).




In effetti, nei PIC come in altri processori embedded, la procedura di interrupt consiste essenzialmente nella **sostituzione del valore contenuto nel Program Counter** con un indirizzo fisso, predeterminato dalla struttura del componente.
Questo fa si che l' istruzione successiva all' evento di interrupt sia la prima che si trova a questo indirizzo, che è chiamato **vettore di interrupt** (*interrupt vector*).

In sostanza, da una qualsiasi posizione attuale nella memoria programma, si sospende il flusso di istruzioni e si passa ad eseguire le istruzioni ad un indirizzo determinato, analogamente a quello che succede per una subroutine.

Il termine dell' esecuzione di quanto contenuto nelle istruzioni di gestione dell'interruzione viene determinato da una speciale istruzione di ritorno-da-interrupt (**RETFIE** - *RETurn From IntErrupt*) **ripristina il Program Counter** nella posizione da cui si era distaccato, riprendendo il flusso di istruzioni principale, analogamente al RETURN delle subroutine.

Quindi, apparentemente, è una operazione del tutto analoga a quella della chiamata di una subroutine, ma in pratica esistono alcune fondamentali differenze:

| | Subroutine | Interrupt |
|---|---|---|
| **Azione** | Devia l'esecuzione ad un indirizzo contenuto nella chiamata, caricandolo nel PC | Devia l'esecuzione ad un indirizzo fisso (vettore di interrupt), caricandolo nel PC |
| **Avvio** | Inizia in seguito all'esecuzione dell'istruzione **call** | Inizia in seguito ad un evento in una periferica |
| **Stack** | L'indirizzo di ritorno è caricato nello stack | L'indirizzo di ritorno è caricato nello stack |
| **Ritorno** | E' comandato dall'istruzione **return** o **retlw** | E' comandato dall'istruzione **retfie** |

La prima differenza essenziale questa:

- la subroutine è **chiamata volontariamente** durante l' esecuzione del programma da una istruzione call
- l' interrupt è **chiamato in modo automatico da un evento hardware**, esterno al programma, e che può capitare in qualsiasi momento dell' esecuzione del programma stesso

Se vogliamo proseguire nell' esempio iniziale, la subroutine è analoga al fatto che sto leggendo, ma sono le cinque e a quell' ora ho l' abitudine di bere un tè, per cui sospendo volontariamente la lettura, vado in cucina, preparo il tè. Poi ritorno a riprendere la lettura dove l' avevo sospesa. L' operazione di fare il tè è la stessa tutti i giorni, per cui si tratta di una funzione ripetuta costantemente.
Nel caso dell' interrupt, invece, mentre sto leggendo, o anche facendo il tè o qualsiasi altra cosa, ecco che suona il telefono: devo sospendere quello che sto facendo per rispondere ad un evento che era dato per possibile, ma non programmato per quel determinato momento, potendo accadere in un qualsiasi istante della giornata.

Il secondo punto riguarda le routines di gestione.

- La subroutine, richiamata dall'istruzione call, è usata per eseguire una certa funzione quando la logica del programma ne ha necessità. Ad esempio, potrà trattarsi della conversione di un valore da esadecimale a BCD oppure di un ritardo. Il programmatore, a quel punto dell' esecuzione, richiama la funzione che gli serve, passando eventuali parametri e raccogliendo in uscita i risultati voluti.
- Anche la richiesta di intervento dovuta all' interrupt va prevista e "servita" in modo adeguato, in quanto anche all' evento che ha generato la richiesta di interrupt è indispensabile fornire una giusta gestione. Con un esempio: se suona l'allarme anti incendio, ma non avete previsto questa possibilità e non sapete cosa fare, potete rimetterci la vita.

> Semplicemente, questo evento, previsto e fornito dell'adeguato insieme di istruzioni per gestirlo, accade "casualmente", in modo asincrono rispetto al flusso principale delle istruzioni.

Dunque, **sia alla call che all' interrupt devono corrispondere delle sezioni di programma che svolgono le funzioni richieste dagli eventi**.

Da questo punto di vista, chiamata di subroutine e chiamata di interrupt non hanno alcuna differenza: nello svolgersi del programma accade un qualche fatto e di conseguenza occorre che il programma lo gestisca.
Anche se sia nella chiamata a subroutine che nella chiamata a interrupt i meccansimi della CPU **sostituiscono il contenuto del Program Counter con un indirizzo preciso, lo fanno in modo sensibilmente differente:**

- **la subroutine inserisce nel PC l' indirizzo che è l'oggetto del call**, indirizzo che può essere in una qualunque posizione ammessa della memoria programma. Il compilatore assegnerà il giusto valore.

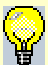
- **l' interrupt**, invece, **forza nel PC un indirizzo fisso**, **prestabilito dalla struttura del microcontroller**, qualunque sia la causa della chiamata. A questo indirizzo fisso si dovrà trovare la routine di controllo dell'evento.

Una differenza sensibile:

- si **possono creare label a volontà** e scrivere blocchi di codice corrispondenti per eseguire ogni genere di funzione, disposti come meglio si desidera nella memoria programma; al momento del call, il PC assumerà il valore dell' indirizzo della label oggetto, quella tra le tante che è indicata nel sorgente.

- Per l' interrupt, invece, solitamente **esiste un unico indirizzo fisso** a cui il PC è deviato a seguito dell' interrupt. Questo vuol dire che a quell' indirizzo occorrerà scrivere un blocco logico in grado di fare fronte a tutte le necessità richieste dall' interrupt, necessità che diventano più complesse se si ha a che fare con sorgenti di interrupt multiple.
  Ad esempio, si deve gestire una comunicazione seriale assieme ad un display multiplexato e una pulsantiera; le diverse periferiche generano ognuna una richiesta di interruzione. Dato che, qualsiasi sia la fonte, il salto con l'interrupt invia ad una sola locazione, ecco che qui sarà necessaria non solo la gestione degli eventi, ma, prima di questo, la discriminazione necessaria per individuare quale periferica debba essere servita.

Dal punto di vista pratico, per i PIC Midrange questo indirizzo (vettore di interrupt) è posto a 004h (per i PIC18F ne esiste un secondo a 0008h; per i PIC superiori sono presenti molti vettori di interrupt).

 In pratica: **scrivere una gestione dell' interrupt equivale a scrivere una subroutine che inizia al vettore di interrupt e viene terminata con l' istruzione retfie.**

Nell'istruzione di ritorno c'è, infatti, una ulteriore differenza tra subroutine e interrupt :

- **la subroutine rientra con un return o retlw**

- **l' interrupt rientra con una istruzione di return chiamata retfie**.

Entrambe le istruzioni modificano il PC prelevando dallo stack l' indirizzo di rientro, ma:

- **return** o **retlw** eseguono solo questa funzione
- **retfie** abilita anche il meccanismo di interrupt che era stato disabilitato all' ingresso nella gestione della chiamata di interrupt.

Questa azione di **retfie** è necessaria in quanto, nei Midrange, che dispongono di un solo livello di interruzione, in condizioni normali, **una chiamata di interrupt non deve essere interrotta da un'altra chiamata interrupt** che avvenga nello stesso momento: solo quando la prima è stata servita, se ne potrà eseguire un'altra.
Questo è ottenuto disabilitando il bit **`GIE`** all'ingresso del vettore di interrupt e ri abilitandolo in uscita all'esecuzione di retfie. Quindi, nella routine di interruzione non occorre aggiungere istruzioni per disabilitare **`GIE`**, ne tanto meno riabilitarlo in uscita.

Da un punto di vista generale, quindi, le due istruzioni sarebbero analoghe per quanto riguarda il PC, ma da un punto di vista pratico **non è per niente opportuno** utilizzare un retfie a chiusura di una subroutine, dato che l' abilitazione degli interrupt potrebbe non essere un evento previsto e quindi causa di grave blocco del programma.
Nè è possibile utilizzare return per l' uscita dalla gestione dell' interrupt, dato che questa istruzione non riabilita il meccanismo di interrupt, che resterebbe inattivo, impedendo che le successive chiamate possano essere accolte.

Per inciso, questa è la struttura base dei microcontroller più semplici. E' evidente che si tratta di una gestione che può presentare limiti sensibili nel caso in cui il servire uno o l'altro evento di interruzione non abbia la stessa priorità. Ad esempio, la gestione di una ram dinamica ha la precedenza sulla elaborazione del dato seriale ricevuto, in quanto, mancando il refresh con la giusta cadenza temporale, il contenuto della memoria andrebbe perso.

Per microcontroller più complessi, esistono metodi per risolvere le varie situazioni. Il più semplice è quello impiegato nei PIC18F, dove esistono due livelli di priorità programmabili per ogni periferica: una interruzione a livello basso potrà essere "interrotta" da una richiesta proveniente da una sorgente di livello alto, ma non l'opposto.
Altri chip superiori dispongono di vettori a più livelli.

---

# Ma quale è il senso dell' interrupt e la sua necessità?

Come abbiamo visto nell' esempi della telefonata, il microcontroller esegue un certo flusso di istruzioni e la sua attività può interessare un elevato numero di periferiche; queste possono avere tempi di risposta propri e che possono essere del tutto asincroni rispetto all' attività principale. Basta pensare alla differenza tra l'overflow del Timer0, alimentato dal clock del processore e l'evento della pressione di un pulsante: il primo è determinabile, visto che avverrà dopo un certo numero di cicli istruzione; il secondo è del tutto casuale.

Nell' esempio, so che il timeout avverrà ogni 100ms; ho il tempo tra un evento e il successivo per effettuare altre operazioni, per poi andare a verificare l'overflow con un polling. Però, non posso sapere quando l'utente premerà un tasto e non è evidentemente possibile rimanere in attesa dell'evento con un polling, che richiederebbe tutto il tempo disponibile della CPU.

La chiamata dell' interrupt interrompe ciò che il microcontroller sta facendo per andare a rispondere alle necessità della periferica che ha inviato la richiesta, in qualsiasi momento questo avvenga.

Quindi, posso essere in polling nell' attesa del pulsante, ma non perderò l'overflow , perchè questo interromperà il polling; o viceversa. Oppure anche entrambi: mandiamo il processore in sleep, a basso consumo, e lo facciamo risvegliare a seguito dei due eventi. Abbiamo una risposta immediata

pur avendo il minimo consumo energetico.

L'interrupt garantisce diverse cose:

- finchè le periferiche non hanno alcuna attività, il microcontroller può svolgere altre azioni e non deve necessariamente restare bloccato in loop di polling per verificare la situazione. Per tornare all'esempio fatto all'inizio, non occorre stare davanti al telefono in attesa della chiamata, ma è possibile fare altro.

- la risposta è, comunque, rapida. Allo scatto della chiamata di interrupt, solamente pochi cicli istruzione sono necessari per deviare l'esecuzione alle istruzioni di gestione dell'evento.

- Il fatto che la richiesta di una determinata funzione sia "automatizzato" consente al programma di effettuare in modo altrettanto automatico ed efficiente rapide azioni di intervento a supporto delle periferiche, dato che il programmatore avrà predisposto quanto necessario nel software.

- non si rischia di perdere eventi. Essi sono segnalati da opportuni flag che, fin a quando non sono azzerati dal programma, continuano a segnale l'evento e a richiedere un intervento in interrupt.

Queste sono le chiavi per la gestione di diverse azioni (multi task) contemporanee ed è una funzione essenziale dei microcontroller.
Senza interrupt, gran parte delle applicazioni con più task sarebbe irrealizzabile. E' abbastanza raro che un sistema embedded non faccia uso di almeno un interrupt al di fuori di applicazioni banali.

**Interrupt è una delle caratteristiche più potenti e utili disponibili nei sistemi embedded**, rendendo il sistema più efficiente e più rispondente ad eventi critici e il software più facile da scrivere e capire.

Tuttavia, dove non ne venga compresa la struttura, gli interrupt possono essere fonte di confusione e di errori nel programma; alcune persone li evitano per questo motivo, anche se ogni programmatore che non sia un principiante confusionario dovrebbe essere di casa con le interruzioni, utilizzandole come strumenti essenziali e non come cose oscure da evitare.

---

# Interruptfobia.

Per quanto detto, vogliamo qui fare una digressione sul fatto che l'interrupt sia, per molti di quelli che si avvicinano ai microprocessori, una vera bestia nera. C'è una reale interruptfobia che finisce per condizionare la realizzazione dei programmi.

E questo è molto riduttivo, in quanto le azioni di interrupt sono assolutamente indispensabili per qualsiasi applicazione poco più che semplicistica; strutture di temporizzazione, comunicazioni seriali, USB, LAN, gestione di tastiere o encoder, ecc, ecc. non hanno alcuna possibilità di funzionare in modo anche solo poco più che elementare senza interrupt. E la "timidezza" ad affrontare questo argomento dipende esclusivamente dal fatto di non averne afferrato il meccanismo.

Ma questa posizione è facilmente superabile considerando quanto abbiamo finora detto:

- interrupt non è altro che una particolare forma di gestione di una subroutine

semplicemente si tratta di scrivere un tratto di programma che faccia fronte ad un evento, fornendo l' opportuna funzione.

Se, volendo sommare due numeri, scriverò una subroutine adeguata e la richiamerò ogni volta che ne ho la necessità, altrettanto, nel caso dovuto all' esaurimento di un timer dovrò scrivere un tratto di programma che, ad esempio, ricarichi il timer ed aggiorni un contatore.
Nel caso della somma dei due numeri, sarò io, nel programma, a richiamarla e fare eseguire le istruzioni con una istruzione call quando mi servono; per l' interrupt è la stessa identica cosa, solo che sarà l' hardware a richiamare da se le istruzioni quando necessario. Istruzioni che, comunque, avrò definito io, in modo tale da avere le necessarie azioni.

Comprendiamo bene che in entrambi i casi si tratta semplicemente di aver scritto una serie di istruzioni che svolgono una data funzione e di metterle a disposizione nel programma ogni volta che quella funzione è necessaria: il fatto che sia l' hardware a decidere il momento in cui questo viene messo in atto non è una complicazione, ma, anzi, è una semplificazione.

Infatti, quale è l' alternativa ad una gestione dell' interrupt ? E' il cosiddetto polling, ovvero mettersi in attesa dell' evento, non potendo fare altro che quello, e, quando esso si presenta, chiamare la subroutine adeguata.

E' evidente che se il microcontroller non ha altro da fare, attendere che un timer abbia esaurito il suo conteggio è certamente possibile. Ma non lo è se il microcontroller deve nel contempo tenere sotto controllo altre funzioni, anche solo una trasmissione seriale o il controllo di un display multiplexato.

Nei casi in cui siano diverse le periferiche e le attività concorrenti (task), ciascuna con le sue necessità di controllo e con i suoi tempi di esecuzione, il meccanismo dell' interrupt consente al processore di seguire una linea principale di programma e balzarne fuori per gestire un evento e poi rientrare, anche per eventi che capitano in modo del tutto non sincrono tra di loro. L'alternativa del polling costringe il programmatore ad un gravoso gioco di incastro delle varie azioni, dove il multi task si trasforma in una concatenazione critica di eventi successivi e che può essere messo in atto solamente in un numero limitato di casi.

Cerchiamo di chiarire ancora meglio.
Supponiamo di gestire un display multiplexato a 7 segmenti. Il multiplex consiste nell' accendere una cifra per volta con una cadenza tale da far si che l' occhio le percepisca senza sfarfallio.
La soluzione più semplice è quella di impostare un timer per il tempo richiesto tra l' accensione di una cifra e l' altra.

Il timer conta e, dopo il tempo voluto, dipendente dal clock del processore e dal valore caricato nel timer, esaurisce il conteggio e si azzera. A questo punto è richiesta una semplice azione: ricaricare il timer con il valore richiesto.
Dunque la funzione che corrisponde alla fine del conteggio è questa.
Posso allora procedere in due modi:

- **polling**: attendo la fine del conteggio verificando il contenuto del contatore del timer e chiamo una subroutine che ricarica il timer
- **interrupt**: al vettore di interrupt piazzo la routine che ricarica il timer. Alla richiesta di interrupt, dovuta all'overflow, questa sarà eseguita automaticamente

E' evidente che il secondo metodo è certamente il migliore. Ma supponiamo che, colpiti da interruptfobia, vogliamo adottare il polling. Posso certamente scrivere un programma che faccia questo: carico il timer, accendo una cifra, mi metto in attesa della fine del timer, lo ricarico, accendo la cifra successiva, ecc. Facile da farsi.
Però...

Se il display è fine a se stesso, non ci sono problemi: il processore deve solo passare da una cifra alla successiva. Ma se si tratta delle cifre di un cronometro, di un voltmetro, di terminale ? Come posso, assieme alla gestione del display multiplexato, gestire anche l' ADC o l' RTC o l' UART?

Evidentemente questo è possibile solo in minima parte, in quanto il processore deve seguire il polling sul timer; diventa possibile realizzando pseudo multi task che in effetti sono una organizzazione concatenata degli eventi, lasciando momentaneamente il polling per eseguire le altre azioni, ma dovendoci tornare prima dell'overflow. Questo riduce il tempo disponibile per ogni azione a ben determinate finestre temporali che, nel caso di task del tutto asincrone, i cui eventi sono dipendenti dall'esterno del microcontroller, possono diventare ingestibili.

Si rischia di essere impegnati in una di queste e trascurare le altre, ottenendo, ad esempio, un display saltellante o perdendo dati sulla seriale, oppure riducendo drasticamente le frequenze, dato che questo genere di multitask è basato essenzialmente sulla "forza bruta" del tempo di ciclo delle istruzioni. Per poter fare "di più" occorreranno tempi di esecuzioni minori, ovvero un clock più elevato: potenza di calcolo utilizzata in modo poco efficace, con un aumento ingiustificato del consumo energetico.

La gestione in interrupt, invece, consiste solamente nello scrivere le funzioni richieste da ogni evento, cosa che devo comunque fare anche per il polling, e piazzarle in modo logico al vettore di interrupt, lasciando che siano gli stessi eventi a chiamare le gestioni corrispondenti.

Scrivere una gestione di un singolo evento di interrupt non è particolarmente più complesso che gestire la stessa in polling. Occorre qualche istruzione in più, ma queste sono banali.
Quello che è differente è la logica di intervento, che non è più strettamente sequenziale; avere le idee ben chiare su quello che si sta scrivendo richiede di ricorrere ai tanto trascurati flowchart, che sono uno degli aiuti più potenti nella stesura di un programma appena un poco complesso.

Dunque, non c'è ragione alcuna per l' interruptfobia: compreso il meccanismo, si tratta solo di aggiungere quei pochi elementi necessari. Se l' interruptfobia persiste, si tratta di un approccio totalmente errato alla programmazione per mancanza di una analisi minima di quanto si vuole ottenere e delle possibilità disponibili per farlo. Questo ha come conseguenza la perdita di una logica di azione efficace e quindi la difficoltà a gestire le interruzioni.

---

# Interrupt e stack.

Sia subroutine che interrupt usano lo stack per caricarvi l'indirizzo di ritorno e quindi ne va tenuto conto nell'impiego pratico, dato che lo stack dei Midrange ha un numero di livelli più ampio di quello offerto dai Baseline (8 livelli contro 2), ma non infinito ed occorre evitarne il tracollo.

Da questo punto di vista le istruzioni di rientro sono analoghe. Lo stack è ampio come il PC e contiene l'intero indirizzo di rientro: non occorre alcuna manovra dei bit di cambio pagina.

---

# Sorgenti di interrupt.

Praticamente tutte le periferiche integrate sono sorgenti di interrupt:

- tutti i timer, all'overflow
- modulo ADC, al termine della conversione
- comparatori, al cambio di stato dell'uscita
- UART/USART, all'arrivo di un byte o allo svuotamento del buffer di trasmissione
- modulo CCP
- modulo MSSP

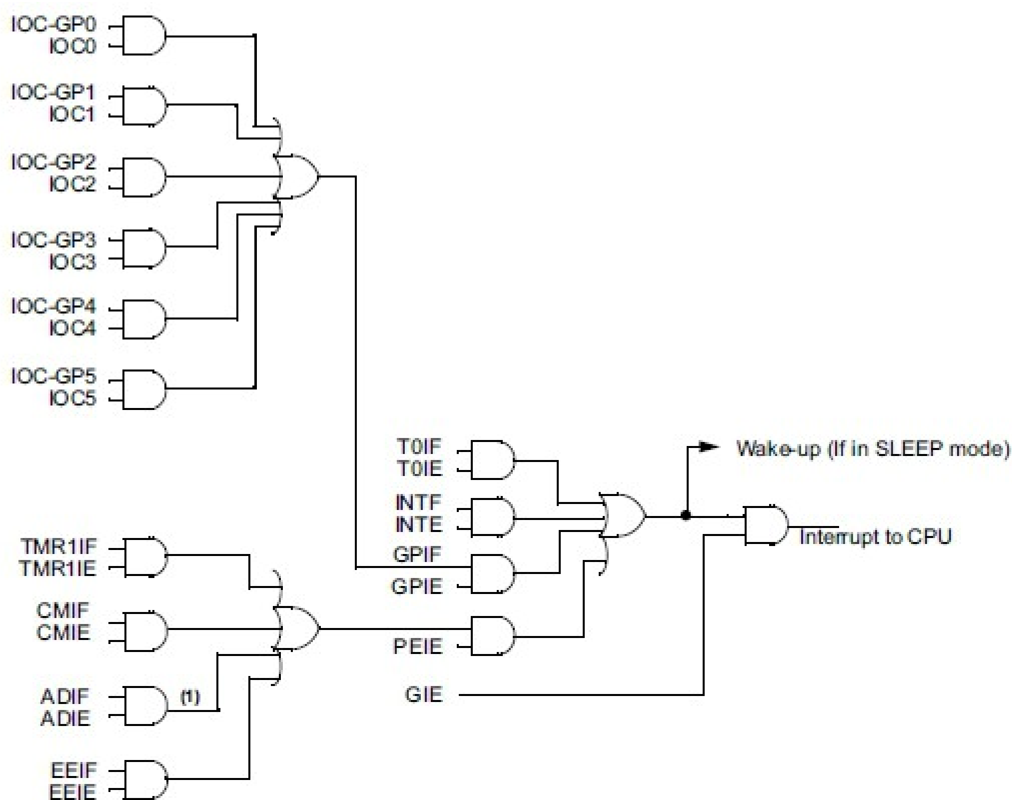
Inoltre si generano interrupt al cambio di livello dei pin o da una variazione su uno specifico pin (INT).

Tutte le periferiche dispongono negli specifici registri di controllo di alcuni bit relativi alla funzione di interrupt.

---

# Graficamente...

Vediamo un diagramma in cui è schematizzato il sistema di interruzioni dei 12F629/675:




Da un punto di vista logico, rappresentiamo con i simboli booleani la situazione dei segnali di interrupt (**xxIF**) e dei flag di abilitazione (**xxIE**).

Prendiamo ad esempio l'interrupt generato dall'overflow di TIMER0.

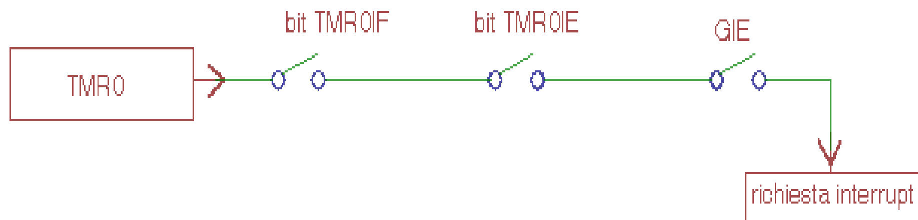
Se il timer termina il conteggio, passando da FFh a 00h, il bit T0IF va a livello 1. Questo bit è in AND con i bit di abilitazione T0IE : se questo è a livello 1, anche l' uscita dell' AND va a livello 1.

Questa uscita è in OR con altre catene similari: dunque basta che uno solo degli ingressi dell' OR vada a livello 1 perchè l' uscita del gate sia pure a livello 1.

Questa uscita viene riportata ad un ulteriore AND che ha all'altro ingresso il bit di abilitazione generale **GIE**: se questo è a livello 1, l' uscita dell'AND va a livello 1 e attiva la chiamata di interrupt.

Nello stesso tempo, se la funzione è abilitata a questo, un gate OR comanda il wake up dalla condizione di sleep.

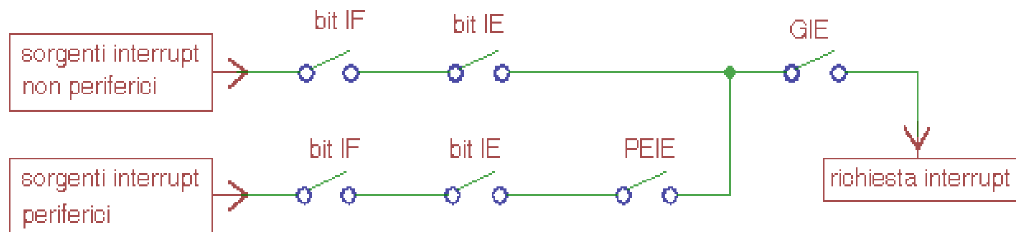
Se esemplifichiamo quanto detto con una catena di interruttori, otteniamo:




Ugualmente le altre fonti di interrupt dispongono di bit **IE** e **IF** e sono condizionate dal bit generale **GIE**.

Se osserviamo il diagramma e seguiamo la richiesta di interrupt del Timer1

Notiamo che nel percorso è inserito un ulteriore AND con il segnale **PEIE**.
Visto come interruttori in serie:




Si tratta di una divisone tra le sorgenti di interruzione tipica di Microchip.

# INTERRUPT Periferici e non periferici

Una caratteristica specifica di progetto deI PIC è quella di distinguere due categorie di sorgenti di interrupt :

- **interrupt "non periferici"**
- **interrupt "periferici"**

La differenza consiste in questo:

- le sorgenti di interrupt "non periferici" sono relative **a funzioni di base presenti su tutti i chip**, come Timer0, Pin Level Change, INT. Queste hanno i bit **IE** e **IF** nei registri **INTCON**.

- Le sorgenti "periferiche" sono quelle relative, appunto, alle periferiche, come UART, MSSP, CCP , Timer2, ecc.; esse hanno i bit di controllo nei registri **PIR/PIE**.

Ovvero: le sorgenti di interrupt di tipo periferico possono essere disabilitate in blocco dallo switch **PEIE** senza alterare il **GIE**.

Ci si può chiedere a cosa possano servire gli "interruttori generali" , i quali sono azionabili durante il programma.
In generale, la loro funzione è quella di disabilitare momentaneamente le sorgenti di interrupt perchè viene eseguita una operazione che non deve essere interrotta; un esempio di queste operazioni che non vanno interrotte, pena il loro fallimento, è costituito dalla scrittura della sequenza-chiave per la EEPROM. Siccome potrebbero essere attive varie sorgenti di interrupt, un "interruttore generale" è molto più efficace e semplice da azionare che non la modifica dei singoli bit IE .

# Gestire l' interrupt.

Come abbiamo visto, possono essere disponibili numerose fonti di interrupt, derivanti da un altrettanto elevato numero di periferiche.

Però, **non è certo necessario gestire tutte le fonti di interrupt disponibili**: nella maggior parte dei casi l'applicazione utilizzerà solamente alcune di queste periferiche, mentre le altre non verranno neppure considerate.

Ne deriva che è necessario almeno un paio di bit di controllo per ogni sorgente di evento:

- **un bit per abilitare o disabilitare la possibilità della periferica**: se mi serve, abiliterò la generazione dell'interrupt; se non mi serve la disabiliterò. Così ho la necessità di trattare solamente le periferiche necessarie all'applicazione, mentre le altre resteranno dormienti.

- **un bit per avvisare che la periferica ha richiesto l'interruzione**. Questo è indispensabile per identificare la sorgente che ha attivato la richiesta.

Questi due bit di controllo sono comuni a Midrange e PIC18F (i Baseline non hanno interrupt):

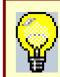
| bit | funzione |
|---|---|
| **IE** - *Interrupt Enable* | **bit di abilitazione**, la cui sigla termina tipicamente con IE; portando a 1 questo bit si abilita la sorgente a generare interruzioni |
| **IF** - *Interrupt Flag* | **bit di flag**, la cui sigla termina con IF.<br>Il livello logico 1 di questo bit indica l'avvenuto evento. |

Questi sono i bit che occorre considerare nella chiamata ad interrupt e che, invece, non sono usati per le chiamate a subroutine. Vediamoli in dettaglio.

---

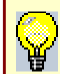
# Bit IE

Occorre tenere presente che:

 **ogni sorgente di interrupt è una sorgente potenziale, ovvero non chiama alcun interrupt fino a che il programmatore non abilita la sorgente stessa**

Questo è ottenuto portando a 1 un bit IE (*Interrupt Enable*) contenuto nel registro di controllo di ogni periferica. Così il Timer0 avrà un bit **TMR0IE**, l' ADC avrà un **ADIE**, ecc.
Cosa vuol dire questo ?

- Se il bit **IE** della periferica è a **0** (valore al default dopo il reset) **la periferica funziona regolarmente, ma non produce chiamate ad interrupt.**
- La chiamata ad interrupt sarà generata dalla periferica solo se il bit **IE** è stato posto a **1**

 **Al POR, nessuna fonte di interrupt è attiva: sta all' utente accendere le sorgenti volute**

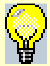
**portando a 1 il relativo bit IE.**

Ovviamente, in qualsiasi momento, riportandolo a 0, la sorgente cesserà di generare interrupt.

---

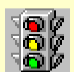
# Bit IF

Quando la periferica richiede una interruzione, lo fa attraverso un bit di flag **IF** (*Interrupt Flag*), che viene portato a 1.
Andando a testare lo stato di questo bit si potrà identificare la periferica che ha richiesto l' interruzione.
Ogni periferica dispone del proprio, per cui si avranno, ad esempio, **`TMR0IF`**, **`ADIF`**, ecc.

 **E' molto importante notare che il flag IF va a livello 1 anche se l' interrupt non è abilitato, ovvero se IE di quella periferica è a 0.**

**IF** è una "bandierina" (flag) sollevata per permettere il riconoscimento del "colpevole" della chiamata.
**IF** è legato al funzionamento della periferica e si attiva quando si verifica l'evento specifico; questo flag è "funzionante" in modo indipendente dal fatto che l' interrupt sia abilitato o meno. Così, ad esempio, per sapere quando l' ADC ha terminato la conversione, sarà sufficiente testare lo stato di 1 del bit ADIF e per vedere se TIMER0 ha esaurito il conteggio non occorre verificare se il registro TMR0 è andato a 0, ma basta verificare lo stato di TMR0IF.

Occorre qui un momento di **attenzione particolare**:

 **per consentire una gestione dell' evento, il flag IF resta a livello 1 fino a che non viene azzerato dall'utente che ha inserito nel programma le necessarie istruzioni.**

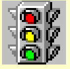
Perchè questo? semplicemente perchè dal momento dell' evento al momento in cui il programma inizia la relativa gestione può trascorrere un certo tempo, ad esempio quando ci sono più chiamate di interruzione sovrapposte. Quindi la "bandierina", una volta alzata, resta alzata fino a che l' utente, dopo aver servito l'evento, non va ad abbassarla (questo azzeramento si fa di solito in uscita dalla routine di gestione dell' interrupt).
Ovvero:

**<u>la routine di gestione comporta la necessità assoluta di resettare il flag una volta servito.</u>**

Questa azione è indispensabile per la maggior parte delle periferiche (alcune, poche, si azzerano da sole una volta servite in modo adeguato) in quanto il portare a 0 il flag cancella la chiamata di interrupt e fa si che gli automatismi interni si predispongano per accogliere un evento successivo.

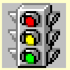
Se non si portasse a 0 il flag, la chiamata di interrupt resterebbe attiva e bloccherebbe l'esecuzione del programma: non appena usciti dalla routine di interrupt chiamata da quella periferica, ci si

ritornerebbe immediatamente, dato che il flag IF non è stato "abbassato", con i risultati immaginabili.

 **La gestione di un interrupt deve SEMPRE prevedere la cancellazione del flag della periferica che ha generato la chiamata.**

In caso contrario non sarà possibile uscire da un loop sul vettore dell' interrupt: se il flag non è cancellato, al retfie si rientrerà al programma principale, ma solo per tornare immediatamente al vettore di interrupt !

Questa gestione del flag **IF** è la cosa principale che occorre non trascurare nella scrittura della gestione dell' interrupt.

 In particolare, **se usiamo in polling una periferica testando il flag IF**, senza avere abilitato il relativo IE, dobbiamo comunque azzerare IF nella subroutine di gestione, altrimenti lo stato della "bandierina" rimarrebbe inalterato e non sarebbe possibile rilevare un evento successivo.

Per riassumere in poche parole:

- nel main, aggiungere una abilitazione dei bit **IE** delle periferiche da cui vogliamo chiamate ad interrupt quando questo è necessario. Le altre periferiche per default non saranno abilitate e quindi possono essere non considerate.
- si deve introdurre la routine di gestione dell' evento, linkandola al vettore di interrupt
- in questa routine si deve prevedere la cancellazione del flag **IF** della periferica chiamante

Vedremo più avanti che l' **abilitazione dei bit IE** non è sufficiente ad attivare l' interrupt: altri bit fungono da "interruttore generale" di sicurezza e sarà necessario settare anche questi perchè l' evento di interrupt abbia effetto.

---

## Gli interruttori generali GIE e PEIE.

Occorre ancora focalizzare un elemento: se disabilitiamo il bit **IE** di una sorgente di interrupt, questa azione NON attiverà l'interrupt. Però le sorgenti possono essere varie e, nella necessità di bloccare tutti gli interrupt, si dovrebbe ricorre ad una serie di disabilitazioni singole.

In effetti, esiste un "interruttore generale" : si tratta del bit **GIE** (*Global Interrupt Enable*) del registro **INTCON.**
**Questo bit, è, per default al POR, a livello 0, impedendo qualsiasi interrupt**, a prescindere dal valore dei bit IE.
**Per disporre della funzione, occorre portare da programma questo bit a 1**.

Quindi, volendo attivare l'interrupt di una sorgente, la sequenza sarà:

1. abilitare il bit **IE** della sorgente voluta
2. abilitare il bit **GIE** per attivare la funzione di interrupt

Va ancora aggiunto un elemento: a seguito della divisione vista sopra tra sorgenti periferiche e non periferiche, i bit di abilitazione globale, in realtà, sono due: il già citato GIE e un **PEIE** (*PEripheral Interrupt Enable*) .
Questo bit abilita o disabilita solamente gli interrupt periferici.

In pratica, volendo abilitare l'interrupt di una sorgente periferica occorrerà:

1. abilitare il bit **IE** della periferica voluta
2. abilitare il bit **PEIE** se si tratta di un interrupt periferico
3. abilitare il bit **GIE** per attivare la funzione di interrupt generale

| | **sorgente non periferica** | **sorgente periferica** |
|---|---|---|
| **abilitazione generale** | **GIE** | **GIE + PEIE** |
| **abilitazione specifica** | **IE** | **IE** |

Volendo disabilitare momentaneamente gli interrupt periferici, basterà agire su **`PEIE`**.
Volendo disabilitare momentaneamente tutti gli interrupt, si agirà su **`GIE`**.

Vediamo di chiarire con qualche diagramma.

Lo schema seguente utilizza delle funzioni logiche AND e OR per spiegare il funzionamento della catena di bit preposti al controllo di un interrupt.

# I registri di gestione.

Nei Midrange il registro **INTCON** ha, in generale, questa struttura (dal foglio dati di 16F690):

**INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh OR 18Bh)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-x |
|---|---|---|---|---|---|---|---|
| GIE | PEIE | T0IE | INTE | RABIE | T0IF | INTF | RABIF |
| bit 7 | | | | | | | bit 0 |

Da notare che dispone di indirizzi alternativi, ovvero è accessibile identicamente da 4 banchi (o dai due dei PIC minori) e non richiede switch di banco.

In pratica, **INTCON** contiene gli "interruttori generali" **GIE** e **PEIE** e gli interruttori non-periferici, ovvero da Timer0 (**T0IE**), del pin **INT** (**INTE**) e del Pin Level Change (**RABIE** per i chip con **PORTA** e **PORTB** o **IOCIE** per quelli con **GPIO**).
Inoltre contiene i flag di segnalazione degli stessi, ovvero **T0IF**, **INTF** e **RBIF** (o **IOCIF** a seconda del chip).

Per le altre periferiche, i registri **PIE** contengono i bit di abilitazione. Possono esserci più di un registro **PIE** , a seconda del numero delle periferiche installate. Ecco, ad esempio, **PIE1** e **PIE2** di 16F690:

**PIE1 – PERIPHERAL INTERRUPT ENABLE REGISTER 1 (ADDRESS: 8Ch)**

| U-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| — | ADIE | RCIE(2) | TXIE(2) | SSPIE(2) | CCP1IE(1) | TMR2IE(1) | TMR1IE |
| bit 7 | | | | | | | bit 0 |

**PIE2 – PERIPHERAL INTERRUPT ENABLE REGISTER 2 (ADDRESS: 8Dh)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | U-0 | U-0 | U-0 | U-0 |
|---|---|---|---|---|---|---|---|
| OSFIE | C2IE | C1IE | EEIE | — | — | — | — |
| bit 7 | | | | | | | bit 0 |

Osserviamo che hanno una sola posizione in memoria (precisamente nel banco 1) per cui sarà necessario utilizzare lo switch di banco banksel quando si deve accedere a questi SFR.

Altrettanto per i registri **PIF**, con i flag di identificazione della periferica chiamante. Sempre per 16F690 si tratta di due SFR: **PIR1** e **PIR2**

**PIR1 – PERIPHERAL INTERRUPT REQUEST REGISTER 1 (ADDRESS: 0Ch)**

| U-0 | R/W-0 | R-0 | R-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| — | ADIF | RCIF(1) | TXIF(1) | SSPIF(1) | CCP1IF(2) | TMR2IF(2) | TMR1IF |
| bit 7 | | | | | | | bit 0 |

Anch

Nel c

**PIR2 – PERIPHERAL INTERRUPT REQUEST REGISTER 2 (ADDRESS: 0Dh)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | U-0 | U-0 | U-0 | U-0 |
|---|---|---|---|---|---|---|---|
| OSFIF | C2IF | C1IF | EEIF | — | — | — | — |
| bit 7 | | | | | | | bit 0 |

Ecco un elenco di alcuni di questi bit, giusto per dare una idea delle possibilità disponibili:

| Periferica | bit IE | bit IF |
|---|---|---|
| Timer1 overflow | TMR1IE | TMR1IF |
| Timer1 gate | TMR1GIE | TMR1GIF |
| CCP1 | CCP1IE | CCP1IF |
| Sychronous serial port | SSPIE | SSPIF |
| USART ricezione | RCIE | RCIF |
| USART trasmissione | TXIE | TXIF |
| Modulo ADC | ADCIE | ADCIF |
| Slope AD TMR Overflow | OVIE | OVIF |
| Parallel slave port | PSPIE | PSPIF |
| EEPROM | EEIE | EEIF |
| LCD drive | LCDIE | LCDIF |
| Comparatore | CMIE | CMIF |
| Oscillator failure | OSCFIE | OSCFIF |
| USB | USBIE | USBIF |
| MSSP bus collision | BCLIE | BCLIF |
| Low Voltage detector | HLVDIE | HLVDIF |

In generale possiamo dire che ogni periferica ha almeno una sorgente di interruzione e la coppia di flag relativi.

# La priorità.

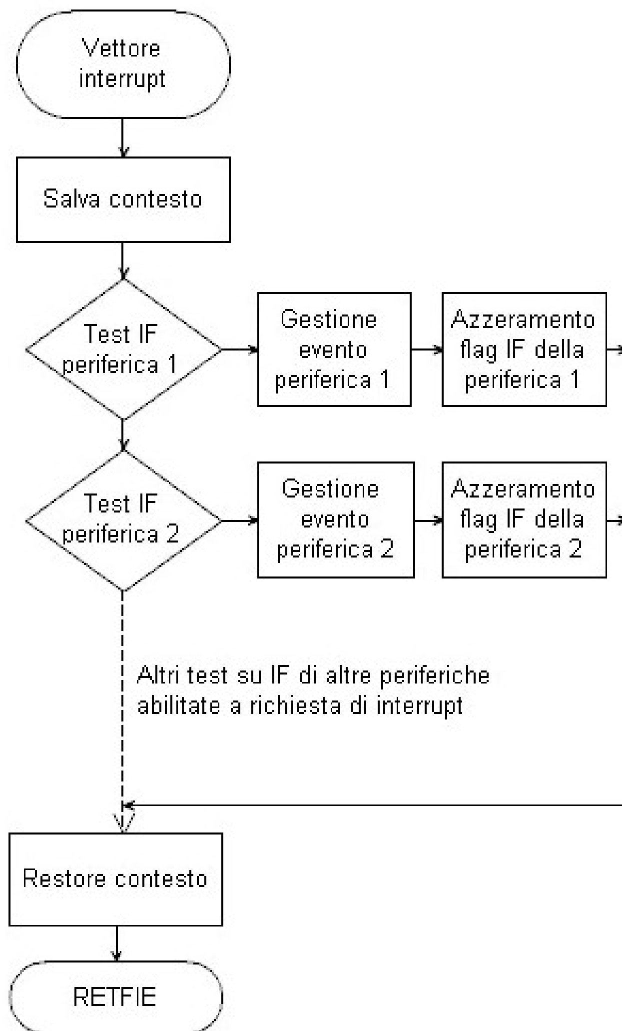
Il fatto che esista una sola locazione in cui convergono tutte le possibili chiamate dell' interrupt è una limitazione in termini di efficienza, in quanto il microcontroller, se sono attivate più sorgenti di interruzione, deve per prima cosa determinare quale è quella da seguire.
Quindi, la routine di interrupt dovrà testare i bit IF delle sorgenti attive e, per ognuna di esse, disporre della sequenza di istruzioni appropriata.

Nelle applicazioni critiche è probabile che esistano sorgenti di interrupt che devono essere prioritarie rispetto ad altre di importanza minore. Disponendo di un solo vettore di interrupt e di più sorgenti, occorre che la parte iniziale della routine di gestione contenga una qualche logica per poter effettuare non solo la corretta gestione di tutte le sorgenti di interrupt, ma anche di dare ad alcune una precedenza sulle altre. Nel caso dei PIC18F, la situazione è risolta introducendo due livelli di priorità programmabili con due differenti vettori di interrupt. In sostanza, ogni livello è collegato ad un diverso vettore, ovvero punta ad una diversa sequenza di istruzioni.
Inoltre, particolare importante, **un evento a priorità bassa può essere interrotto da un evento a priorità alta, ma non viceversa.** Ovviamente utilizzando due livelli si potrà avere una risposta molto più accurata agli eventi di interrupt, assegnando alla priorità elevata l'interrupt degli eventi più determinanti.
Esistono differenze sensibili tra li varie famiglie: nei PIC superiori agli 8 bit ogni periferica dispone di un proprio vettore di interrupt, rendendo così immediata la sua azione. In pratica, aumentando le prestazioni del chip, anche la gestione dell' interrupt utilizza meccanismi più complessi:

| | |
|---|---|
| **Baseline** | nessun interrupt |
| **Midrange** | interrupt a 1 livello |
| **PIC18F** | Interrupt con priorità a due livelli |
| **PIC24/dsPIC** | ogni periferica ha un proprio vettore di interrupt |
| **PIC32** | hanno un numero elevato di interrupt,<br>con registri di controllo per la priorità e la sub-priorità. |

## Priorità "software".

In un sistema con priorità singola, come quello offerto dai Midrange, se abbiamo una sola sorgente di interrupt, non ci sono problemi: la routine di gestione sarà specifica per questa periferica.
Se, però, abbiamo più di una sorgente, qualunque di esse richieda una interruzione, devierà il PC all'unico vettore di interrupt.  Tipicamente, la gestione avrà questa struttura:




Il Program Counter viene deviato al vettore di interrupt ed inizia la routine di gestione.

Questa determina quale periferica sia la "colpevole" della chiamata, verificando quale bit IF è andato a 1, in modo da eseguire la relativa gestione.

Consideriamo che siano attive diverse sorgenti di interrupt e il vettore sia unico, come capita nei Midrange. Tra la richiesta di interruzione e la deviazione del program counter è necessario un certo numero di cicli istruzione; può capitare che, durante questo tempo, anche le altre periferiche richiedano una interruzione, portando a 1 i relativi flag.

Il flow chart descrive l' azione dell' algoritmo: viene verificata la periferica 1, poi la 2, poi la 3 e così via.

Se è stata la periferica 2 ad effettuare la chiamata e solo poco dopo anche la 1 lo ha fatto, sarà eseguita, in ogni caso, per prima la gestione della periferica 1

All'uscita della routine sarà cancellato il flag **IF** di questa periferica.

Al ritorno al main, ci sarà immediatamente un rientro al vettore di interrupt, in quanto i flag IF della Due è ancora acceso. Ora la routine troverà azzerato il flag della periferica Uno e servirà quello della Due e così via.
Se fosse stata la Tre ad avere, per pochi microsecondi, richiesto per prima l'interrupt, si troverebbe comunque ad essere l'ultima servita.

Se osserviamo bene, vediamo che, in presenza di un solo vettore, si determina comunque una priorità "software" agli interrupt. Quindi dove è richiesto, si potrà attribuire una priorità alle varie sorgenti analizzandole in ordine di importanza, partendo da quella più determinante e le altre in successione.

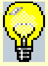
Questo accade anche se si hanno due livelli di priorità con più di due sorgenti di interruzione. La priorità più elevata sarà attribuita all'evento più significativo per l'esecuzione del programma, ma le altre sorgenti dovranno essere gestite come visto ora.

Da queste considerazioni si può apprezzare la funzionalità presente nei PIC superiori di una presenza di vettori distinti per ogni periferica e di una gestione dei livelli più articolata, che permette di intervenire con tempi di esecuzione quanto mai abbreviati.

# Una breve nota.

L'apparente complicazione data dalle numerose possibilità che offrono i chip che integrano varie periferiche è troppo spesso un ostacolo che l'utente pensa di non riuscire a superare.

Però, come per la gran parte degli elementi periferici del microcontroller va ricordato che, in generale, la filosofia di progetto dei chip embedded è tale per cui **la presenza di una molteplicità di periferiche, funzioni, istruzioni, non deve essere vista come un aggaravio della gestione rispetto ad un chip meno ricco.**

 **Per default, le funzioni al di là di quelle basilari sono normalmente disabilitate. Entrano in gioco solamente quando l'utente le attiva settando gli opportuni bit da programma.**
**In caso contrario è come se non ci fossero !**

Nel diagramma precedente sono visibili numerose fonti di interrupt; però, **per default all' avviamento, nessuna di queste fonti è attivata**.

Ne deriva che questa apparente complicazione esiste solamente nel caso in cui, volontariamente, attraverso le istruzioni, si vada ad abilitare queste funzioni. In caso contrario, nessun interrupt è attivo e ne posso abilitare anche uno solo, quello che mi interessa, lasciando "dormienti" tutti gli altri: come se non ci fossero.

Quindi, non esiste alcuna difficoltà nello scrivere un programma per un vecchio 16F84 o per un recente 18F14k22.
In entrambi, se intendo usare solamente il Timer0, abiliterò l'interrupt solo di questo, mentre per tutte le altre periferiche che non uso, non mi interesserà neppure leggere (per il momento...) i relativi capitoli del foglio dati, venendo così a mancare anche il motivo di panico che coglie molti davanti a data sheets da 600 e passa pagine.

# Salviamo il contesto.

Abbiamo notato nel flow chart precedente la presenza di due blocchi: uno, iniziale, di "salvataggio del contesto" e uno finale di "restore del contesto". Vediamo cosa significano.

Il salto ad una subroutine non è, in realtà, una sospensione del flusso logico principale, ma l'inserimento in questo di un blocco di istruzioni che è richiesto per completare una azione.
Ad esempio, è stato elaborato un dato ed ora è richiesta la sua trasmissione su una uscita seriale; si richiamerà la subroutine opportuna.
Al rientro, il programma procederà all' elaborazione successiva.

E' ovvio che, nel passare ad una subroutine, il programmatore ha ben chiaro il contesto in cui questo accade, ovvero della situazione dei registri prima e dopo la chiamata. Ad esempio, la routine di trasmissione riceve attraverso WREG il dato da trasmettere e ritorna con un flag indicatore dell'esecuzione.

Un interrupt, invece, **"interrompe" in un momento del tutto casuale** una qualsiasi attività dell'unità centrale per eseguire le istruzioni di gestione di un evento. Questo non vuol dire che sia "inaspettato": infatti lo abbiamo abiltato noi da programma, scrivendo al vettore di interrupt una opportuna routine di gestione dell'evento.

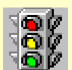
Ma questo vuol dire che la situazione dei registri principali del sistema (**WREG, STATUS, PCL, PCLATH, FSR**) può venire cambiata in modo determinante durante l'esecuzione delle istruzioni relative al vettore di interrupt.

Facciamo un esempio: il programma sta eseguendo un algoritmo matematico in cui sono coinvolti **WREG** e **STATUS**. Questo processo viene interrotto in un istante qualsiasi dalla richiesta dell'UART che ha ricevuto un carattere. L'algoritmo matematico è sospeso e si passa alla gestione del carattere arrivato; la routine lo piazza in un buffer, ma durante la sua esecuzione ha modificato **WREG, STATUS, FSR**.

Al termine della gestione dell'evento, il **retfie** riporta il PC all' istruzione successiva a quella in cui è avvenuta la chiamata, ovvero il programma riprende ad eseguire l'algoritmo matematico al punto in cui era stato interrotto.

Però lo fa con valori nei registri **WREG** e **STATUS** diversi da quelli precedenti: il risultato dell'operazione matematica sarà alterato !

Dunque:

 **Prima di entrare nella routine di gestione dell'interrupt è necessario salvare il contesto, ovvero i registri principali, come STATUS e WREG.**

All'uscita della routine, i parametri salvati saranno riscritti nei registri (restore), in modo tale da ripristinare la continuità dell'esecuzione in modo corretto.

La famiglia dei Midrange non ha un meccanismo automatico per il salvataggio del contesto (come invece hanno i PIC18F e gli Enhanced Midrange) e, oltre a ciò, ha un certo numero di problemi che complicano questa azione:

1. Tutti gli spostamenti e operazioni logiche sui files passano attraverso il registro **W**.

2. L'istruzione **movf** modifica lo **STATUS** nell'esecuzione, agendo sul flag **Z**.

3. La RAM dati è divisa in banchi, in cui è mescolata con i registri delle funzioni speciali (SFR). C'è un'area RAM ad accesso shared da tutti banchi, ma ha ampiezza piuttosto limitata (tipicamente 16 locazioni).

4. Non tutti i chip hanno risorse RAM comuni, cosi come quantità che come banchi.

5. Anche la memoria programma può essere divisa in pagine (da 2048 word ciascuna).

6. Non c'è modo di accedere allo stack per inserirvi o estrarre dati; lo stack, comunque, ha una profondità limitata.

Queste condizioni interferiscono con una gestione semplice del salvataggio del contesto e devono essere gestite dall'utente nella routine di servizio di interruzione.

Ne consegue che, per salvare correttamente il contesto in interrupt occorrono alcune precauzioni ed una sequenza precisa:

1. Il registro **W** deve essere salvato in RAM
   Questa è la prima azione, dato che **W** dovrà essere usato per le azioni successive

2. Lo **STATUS** deve essere salvato, dato che contiene i falg di risultato (C, Z, DC) e i bit di switch dei banchi (RP1:0).
   Per spostarlo in RAM occorre passare attraverso il registro **W** .
   Non possiamo fare questo con l'istruzione **movf STATUS,w**, dato che essa cambia il valore del flag Z.
   Quindi, si rende necessario usare l'istruzione **swapf STATUS, w**, dato che **swapf** non modifica i flag dello **STATUS**. Ne risulta che il nibble basso e il nibble alto saranno scambiati e che, queindi, risulterà necessario usare ancora **swapf** per il restore.

3. E' opportuno dedicare un banco alla memoria RAM necessaria durante la gestione dell'interrupt.
   La memoria shared di solito è limitata, ma se non serve in altre azioni, è ideale. Altrimenti si potrà usare un altro banco; non è obbligo usare RAM nel banco0, ma è quello il cui accesso richiede meno istruzioni. In ogni caso, è opportuno che gli elementi del contesto siano salvati in una area determinata.

4. il registro **PCLATH** deve essere salvato. Questo è necessario perchè, se si lavora su più pagine o si usano **tabelle retlw** posizionate diversamente in memoria, la perdita del contenuto del **PCLATH** porterebbe, al ritorno, a generare salti errati, con il relativo blocco del programma.

5. altri registri usati sia nel main che nell'interrupt vanno salvati. Ad esempio, è importante il registro FSR, se viene usato l'indirizzamento indiretto. I registri sono scritti in RAM in successione, dopo i primi fondamentali, ma non ha importanza l'ordine di salvataggio.

In istruzioni:

```
saveram    UDATA_SHR          ; RAM shared
w_tmp res 1

saveram    UDATA 0x20         ; RAM in banco0
status_tmp res 1
pclath_tmp res 1
fsr_tmp res 1

         movwf    w_tmp      ; salva W
         swapf    STATUS,w
         clrf     STATUS     ; forza il banco 0
         movwf    status_tmp ; salva STATUS
         movf     PCLATH,w
         movwf    pclath_tmp ; Salva PCLATH
         movf     FSR,w
         movwf    fsr_tmp    ; salva FSR
```

Da notare che :

- **W** deve essere salvato in RAM shared, dato che non è possibile indicare un banco, cosa che modificherebbe **RP1:0** dello **STATUS**.
  Si potrebbe, con un maggiore lavoro software, usare in una locazione riservata uguale in tutti i banchi, ad esempio 0x020, 0x0A0, 0x120 e 0x1A0; in tal caso, va bene qualsiasi banco sia puntato da RP1:0 al momento dell'ingresso in interrupt, copiando poi successivamente nelle altre locazioni, per essere sicuri che la memorizzazione di **W** non sovrascriva qualsiasi altra variabile o essere sovrascritta.

- Una volta salvato `WREG` e lo `STATUS` è trasferito in `WREG`, si è liberi di scegliere l'area di RAM per salvare i registri.
  Occorre scegliere un banco in quanto, all'ingresso della routine di interrupt, è selezionato un banco qualsiasi, dipendentemente da cosa si svolgeva nel tratto di programma interrotto.
  Se pure è possibile determinare il banco analizzando i bit `RP1:0`, è molto più semplice e chiaro definire un banco dove salvare i registri. Questo evita di sovrascrivere aree di RAM usate in altre parti del programma.
  Quando si esce dall'interrupt, l'azione di ricaricare lo `STATUS` salvato all'inizio ripristina gli switches di banco come erano nel programma chiamante.
  Se si usa RAM shared, questa azione non è richiesta.

- Se si usa RAM banked e non shared, l'impostazione del banco di ram da utilizzare deve essere fatta dopo che lo `STATUS` è stato copiato in `W,` altrimenti si modificherebbero `RP1:0` nello `STATUS` stesso.
  Quindi, occorre prima copiare lo `STATUS` precedente, poi modificare quello attuale per il banco voluto.
  La soluzione più semplice è quella di usare il banco 0 semplicemente scrivendo 0 nello `STATUS`: questo porta a 0 i bit di switch dei banchi e va bene sia per il caso in cui il chip ha 2 o 4 banchi.
  Attenzione: `clrf STATUS` non azzera lo `STATUS`, perchè il flag `Z` va a 1, ma cancella `RP1:0` in una sola istruzione, mentre un `banksel 0` , anche se più chiaro, potrebbe essere compilato in due righe per chip con 4 banchi. Non cambia nulla, ma la tendenza è quella di minimizzare il tempo necessario per la gestione dell'interrupt.

Volendo usare altri banchi si impiegherà il classico **banksel saveram** al posto della linea **clrf STATUS**
Se si è usata RAM shared nessuna delle due linee è necessaria.

Attenzione perchè alcuni chip Midrange, come 12F629, hanno solo RAM shared: definendo RAM banked si genererà un errore in compilazione.

Una nota può essere fatta per il **PCLATH**: come si arriva nella gestione dell'interrupt con il banco pre determinato dal tratto di programma in cui l'interruzione si è inserita, così pure per la pagina. Se, durante l'interruzione, si fa uso di **call** o **goto** che puntano a pagine diverse, occorre adeguare il **PCLATH**: la mancanta attenzione alla situazione delle pagine rischia di mandare in blocco il programma.
All'uscita dall'interrupt, il restore ripristinerà il **PCLATH** adeguato.

Per ripristinare correttamente il contesto, prima del ritorno con l'istruzione **retfie**:

1. se si è usata RAM in banchi occorre puntare correttamente il banco. L'uso del banco 0 consente una gestione più rapida (nell'esempio). Altrimenti si utilizzerà il **banksel**.
   Se si è usata RAM shared la linea non è necessaria
2. vanno ripristinati i registri salvati a partire dall'ultimo (ad esempio, FSR).Eventuali altri registri salvati vanno ripristinati nell'ordine.
3. va ripristinato il registro **PCLATH**
4. il registro **STATUS** deve essere ripristinato con **swapf** per riportare i nibble nell'ordine giusto.
5. A sua volta, il registro **W** deve essere ripristinato senza modificare il valore del registro di stato, utilizzando l'istruzione **swapf**

In istruzioni:

```
clrf    STATUS              ; Select Bank0
 movf    fsr_tmp,w
 movwf   FSR                 ; Restore FSR
 movf    pclath_tmp,w
 movwf   PCLATH              ; Restore PCLATH
 swapf   status_temp,w       ; recupera STATUS salvato
 movwf   STATUS              ; senza modificare Z
 swapf   w_temp,f            ; recupera W senza modificare
 swapf   w_temp,w            ; lo STATUS
 retfie
```

Nei **PIC** delle famiglie superiori ai Midrange, il salvataggio e il restore del contesto sono effettuati automaticamente, operazione che non richiede tempo addizionale, nè locazioni della RAM, dato che utilizzano aree di memoria dedicate. Questo è un notevole sollievo per il programmatore, che non deve aggiungere istruzioni e che non deve tenere conto di tempi addizionali nell'esecuzione, banchi, ecc.

Per quanto riguarda i linguaggi come il C, in generale le operazioni di salvataggio del contesto sono automatizza ed adeguate alle caratteristiche di ogni chip.

Questi automatismi, comunque, indicano che vengono salvati i registri principali: in tutti i casi, **se occorre salvare altre variabili oltre quelle previste dagli automatismi, occorre procedere aggiungendo opportune istruzioni.**

# Problemi con le routines di interrupt.

I problemi più comuni nella codifica di una routine di interruzione sono:

- Il registro **`PCLATH`** non viene salvato. Questo comporta che al ritorno ci si trovi con puntata una pagina diversa da quella necessaria

- non si considera il problema delle pagine. La pagina in cui si trova la gestione dell'interrupt è solitamente la pagina 0, dato che il vettore si trova a 0x04. Nulla toglie, però, che la routine sia rinviata in una altra locazione o il programma si allarghi su più pagine o nella gestione dell'interruzione si faccia richiamo di routines che si trovano in altre pagine.
  Così, utilizzare **`call`** e **`goto`** all'interno della gestione dell'interrupt senza tenere conto del paging crea gravi problemi: la conseguenza è normalmente un blocco del programma con salti inaspettati a locazioni non corrette.

- non si considera il problema dei banchi: il banco in cui si entra nella gestione dell'interrupt non è quello che desiderate voi, ma quello ereditato dalla situazione di **RP1:0** nello **`STATUS`** del tratto di programma interrotto.
  Se il programma necessita di poca RAM, è probabile che non si esca dal banco 0, ma se la RAM richiesta supera la capacità del banco, ecco che, con la mancata cura nella gestione dei banchi, gravi problemi sono in agguato, dovuti alla scrittura/lettura di locazioni RAM che non sono quelle richieste.

- Si usano registri che sono impiegati anche nel flusso principale senza salvarli. A seconda di come è organizzato il programma, è possibile che non serva salvare altro che W e STATUS, ma è possibile che si richieda di salvare anche PCLATH, FSR e altri registri che vengono sovra scritti durante la gestione dell'interruzione.
  Verificare con cura questo aspetto.

- Si modifica lo stato del bit **`GIE`** durante la gestione dell'interrupt. **`GIE`** viene disabilitato all'ingresso nell'interrupt e riabilitato automaticamente alla fine, dato che i Midrange non gestiscono che un solo livello di interruzione. Disabilitarlo durante l'interrupt non serve a nulla, ma abilitarlo durante l'esecuzione dell'interrupt crea rischi se le sorgenti di interruzione sono più di una.
- si dimentica banalmente di cancellare il flag della periferica che ha generato l'interruzione, col risultato di bloccare il programma. Se il flag non è azzerato, all'uscita dell'interruzione il suo stato a 1 farà si che si attivi immediatamente una nuova chiamata a interrupt, col risultato di bloccare il programma in un loop inaspettato.

Conseguenza degli errori sopra elencati sono tipicamente:

- **un algoritmo, che funzionava correttamente senza interrupt, non rende i giusti risultati**
  Questo è senz'altro dovuto al fatto che non è stato correttamente salvato il set dei registri che l'algoritmo usa e che questi sono stati modificati nella routine di interruzione: al rientro, i registri resi diversi da come dovevano essere, alterano il funzionamento dell'algoritmo.
- **Il programma si blocca o si comporta stranamente.**
  Anche questo è dovuto ad un errato o parziale salvataggio del contesto, in particolare dello STATUS e del PCLATH: questo crea al rientro dall'interruzione un posizionamento degli switch di banco e pagina errati, con le relative conseguenze

Oppure si è usato, nella routine di interrupt, alcune risorse che sono comuni al programma principale, alterandole.
Ancora, non è stato cancellato il flag di interrupt della periferica, col risultato di bloccare l'esecuzione in un loop indeterminato.

Inoltre, il mal funzionamento di una gestione in interrupt può dipendere anche da fattori di logica:

- non viene valutato correttamente cosa succede nell'hardware al momento dell'interruzione e si impongono scelte errate. Ad esempio, non conoscendo il funzionamento della periferica.
- non viene valutato correttamente quanto necessario per passare dal programma principale, che viene sospeso, alla routine dell'interruzione, ad esempio non vengono salvati tutti i registri necessari.
- non viene considerata correttamente la gestione di più interrupt concorrenti.
- si impongono routine di interrupt di dimensioni eccessive rispeto alla frequenza dell'interruzione, il che non lascia spazio sufficiente all'esecuzione del resto del programma.

**Questi problemi possono essere risolti semplicemente tracciando un flow chart, o diagramma di flusso, di quanto si vuole eseguire.**

# Interrupt Latency.

E' evidente che le operazioni associate, ma che non hanno direttamente a che fare con la gestione vera e propria dell'evento di interrupt aggiungono tempi di esecuzione che può essere necessario conteggiare.

Dal momento di uno specifico evento, la periferica interessata attiva la chiamata interrupt e il programma viene deviato immediatamente verso la sua gestione.

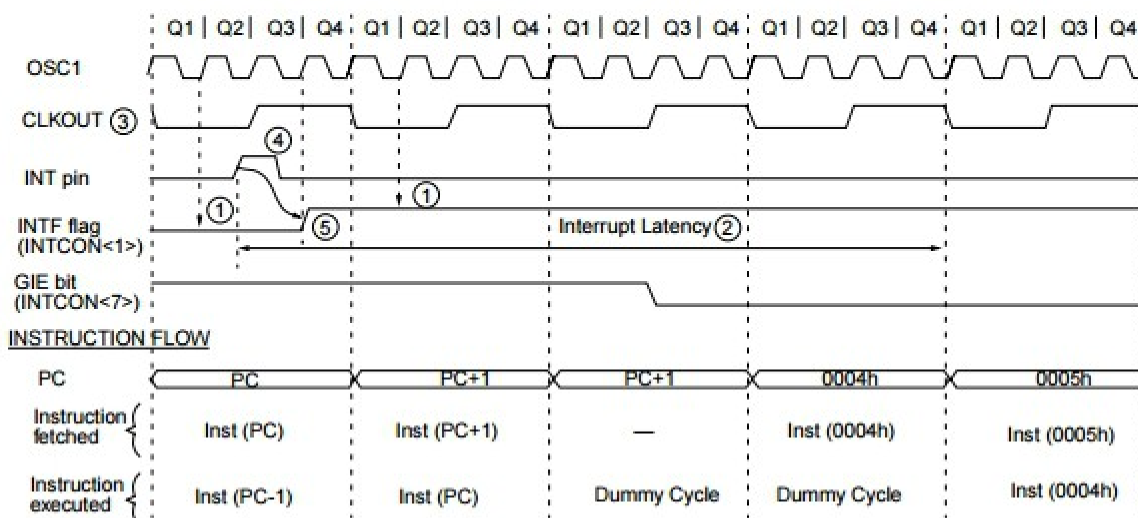
Ma quanto "immediatamente", dato che tutte le azioni del processore sono determinate dal clock del sistema?
In effetti va considerato che esiste una interrupt latency, ovvero un tempo necessario alla logica interna per individuare la richiesta di interrupt e modificare il PC.

Questo tempo, in genere trascurato, può rivestire importanza in applicazioni particolari, dove il progettista ha necessità di controllare che l'interrupt risponde abbastanza velocemente per l'applicazione e che il carico di interrupt non pregiudichi l'esecuzione dell'applicazione oppure si debba tenere conto di questo tempo in applicazioni di precisione, come RTC o misura di frequenze.

Si può essere, quindi, nella necessità di chiedersi: quanto tempo impiega l' esecuzione dell'interrupt?

Va fatta una prima considerazione: il segnale di richiesta di interruzione deve essere sincronizzato con il clock della unità centrale e deve essere completata l'istruzione in corso.

Per i Midrange, questo comporta un tempo di latenza è di 3 cicli istruzione (tcyc) nel caso in cui la periferica chiamante è sincrona con il clock. Nel caso di periferiche "asincrone", come UART o INT, il tempo di latenza può aumentare ad un massimo di 3.75 e dipende dal momento in cui l' evento avviene rispetto all'andamento del clock.
Ad esempio, per il pin **INT**:




Note 1: INTF flag is sampled here (every Q1).
2: Interrupt latency = 3-4 TCY where TCY = instruction cycle time.
Latency is the same whether Instruction (PC) is a single cycle or a 2-cycle instruction.
3: CLKOUT is available only in RC oscillator mode.
4: For minimum width of INT pulse, refer to AC specs.
5: INTF is enabled to be set anytime during the Q4-Q1 cycles.

3.75 potrebbe risultare un numero strano, ma ricordiamo che nei PIC il clock principale è 4 volte il tcyc e le quattro fasi Q1/2/3/4 sono utilizzate dalla logica interna per sincronizzare varie azioni.

**Anche l'esecuzione dell'istruzione retfie impegnerà 2 cicli.**

# Interrupt vs Polling

Interrupt o Polling? Vediamo qualcosa a questo riguardo.

In un computer, **polling** significa determinare lo stato di un dispositivo di I / O con azione diretta della unità centrale che effettua una continua interrogazione del dispositivo. Questo è possibile se le istruzioni hanno cicli molto più brevi dei tempi di risposta della periferica interessata, così da non perdere il momento dell'evento.

Il programma e le caratteristiche della CPU determinano la frequenza del polling.
In apparenza, è una azione molto semplice: ad esempio, nei Baseline, privi di interrupt, possiamo verificare l'overflow del Timer0 (solamente...) controllando quando il contatore TMR0 va a zero. Basterà impostare un loop in cui si verifica questa condizione.

Però, se il clock del timer è quello principale, l'azione diventa problematica, e al limite impossibile, se non viene utilizzato un prescaler, dato che l'avanzamento del timer è pari al ciclo di una istruzione e per il polling più stretto ne occorrono due. E quanto è più piccolo il prescaler, tanto più stretto dovrà essere il loop di verifica del **`TMR0`**.

Lo scotto da pagare per la "semplicità" del polling è l'impossibilità di fare altre operazioni la cui esecuzione superi l'intervallo tra un evento e il successivo verificati dal polling. Un multitask è possibile solo riducendolo ad un loop concatenato di polling e di altre operazioni.
Questo diventa impossibile dove gli eventi sono determinati da dispositivi esterni non dipendenti dal microcontroller, ad esempio il trattamento dati su una linea di comunicazione, la pressione di pulsanti, ecc. E quanto maggiori sono le possibili fonti di eventi, tanto meno diventa possibile il procedere con il polling.

Ad esempio, è possibile, senza difficoltà, impostare algoritmi che risolvano via software trasmissioni seriali, I2C, SPI, con i cosiddetti bit-bang, ma questo consente solamente connessioni half-duplex, impedisce al processore qualsiasi altra attività durante la comunicazione e non consente di raggiungere le frequenze che invece sono possibili usando le periferiche dedicate (USART, MSSP, ecc.) in interrupt.

**Interrupt** è il segnale inviato all'unità centrale per avvisarla di un evento che richiede attenzione.
Il processore deve fermare quanto ha in corso per controllare la periferica.
Solamente se si è liberi dai loop di polling si possono trattare situazioni in cui convergano eventi non sincroni con il programma.

Con una gestione a interrupt ci sono molti vantaggi:

- il microcontroller può servire molti dispositivi e ogni dispositivo può ottenere il servizio in base alla priorità assegnata. In polling questo non è possibile, in quanto più task possono essere seguite unicamente in un loop pre determinato e con temporizzazioni inalterabili.

- si possono escludere o includere dinamicamente da programma le varie periferiche come sorgenti di interrupt, agendo su specifici bit. Questo permette di gestire eventi in modo determinato da quanto definito dal programma per ottenere le migliori prestazioni.

- si ottiene una maggiore precisione negli eventi dipendenti dal tempo RTC, misure di frequenza e periodo, ecc, in quanto è determinabile con sufficiente precisione quanto tempo

passa dall' evento alla sua gestione

- interrupt non richiede alcuna elaborazione quando non accade nulla: diventa possibile un notevole risparmio di energia, mandando il sistema in sleep e risvegliandolo solamente a seguito di un evento. Questa è la soluzione ideale per dispositivi alimentati a batteria.

- non dovendo essere impegnati in polling, la frequenza del clock può essere ridotta, con un ulteriore riduzione del consumo.

- il programma è meno contorto, più leggibile e facile da gestire.

In generale, il microcontroller è più efficiente e può eseguire task concorrenti in modo impossibile con il polling.

Usare polling o interrupt, però, dipende da ogni specifica situazione.

Un polling è facile da comprendere, scrivere e testare (lo diventa molto meno se ci si trova a gestire sovraccarichi di dati, messaggi persi, ecc.).
Per contro, una gestione in interrupt richiede un maggior impegno nella comprensione dei meccanismi degli eventi e della loro organizzazione nel programma. Può essere meno semplice da scrivere e da testare, ma è assai più efficiente ed evita la possibilità di perdita di dati o di effettuare azioni "fuori tempo massimo".

Inoltre, l'uso di interrupt consente di ridurre i consumi energetici, evitando di avere un continuo frullare di istruzioni in loop di polling. Sistemi a batteria possono avere grandi vantaggi da sleep e dall'uso di clock a frequenze basse.

Però, nulla vieta di utilizzare in un programma solamente polling, quando la necessità di interrupt non sia evidente e così pure l'impiegare tanto interrupt quanto pollig nello stesso programma, a seconda delle necessità o della comodità di programmazione. Eventi non critici sul tempo di risposta potranno sempre essere verificati da un polling, come pure si potranno usare i flag IF con il polling, senza attivare interrupt.

**L' importante è acquisire familiarità con i vari metodi e applicarli quando è più logico farlo.**

# FAQs

1. ***Si dice correntemente che la routine di interrupt deve essere la più breve possibile. E' corretto?***
   Questo non del tutto esatto.
   In realtà, la permanenza nella routine deve impegnare un tempo tale da permettere al resto del programma di operare correttamente e la sua durata deve essere inferiore al periodo di ripetizione della richiesta di interrupt, in modo da poter essere servita completamente senza perdita di un evento successivo.
   In generale, un programma è costituito da un loop primario, la cui esecuzione richiede un certo tempo. In generale, quando non si ha modo di valutare i tempi nei dettagli, certamente la routine di interrupt è opportuno sia la più breve possibile. Per certo, deve essere più breve del loop principale, limitandosi alla gestione dell' evento e rimandando con flag e buffer l'elaborazione dei dati al processo principale
   Però, va anche tenuto conto che l'interruzione deve capitare con una frequenza tale da permettere il completamento delle altre operazioni.
   Ad esempio, se dobbiamo effettuare il refresh di un display ogni 10ms, possiamo usare un timer per ottenere questa cadenza di tempo per le chiamate interrupt, ma l'operazione di refresh deve ovviamente durare meno di 10ms. Se abbiamo ridotto la durata di esecuzione della routine di interrupt ad un minimo di 2ms, ma abbiamo altre sorgenti di interruzione che, per numero o per frequenza di intervento, sono tali da richiamare l'interruzione ogni ms, il programma sarà bloccato in queste gestioni senza poter fare altro.

2. ***Si sostiene che interrupt va utilizzato con eventi "non frequenti" o casuali. E' esatto?***
   No, per nulla. L' affermazione non ha senso.
   Cosa vuol dire "non frequenti"? Che capitano una volta al giorno? o una volta al mese? o una volta al minuto?
   L'arrivo di una trasmissione da una linea seriale può essere "poco frequente" nel complesso del tempo di funzionamento del processore, ma quando arriva una stringa composta da decine di caratteri, questi si susseguono alla cadenza del baud rate per un certo tempo. E in questo tempo non sono affatto "non frequenti". E in un RTC la cadenza di interrupt del timer preposto sarà per nulla casuale.
   La "frequenza" dell' evento interrupt può essere qualsiasi.
   Il suo solo limite è quello di non poter capitare con una cadenza tale da impedire l'esecuzione della sua gestione. Come nella questione precedente, se il ciclo di istruzioni è 1us, non sarà possibile seguire eventi che capitano ogni 100us, assieme ad altre operazioni che richiedono 150 istruzioni; non ce ne sarebbe il tempo. Se, però, il ciclo di istruzione diventa 50ns, questo può diventare possibile. Non sono questioni di "assoluti", ma sono valutazioni da fare relativamente ad ogni situazione specifica.

3. ***Si legge che interrupt è meglio in quanto evita di sprecare i cicli di CPU del polling...***
   La questione è priva di senso.
   Se sto facendo un polling, un loop di istruzioni è certamente sempre in esecuzione.
   Se utilizzo un interrupt, posso mettere il processore in sleep ed attendere a consumo minimo l'evento programmato. In questo senso l'azione dell' interrupt è vantaggiosa.

   Se, però, siamo in situazioni diverse, non esiste mai "uno spreco di istruzioni" (se chi scrive il sorgente ha una minimo buon senso e competenza...).
   Le istruzioni del loop di un polling servono a verificare il bit voluto; non ne posso mettere di meno. Così come un ciclo di delay basato sull'esecuzione di istruzioni (waste time) non

spreca nulla: le istruzioni eseguono il tempo di attesa richiesto; non ne posso mettere nè di meno, nè di più. E un ritardo non è uno spreco di istruzioni, ma una necessità del programma che gestisce una determinata azione.
Certamente, usando interrupt, potrò realizzare attese con i timer, lasciando libera l'unità centrale di svolge altre attività in attesa dell'overflow. Ma dove questo non serve, polling e waste time sono perfettamente adeguati.
Diverso sarà il caso in cui occorra una minimizzazione della corrente assorbita dall' applicazione.

La scrittura di un programma efficiente molto probabilmente comprenderà meno istruzioni di uno inefficiente; sta al programmatore stendere sorgenti adeguati all' applicazione, ma che **non diventino contorti e criptici nel tentativo di impiegare il minor numero di istruzioni**. Il sorgente deve essere anche sufficientemente elegante e leggibile da poter essere compreso, modificato e aggiornato con facilità.

4. ***L'interrupt è critico e complicato da gestire...***
   In primo luogo, se si ricorresse con maggiore assiduità alla stesura di un diagramma di flusso prima di iniziare a scrivere istruzioni, si risolverebbero molti dubbi.
   La "complicazione" nella gestione di eventi in interrupt piuttosto che in polling consiste solamente nella necessità di un approccio logico un poco diverso. Superato questo, non esiste alcuna reale difficoltà. Anzi, in generale, la gestione in interrupt di task concorrenti è logicamente molto più semplice e immediata di una soluzione in polling.
   Il concetto guida è quello di far eseguire nella routine interrupt esclusivamente la gestione della periferica interessata, lasciando al main le operazioni logiche conseguenti. Ad esempio, la gestione di un UART in ricezione consisterà nello spostare il carattere ricevuto in un buffer e segnalare l'evento con un flag. Sarà poi compito della logica del main il trattare opportunamente il carattere.
   Quanto alla criticità, con questo solitamente si riduce al fatto di non aver ben chiaro il problema del flag IF; tipicamente, lo scordarsi di cancellarlo crea situazioni di mancato funzionamento del software, apparentemente difficili da diagnosticare. Basta solo comprendere che i flag sono bandierine "alzate" per dire alla logica del programma che è accaduto un evento e che sta a noi "abbassare" una volta identificate.

   Certamente potranno esserci casi in cui siano convergenti numerose sorgenti di interrupt, il che richiede un certo impegno nello studio della migliore struttura logica per far convivere correttamente le varie azione.
   Oppure si ha a che fare con tempi di intervento critici, in quanto di ordine vicino a quello del ciclo di istruzione; in queste condizioni, se non è possibile un aumento della frequenza del clock, occorre una certa cura nella costruzione dei vari algoritmi e loop. Ma questi casi non hanno direttamente a che fare con la struttura dell'interrupt; piuttosto, con la necessità di seguire hardware complessi o veloci. E, per altro, si tratta di condizioni che non sarebbe possibile in alcun caso affrontare in polling.

5. ***Si parla di gestione degli interrupt su PIC12, !6, ecc...***
   In questo c'è poca correttezza.
   La sigla dei PIC identifica in qualche modo a quale famiglia appartiene il componente, ma esistono PIC12 e PIC16 tanto nei **Midrange** (es 12F629) quanto nei **Baseline** (es.12F509), solo che questi ultimi non dispongono di interrupt....
   E' anche errato limitare i Baseline ai PIC10F: lo sono, senza gestione interrupt, ad esempio, i PIC10F200, ma i PIC10F320 hanno gestione interrupt e non sono Baseline.

6. ***I pic della serie MidRange ... salvano in automatico ...i registri W e STATUS in registri temporanei ...***
   Proprio per nulla. I Midrange non salvano alcunchè.
   Il ***Program Counter*** viene “salvato”, per qualsiasi PIC, nello **stack** , dove trovano posto gli indirizzi di ritorno dalle chiamate **call**. Questo fa parte dei meccanismi di esecuzione dell'istruzione **call, return, retlw 0, retfie** e della chiamata hardware di interrupt.
   Oltre a questa azione, i Midrange non operano alcun salvataggio automatico del contesto: deve essere l'utente ad eseguire quanto gli necessita.
   Nel segmento degli 8bit, sono gli **Enhanced Midrange** e i **PIC18F** ad operare un salvataggio ed un ripristino automatico di alcuni registri in locazioni particolari.
   Il fatto che alcuni compilatori C provvedano ad eseguire salvataggio e restore in modo trasparente all'utente non ha niente a che vedere con il funzionamento del chip, ma è una azione software aggiunta.

7. ***Si può chiarire meglio il problema dell'uso di swapf con il salvataggio dello status?***
   La necessità di usare **`swapf`** per movimentare lo STATUS è richiesta da una particolarità della più comune istruzione **`movf`**
   Se verifichiamo la sezione relativa alle istruzioni di un qualsiasi foglio dati Midrange, troviamo questa indicazione:

| MOVF | f, d | Move f | 1 | 00 1000 dfff ffff | Z |
|---|---|---|---|---|---|

Questo vuol dire che, ogni volta che eseguiamo un **`movf`**, modifichiamo nel contempo il flag Z dello **`STATUS`**:

- se il file oggetto dell'istruzione ha valore 0, il flag Z viene settato
- se ha valore diverso da 0, il flag Z viene azzerato

In questo modo è possibile usare l'istruzione come test del contenuto del file:

```
movf  target,f    ; test se target = 0
skpz              ; si - salta prossima istruzione
```

Ora, se applichiamo l'istruzione movf sullo STATUS, otteniamo un risultato indesiderato:

```
movf  STATUS,w
```

- **se il contenuto di `STATUS` è diverso da 0, il flag Z viene azzerato**.
  Così, se Z era a 1, viene cancellato
- **se il contenuto di `STATUS` è 0, il flag Z viene settato**.
  Così, se Z era a 0, viene portato a 1.

Se il flag **Z** era usato nell'elaborazione attiva nel main al momento della interruzione, il ripristino con un valore errato alla fine della routine di interrupt fa si che si creino errori non facilmente risolvibili.

Occorre far si che lo spostare **`STATUS`** in una locazione di backup e poi recuperarlo, non crei errori.

Si ricorre, allora, all'istruzione swapf:

| SWAPF | f, d | Swap nibbles in f | 1 | 00 1110 dfff ffff | |
|---|---|---|---|---|---|

Osserviamo che **`swapf`** non modifica alcun flag dello **`STATUS`**, quindi:

```
swapf   STATUS,w     ; swap dello status
movwf   status_tmp   ; salva STATUS
```

fa si che lo **STATUS** sia poeto nella locazione voluta senza essere modificato.

Certamente, **swapf** provoca lo scambio dei nibble, ma basta usare ancora la stessa istruzione per ripristinare lo **STATUS** correttamente.

```
swapf    status_temp,w    ; restore STATUS
movwf    STATUS           ;
swapf    w_temp,f         ; restore W
swapf    w_temp,w         ;
```

Va notato che è necessario usare **swapf** anche per ripristinare **W**, sempre per il problema dell'azione del **movf** su flag **Z**; l'istruzione è ripetuta due volte, dato che **W** era stato salvato con un **movwf** (che non altera lo **STATUS**), senza lo scambio dei nibble.

La documentazione di Microchip presenta spesso la versione:

```
movwf    w_temp           ; salva W
movf     STATUS,W         ; copia STATUS in W
```

senza particolari commenti che chiariscano il problema.

In effetti, esistono delle ragioni che supportano anche una gestione come questa:

```
INTERRUPT
        movwf    w_temp           ; salva W
        movf     STATUS,W         ; copia STATUS in W
; movf modifica lo STATUS successivo all'istruzione, ma quello precedente
; è già copiato in W

        banksel status_temp      ; cambia banco
        movwf    status_temp      ; salva STATUS (in W) nella RAM
        movf     PCLATH,W         ; salva PCLATH
        movwf    pclath_temp
        ....
ISR_end:
        movf     pclath_temp,W    ; restore PCLATH
        movwf    PCLATH           ;
        movf     status_temp,W    ; restore STATUS
        movwf    STATUS           ;
        swapf    w_temp,F         ; restore W senza modificare STATUS
        swapf    w_temp,W         ;
        retfie                    ;
```

Questo perchè in **movf STATUS,W** la lettura dallo **STATUS** avviene nel segmento di tempo **Q2** del ciclo di esecuzione a 4 tempi dell'istruzione, mentre il flag **Z** viene aggiornato nel successivo **Q4**: in quel momento, però, il vecchio valore di **STATUS** è già copiato in W.
Non è una cosa ovvia, ma si può dedurre dall'applicazione ***DS33023A***.

Dal punto di vista pratico non c'è alcuna differenza nel numero delle istruzioni richieste tra questa versione e quella con l'uso di **`swapf`**. Si può ritenere preferibile la versione con swapf.

---

# Esempi di subroutine save/restore (Midrange).

**Sotto forma di subroutine rilocabile**. RAM shared.
Va usata con il Linker.

```
;*********************************************************************
; Savereg.asm
; Titolo       : Subroutine salvataggio registri per interrupt.
;                Versione rilocabile.
;
; PIC           : 8bit - Midrange
; Supporto      : MPASM
; Versione      : 1.0
; Data          : 01-05-2013
; Ref. hardware :
; Autore        : afg
;
;*********************************************************************
;
NOLIST
;---------------------------------------------------------------------

 #include <p16F690.inc> ; qualsiasi Midrange

    GLOBAL  status_temp
    GLOBAL  pclath_temp
    GLOBAL  fsr_temp
    GLOBAL  w_temp

 SAVE_ALL        udata_shr
 w_temp          res 1       ; area salvataggio contesto
 status_temp     res 1       ; 4 bytes
 pclath_temp     res 1
 fsr_temp        res 1


;**********************************************************************
RESVEC     CODE   0x00
      nop                     ; richiesto se si usa ICD
      goto    Main

     GLOBAL    ISR_entry, ISR_done
     EXTERN    isr_select

ISRVECTOR  CODE   0x04
ISR_entry:
      movwf    w_temp          ; salva W
```

```
      swapf    STATUS,w         ; copia STATUS in W
      banksel  BANK0            ; banco 0 per il resto della ram
      movwf    status_temp      ; salva STATUS
      movf     PCLATH,w         ; salva PCLATH
      movwf    pclath_temp      ;
      movf     FSR,w            ; salva FSR
      movwf    fsr_temp         ;
      movlw    HIGH isr_select ; aggiusta PCLATH
      movwf    PCLATH           ;
      goto     isr_select
      ....
 ISR_done:
      banksel  BANK0
      movf    fsr_temp,w
      movwf   FSR              ; restore FSR
      movf    pclath_temp,w    ; restore PCLATH
      movwf   PCLATH           ;
      swapf   status_temp,w    ; restore STATUS
      movwf   STATUS           ;
      swapf   w_temp,f         ; restore W
      swapf   w_temp,w         ;
      retfie                   ; return from interrupt

;*******************************************************************
 LIST
        END
```

**Sotto forma di macro**. RAM shared.
Va inclusa all'inizo del sorgente.

La forma macro è preferibile perchè consente di risparmiare i cicli istruzione della **call** e del **return.**

```
;*******************************************************************
; Savemacro
; Titolo        : Macro salvataggio registri per interrupt.
;
; PIC           : 8bit - Midrange
; Supporto      : MPASM
; Versione      : 1.0
; Data          : 01-05-2013
; Ref. hardware :
; Autore        : afg
;
;*******************************************************************
;
;-------------------------------------------------------------------

Saveram    UDATA_SHR           ; RAM shared
w_temp      res 1              ; 4 locazioni
status_temp res 1
fsr_temp    res 1
pclath_temp res 1
```

```
       CODE
SAVEREG   MACRO
        movwf  w_temp          ; salva W
        swapf  STATUS,w        ; swap status
        movwf  status_temp     ; e salva
        movf   FSR , w         ; salva FSR
        movwf  fsr_temp
          ENDM

RESTOREG  MACRO
        movf   fsr_temp , w     ; recupera FSR
        movwf  FSR
        swapf  status_temp,w    ; recupera STATUS salvato
        movwf  STATUS           ; senza modificare Z
        swapf  w_temp,f         ; recupera W senza modificare
        swapf  w_temp,w         ; lo STATUS
          ENDM

;*********************************************************************
```

La stessa con ram in banco 0, per PIC con RAM da 00h a 7Fh :

```
;*********************************************************************
; Savemacro
; Titolo         : Macro salvataggio registri per interrupt.
;
; PIC            : 8bit - Midrange
; Supporto       : MPASM
; Versione       : 1.0
; Data           : 01-05-2013
; Ref. hardware  :
; Autore         : afg
;
;****************************************************************
;
NOLIST
;--------------------------------------------------------------------

SaveW      UDATA_SHR          ; RAM shared
w_temp      res 1             ; 1 locazione

Saveram    UDATA  0x6D        ; ultime tre locazioni di banco 0
status_temp res 1             ; prima dell'area shared
fsr_temp    res 1
pclath_temp res 1


    CODE
SAVEREG   MACRO
       movwf  w_temp          ; salva W
       swapf  STATUS,w        ; swap status
       clrf   STATUS
       movwf  status_temp     ; e salva
```

```
        movf   FSR , w        ; salva FSR
        movwf  fsr_temp
          ENDM

 RESTOREG  MACRO
        clrf   STATUS
        movf   fsr_temp , w    ; recupera FSR
        movwf  FSR
        swapf  status_temp,w   ; recupera STATUS salvato
        movwf  STATUS          ; senza modificare Z
        swapf  w_temp,f        ; recupera W senza modificare
        swapf  w_temp,w        ; lo STATUS
          ENDM

 ;*********************************************************************
```

Ovviamente si potranno adattare gli esempi alle proprie necessità, sempre considerando quanto detto prima. Ad esempio, se non è usato in nessuna parte del programma, non occorre salvare FSR.

Comunque, in generale, è meglio attenersi ad una unica forma comune a tutti i casi piuttosto che "personalizzare" ogni sorgente, col rischio di non avere più ben chiaro cosa si sta facendo. Il risparmio di un paio di righe di istruzione, in genere, non influisce minimamente sull'esecuzione e non vale il rischio di ritrovarsi con un programma non funzionante.

# Un breve riassunto.

- I **Midrange sono dotati di interrupt**

- L'interruzione è una sospensione del flusso principale per passare all'esecuzione di un tratto di programma che inizia alla **locazione 04h ( vettore di interrupt)**.

- E' analogo ad una subroutine, ma non viene chiamato da **call**, bensì da un evento hardware che interessa una periferica. Il rientro avviene, d'obbligo, con l'istruzione **retfie**.

- Praticamente tutte le periferiche integrate possono chiamare interrupt in seguito ad un particolare evento: ad esempio, per i timer, l'evento è l'overflow; per un UART, la ricezione o il completamento della trasmissione di un carattere, ecc.

- **Le periferiche che generano interrupt hanno due bit di controllo:**
  - un bit **xxIE** che, posto a 1, abilita l'interruzione. Si trova nei registri **PIE**.
  - un bit **xxIF** che segnala, andando a 1, l'evento avvenuto. Si trova nei registri **PIR**.
  Questo permette di abilitare solo le sorgenti di interrupt richieste dall'applicazione e di capire quale ha generato la chiamata.
  Possono esserci più registri **PIE/PIR** a seconda del numero delle periferiche integrate.

- Esiste un bit di abilitazione generale **GIE**. Per ottenere un interruzione all'evento, occorre che sia il bit **IE** relativo alla periferica che il bit **GIE** siano a 1.
  Il bit **GIE** a 0 blocca qualsiasi interruzione.
  Da osservare che il flag **IF** viene settato all'evento anche se l'interrupt non è abilitato

- Le periferiche integrate, come fonte di interrupt, sono divise in due blocchi:
  - **interrupt non periferici**
  Riguardano le periferiche presenti in tutti i chip, ovvero **Timer0, INT, Pin Level Change e scrittura EEPROM.** I bit **IF** e **IE** di queste periferiche sono contenuti nel registro **INTCON**.
  - **interrupt periferici**
  Riguardano tutte le altre periferiche che hanno i bit **IF** e **IE** nei registri **PIR/PIE**.
  Per avere interruzione da queste occorre agire anche su un altro bit, detto **PEIE**.

  Quindi
  **- le fonti non periferiche richiedono IE e GIE a 1**.
  **- quelle periferiche richiedono IE, PEIE e GIE a 1**.

- **I flag indicatori IF vanno obbligatoriamente cancellati da programma una volta utilizzati; se questo non viene fatto si avrà un blocco del programma che continua a richiamare l'interruzione**

- Entrando nella gestione dell'interruzione, **occorre salvare i principali registri e ripristinarli all'uscita.**

- La lunghezza della routine scritta per gestire le interruzioni deve essere proporzionata alla frequenza delle interruzioni e lasciare sufficiente tempo per lo svolgimento del resto delle istruzioni del programma.

- Può essere necessario consultare i fogli dati del componente se si intende usare la sorgente di interrupt in sleep: alcune periferiche non sono operative in questa modalità. Altrettanto per conoscere la situazione dei registri dopo un reset.

- La stesura di un diagramma di flusso semplifica la scrittura delle istruzioni e riduce gli errori.

# Altre informazioni.

- Microchip DS31008a
- Microchip DS31001a
- Microchip AN566