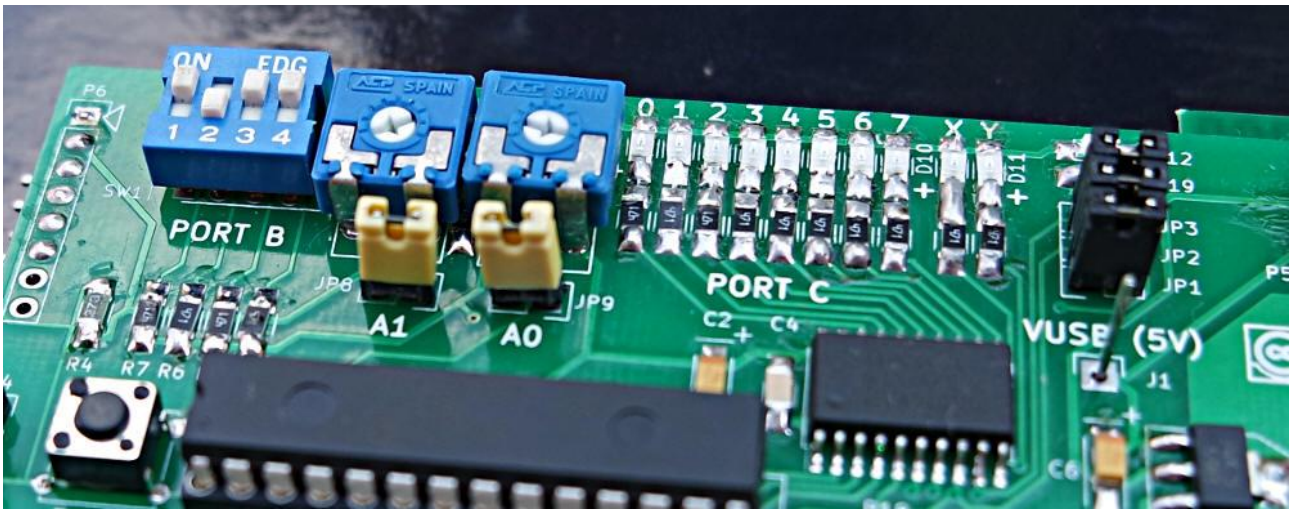
PIC18: subroutines in assembly



Una subroutine (o anche *routine* o *sottoprogramma*) è una sequenza di istruzioni assembly che realizza un determinato compito.

Questa sequenza non è autonoma, ma viene *chiamata* (`call`) all'interno di un generico programma; al termine della esecuzione della subroutine, il **P**rogram **C**ounter *ritorna* (`return`) al suo valore precedente, per eseguire l'istruzione immediatamente seguente la `call`.

Una subroutine è generalmente identificata da una *label* e deve terminare con l'istruzione `return`.

Un esempio semplice

Il seguente esempio d'uso mostra il programma principale `main` al cui interno, per due volte, è invocata la stessa subroutine `delay`:

```
12  #include "p18f26k20.inc" ; il file con la definizine dei registri specifici del PIC in uso
13
14      UDATA_ACS
15  contatore res 2          ; Variabile a 16 bit usata per il ritardo
16
17  main    CODE    0x0000  ; vettore di reset, indirizzo 0. Il codice inizia qui
18
19      movlw 0              ; azzera il registro W (accumulatore)
20      movwf TRISC          ; copia il contenuto del registro W nel registro speciale TRISC
21                           ; (imposta PORTC come uscita)
22  superloop
23      movlw  0xAA          ; copia in W il numero binario 1010 1010
24      movwf LATC           ; copia il contenuto del registro W nel registro speciale LATC
25      call delay
26
27      movlw  0x55          ; copia in W il numero binario 0101 0101
28      movwf LATC           ; copia il contenuto del registro W nel registro speciale LATC
29      call delay
30
31      bra superloop
32
33  delay
34  ; **** Ciclo di ritardo - un quarto di secondo circa @ 1 MHz ****
35      movlw 0x30
36      movwf contatore+1
37      clrf contatore
38  ripeti
39      decfsz contatore
40      bra ripeti
41      decfsz contatore+1
42      bra ripeti
43  ; **** Fine del ciclo di ritardo ****
44      return
45
46      END                  ; Fine del file
```

Il funzionamento è intuitivo:

- Il codice alle righe 23-24 accende i quattro LED 7, 5, 3 e 1
- L'istruzione alla riga 25 (`call delay`) porta il *program counter* alla riga 33
- L'esecuzione delle righe da 33 a 42 richiede circa un secondo.
- L'istruzione alla riga 44 fa tornare il *program counter* alla riga 26 (successiva della 25)
- Il codice alle righe 27-28 accende inverte l'accensione degli otto LED
- L'istruzione alla riga 29 (`call delay`) porta il *program counter* alla riga 33
- L'esecuzione prosegue e l'istruzione alla riga 44 fa tornare il program counter alla riga 30 (successiva alla 29).

Alcune osservazioni:

- La stessa istruzione `return` di riga 44 una volta fa "tornare" il programma alla riga 26 ed una volta alla riga 30, in relazione alla posizione della corrispondente `call`
- L'indirizzo a cui il *program counter* deve tornare al termine della esecuzione della subroutine viene salvato nello *stack hardware* del PIC18 (*return address stack*). Questa soluzione permette di richiamare una subroutine da più punti diversi di un programma, garantendo sempre il ritorno all'istruzione successiva alla `call`
- La profondità dello stack hardware del PIC18 è pari a 31 e quindi occorre limitare a tale valore il numero di subroutine che richiamano subroutine che richiamano subroutine...

Vediamo nel dettaglio il funzionamento dell'istruzione `call`, con riferimento ai fogli tecnici:

CALL	**Subroutine Call**
Syntax:	CALL k
Operands:	$0 \leq k \leq 1048575$
Operation:	$(PC) + 4 \rightarrow TOS$, $k \rightarrow PC<20:1>$,
Description:	Subroutine call of entire 2-Mbyte memory range. First, return address (PC + 4) is pushed onto the return stack. Then, the 20-bit value 'k' is loaded into PC<20:1>.

- L'istruzione `call` ha un solo operando, un numero tra 0 e 1 048 575, cioè un qualunque indirizzo compreso all'interno della memoria FLASH (un milione di parole da 16 bit, cioè 2 MB). Questo operando è l'indirizzo a cui si trova la subroutine ed in genere è indicato da una label (`delay` nell'esempio)
- La `call` memorizza in cima allo *stack hardware* (**T**op **O**f **S**tack) l'indirizzo dell'istruzione successiva, cioè l'istruzione a cui il codice dovrà tornare dopo la conclusione della subroutine. Questa operazione è a volte indicata come *Push*

- Il valore di `k` viene messo nel **P**rogram **C**ounter e quindi l'esecuzione del codice passa alla subroutine

Vediamo nel dettaglio il funzionamento dell'istruzione `return`, con riferimento ai fogli tecnici:

RETURN	Return from Subroutine
Syntax:	RETURN
Operation:	(TOS) → PC,
Description:	Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter.

- L'istruzione `return` non ha argomenti
- La `return` preleva dalla cima allo *stack hardware* (**T**op **O**f **S**tack) l'indirizzo a cui ritornare, indirizzo precedentemente memorizzato dalla `call` corrispondente. Questo indirizzo viene posto nel **P**rogram **C**ounter, permettendo di tornare all'istruzione successiva alla `call`

Un secondo esempio

La soluzione proposta nel primo esempio ha vari difetti:

- Richiede permanentemente l'uso di due celle di memoria RAM, risorse limitata, anche quando la subroutine non è in esecuzione. Si pensi agli effetti in un programma di grosse dimensioni, con centinaia di variabili
- Come effetto collaterale, riscrive il contenuto del registro `STATUS` (i *flag*), distruggendo il precedente valore.
- (non in questo caso) Potrebbe riscrivere i contenuti anche dei registri `WREG` e `BSR`

Il prossimo esercizio crea, a livello umano, un ritardo molto breve. Un'idea potrebbe essere quella di scrivere una subroutine che richiama decine di volte `ritardo_breve`, per ottenere un ritardo più lungo. Per esempio il

codice seguente mostra `ritardo_lungo_1`, ovviamente dalla durata di circa 30 ms

```
18    superloop
19        incf LATC
20        call ritardo_lungo_1
21        bra superloop
22
23    ritardo_lungo_1
24        call ritardo_breve
25        call ritardo_breve
26        call ritardo_breve
27        call ritardo_breve
28        call ritardo_breve
29        call ritardo_breve
30        call ritardo_breve
31        call ritardo_breve
32        call ritardo_breve
33        call ritardo_breve
34        return
```

Si tratta di una soluzione molto rigida e ben poco elegante; sarebbe più comodo usare un *loop* che richiama `ritardo_breve` qualche decina di volte, decrementando per esempio da venti a zero un'apposita variabile `contatore`:

```
25    ritardo_lungo_2
26        movlw .20
27        movwf contatore
28    ripeti_2
29        call ritardo_breve
30        decfsz contatore
31        bra ripeti_2
32        return
```

Ancora meglio sarebbe usare `WREG` invece che `contatore`, per non sprecare una cella di memoria. Peccato che tale soluzione non funziona (perché? La risposta poco più avanti, ma vale la pena provare a rispondere usando la propria intelligenza...).

Una subroutine, per essere davvero utilizzabile, non deve come effetto collaterale modificare i registri che descrivono lo stato del processore, cioè i registri `WREG`, `STATUS` e `BSR`.

Una possibile soluzione è quella di creare alcuni *registri temporanei* che permettono di salvare i valori di tali registri all'inizio del codice della subroutine e di recuperarli al termine.

Questa tecnica è mostrata negli esempi **Delay_subroutine.asm**

Questo programma contiene due subroutines:

- La prima `ritardo_breve` è sostanzialmente identica a quelle appena mostrate, se non nell'uso di registri temporanei per salvare `WREG` e `STATUS`
- La seconda `ritardo_lungo`, invoca per 100 volte la subroutine `ritardo_breve` per ottenere un ritardo di circa 300 ms, significativo anche a livello umano

Il salvataggio nei registri temporanei di `WREG` e `STATUS` è reso necessario dal fatto che entrambe le subroutines usano gli stessi registri: non salvarli avrebbe causato un'interferenza tra le due subroutines e, in definitiva, il loro mancato funzionamento.

Si noti che per quanto riguarda `ritardo_lungo` non è stato implementato alcun meccanismo di registri temporanei; sarebbe stata necessaria la creazione di ulteriori registri temporanei, con ulteriore spreco di spazio e potenziali rischi difficili da controllare nel caso di omonimie.

Un esempio avanzato

L'ultimo esempio mostra l'utilizzo dello **stack software** per il salvataggio dei registri: in pratica ogni variabile che verrà modificata all'interno della subroutine viene salvata all'inizio e recuperata prima del `return`.
Questo esempio è contenuto nel programma **Delay_Stack.asm**

Come prassi tutte le subroutines iniziano con il *salvataggio del contesto* nello stack software:

```
movwf POSTDEC1          ; salva WREG nello stack
movff STATUS, POSTDEC1 ; salva i flags nello stack
movff BSR, POSTDEC1     ; salva BSR nello stack
```

Terminano con il *ripristino del contesto*, prelevando i contenuti originali dei registri dalla stack software:

```
movff PREINC1, BSR     ; ripristina il registro BSR
movff PREINC1, STATUS ; ripristina i flags
movf  PREINC1, W       ; ripristina il registro WREG
```

Si noti la simmetria tra l'ordine di salvataggio e quello di ripristino; non ha invece particolare importanza l'ordine di salvataggio, purché venga mantenuta la simmetria al momento del ripristino.

Questo esempio può essere considerato una soluzione definitiva di ritardo.