# 1 Programmi modulari in Assembly

In questo capitolo si tratta dei programmi che sono costituiti di componenti racchiusi in file separati, eventualmente scritti in linguaggi di programmazione diversi.

La principale utilità del linker, già discussa nel volume precedente, sta nel fatto che può collegare programmi compilati in tempi, luoghi e con strumenti diversi.

Un programma può quindi essere composto da diversi "**moduli**", ciascuno dei quali viene compilato in modo del tutto indipendente dagli altri. Il linker ("collegatore") ha proprio il compito di mettere assieme i file oggetto prodotti a partire dai moduli e di realizzare un unico programma eseguibile.

Nel capitolo illustreremo come realizzare programmi divisi in moduli usando diversi strumenti di sviluppo.

Si definisce "modulo" di un programma una sua parte che può essere compilata indipendentemente dalla altre.

## *1.1 Linking statico*

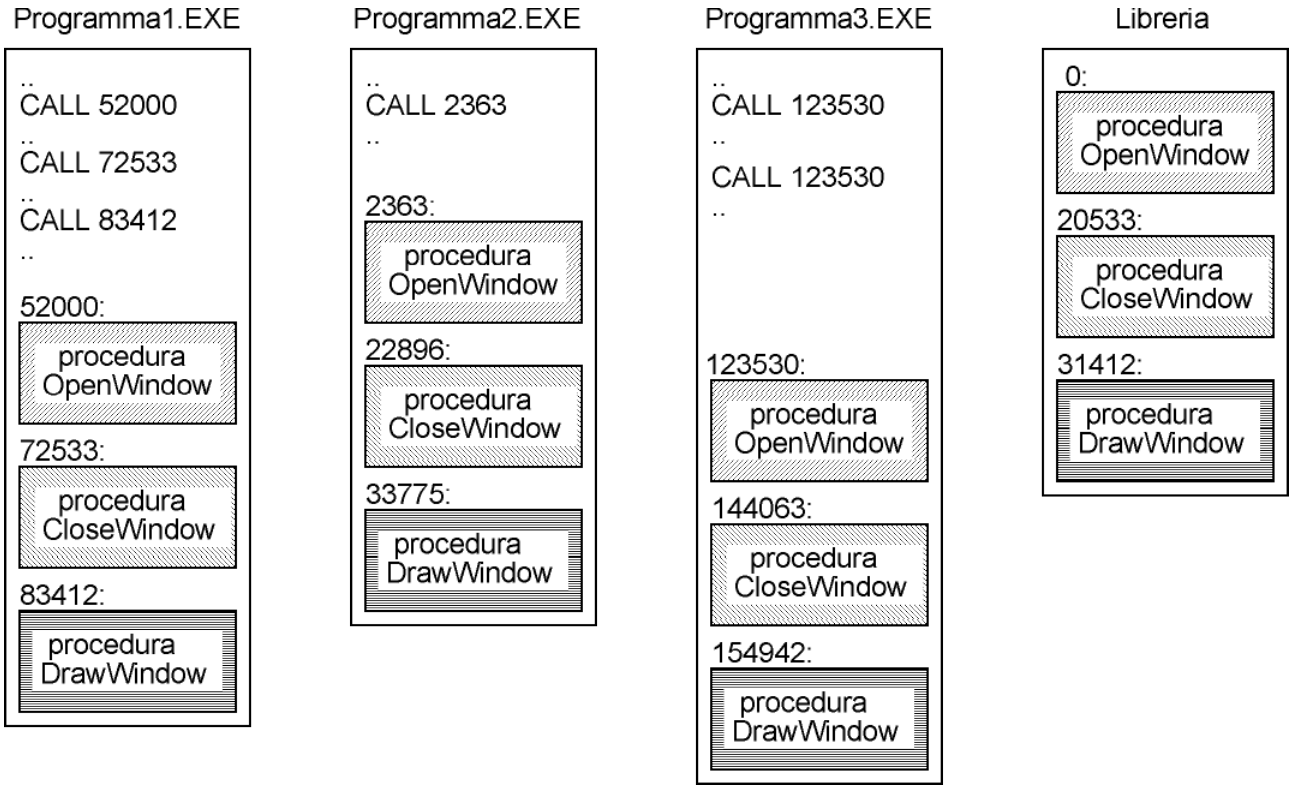



Figura 1 linking statico o dinamico

Se un programma suddiviso in file diversi deve diventare un unico file eseguibile è necessario che nei diversi file si possa far riferimento a locazioni di memoria "esterne", definite in altri file. Il riferimento può riguardare sia un'istruzione, come p.es. l'indirizzo di una procedura, sia un dato, come p.es. una "variabile" condivisa fra due moduli.
La presenza di un riferimento esterno al modulo significa che il compilatore non ha modo di sostituire un indirizzo ad un'etichetta, dato che la sua definizione è in un altro file ed il compilatore traduce un file alla volta.

È compito del linker collegare tutti i riferimenti esterni che trova nei vari moduli che costituiscono il programma.

Se questo collegamento è costituito da un indirizzo esso viene definito **link statico**.

Se un programma usa un linking statico tutti i riferimenti esterni sono sostituiti con i relativi indirizzi a tempo di collegamento (link time).
Questo tipo di collegamento (**static linking**) è molto efficiente, perché ogni chiamata a procedura salta direttamente al giusto indirizzo ove è il codice ed ogni locazione di memoria utilizzata dal programma per i dati è accessibile con indirizzamenti assoluti.

## *1.2 Link dinamico*

Un metodo di collegamento meno efficiente ma per molti versi più flessibile è il "**linking dinamico**", modo di collegamento sul quale sono basati molti importanti servizi dei Sistemi Operativi (S.O.) moderni.
Supponiamo che il codice di una libreria debba essere usato da tutti i programmi che sono in esecuzione "contemporaneamente" su un Sistema Operativo multiprogrammato (*), che possono anche essere molte decine.
<NOTA>
(*) vedi nel seguito per la definizione di S.O. monoprogrammato e multiprogrammato
</NOTA>
!!!!AGGIUSTARE!!!!
Se la libreria viene collegata in modo statico si presenta il problema !!!!!!! che ognuno dei programmi che usa la libreria ha al suo interno una sua copia della libreria, per cui il codice della libreria occupa la memoria per tante volte quanti sono i programmi che lo usano contemporaneamente.
Questo non è un problema se il Sistema Operativo è monoprogrammato (*), in quanto può eseguire un solo programma alla volta ed il programma precedentemente eseguito viene scaricato dalla memoria prima che esegua il successivo.
Quando invece il S.O. è multiprogrammato molti programmi possono essere presenti contemporaneamente nella memoria principale del computer, per cui la replicazione del codice delle librerie è uno spreco di risorse.

Per risolvere questo problema è stato introdotto il concetto di "**libreria dinamica**" o "**libreria condivisa**" (in Windows **DLL** (**D**ynamic **L**ink **L**ibrary, in Linux "**shared library**"; to share = condividere).
Questo meccanismo, sia pur molto più complicato e meno veloce del linking statico, permette di risparmiare memoria, dato che il codice della DLL viene caricato una sola volta ed usato da molti programmi contemporaneamente.

### 1.2.1 DLL in Windows

Una libreria dinamica è una collezione di procedure, funzionalmente non dissimile da tutte le altre librerie; diverso è il modo con cui si accede alle sue procedure.
Quando il linker incontra un riferimento esterno ad una procedura di una libreria dinamica non lo risolve in un indirizzo, ma lascia nel file eseguibile in riferimento simbolico, che dovrà specificare il nome della procedura che si vuole eseguire ed il nome della libreria in cui la si deve cercare.
Il programma eseguibile mantiene perciò "in sospeso" le chiamate alle procedure che appartengono a DLL; esse che salteranno ad un indirizzo che ancora non si conosce.
Per conoscere quali sono le procedure contenute in una DLL il linker necessita di una "import library", un file .LIB che viene prodotto quando la DLL viene generata. Con le informazioni contenute nella import library il linker è in grado di completare l'eseguibile, lasciando al suo interno i riferimenti sufficienti perché l'indirizzo delle procedure possa essere trovato a tempo d'esecuzione.
Quando il programma va in esecuzione esegue una chiamata al S.O. richiedendo l'indirizzo delle procedure che utilizza che sono contenute in librerie dinamiche (in Windows la chiamata è "GetProcAddress").
Il S.O. tiene traccia di tutte le DLL che ha caricato in memoria.
Se la DLL richiesta è già in memoria, perché già in uso da un altro programma, il S.O. restituisce al programma richiedente l'indirizzo della procedura.
Se invece la DLL non è ancora stata caricata, allora il S.O. la cerca sull'hard disk (il programma deve passare anche il nome simbolico della DLL ed eventualmente il percorso di directory da utilizzare), la carica in memoria e restituisce l'indirizzo della procedura al programma che l'aveva richiesto. Il programma potrà ora eseguire.
Nel suo lavoro con le DLL il S.O. tiene traccia anche del numero di programmi che hanno fatto richiesta di una specifica DLL. Ogni volta che un nuovo programma fa una richiesta di quella DLL il S.O. incrementerà un contatore, che rappresenterà il numero di programmi che stanno attualmente usandola.
Ogni volta che un programma che sta usando la DLL comunica al S.O. che ha terminato la sua esecuzione il contatore verrà decrementato. Quando il contatore assume il valore di zero il S.O. cancellerà la DLL dalla memoria, dato che nessun programma ne ha più bisogno.

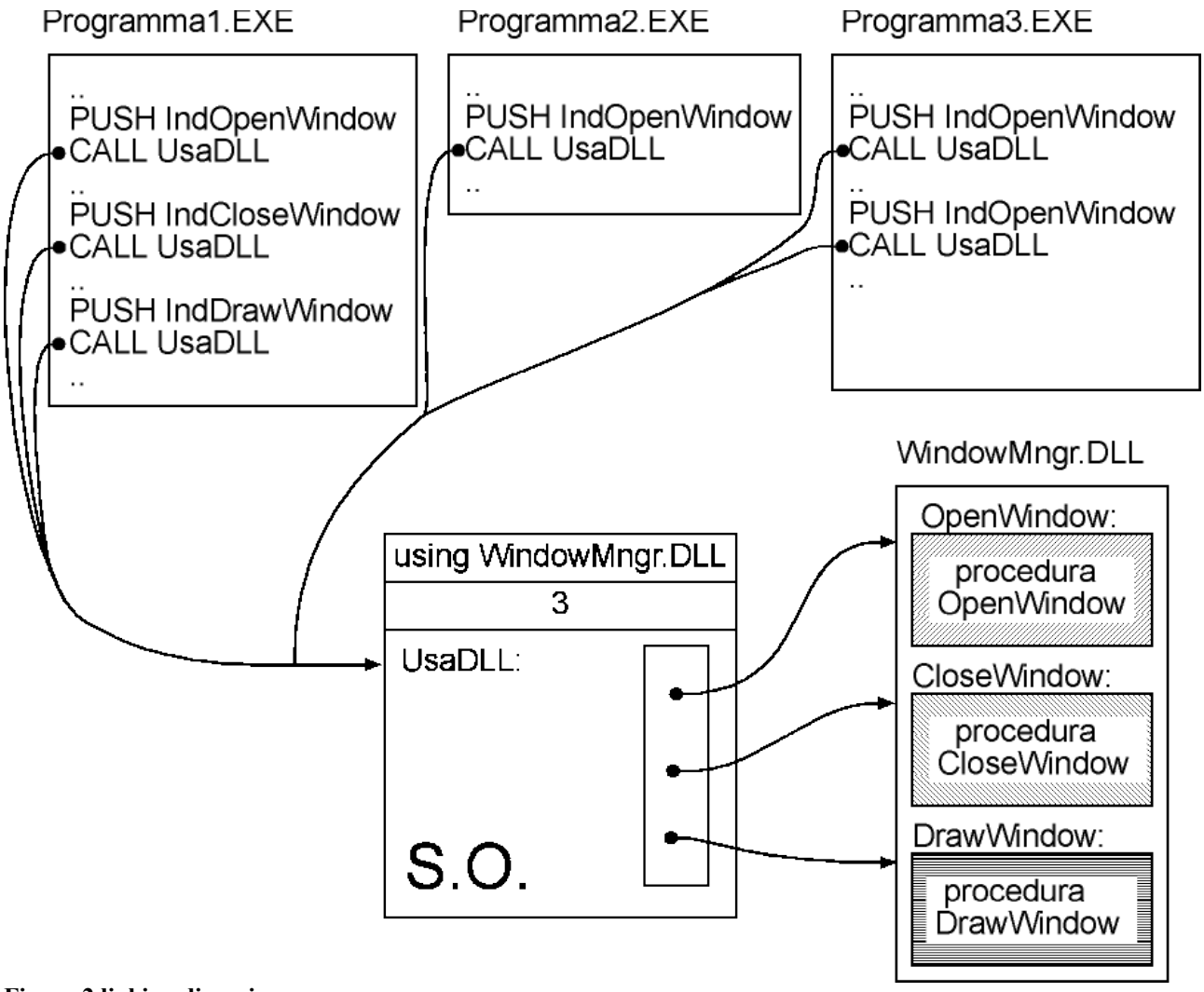



**Figura 2 linking dinamico**

## *1.3 File di inclusione*

Molti linguaggi di programmazione hanno una particolare clausola per indicare l'inclusione di file.
Quando un file viene "incluso" nel sorgente di un programma tutto funziona come se il suo contento fosse scritto nel punto del programma "ospite" in cui la clausola di inclusione viene citata.

Un file di include non è un modulo, almeno nel senso che abbiamo dato qui alla parola "modulo", perché non viene compilato "separatamente". Il contenuto del file include è costituito da istruzioni "legali" del linguaggio di programmazione utilizzato, ma di solito non è compilabile separatamente, perché è fatto per essere messo fra le righe di un altro programma. Il linker non interviene quando ci sono file include perché il lavoro di inclusione viene fatto direttamente dal compilatore.

## *1.4 Programmi Assembly su più moduli*

Nel volume precedente avevamo visto come la produzione del programma eseguibile in due fasi permetta di unire insieme, nella fase di linking, parti di programma prodotte e compilate in tempi, luoghi e da persone diverse.

In questo capitolo vedremo come realizzare programmi in Assembly mettendo insieme parti compilate separatamente.

E' dunque possibile costruire programmi che siano composti da diversi file sorgente, separatamente compilati, già in forma di .OBJ, ed uniti tramite il linker. Vediamo come si deve fare con il Turbo Assembler (TASM). Analoghe saranno le operazioni da fare se si userà MASM.

Se si hanno più moduli .OBJ da unire, si può usare il linker in questo modo:

```
TLINK <Modulo principale> + <Modulo secondario 1> + <Modulo secondario 2> ..
```

Il linker farà partire il programma complessivo dal primo modulo scritto nell'elenco di moduli, il file eseguibile avrà nome <Modulo principale>.EXE.

Naturalmente, perché abbia senso unire insieme moduli separati, essi debbono avere qualcosa in comune, cioè codice e/o dati. Per indicare al compilatore che alcuni indirizzi sono condivisi fra più moduli, si usano le direttive seguenti.

### 1.4.1 EXTRN

L'indirizzo di locazioni di memoria non definite

Indica al compilatore che la label non è definita in questo file (modulo). Il compilatore non deve perciò dare errore quando non troverà l'etichetta, ma deve solo inserire nel file .OBJ il nome simbolico dell'etichetta. Ci penserà il linker a 'risolvere' il riferimento esterno, cioè ad assegnare il giusto indirizzo all'etichetta.

Sintassi:

```
EXTRN <Label> [: <Tipo>]
```

per etichette di procedure <Tipo> è FAR o NEAR, per etichette di dati si può esprimere il parallelismo dei trasferimenti (BYTE, WORD ..).
Esempi:

```
EXTRN Min, Max
EXTRN TrovaMax: FAR
EXTRN Max: WORD
```

L'aggiunta del tipo permette al compilatore di fare alcuni controlli che possono evitare degli errori, ma non è obbligatoria.

### 1.4.2 PUBLIC

Si usa per far sapere al compilatore che un indirizzo in memoria (cioè una etichetta) può essere usato anche da altri moduli, compilati separatamente. Il compilatore inserisce nel file .OBJ l'informazione simbolica dell'etichetta 'esportata' e l'indirizzo che esso le ha assegnato nella sua symbol table. Successivamente il linker assegnerà proprio quell'indirizzo a tutti i nomi uguali non risolti che troverà nei file che deve collegare.

Sintassi:

```
PUBLIC <Label> [: <Tipo>]
```

### 1.4.3 GLOBAL

Nel caso in cui nel modulo in cui compare la GLOBAL sia presente la definizione della corrispondente label , GLOBAL si comporterà come PUBLIC, altrimenti come EXTRN.

```
GLOBAL <Label> [: <Tipo>]
```

**Figura 3: moduli oggetto con informazioni simboliche**

### 1.4.4 PROC

La direttiva che definisce la procedura può anche dire se essa deve essere NEAR o FAR. Nelle procedure chiamate fra moduli diversi di solito deve essere chiaro se sono near o far, per cui non è opportuno ometterlo. Il default è NEAR.

Sintassi:

```
<Label della procedura> PROC [NEAR | FAR]
```

Indicando al compilatore se la procedura deve essere near o far, esso può mettere il giusto codice operativo della RET, near o far. La zona di codice che costituisce la procedura finisce con la direttiva:

```
<Label della procedura> ENDP
```

## *1.5 Librerie di procedure*

La possibilità di collegare moduli diversi in tempi diversi apre la strada alla costruzione di "**librerie**" di procedure.

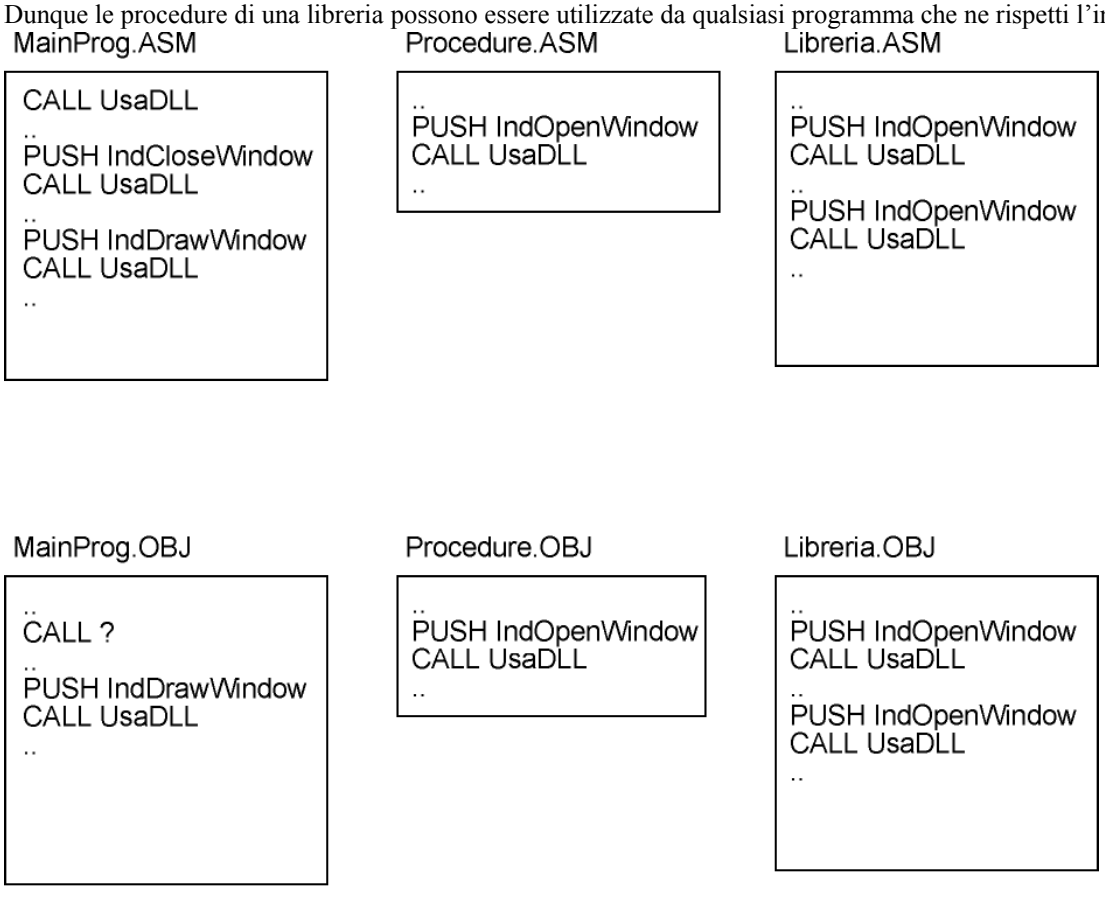
Una libreria è una collezione di procedure già compilate che possono essere utilizzate da un programma principale.

Una libreria possiede una serie ben definita di specifiche, che descrivono i nomi delle procedure in essa contenute, le convenzioni di chiamata ed il significato di ogni parametro che si può passare alle procedure.

Di solito le specifiche delle procedure di una libreria vengono chiamate "**interfaccia di programmazione**" di quella libreria.

Spesso in questi casi si ricorre all'acronimo **API** (**A**pplication **P**rogram **I**nterface).

Dunque le procedure di una libreria possono essere utilizzate da qualsiasi programma che ne rispetti l'interfaccia di pro-




grammazione, indipendentemente dal linguaggi che si usa nel programma principale (almeno in linea di principio). Inoltre le librerie stesse possono essere realizzate con tecniche e linguaggi diversi.

Di solito i Sistemi Operativi e/o i linguaggi di programmazione supportano almeno un formato per il file di librerie, che può essere leggermente diverso da quello del file oggetto.

Esempio:

I file di libreria di MSDOS hanno estensione .LIB, mentre quelli creati dal compilatore gcc per Linux terminano per .a

Quando un programma non utilizza tutte le procedure contenute in una libreria, in base alle caratteristiche del linker possono accadere due cose:

vengono collegate solo le procedure effettivamente utilizzate nel programma che si sta producendo
viene comunque collegata tutta la libreria, comprese le procedure non usate (questo è il caso più comune).
Di solito si possono accorpare in una singola libreria diversi file oggetto, prodotti in modi e tempi diversi ed eventualmente anche con compilatori diversi.

### 1.5.1 Utilizzazione di librerie

La versione più completa della sintassi del comando TLINK è la seguente:

```
TLINK <Elenco dei moduli OBJ> [,<Nome del file eseguibile>]
[,<nome del file .MAP, di info] [,<Elenco delle librerie>]
```

Gli elenchi sono una serie di nomi di file, ciascuno separato da uno spazio o da un segno "+".

Si nota la possibilità di usare librerie, che vengono specificate in un elenco separato da virgole.

Il formato dei file di libreria è leggermente diverso da quello degli OBJ, per cui ci vuole un programma specifico per generarle. Il default dell'estensione dei nomi delle librerie è .LIB.

LIB

LIB è un programma compreso in tutti gli strumenti di sviluppo per MS-DOS. Esso permette di creare librerie a partire da un insieme di file oggetto, di modificarle o di estrarne dei pezzi in un nuovo file oggetto.

Sintassi:

```
LIB [<opzioni>] [<comandi>] [,<lista di file .OBJ>] [, Nome della libreria]
```

<lista di file .OBJ> è l'insieme dei nomi che si vogliono aggiungere, ciascuno separato dal segno +.

Per i comandi e le opzioni per manipolare le librerie si veda l'help in linea del DOS (LIB /? oppure LIB /HELP).

### 1.5.2 Un programma di esempio con due moduli

```
; ************************** inizia il primo modulo **************************
; MODULO1.ASM
; MODULO1 è il modulo principale

; da collegare a MODULO2.OBJ con il linker:
; TLINK MODULO1.OBJ + MODULO2.OBJ
; Programma di esempio per TASM - MASM con chiamate, anche indirette,
; near e far, fra moduli e segmenti diversi e con memoria condivisa fra moduli
; Il programma cambia semplicemente i valori di alcune variabili comuni, con
; procedure e valori messi in file diversi.
; **** memoria importata ed esportata:
PUBLIC VarComuneIn1      ;Byte in memoria condivisa fra i due moduli
                         ; definita in MODULO1

EXTRN VarComuneIn2       ; word in memoria condivisa fra i due moduli
                         ; definita in MODULO2

; **** procedure importate ed esportate:
EXTRN ProcInM2           ; in MODULO2, procedura chiamata da MODULO1,
                         ; si può definire se NEAR o FAR (TASM)

PUBLIC ProcFARInM1       ; in MODULO1, procedura chiamata da MODULO2

dati1   SEGMENT
    ; qui ci sono le normali direttive per l'allocazione della memoria
```

```
    IndirizzoFar DD ?    ; una DWORD per memorizzare un indirizzo far
    VarComuneIn1 DB ?    ; locazione, già definita PUBLIC,
; che verrà usata dal modulo secondario
dati1   ENDS
; ************** Inizia un primo segmento di codice ****************
codice1  SEGMENT
; nel seguito useremo DS per puntare al segmento dati1 ed ES per DatiEsterno:
ASSUME CS:codice1, DS:dati1
; ****** Main program ******
inizio:
; inizio è la label che stabilisce l'inizio del programma, a patto che il
; file MODULO1.OBJ sia linkato come il pricipale, cioè il primo della
; lista dei file da collegare.

; mettiamo in DS l'inizio del segmento dati1
    MOV  AX, SEG dati1   ; il loader non carica DS, lo dobbiamo fare noi,
    MOV  DS, AX  ; passando per AX perché MOV DS, SEG dati non è
                 ; possibile, SEG dati fa prendere dal PSP
                 ; l'indirizzo del segmento dei dati assegnato
                 ; dal DOS al momento del lancio del programma

; mettiamo in ES l'inizio del segmento dati del modulo esterno:
MOV  AX, SEG VarComuneIn2
MOV  ES, AX
; ES punta al primo Byte del segmento dati esterno (DatiEsterno)
; perché SEG VarComuneIn2 = SEG DatiEsterno
; (VarComuneIn2 è nel segmento DatiEsterno)

    CALL PvicinaInM1     ; chiamata near

    CALL FAR PTR ProcInM2; chiamata alla procedura nel modulo esterno (MODULO2.ASM)
                         ; senza FAR PTR non funziona!

    CALL FAR PTR ProcFARinM1 ; chiamata far nello stesso segmento

; SALTO INDIRETTO a PlontanaFuori, uso del primo segmento di dati
; prepara in memoria l'indirizzo per una CALL
MOV WORD PTR [IndirizzoFar], OFFSET ProcInM2
MOV WORD PTR [IndirizzoFar + 02h], SEG ProcInM2
; chiamata indiretta, ad un indirizzo far che è in memoria:
CALL [IndirizzoFar]

MOV  AH, 4ch ; fine programma
INT  21h

; **** procedure ****
PvicinaInM1 PROC NEAR
; procedura near scritta prima del codice del main, nello stesso segmento
; aumenta di 1 le variabili comuni
INC [VarComuneIn1] ; il segmento di VarComuneIn1 è puntato da DS
INC ES:[VarComuneIn2] ; il segmento di VarComuneIn2 è puntato da ES
RET
PvicinaInM1 ENDP
ProcFARinM1 PROC FAR
; procedura far nello stesso segmento, aggiunge 4 alle variabili comuni
ADD [VarComuneIn1], 4
ADD ES:[VarComuneIn2], 4
RET
ProcFARinM1 ENDP
codice1  ENDS

END inizio ; fine di tutto, si indica anche da dove partire con il codice
; ************************ inizia il secondo modulo *************************
; MODULO2.ASM
; MODULO2 è il modulo secondario

; Il main program può benissimo essere assente, dato che di questo modulo
; si usano le procedure.
; Peraltro è altrettanto lecito mettere un programma principale che non
; viene usato dal modulo che chiama le procedure ma potrebbe essere utile,
; per esempio, per poter provare le procedure del modulo
; senza il bisogno di collegarle con un programma esterno.
; **** dichiarazioni dei nomi importati ed esportati:
PUBLIC ProcInM2      ; in MODULO2, procedura chiamata da MODULO1
EXTRN ProcFARInM1    ; in MODULO1, procedura chiamata da MODULO2
```

```
EXTRN VarComuneIn1   ; byte in memoria, condiviso e definito in MODULO1
PUBLIC VarComuneIn2  ; word in memoria, condivisa e definita in MODULO2

; **** segmento dati di questo modulo:
DatiEsterno SEGMENT
VarComuneIn2 DW ?
DatiEsterno ENDS
; **** segmento codice di questo modulo:
CodiceEsterno SEGMENT
; **** dichiarazione dell'uso dei segmenti:
ASSUME CS:CodiceEsterno
ProcInM2 PROC FAR
; toglie 1 a VarComuneIn1 e aggiunge 1 a VarComuneIn2, poi chiama
; ProcFARInM1 (che aggiunge 4 a entrambi)
; non modifica DS ed ES, che sono già sistemati nel programma principale.
DEC [VarComuneIn1] ; il segmento di VarComuneIn1 è puntato da DS
INC ES:[VarComuneIn2] ; il segmento di VarComuneIn2 è puntato da ES
CALL FAR PTR ProcFARInM1
RET
ProcInM2 ENDP
CodiceEsterno ENDS
; il linker può anche mettere assieme i segmenti di due moduli facendoli
; diventare un unico segmento globale. Questo caso non viene illustrato
; per semplicità (vedi i moduli COMBINE1.ASM e COMBINE2.ASM nel CD ROM).
END
```

È istruttivo provare con il debugger, passo per passo, questo programma. I file MODULO1.ASM a MODULO2.ASM si trovano nel CD-ROM allegato.

Il linker può essere istruito, con le pseudoistruzioni di "combine", per concatenare segmenti separati nei vari moduli in un unico segmento nel programma eseguibile. Le combine non sono trattate in questo testo, un esempio, sia pur non troppo esplicativo, è presentato nei file sorgente del CD-ROM allegato; file COMBINE1.ASM e COMBINE2.ASM.