PROVA

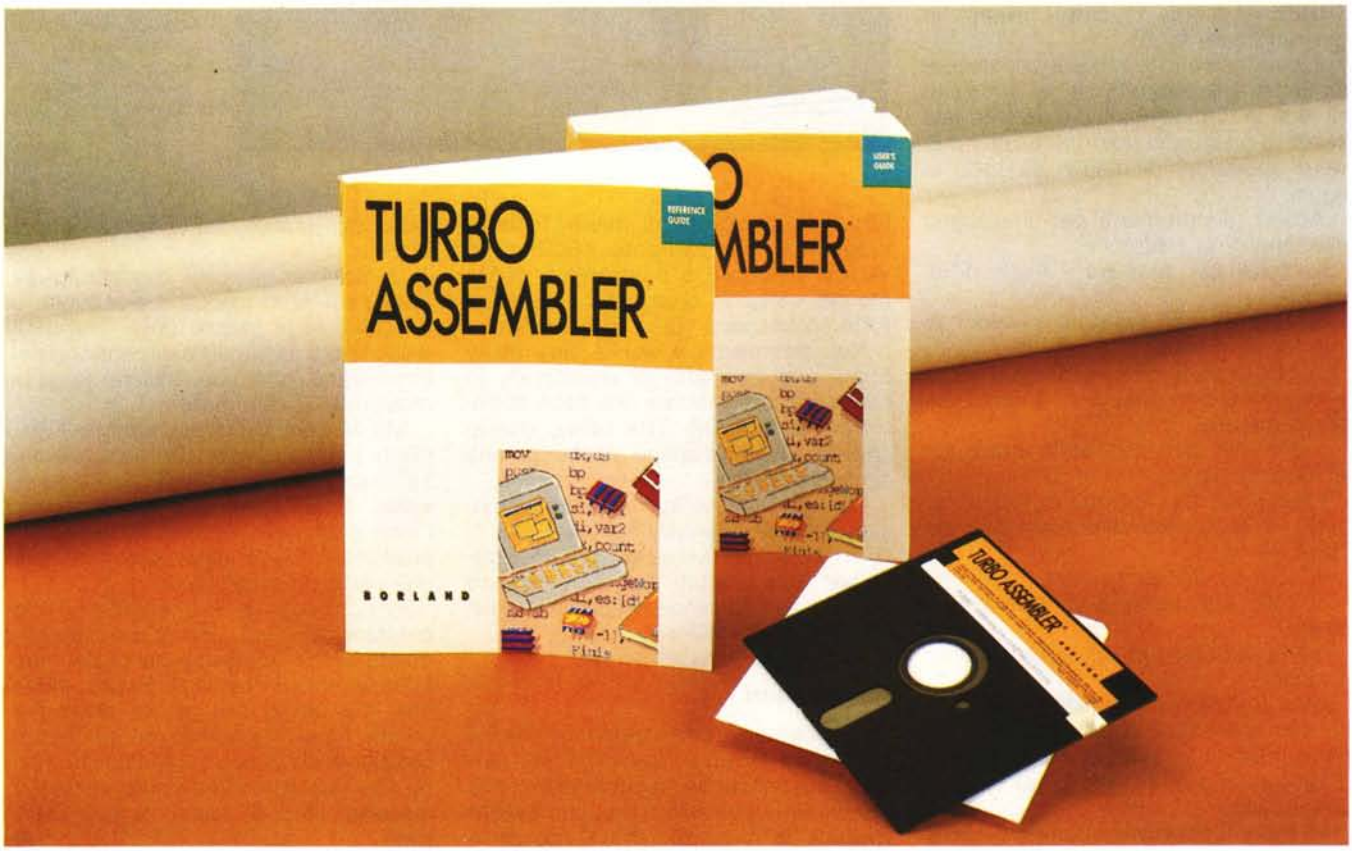
Borland Turbo Assembler 1.0

di Sergio Polini

Nel luglio 1986 la Microsoft stabilì un nuovo standard. Acquistando la versione 4.0 del suo compilatore C si otteneva il CodeView, un debugger simbolico di ottimo livello, un decisivo passo in avanti rispetto al «vecchio» SYMDEB ed anche rispetto ad altri prodotti analoghi già sul mercato. Con il CodeView, programmi che davano risultati pazzi, o programmi che inchiodavano la macchina, cessavano di essere un incubo. Non era più necessario disseminare nei sorgenti decine di istruzioni write *o* printf, *solo per tenere sotto controllo lo svolgimento delle operazioni: se qualcosa non andava per il verso giusto ci pensava il CodeView a farci eseguire passo passo il programma, tenendo costantemente sott'occhio i valori delle nostre variabili, i registri del microprocessore, qualsiasi locazione di memoria. Trovare un bug era e rimase un'impresa tutt'altro che banale, ma con un simile strumento a disposizione i tempi di sviluppo di un programma si sono abbreviati sensibilmente.*

Un po' alla volta tutti i compilatori Microsoft, ed anche l'assembler, sono stati confezionati con il CodeView. È proprio questo il nuovo standard: nella prova di un compilatore o di un assembler non si può prescindere dalla compatibilità con il CodeView o dalla disponibilità di un prodotto almeno equivalente, e ne ha fatto le spese, tra gli altri, anche la Borland. Un paragone tra il Pascal della Microsoft e il Turbo Pascal non si pone nemmeno, tanto è il successo che centinaia di migliaia di utenti hanno decretato a favore del secondo. Ben diversamente è andata però per il C: il debugger simbolico integrato nel QuickC, il CodeView offerto fin dal C 4.0 hanno indubbiamente fatto sentire il loro peso. Tutti sapevamo che la Borland non avrebbe dormito su instabili allori; ci chiedevamo solo quando avrebbe fatto la prossima mossa. Bene, ci siamo. Il Turbo C 2.0 e il Turbo Pascal 5.0 hanno ora un debugger simbolico integrato, più o meno analogo a quello del QuickC; è possibile però anche acquistare le versioni «Professional» dei due compilatori, che comprendono un Turbo Debugger separato e un Turbo Assembler. Oppure si possono acquistare le versioni «normali», integrandole poi con un Turbo Assembler/Debugger, la confezione con le due maggiori novità.

Il tutto con il consueto stile Borland: il Turbo Debugger verrà esaminato il mese prossimo, ma già vi possiamo anticipare che... sarà ora la Microsoft a non poter dormire sugli allori! Quanto al Turbo Assembler, giudicate voi.



Turbo Assembler 1.0

Produttore:
Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001
Distributore:
Edia Borland Srl
Via Cavalcanti, 5 - 20127 Milano
Telefono: 02/2610102
Prezzo *(IVA 9% esclusa):*
Turbo Assembler/Debugger 1.0 *L. 249.000*

A cosa serve un assembler? La domanda sembra banale, ma secondo la Borland non lo è.

Può essere utilizzato per scrivere brevi programmi di utilità o applicazioni complesse, ma anche per ottenere la massima efficienza nelle sezioni più critiche di un programma scritto in un linguaggio di alto livello (quelle che maggiormente incidono sulle prestazioni velocistiche del tutto), magari ottimizzando «a mano» il codice prodotto dal compilatore.

In fondo è anche per questo che vi sono diversi assembler sul mercato, tra i quali potremmo includere perfino il buon vecchio DEBUG del DOS, vista la possibilità di usarlo per produrre brevi file .COM.

Tra tutti è comunque sempre emerso il Macro Assembler (MASM) della Microsoft: una sintassi relativamente agile ne consente l'uso anche per il piccolo cabotaggio, mentre le numerose direttive e le potenti macro ne hanno sempre fatto lo strumento preferito per la programmazione «seria». Solo recentemente è apparso un pericoloso concorrente: l'OPTASM della SLR Systems che, pur non riconoscendo le istruzioni specifiche dell'80386/387, si presenta come un prodotto di ottima fattura, compatibile con il MASM (nelle sue diverse versioni) e espressamente orientato alla programmazione professionale, alla produzione di programmi complessi.

Il mercato è quindi diventato dinamico, si è cioè aperta una corsa alla palma del «migliore», nella quale i concorrenti sono subito diventati tre: il Turbo Assembler (TASM) della Borland appare avere tutte le intenzioni di dire la sua, anche per la produzione di intere applicazioni, pur essendo espressamente dedicato agli utenti dei compilatori Borland. Compatibile con il MASM, completo delle istruzioni dell'80386/387, risulta molto interessante per chi voglia usarlo da solo, ma eccelle soprattutto come strumento ausiliario.

Sembrerebbe che alla Borland credano (a ragione) che non ha più molto senso scrivere interi programmi in assembler: oggi è infatti possibile disporre di compilatori molto potenti, anche per quanto riguarda l'accesso all'hardware e l'interfaccia con il sistema operativo, e l'esperienza ha ormai insegnato che non sempre è vera l'equazione «assembler = codice più veloce».

Spesso l'elemento decisivo è l'algoritmo usato, e l'uso di un linguaggio di alto livello rende più efficace la ricerca e l'implementazione dell'algoritmo migliore.

In un mercato sempre più competitivo è poi anche più importante un'altra velocità: quella dell'attività di programmazione; è chiaro che in ogni caso l'uso di un linguaggio di alto livello consente tempi di sviluppo più brevi.

Solo in una seconda fase, dopo aver individuato i «colli di bottiglia» del programma finito, è lecito intraprendere una conversione in assembler delle sezioni «critiche».

Questo spiega la più evidente differenza tra il Turbo Assembler e i compilatori fratelli: l'assenza di una interfaccia utente «alla Borland».

Il TASM è fatto per essere usato prevalentemente da un *makefile*, per la preparazione di file .OBJ da «linkare» poi a quelli prodotti dai compilatori Basic, C, Pascal o Prolog.

I manuali

Un'altra evidente differenza è nello stile dei manuali. A parte forse quello del Turbo Prolog, i manuali della Borland non hanno mai preteso di essere dei *tutorial*. «Nonostante i molteplici esempi, [questo manuale] non vuole essere né un libro di testo né un manuale autodidattico e presume quindi almeno una conoscenza di base del linguaggio»; così cominciavano i manuali delle prime versioni del Turbo Pascal. Ora abbiamo una *User's Guide* e una *Reference Guide*; nella seconda si descrivono le variabili predefinite, gli operatori, le direttive, la sintassi dell'assembler, le estensioni rispetto al MASM, i programmi di utilità (MAKE, TOUCH, TLINK, TLIB, GREP, OBJXREF, TCREF), i messaggi d'errore; nella *User's Guide*, invece, si mostra con estrema gradualità l'uso dell'assembler e delle diverse opzioni senza nulla dare per scontato.

L'utente si trova come a discorrere pacatamente con un esperto e saggio programmatore, che non solo lo introduce piano piano ad un mondo sicuramente ostico per chi vi si accosti per la prima volta, ma si preoccupa di metterlo in guardia sia dalle «cattive abitudini» che dagli errori in cui è più facile incorrere.

Ecco ad esempio come «il saggio» spiega perché è meglio scrivere una *label* da sola su una riga, rimandando alla successiva le corrispondenti istruzioni: «In primo luogo, se mettete ogni label da sola su una riga, è più facile usare nomi lunghi per le label senza sconvolgere il formato del vostro sorgente [...].

In secondo luogo, è più facile aggiungere una nuova istruzione subito dopo una label se non vi sono istruzioni sulla stessa riga in cui è la label».

Sono osservazioni che possono risultare banali solo a chi non ha mai scritto più di tre righe in assembler, ma soprattutto danno un'idea dello spirito con cui il manuale è stato scritto.

Un altro esempio: ogni testo o manuale sull'assembler 80x86 deve spiegare a cosa serve e perché è necessaria la direttiva ASSUME, l'oggetto forse più misterioso per i principianti (e non solo per loro). Bene: nessun dubbio può restare dopo aver letto le pagine 116-119 della *User's Guide*.

Si potrebbe continuare: si consiglia di non usare mai la direttiva «.RADIX 16» e se ne spiegano con chiarezza gli inconvenienti; si illustra l'utilità, non a tutti evidente, dell'istruzione JCXZ; ci si sofferma con puntiglio sugli incrementi/decrementi dei registri SI, DI e CX operati dalle istruzioni di stringa (LODS, STOS, MOVS, SCAS, CMPS) e sulla confusione che possono generare, ecc.

Manca un capitolo dedicato all'esposizione sistematica delle diverse istruzioni, proprio perché queste vengono prima elencate tutte insieme, poi esaminate una ad una attraverso numerosi esempi, ponendo sempre l'accento sull'uso corretto di ognuna.

Un indice analitico molto ben fatto fa il resto.

Dopo le oltre 270 pagine dedicate a questa illustrazione, ben trenta pagine riassumono concisamente gli errori e i trabocchetti più comuni: un vero e proprio tesoro di preziosi consigli. L'unico appunto potrebbe riguardare la brevità forse eccessiva della sezione dedicata ai salti condizionali, alla possibile confu-

sione tra un JA, ad esempio, ed un JG (da usare il primo per valori «segnati», ovvero con il bit più significativo dedicato a rappresentare un segno positivo o negativo, il secondo per valori «non segnati»), ma può anche darsi che il giudizio sia condizionato dal fatto che il sottoscritto è incappato recentemente proprio in un errore del genere...

A questo punto chi è già esperto si chiederà cosa potrebbe farsene di un manuale così fatto. Vi sono due risposte ad una simile domanda: da una parte è probabile che l'esempio di uso dei RECORD (con routine per l'accesso e la modifica dei bit del *BIOS equipment flag*) o quello relativo ai nuovi modi di indirizzamento consentiti dall'80386 siano comunque degni di nota, dall'altra si deve rilevare che con la stessa ricchezza di esempi e con la stessa chiarezza espositiva vengono man mano illustrate anche le numerose estensioni che il Turbo Assembler propone rispetto al MASM.

Istruzioni

/A	I segmenti vengono ordinati alfabeticamente nel file oggetto
/B	Inclusa solo per compatibilità con il MASM, non ha alcun effetto
/C	Aggiunge informazioni di «cross reference» al file listato
/D	Definisce un simbolo (come con la direttiva =)
/E	Genera codice compatibile con una libreria di emulazione dell'80x87
/H	Mostra una schermata di help (anche /?)
/I	Specifica un path per i file da includere
/J	Definisce una direttiva iniziale (quali IDEAL, JUMPS, ecc.)
/KH	Imposta il numero massimo di simboli (default = 8192; max = 32768)
/KS	Imposta la memoria massima per le stringhe (max = 255K)
/L	Genera un file listato
/LA	Genera un file listato completo
/ML	Distingue per tutti i simboli tra lettere maiuscole e minuscole
/MU	Converte tutti i simboli in lettere maiuscole
/MX	Distingue tra maiuscole e minuscole per i simboli EXTRN e PUBLIC
/N	Non include la tavola dei simboli nel file listato
/P	Controlla che non vengano generate istruzioni problematiche se eseguite da un 80286 o 80386 in «modo protetto»
/R	Genera istruzioni eseguibili da un 80x87 (cfr. /E)
/S	I segmenti vengono assemblati nel file oggetto con lo stesso ordine in cui compaiono nel sorgente
/T	Elimina messaggi su video in caso di esecuzione senza errori
/V	Inclusa solo per compatibilità con il MASM, non ha alcun effetto
/W	Controlla l'emissione di messaggi di avvertimento
/X	Provoca l'inclusione completa delle direttive condizionali nel file listato
/Z	Mostra la corrispondente riga del sorgente insieme ai messaggi d'errore
/ZD	Aggiunge informazioni sui numeri di riga nel file oggetto
/ZI	Aggiunge informazioni su simboli e numeri di riga nel file oggetto

Figura 1. Le opzioni che è possibile usare nella riga comando del Turbo Assembler. Vengono riconosciute tutte le opzioni del MASM 5.1, anche se /W ha una sintassi estesa e altre sono nuove: /KH, /KS, /R (presente nelle precedenti versioni del MASM ma non nella 5.x). Manca solo l'opzione per produrre il listato dopo la prima passata, in quanto il TASM è un assembler a una sola passata.

*TASM **

Cominciamo dalla riga comando. Chi è abituato al MASM avrà subito notato che non è più necessario assemblare un file per volta, ma è possibile usare l'asterisco o il punto interrogativo per indicare un insieme di file da lavorare tutti insieme: avrà anche notato che non è necessario terminare una riga comando abbreviata con quel benedetto punto e virgola che si dimentica sempre.

Per il resto la sintassi della riga comando è analoga a quella del MASM: opzioni (vedi figura 1), nome del sorgente, nome del file oggetto, nome del file listato, nome del file di *cross reference*. I nomi dei file vanno separati da virgole, una virgola non seguita da un nome indica che si vuole produrre un file del tipo corrispondente ad una data posizione nella riga comando avente per nome quello del sorgente e per estensione, secondo i casi, OBJ, LST o XRF. Ad esempio:

```
TASM pippo,,test,
```

produce PIPPO.OBJ, TEST.LST e PIPPO.XRF. «TASM pippo» si limita a produrre il solo file oggetto.

Alcune opzioni sono mantenute solo per una eventuale esigenza di compatibilità con assembler ormai «antichi» (in particolare /A), altre vengono usate raramente; vi possono essere tuttavia opzioni che si vorrebbe usare sempre (quali /MX per linkare poi i file OBJ a un programma in C, /Z per messaggi d'errore più chiari, /ZI per poter poi esaminare il programma con il debugger simbolico).

Il MASM propone di soddisfare questa esigenza creando una variabile di nome «MASM» nell'environment, alla quale si può assegnare come valore una sequenza di opzioni (tranne quelle del tipo: /DSimbolo=Valore, in quanto la stringa assegnata ad una variabile dell'environment non può contenere un segno di uguale).

Il Turbo Assembler preferisce invece un file testo TASM.CFG, creabile con un qualsiasi editor, contenente le opzioni desiderate.

Ogni soluzione presenta vantaggi e svantaggi. La soluzione Microsoft è sicuramente più veloce, in quanto non richiede la lettura di un file, ma può risultare «troppo globale»: l'environment (salvo sottili disquisizioni che qui non è il caso di fare) è sempre lo stesso dovunque e comunque si usi l'assembler. Un file di configurazione fa perdere qualche attimo (compensato peraltro dalla velocità di esecuzione; ne riparleremo tra breve), ma risulta più flessibile: il file TASM.CFG viene cercato infatti per prima cosa nella directory corrente, e diventa così possibile collocare diversi file di configurazione in diverse subdirectory.

Il Turbo Assembler consente infine di inserire nella riga comando alcuni nomi di file preceduti da un carattere «chiocciola»: il contenuto di tali file diventa parte integrante della riga comando. Ad esempio:

```
TASM @opzioni @sorgenti @librerie
```

assembla i file elencati nei file SORGENTI (routine specifiche) e LIBRERIE (routine di uso generale), secondo le opzioni contenute nel file OPZIONI.

```
        LOCALS
Proc1   PROC
        sub     ax, ax
@@Ciclo:
        add     ax, [bx]
        inc     bx
        inc     bx
        loop    @@Ciclo
        ret
Proc1   ENDP
        .
        .
Proc2   PROC
        sub     ax, ax
@@Ciclo:
        add     ax, [bx]
        adc     dx, [bx+2]
        add     bx, 4
        loop    @@Ciclo
        ret
Proc2   ENDP
```

Figura 2. La direttiva LOCALS (disattivabile, anche solo temporaneamente, con NOLOCALS) rende locali alle rispettive procedure le due label @@Ciclo.

Le direttive

Salvo errori od omissioni, sono 189: nonostante alcune duplicazioni (ben motivate), è ovviamente impossibile descriverle tutte. Si tratta perlopiù di quelle già ben note agli utenti del MASM, con interessanti integrazioni. Prendiamo ad esempio le direttive che controllano i file listato: è possibile scegliere tra inserimento e non inserimento nel listato dei file inclusi (%INCL e %NOINCL), oppure sospendere temporaneamente tutte le direttive di listato e poi riattivarle (%PUSHLCTL e %POPLCTL); è anche possibile ottenere listati comodamente esaminabili su video, o chiare stampe anche a 80 colonne, regolando a proprio piacimento la larghezza dei vari campi di ogni riga di un file LST (%DEPTH per il livello di espansione di macro o di inclusione di file, %LINUM per il numero di riga, %PCNT per l'offset di dati o istruzioni nel rispettivo segmento, %BIN per il codice esadecimale e %TEXT per il sorgente).

Abbiamo naturalmente le direttive EXTRN e PUBLIC, che consentono l'accesso da un file ASM a variabili o procedure definite in un altro, come pure COMM, che permette di definire in un *include file* variabili comuni a più file ASM e quindi di evitare di dichiararle tutte PUBLIC da una parte e EXTRN dall'altra. Il limite della direttiva COMM è che può essere usata solo per variabili non inizializzate; la Borland ci propone quindi una direttiva GLOBAL, «sintesi» di EXTRN e PUBLIC, di impiego analogo a COMM ma usabile anche per variabili inizializzate: sarà il Turbo Assembler a interpretare come PUBLIC la variabile GLOBAL inclusa in un file che la definisce (ad esempio con un DW), a interpretarla come EXTRN negli altri file.

La direttiva STRUCT è stata poi potenziata ed affiancata da una direttiva UNION (analoga alla *union* del C): è possibile combinarle insieme, anche per creare strutture nidificate.

Abbiamo anche qualcosa di molto nuovo, come le direttive JUMPS e NOJUMPS. I salti condizionali, nonostante l'ovvio largo impiego, presentano una noiosa limitazione: non consentono di raggiungere una destinazione che disti più di 128 byte. Ciò comporta che talvolta si sbagli il conto (con conseguente messaggio di errore da parte dell'assembler), oppure che ci si rassegni ad usare anche più del necessario l'unica tecnica disponibile; invece di un semplice «jz Zero» si ricorre ad un aggiramento:

```
    jnz  NonZero
    jmp Zero
NonZero:
```

Con JUMPS il Turbo Assembler provvede automaticamente a convertire in sequenze come quella appena vista tutti i salti condizionali che lo richiedano. Meglio: poiché è un assembler a una sola passata, converte efficacemente i soli salti «all'indietro», ma converte tutti i salti condizionali «in avanti», anche quelli per i quali non sarebbe necessario (quando incontra un salto condizionato prima della label, infatti, non può giudicare se tra i due vi sono più o meno di 128 byte e adotta l'unica soluzione che consente di assemblare correttamente; quando poi incontra la label della destinazione può tornare sui suoi passi per sostituire un salto condizionato all'aggiramento, ma così facendo genera dei NOP). Ecco perché la Borland consiglia di usare JUMPS prima dei salti «all'indietro» e NOJUMPS prima di quelli «in avanti». Da questo punto di vista l'OPTASM della SLR Systems si comporta meglio (è un assembler che esegue tante passate quante ne risultano necessarie per produrre il codice più compatto), ma non è escluso che tra non molto MC vi proponga un'applicazione molto interessante, per la quale la direttiva JUMPS, pur se non perfetta, risulterà preziosa.

Una volta tutte le label erano globali, e ciò costringeva il programmatore ad un intenso sforzo di fantasia: guai ad assegnare uno stesso nome a diverse label in uno stesso file! Il Turbo Assembler riconosce invece una direttiva LOCALS, grazie alla quale vengono trattate come locali tutte le label il cui nome inizia con due «chiocciole» o con altri due caratteri a scelta (da indicare subito dopo la direttiva). L'ambito di validità di una label locale è delimitato dalle più prossime label globali, comprese quelle definite con PROC (v. la figura 2). L'utilità dell'innovazione è con evidenza direttamente proporzionale alla lunghezza e alla complessità dei vostri sorgenti.

Parametri e variabili locali

Il TASM vuole comunque essere comodo soprattutto per chi abbia bisogno di routine da inserire in programmi scritti in un linguaggio di alto livello. Da questo punto di vista, l'aspetto più noioso è sicuramente rappresentato dalle convenzioni adottate dai compilatori in relazione ai diversi modelli di memoria: nella figura 3 potete vedere quanto bisogna scrivere intorno anche a pochissime istruzioni, perché queste siano poi «linkabili» ai file OBJ prodotti dal Turbo C per un modello *small*. Analoga la situazione per i compilatori Microsoft, tanto che già il MASM 5.0 consentiva le cosiddette «direttive semplificate»: tutto il cappello si riduceva così ad un semplice MODEL seguito, secondo i casi, da SMALL, MEDIUM, COMPACT, LARGE o HUGE, mentre con .CODE, .DATA e simili si poteva agilmente fare riferimento ai diversi segmenti. La figura 4 consente di apprezzare la differenza.

Quelle direttive non hanno però risolto tutti i problemi. I parametri passati ad una procedura sono accessibili aggiungendo un opportuno valore al registro BP, valore che cambia secondo il linguaggio e il modello di memoria; per le variabili locali va decrementato il registro SP sottraendo il numero di byte richiesto. Nulla vieta di usare la direttiva EQU, è certo possibile farsi un po' di conti, ma la Borland ha ritenuto di proporci una nuova direttiva ARG che, accanto a versioni di LOCAL e PROC simili a quelle introdotte dal MASM 5.1,

```
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP,ss:DGROUP
_DATA   segment word public 'DATA'
        ; dichiarazione di variabili EXTRN
        ; variabili inizializzate
_DATA   ends
_BSS    segment word public 'BSS'
        ; variabili non inizializzate
_BSS    ends
_TEXT   segment byte public 'CODE'
        ; istruzioni
_TEXT   ends
        end
```

Figura 3. Lo «scheletro» di un sorgente assembler compatibile con il Turbo C (modello di memoria «small»).

```
        DOSSEG
        .MODEL  SMALL
        .DATA
; dichiarazione di variabili EXTRN
; variabili inizializzate
        .DATA?
; variabili non inizializzate
        .CODE
; istruzioni
        END
```

Figura 4. Uno «scheletro» del tutto equivalente a quello della figura 3, realizzato mediante impiego delle direttive semplificate del Turbo Assembler.

rende tutto automatico. Ponendo due righe come

```
ARG    a:WORD,b:WORD
LOCAL Array:BYTE:100,i:WORD=Size
```

subito dopo la direttiva PROC, il Turbo Assembler genera simboli locali alla procedura; i primi due («a» e «b») sono equivalenti a «[BP+m]» e «[BP+n]», dove «m» e «n» sono i valori da aggiungere a BP per accedere ai parametri passati dalla routine chiamante; i secondi due («Array» e «i») equivalgono a «[BP-p]» e «[BP−q]», dove «p» e «q» sono i valori da sottrarre a BP per accedere alle variabili locali; l'ultimo («Size») è il valore da sottrarre a SP per creare lo spazio per le variabili locali. I valori assegnati ai simboli definiti con ARG tengono conto del modello di memoria adottato (scelto con la direttiva MODEL). Non solo. In C i parametri vengono passati sullo stack «al contrario»: prima l'ultimo, poi il penultimo, infine il primo. In Pascal si segue invece lo stesso ordine che hanno i parametri nella chiamata della procedura (o funzione). Normalmente ARG segue la convenzione del C, e quindi quando si scrive una routine da incorporare poi in un programma Pascal (o Prolog) bisogna elencare i parametri dopo ARG nell'ordine inverso; è però anche disponibile un MODEL TPASCAL, che non solo tiene conto del particolare «modello di memoria» adottato dal Turbo Pascal (un *code segment* e uno *heap* di tipo LARGE, un *data segment* di tipo SMALL, tutti i puntatori ai dati di tipo *far*), ma fa sì che ARG rispetti l'ordine con cui i parametri vengono passati in Pascal.

Come se non bastasse, il MODEL TPASCAL consente di sfruttare fino in fondo tutte le sfumature delle direttive ARG e PROC. In C lo stack viene liberato dai parametri passati ad una procedura dopo che questa è tornata alla istruzione immediatamente successiva alla CALL, mediante un incrememnto del registro SP; in Pascal, invece, è la stessa routine chiamata che si fa carico di aggiustare SP, tornando alla chiamante con una istruzione RET avente come argomento un opportuno valore numerico. Non è necessario indicare, né tanto meno calcolarsi, questo valore: ci pensa l'assembler. Non occorre neppure iniziare la routine con le solite «PUSH BP» e «MOV BP,SP» o terminarla con «POP BP»: anche a questo pensa l'assembler, che, se istruito con una direttiva USES, provvede persino automaticamente a salvare e poi ripristinare i registri alterati dalla procedura.

Inutile dire che in MODEL TPASCAL è facilitato anche il passaggio dei risultati delle funzioni: il Turbo Pascal qualche volta usa i registri, qualche volta (in particolare per le stringhe) usa lo stack. Sia ARG che PROC prevedono una clausola RETURNS, mediante la quale si può far sì che il RET automatico non elimini dallo stack il puntatore (di tipo DWORD) alla stringa ritornata da una funzione.

```
        DOSSEG
        .MODEL TPASCAL
        .CODE
Stu     PROC    FAR strg:DWORD RETURNS result:DWORD
        PUBLIC  Stu
        USES    DS
        lds     si,strg
        les     di,result
        lodsb
        mov     cl,al           ; byte di lunghezza
        inc     cl
Ciclo:
        stosb
        dec     cl
        jz      Fine
        lodsb
        cmp     al,'a'
        jb      Ciclo
        cmp     al,'z'
        ja      Ciclo
        sub     al,20h
        jmp     short Ciclo
Fine:
        ret
Stu     ENDP
        END
```

Figura 5. Un esempio di programmazione in MODEL TPASCAL. La procedura Stu è in realtà una funzione che accetta come parametro una stringa e ritorna una stringa uguale a questa ma con tutte le lettere maiuscole. La specificazione dei tipi del parametro e del risultato (ambedue puntatori) nell'ambito della direttiva PROC automattizza sia l'accesso al parametro nello stack (occorrerebbe un LDS SI,[BP+06]) sia quello al risultato (LES DI, [BP+10]). Non sono necessarie inoltre, in quanto anch'esse automatiche, né le istruzioni PUSH BP e MOV BP,SP all'inizio e POP BP alla fine, né, grazie a USES, le istruzioni per salvare e poi rispristinare il registro DS; il «ret» viene convertito in RETF 4.

Oltre 120 pagine del manuale sono dedicate ad illustrare questi ed altri aspetti, offrendo un esame completo dell'uso della *keyword* **asm** del Turbo C e dell'interfacciamento tra Turbo Assembler da una parte e Turbo C, Turbo Pascal, Turbo Basic e Turbo Prolog dall'altra.

Si nota qua e là qualche incertezza espositiva, nel senso che non sempre la *User's Guide* va d'accordo con la *Reference Guide* (ma basta fidarsi sempre di quest'ultima).

È poi necessaria un po' di pratica, ad esempio ricordare che in C un *char* è convertito in *int*, che in Pascal un *byte* viene passato come parte meno significativa di una word, ecc. In complesso tuttavia non si può che riconoscere che il TASM facilita enormemente la scrittura di routine da linkare poi a programmi scritti in altri linguaggi, e che i manuali costituiscono una valida guida e un sicuro supporto.

L'Ideale e le Stranezze

Alcune caratteristiche del Turbo Assembler sono tali da destare l'interesse anche di chi non abbia intenzione di usarlo solo in coppia con un compilatore. La Borland ha cercato infatti di rendere più elegante e al tempo stesso più potente ed efficiente la sintassi del linguaggio.

Se si adotta la direttiva IDEAL, ad esempio, i membri di una STRUCT non sono più visti come simboli globali; ciò comporta che è possibile assegnare lo stesso nome a membri di strutture diverse, o anche solo di assegnare ad una variabile lo stesso nome che ha un membro di una struttura. La possibile ambiguità viene risolta attribuendo un più preciso valore sintattico al punto (.), che può essere usato solo tra il nome di una struttura e il nome di uno dei suoi membri. È questo il motivo per cui alcune direttive, come dicevamo sopra, sono duplicate (ci sono «.287» e «P287», «.LIST» e «%LIST», «.ERR» e «ERR», ecc.): in modo IDEAL non sono più consentite direttive che comincino con un punto (la direttiva contraria MASM consente di attivare e disattivare a piacimento il modo IDEAL). Viene anche meglio regolata la sintassi dell'indirizzamento.

In MASM sono possibili molte diverse scritture. Ad esempio:

```
mov ax,[bx][si]
mov ax,6[bx]
mov ax,es:[bp+8][si+6]
mov ax,pippo
mov ax,[pippo]
```

In IDEAL esiste una sola semplice regola: quando ci si riferisce al valore di un registro o di un simbolo, questo viene scritto senza parentesi quadre; quando ci si riferisce al valore contenuto nella locazione di memoria cui «punta» un registro o un simbolo si usano le parentesi, che devono racchiudere tutta l'espressione che funge da puntatore. Quindi:

```
mov ax,[bx+si]
mov ax,[bx+6]
mov ax,[es:bp+si+14]
mov ax,[pippo]
```

Gli altri costrutti ammessi dal MASM o sono respinti o provocano un messaggio d'avvertimento. In particolare viene vista con sospetto una istruzione come «MOV AX,PIPPO»: l'utente potrebbe non aver chiaro che un conto è mettere in AX l'indirizzo di PIPPO, tutt'altro è metterne in AX il contenuto.

Quando si usano registri è ben chiara la differenza tra «MOV AX,BX» e «MOV AX,[BX]»; quando si usano simboli succede invece che il loro valore non è altro che un offset rispetto all'inizio del relativo segmento dati, ma in MASM «MOV AX, PIPPO» opera come «MOV AX,[PIPPO]» invece che come «MOV AX,OFFSET PIPPO». L'estensione della regola adottata per i registri anche ai simboli non può che giovare alla chiarezza.

È noto che le *forward reference* fanno lavorare l'assembler un po' più del normale, ma è ovvio che non ci si può limitare ai soli JMP o CALL «all'indietro». Accade tuttavia è che il MASM adotta una sorta di *forward reference* anche quando non sarebbe necessario: nella definizione di una label, di una macro, di una procedura, ecc., richiede che venga scritto prima il nome poi la direttiva che chiarisce la natura di quel nome (ad esempio: «Struttura STRUC»). Il parser deve quindi prima leggere il nome senza sapere di che si tratta, poi leggere la direttiva, quindi tornare sui suoi passi per «appuntarsi» cosa fare di quel nome; in modo IDEAL è invece possibile seguire un ordine che non solo risulta più omogeneo alla sintassi dei linguaggi di alto livello («STRUC Struttura»), ma consente anche un parsing più efficiente.

Ci sarebbe molto altro da dire; in sintesi si può rilevare che il TASM mantiene un'ampia compatibilità con il MASM, del quale riproduce anche comportamenti discutibili (quali il risultato di un operatore OFFSET quando vengono definiti più segmenti dati in un unico GROUP), talvolta richiedendo l'attivazione di direttive specifiche (quali QUIRKS, letteralmente «stranezze», o MASM51, ma solo per i casi estremi). L'obiettivo può dirsi raggiunto. Abbiamo provato ad assemblare sia un vecchio file a suo tempo lavorato con il MASM 2.0 (PRIMI.ASM: trova i primi 100 numeri primi simulando il computer «ideale» MIX di D. Knuth), sia i 15 file ASM generati dal Microsoft C 5.1 con opzione /Fa a partire dai sorgenti C di uno screen editor. Non vi sono stati problemi; si è anzi potuto rilevare che il Turbo Assembler è sensibilmente più veloce (anche in modo MASM) e produce codice più compatto.

Per chi voglia invece affrancarsi dalle «stranezze» fin qui imposte dallo standard, il modo IDEAL propone una sintassi più coerente ed elegante, nonché più efficiente, con l'autorevolezza di chi appare avere buoni argomenti per proporsi a sua volta come standard.

	MASM	TASM
PRIMI.ASM		
tempo di assemblaggio	8"	5"
dimensione file .ASM	50688 byte	50688 byte
dimensione file .OBJ	6824 byte	5297 byte
Editor (15 file)		
tempo di assemblaggio:		
uno per volta (BATCH)	1'42"	1'09"
tutti insieme (TASM *)	--	39"
dimensioni:		
15 file .ASM	272409 byte	272409 byte
15 file .OBJ	53499 byte	50844 byte

*Figura 6. Abbiamo provato ad assemblare sia con il MASM 5.0 che con il Turbo Assembler un file PRIMI.ASM (calcola i primi 100 numeri primi simulando il MIX di Knuth) e i 15 file .ASM prodotti dal Microsoft C 5.1 con opzione /Fa a partire dai sorgenti in C di uno screen editor. Il TASM produce codice più compatto in minor tempo. In particolare, grazie alla possibilità di assemblare più file con una sola chiamata del programma (TASM *), per l'editor è stato sufficiente un tempo inferiore del 60% a quello richiesto dal MASM (la prova è stata condotta su un compatibile AT a 10 MHz e 1 wait-state, con disco rigido da 40 millisecondi).*

Conclusioni

Un ottimo assembler, i consueti programmi di utilità (MAKE, GREP, TOUCH), un *linker* capace di generare anche file .COM, un *librarian*, due programmi di *cross reference* (OBJXREF per i file oggetto, TCREF per i sorgenti), ottimi manuali (ora in inglese, ma sostituibili con la versione italiana a partire da febbraio), una ricca messe di esempi sul dischetto, dai sorgenti di due programmi di utilità (WHEREIS, che cerca uno o più file su tutto un disco; FILT, che formatta in vario modo un file testo) a diversi file contenenti numerose macro da includere nei propri programmi. Il tutto in una confezione che comprende anche un eccezionale debugger simbolico (che, ripetiamo, sarà esaminato il mese prossimo), ad un prezzo sicuramente giustificato.

Un prodotto senza dubbio appetibile per chi voglia o debba programmare in assembler.

C'è comunque anche un altro aspetto. Chi abbia un Turbo Pascal 4.0 o un Turbo C 1.5 avrà certo da pensare alla possibilità di un passaggio alle versioni più recenti, magari aiutato dalle prove che vi proponiamo altrove sulla rivista. Non ci sono tuttavia solo il Pascal 5.0 e il C 2.0: ci sono anche versioni dei compilatori che comprendono il Turbo Assembler/Debugger. Se non si ha proprio bisogno dell'assembler, né per scrivere routine ottimizzate né per esaminare fin nei più minuti dettagli il codice prodotto da un compilatore, il problema non si pone. Se invece si vuole arricchire il proprio bagaglio di strumenti, se non si vuole rinunciare né alla possibilità di fare anche quello che con i linguaggi di alto livello non si può fare (o si fa peggio), né alla possibilità di catturare anche il bug più insidioso, la qualità del Turbo Assembler/Debugger è tale da rendere ben giustificato il maggior prezzo delle versioni «Professional».

Considerando alla fine che questo «maggior prezzo» non ha nulla di esoso, vi basta avere le idee chiare sulle vostre esigenze: la Borland sa cosa proporvi senza deludervi. MC