1 MASM32

Microsoft produce, aggiorna e rende disponibile gratis il solo compilatore MASM (programma"ml"), a 16 o 32 bit, ed il linker (programma"link"), ma non supporta in alcun modo questi due prodotti.
MASM32 è il risultato di uno sforzo di un team di sviluppo su Internet, che ha completato il lavoro sull'Assembler MASM di Microsoft. Il team di MASM32 ha scritto una gran quantità di librerie e di file di inclusione, per poter programmare efficacemente in Assembly sotto Windows ed ha chiamato l'insieme "MASM32". Gli strumenti MASM32 realizzano tutte le API del S.O. Windows[1] ed anche altro. MASM32 è disponibile gratuitamente e comprende anche il compilatore ed il linker di Microsoft. MASM32 comprende anche una nutrita documentazione ed una serie di tutorial. A differenza di DOS, ove le chiamate al S.O. erano fatte con interrupt software (es. INT 21h), in Windows ciò avviene con normali chiamate a procedure. Queste procedure sono contenute in librerie a collegamento dinamico (DLL = Dynamic Link Library), caricate in memoria all'accensione del sistema o a richiesta, quando serve. L'insieme di queste procedure viene detto API (Application Program Interface). In MASM32 i nomi delle procedure delle API di Windows sono contenuti in file .INC, da "includere" all'inizio del file .ASM. In questo modo basta usare CALL con il nome delle procedure della API per poterle utilizzare. Il nome delle procedure, il numero e le caratteristiche dei loro parametri si possono trovare nella documentazione di Windows o di MASM32.

Struttura di un programma MASM32

Un programma MASM32 per Windows si presenta in questo modo:

```
.386
.MODEL FLAT, STDCALL
; ***** File di inclusione     *****
INCLUDE <path e nome del file da includere>
    ..

INCLUDELIB <path e nome della libreria da linkare a questo file per generare
l'eseguibile>
    ..

; ***** Riferimenti esterni    *****
EXTRN <simbolo>
    ..

; ***** EQUATE                 *****
<simbolo> EQU <stringa>

; ***** dati inizializzati     *****
.DATA
    <dati inizializzati>
    ..

; ***** dati NON inizializzati *****
.DATA?
   <dati NON inizializzati>
   ..

; ***** costanti *****
.CONST
   <Costanti>
   ..

; ***** codice *****
.CODE
<etichetta d'inizio>:
   <Codice>
   ..
END <etichetta d'inizio>
```

Alcune delle parti del programma qui illustrate possono mancare.
`.386` indica che i codici operativi utilizzati nel programma appartengono al set d'istruzioni della CPU 80386.
`.MODEL FLAT` indica che la memoria è organizzata in modo "flat" (modello "piatto"). Questa è l'UNICA organizzazione della memoria presente nei Windows da Win 98 in poi ed, in generale, in ogni S.O. scritto per CPU della famiglia

1 Le API dei Windows recenti sono dette "Win32 API"

X86 dal 386 in poi (p.es. anche Linux). Nei S.O. per 8086 (MS-DOS) esistevano altre modalità di organizzazione della memoria, ma NON il modello piatto. Con il modello piatto si può considerare di avere a disposizione uno spazio di memoria "virtuale" di 4 Gbyte per OGNI programma che si crea. Il S.O. si incarica di far corrispondere la memoria virtuale di 4GByte che il nostro programma "pensa" di avere con la memoria fisica effettivamente presente sul sistema nel quale il nostro programma gira.
`STDCALL` indica che la modalità di passaggio dei parametri alle procedure è quella standard di Windows (standard call). Con questa modalità i parametri vengono messi nello stack da destra a sinistra (come in tutti i programmi C), mentre la responsabilità del bilanciamento dello stack è del programma CHIAMATO (come faceva il Pascal in DOS). Questa è una convenzione "a metà" fra la convenzione C "pura" e quella Pascal, adottata per sfruttare a pieno il set d'istruzioni X86 (istruzioni RET "n"). E' valida solo per CPU X86 e non è trasportabile ad architetture che non abbiano istruzioni come la RET "n", per questo non viene usata dal C "puro".
`INCLUDE` permette di specificare un file che viene aperto, letto dal compilatore e compilato come se fosse effettivamente scritto nel punto dove, nel nostro sorgente, compare la direttiva INCLUDE. Naturalmente in un programma ci può essere più di una direttiva INCLUDE, scritta nel punto dove si vuole che venga "espanso" il contenuto del file indicato.
`INCLUDELIB` invece specifica un file che contiene una libreria già compilata che deve essere collegata al nostro file, in fase di linkage, per poter ottenere il file eseguibile. Anche le direttive INCLUDELIB possono essere più d'una nello stesso file.
`EXTRN` direttiva che dice al compilatore che il <simbolo> indicato non è definito in questo file. Il compilatore non potrà trovare un indirizzo per quel simbolo, per cui lo inserirà nel file .OBJ per la sua "risoluzione" al momento del linking. Il compito del linker è infatti il collegamento al nostro programma di altri moduli o librerie; l'indirizzo che corrisponde al nostro <simbolo> sarà trovato in uno di questi altri moduli o librerie.
`EQU` la direttiva di "equate" fa "espandere" nel testo del nostro programma la <stringa> al posto di ogni occorrenza del <simbolo> (vedi).
`.DATA` Il codice scritto dopo una direttiva .DATA contiene una "sezione" di dati. Per comodità nella gestione della memoria, Windows distingue fra tre tipi di "sezioni" di memoria, che vengono trattate diversamente al momento della loro allocazione ed utilizzazione. Nella sezione .DATA vanno definiti i dati per i quali siamo interessati ad avere un valore iniziale. I numeri che si definiscono in questa sezione verranno caricati in memoria al momento del caricamento del programma compilato.
`.DATA?` inizia la sezione di dati non inizializzati. L'assembler alloca soltanto la memoria per i dati compresi in questa sezione ma fa in modo che non vi venga caricato nulla al momento del lancio del programma. Perciò quando il programma parte in questa sezione di dati ci saranno numeri non definiti a priori; di solito quelli lasciati dal programma che precedentemente è stato caricato in queste locazioni di memoria.
`.CONST` sezione delle costanti. I dati definiti in questa sezione non potranno mai essere cambiati durante il funzionamento del programma
`.CODE` indica l'inizio della sezione del programma che contiene i codici mnemonici delle istruzioni Assembly, cioè della vera e propria parte "di codice".
`<etichetta d'inizio>` indica al compilatore la prima istruzione del programma. Quando il programma partirà il program counter verrà fatto puntare alla locazione che corrisponde a questa etichetta.
`END <etichetta d'inizio>` END indica la fine di tutto il programma. Il compilatore trascurerà tutto ciò che sta dopo END. <etichetta d'inizio> indica l'etichetta che il compilatore deve considerare come il punto dove il programma comincia.

Compilazione di un programma MASM32

1.0.1 Assembler

L'Assembler si chiama "ml.exe" ed è contenuto in <PATHmasm>\BIN[2]. L'Assembler si chiama da finestra DOS come nel seguente esempio:

```
C:\MASM32\BIN\ml  /c  /coff /Zi /Cp /IC:\MASM32\INCLUDE primo32.asm
```

Opzioni più importanti di ml:
`/coff` indica di usare il formato coff per il file eseguibile (da usarsi sempre in Windows).
`/c` questa opzione non fa eseguire il linking. Infatti il programma ml può "linkare" automaticamente, ma di solito à meglio lanciare il linker direttamente, per controllare con più precisione le opzioni da utilizzare.
`/Zi` inserisce nel file oggetto le informazioni simboliche, per l'uso con i debugger simbolici (es. Visual Studio)
`/I` permette di indicare la directory nella quale l'assembler cerca automaticamente i file di include
`/Cp` compilazione "case sensitive". Con questa opzione si ordina al compilatore di considerare diversi i simboli che abbiano le stesse lettere, ma con diverse combinazioni di maiuscole - minuscole. Infatti il comportamento predefinito dell'Assembler non distingue fra Variabile, vArIaBiLe e VARIABILE, mentre se si lavora con il C può essere necessario

2 Nota: si suppone che il file d'installazione di MASM32 sia stato scompattato nel directory <PATHmasm>. Normalmente il programma di installazione di MASM32 usa come directory "di partenza" C:\MASM32.

considerare i tre nomi appena definiti come cose diverse. Dovendo utilizzare per il debugging uno strumento previsto per il C (vedi oltre) è meglio utilizzare quest'opzione.
`/Cx` mantiene il case solo nei simboli esterni (utile per collegarsi a programmi scritti in C, mantenendo "case insensitive" la sola parte di programma in "puro Assembly").

1.0.2 Linker

Per collegare gli OBJ e generare il progamma eseguibile useremo il programma "link".
Lo illustriamo con un esempio:

```
C:\MASM32\BIN\link primo32.obj C:\MASM32\LIB\kernel32.lib /ENTRY:Inizio /SUBSY-
STEM:CONSOLE /DEBUG /LIBPATH:C:\MASM32\LIB\
```

Questo comando produce, se tutto va bene, il file primo32.exe.
Si noti che questo comando "mette insieme" nell'unico eseguibile primo32.exe i due file "oggetto" primo32.obj e kernel32.lib, effettuando effettivamente un "collegamento" fra i due file.
Le opzioni usate in questo esempio sono spiegate di seguito.
`/ENTRY:<Etichetta>` fa iniziare l'esecuzione del programma dalla linea la cui etichetta è <Etichetta>. Nell'esempio precedente l'etuchetta è "Inizio" (infatti viene usata l'opzione /ENTRY:Inizio) .
`/SUBSYSTEM:` indica se il programma deve funzionare con interfaccia utente a carattere (CUI o console) o in ambiente grafico a finestre (GUI, Windows). In caso di ambiente CUI si deve usare SUBSYSTEM:CONSOLE; altrimenti, nel caso di ambiente GUI, SUBSYSTEM:WINDOWS.
`/DEBUG` fa inserire le informazioni di debug nel programma eseguibile
`/LIBPATH:` permette di indicare la directory nella quale cercare le librerie e perciò di semplificare la loro dichiarazione nella linea di comando e nel codice Assembly.

Un'altra opzione che si potrebbe usare è:
`/DLL` indica al linker di generare una DLL (.DLL) invece che un normale file esegubile (.EXE)
Naturalmente, per poter scrivere una DLL che funzioni, è necessario che il programma Assembly rispetti una serie di vincoli piuttosto complicati, che gli permettono di essere "visto" dal Sistema Operativo come una DLL.

Primo esempio "completo"

Il primo programma che compileremo non fa nulla di importante, ma ci permette di vedere con il debugger come viene allocata la memoria in Windows. Dato il seguente programma, contenuto nel file primo32.ASM:

```
; ************************************
; Programma "minimo" MASM32
; ************************************
.386
.MODEL FLAT, STDCALL

; ***** File di inclusione *****
INCLUDE C:\MASM32\INCLUDE\kernel32.inc

.DATA
A DD 87654321h
S DB "Questa è una stringa scritta 'alla C'", 0

.DATA?
B DD 100 DUP (?)

.CONST

.CODE
Inizio:
    ADD [A], 2
    ADD [A], 2
    ADD [S], 2

    ; passa a ExitProcess il codice dell'uscita senza errore:
    PUSH DWORD PTR 0
    ; chiamata al S.O. che dichiara la fine del programma
    CALL ExitProcess
END Inizio
```

Lo compiliamo con:

```
C:\masm32\BIN\ml /c /coff /Zi primo32.ASM
```

Si noti che non specifichiamo la path del file di include, dato che nel file sorgente la riga:
`INCLUDE C:\MASM32\INCLUDE\kernel32.inc`

comprendeva già tutta la path.
In alternativa avremmo potuto scrivere, nel file sorgente, solo:
`INCLUDE kernel32.inc`

ma poi compilare con:
`C:\masm32\BIN\ml /c /coff /Zi /IC:\MASM32\INCLUDE primo32.ASM`

Specificando la directory dei file di include solo al momento della compilazione.

Per generare l'eseguibile dobbiamo poi usare il linker:
`C:\masm32\BIN\link primo32.OBJ C:\MASM32\LIB\kernel32.lib /SUBSYSTEM:CONSOLE /debug`

Si noti che nel sorgente non è citata la libreria kernel32.lib, che viene dichiarata, dopo il nome dell'OBJ "principale", nell'elenco dei file da utilizzare per produrre l'eseguibile.
Peraltro si sarebbe ottenuto lo stesso effetto dichiarando `INCLUDELIB C:\MASM32\LIB\kernel32.lib` nel programma sorgente e non citando `C:\MASM32\LIB\kernel32.lib` al momento del linkage:
`C:\masm32\BIN\link primo32.OBJ /SUBSYSTEM:CONSOLE /debug`

Dato che il programma appena compilato non fa nessuna visualizzazione, per vedere cosa succede dobbiamo usare un debugger.

Debugging nell'ambiente del Visual C++ 6

Una volta generato l'eseguibile, non dimenticando le opzioni /Zi dell'Assembler e /debug del linker, si può caricare l'eseguibile in Visual C++, con la normale sequenza di menu: File | Open e navigando nei directory fino a quello dove c'è il nostro file eseguibile; poi per poter caricare l'eseguibile, con estensione .exe, è necessario cambiare il "Tipo File" di default, per esempio scrivendo *.exe nella casella "Nome file" e premendo "Invio". Un modo alternativo e più rapido per caricare l'eseguibile in Visual Studio è prenderlo, dalla finestra di Windows Explorer che mostra la cartella dove è stato generato, e "buttarlo" (drag & drop) dentro l'applicazione Visual Studio, precedentemente lanciata senza aprire nessun programma.
Una volta caricato il file si potrà far eseguire la sua prima istruzione premendo F10 od F11. Se Visual C++ trova il file sorgente .ASM nello stesso directory dell'eseguibile, e se l'eseguibile è correttamente compilato, verrà visualizzato anche il sorgente e, all'uscita dall'ambiente VC++ verrà chiesto se si vuole salvare il "workspace" generato. Se si risponde di sì, si potrà in seguito caricare eseguibile e sorgente "insieme" aprendo il workspace. In Visual C++ potremo fare il debugging come lo faremmo per un programma ad alto livello, avendo a disposizione anche i breakpoint (F9 per metterli e toglierli). L'esecuzione del programma si interrompe automaticamente quando giunge ad un breakpoint, permettendo al programmatore di esaminare il contenuto dei registri e della memeoria nei punti critici per l'esecuaione del programma. L'ambiente è fatto per verificare programmi scritti in C, per cui utilizzando l'Assembly bisognerà qualche volta adattarsi.

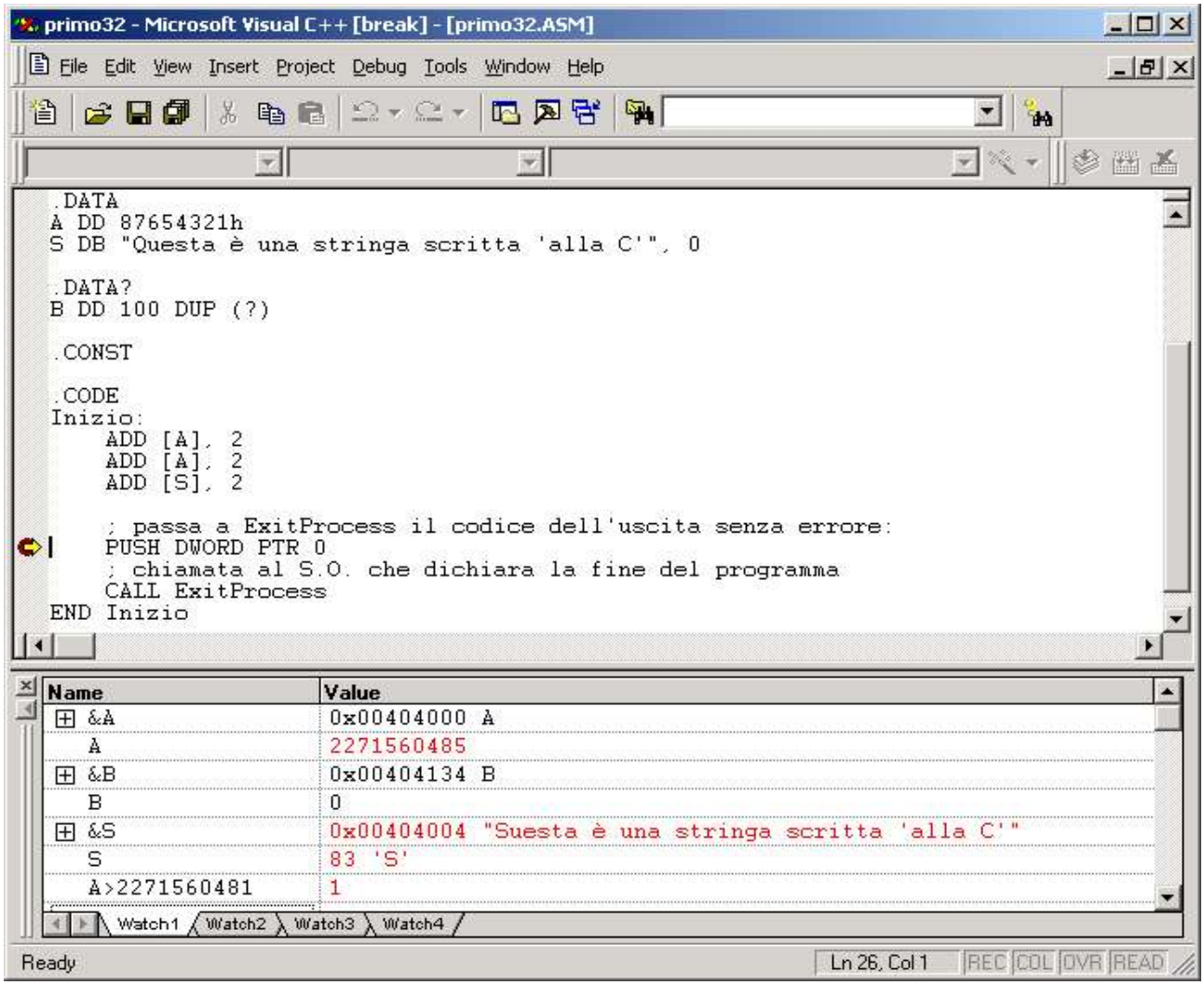

Figura 1

Due finestre importanti dell'ambiente di sviluppo sono mostrate in Figura 1, dove si vede il programma che è stato arrestato su un break point dopo l'istruzione `ADD [S], 2` (la freccia gialla indica il punto ove è giunta l'esecuzione, il pallino rosso, sotto alla freccia gialla, il fatto che su quella linea c'è impostato un breakpoint). La finestra più in alto è quella del sorgente, che indica anche il punto di esecuzione del programma, sotto c'è la finestra "watch" che mostra i valori correnti delle "variabili" e di qualsiasi altra espressione che si voglia tenere sotto controllo. Nella finestra watch sono visibili:

- l'indirizzo della "variabile" A (indicato da &A, naturalmente nella notazione che in C significa "l'indirizzo di A" ("puntatore" ad A)); l'indirizzo è 00404000h ed è indicato come 0x00404000, dato che in C 0x<numero> significa <numero> rappresentato in "esadecimale")
- il valore corrente della "variabile" A, in decimale; in rosso perchè recentemente è cambiato (sono state eseguite le due `ADD [A],2`); il valore al punto in cui il programma è interrotto è 2271560485
- l'indirizzo della "variabile" B (che vale 00404134h; B è un array pieno di numeri sconosciuti)
- il valore del primo degli elementi dell'array B (0)
- l'indirizzo della stringa S (è 00404004h; il programma mostra anche il valore corrente di tutta la stringa, nel quale si vede che la "Q" iniziale della stringa è divenuta "S" per via della `ADD [S],2`)
- il valore corrente di un'espressione "qualsiasi" (A > 2271560481, che dà 1, dato che l'espressione è vera; avrebbe dato 0 se fosse stata falsa).

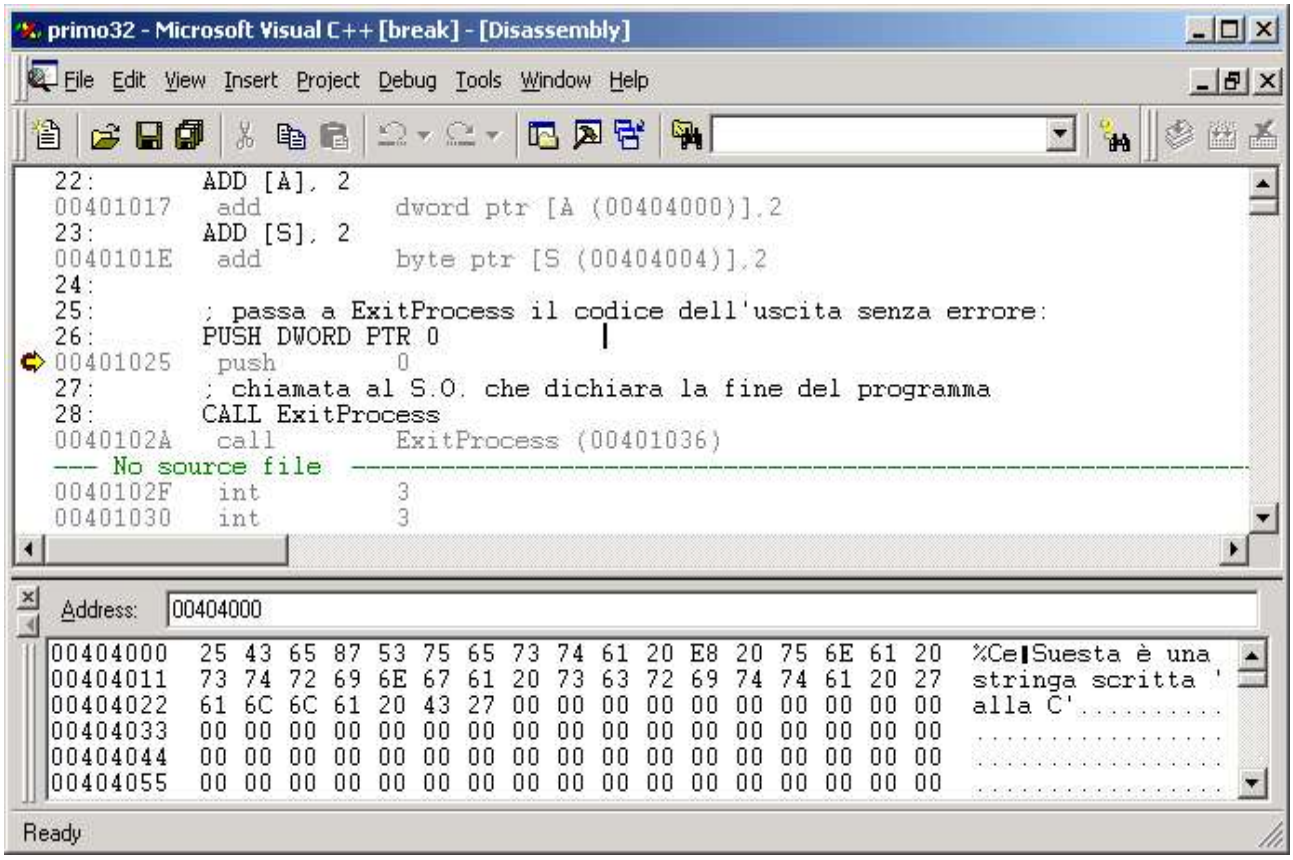

Figura 2

In Figura 2 si vede in alto la finestra "disassembler", nella quale è mostrato, interlacciato, il codice in linguaggio macchina insieme a quello sorgente (quando il sorgente esiste!). In nero si vede il codice sorgente, in grigio i codici operativi ed il disassemblato corrispondenti. In basso nell'immagine si vede la finestra "memory", che mostra il contenuto della memoria. Nella parte sinistra della finestra i dati sono visualizzati A BYTE, come numeri esadecimali, in ordine di indirizzo.
Anche gli indirizzi sono mostrati in esadecimale.
Nella parte a destra GLI STESSI byte sono visualizzati come i caratteri che corrispondono ai rispettivi codici ASCII. In questo caso sono mostrate le locazioni di memoria che partono da 00404000h (l'inizio della Sezione dati inizializzati, cioè la variabile A). Si può individuare il numero 87654325h ("variabile" A), memorizzato in modo "little endian" (25 43 65 87), poi segue subito il valore corrente della stringa S (indirizzo 00404004h, valore corrente: 53, 75, 65, 73, 74, 61, 20, E8 .. eccetera .., cioè "Suesta è una stringa scritta 'alla C'"). Si può notare che la variabile B, che appartiene alla Sezione dati non inizializzati, NON è vicina ai dati della sezione dati inizializzati. Infatti, come si vede nella figura 1 (vicino a &S), il suo indirizzo è 00404134h, e non è allocato subito dopo la fine della sezione dati inizializzati (indirizzo 00404029). Nella figura successiva (Figura 3) si vede il contenuto della sezione dati non inizializzati, che inizia dalla variabile B e che è tutta a zero solo per caso. Infatti, dato che in fase di caricamento in questa area non viene scritto nulla, ci potrebbero essere dei valori qualsiasi, rimasti lì da un programma precedente.

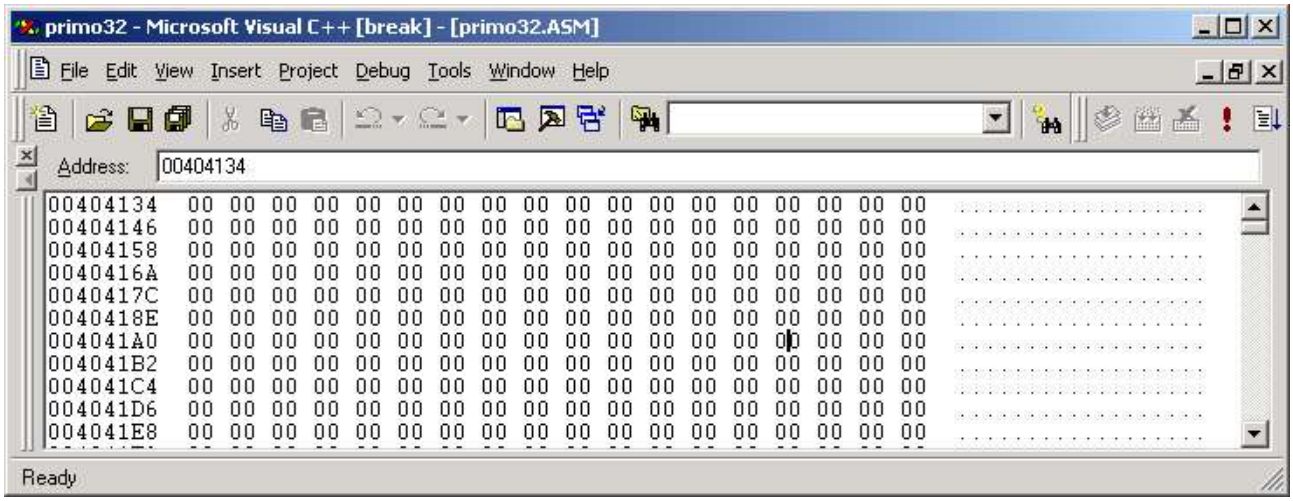

Figura 3

Ecco infine, in figura 4, un'immagine che mostra, in basso a sinistra, la finestra "call stack". Il programma sta eseguendo una procedura che sta in una DLL del Sistema Operativo (NTDLL). Questa procedura è stata chiamata da un'altra parte del S.O. (KERNEL32) e, come si vede nella finestra "call stack" è stato a sua volta chiamato dal programma principale PRIMO32. Nella parte alta si vede una finestra disassembly senza sorgente (la procedura che sta girando fa parte del Sistema Operativo Windows e sono davvero in pochi quelli che possono avere una finestra con il sorgente di kernel32.dll!). In basso a destra c'è la finestra "registers", che contiene il valore corrente di tutti i registri della CPU. A differenza di altri debugger, in Visual C++ 6.0 i valori contenuti nei registri NON si possono modificare a mano.

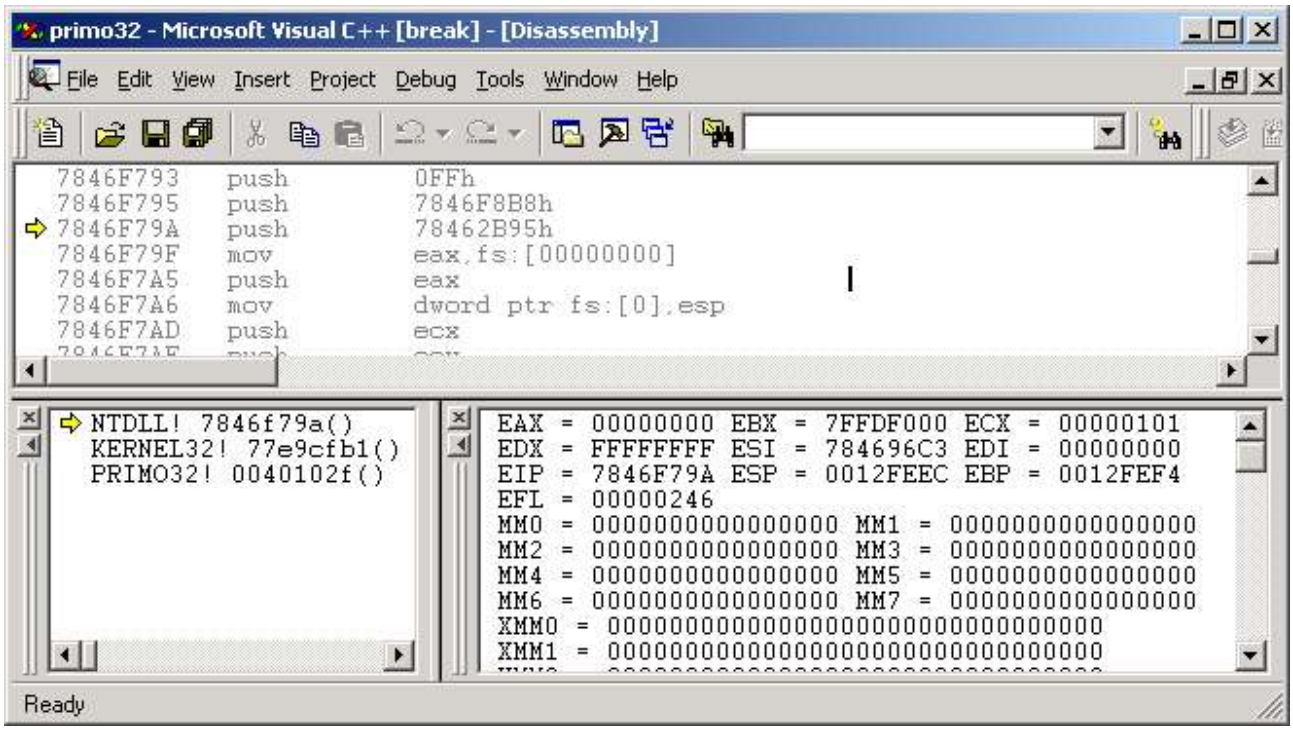

Figura 4

Esiste anche la finestra "output" (qui non visualizzata), cioè una finestra "DOS", che mostra l'uscita su console del programma (in questo caso non mostra niente, dato che il programma primo32 non ha uscite su console!).

1.0.3 Scorciatoie di tastiera (shortcut) di Visual C++ "come debugger" Assembly

Tasti	Funzione
F5	Esegue il programma "a tutta velocità", si ferma solo su un breakpoint od alla terminazione del programma
Shift - F5	Termina l'esecuzione del programma.
F10	"Step over": esecuzione passo-passo senza entrare nelle procedure
F11	"Step into": esecuzione passo-passo entrando nelle procedure
Ctrl - S	Salva il file (Save)
F9	Inserisce/elimina un breakpoint sulla riga attuale.
Ctrl - F	Cerca una stringa (Find)
Alt – 8	Finestra "disassembly": è la finestra principale, quella del codice di macchina, che può contenere anche le informazioni simboliche tratte dal file sorgente
Alt – 3	Finestra "watch": visualizza i valori aggiornati delle "variabili" in memoria
Alt – 7	Finestra "call stack":visualizza quali sono le procedure annidate una dentro l'altra
Alt – 6	Finestra "memory": visualizza in esadecimale ed in ASCII il contenuto della memoria
Alt – 5	Finestra "registrers": visualizza il valore corrente di ogni registro della CPU

Ingressi e uscite in modalità a carattere

1.0.4 Uscita sul video

1.0.5 Ingresso da console

Modalità GUI

1.0.6 Finestre

1.0.7 Output di stringhe

1.0.8 Ingresso da textbox

MASM32 e C